Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής Master thesis Title	**KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D** **KILLSHOT : Online multiplayer FPS / TPS game in Unity3D**
Ονοματεπώνυμο Φοιτητή Full name	**Αλέξανδρος Χατζηαγάπης** **Alexandros Chatziagapis**
Πατρώνυμο Father's name	**Εμμανουήλ** **Emmanouil**
Αριθμός Μητρώου Registration Number	**ΜΠΠΛ16027** **MPPL16027**
Επιβλέπων Supervisor	**Θεμιστοκλής Παναγιωτόπουλος, Καθηγητής** **Themistoklis Panagiotopoulos, Professor**

Παρασκευή 06 **Δεκεμβρίου 2019**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Θεμιστοκλής Παναγιωτόπουλος, Καθηγητής Themistoklis Panagiotopoulos, Professor	Άγγελος Πικράκης, Επίκουρος Καθηγητής Aggelos Pikrakis, Assistant Professor	Χρήστος Δουληγέρης, Καθηγητής Christos Douligeris, Professor

Περίληψη

Το KILLSHOT είναι η Μεταπτυχιακή διατριβή στο πλαίσιο του Μεταπτυχιακού Προγράμματος Σπουδών Πληροφορικής στο Τμήμα Πληροφορικής της Σχολής Τεχνολογιών Πληροφορικής και Επικοινωνιών του Πανεπιστημίου Πειραιώς.

Η εργασία KILLSHOT εμπνεύστηκε από τα διαδικτυακά παιχνίδια πολλών παικτών πρώτου προσώπου / τρίτου προσώπου (FPS / TPS). Εν συντομία, πρόκειται για ένα διαδικτυακό πολεμικό παιχνίδι πολλών παικτών που γίνεται στο Unity3D και χρησιμοποιεί το πληκτρολόγιο και το ποντίκι ως στοιχεία χειρισμού, ο τρόπος σχεδιασμού και εκτέλεσης του συνδυάζει τα χαρακτηριστικά τόσο των πολεμικών παιχνιδιών πρώτου και τρίτου προσώπου. Κάθε παίκτης πρέπει να συνδεθεί μετά από εγγραφή με δικό του λογαριασμό στο παιχνίδι και να συμμετάσχει σε ένα λόμπι γεμάτο με άλλους παίκτες, και να επιλέξει την κατηγορία και την ομάδα του, για να ξεκινήσει ένα γύρο παιχνιδιού. Κάθε παίκτης εκπροσωπείται στο παιχνίδι από έναν στρατιώτη που έχει στη διάθεσή του διαφορετικά όπλα και λοιπό εξοπλισμό ανάλογα με την επιλεγμένη κατηγορία και είναι μέλος μιας από τις δύο ομάδες. Κάθε ομάδα έχει μέχρι 5 μέλη, όταν ξεκινά το παιχνίδι, κάθε ομάδα προσπαθεί να επιτύχει τις περισσότερες εξοντώσεις εναντίον των εχθρικών στρατιωτών και έχει το καλύτερο συνολικό σκορ μέχρι να τελειώσει ο γύρος. Μετά το τέλος του γύρου ο παίκτης μπορεί είτε να επιλέξει να εγκαταλείψει το παιχνίδι, να συμμετάσχει σε άλλο λόμπι είτε να παραμείνει στο ίδιο λόμπι και να ξεκινήσει να παίζει ένα άλλο γύρο του παιχνιδιού.

Αυτό το έργο δημιουργήθηκε στην Unity3D έκδοση 2018.2.11f1. Το έργο είναι εκτεταμένο και χρησιμοποιεί πολλά κομμάτια κώδικα και στοιχεία του Unity3D, έτσι ώστε να αποφευχθεί μια εξίσου μεγάλη τεκμηρίωση, θα υποθέσουμε ότι ο αναγνώστης είναι εξοικειωμένος με το Unity3D και τα εργαλεία του και θα παραθέσουμε μόνο τα σημαντικότερα κομμάτια κώδικα με σχόλια, τα οποία χρησιμοποιούνται στο έργο.

Abstract

KILLSHOT is the MSc thesis in the context of the "Informatics" Master Program, at the Department of Informatics, School of Information and Telecommunication Technologies, of the University of Piraeus.

The Killshot project was inspired by recent online multiplayer first person/ third person shooter (FPS/TPS) computer games. In short, it is an online multiplayer shooting game made in Unity3D that uses the keyboard and mouse as input controls, the gameplay of which mixes the features of both the first and a third person video games. Every player has to login after registering with his own account in the game, and join a lobby filled with other players, select his player class and team, to start a game round. Each player is represented in game by a soldier who has different weapons and gadgets at his disposal depending on their selected class and is a member of one of two teams. Each team has up to 5 members, when the game starts each team tries to achieve the most eliminations against the enemy soldiers and have the best overall score until the round ends. After the end of the round the player can either chose to quit the game, join another lobby or stay in the same lobby and begin playing another round of the game.

This project was created in Unity3D version 2018.2.11f1. The project is extensive, and uses numerous scripts and Unity3D components, so in order to avoid an equally large documentation we shall suppose that the reader is familiar with Unity3D and its tools and we will only provide the most important scripts with comments, wherever necessary, used in the project.

Contents

Introduction

This is the technical document for KILLSHOT, a multiplayer, first and third person shooter, video game developed in the Unity3D game engine. The first chapter is an introductory chapter where we briefly discuss the characteristics of multiplayer games, concentrating mainly on the presentation of first and third person shooter multiplayer video games. We briefly introduce how these two genres are constructed and which concepts used in their implementation were adopted by the Killshot project. We then continue to a brief introduction to game engines such as Unity3D, what they are, and why they are necessary in the development of a modern video game. As mentioned, Unity3D was the chosen game engine for KILLSHOT, and so, to bring this introductory chapter to a close, we explain the reasons behind that choice and identify the capabilities of this engine.

KILLSHOT, like most multiplayer games, uses an account in order for the player to have their profile keep track of their in-game progression. Consequently, the next chapter is dedicated to the registration and login system. We explain how it was constructed, the assets used and the modifications that were made in order for it to fit the project properly. Next, we introduce Unity's lobby, the asset used to create lobbies where the players enter to set up their teams, class selections and color of selection before entering a game, while also explaining how these aspects were created and implemented.

In the next chapter, the main chapter of this documentation, we describe in depth the main gameplay and try to cover how each and every little aspect of the game was created, what purpose it serves and how it all connects together. We explain how Unity's multiplayer system works and how it was used in every sector of the project later on. In addition to the multiplayer module, we provide an extensive presentation of the analysis and design of the Killshot project. We introduce and present the modules of the developed software (i.e. login module, team/class selection module, player module, world/environment handling module, movement/action selection module, combat engagement module, sound module, etc.) as well as inter-module interfaces, user interface, etc. at the design level. We also discuss the difficulties introduced, given that we have decided to develop an online network based multiplayer application, and the implications of such a decision in contrast to developing a single player one. In this chapter, we will make use of a large amount of screenshots to make the text more explanatory and provide the reader with visual presentations.

Finally, in the last chapter, we put together our conclusions, discussing what has been learned from this effort, difficulties met on the way and how they were overcame. We then present ideas for future work, ideas, plans, and decisions to be made concerning the further development of online network-based multiplayer FPS/TPS. The presentation of the thesis ends with a Bibliography and important website addresses. As an Appendix, we have included a short user manual explaining in detail how the build of the game can be run, how to create an account and start or connect to a game server.

Multiplayer Games

Multiplayer games are video games where players are able to play at the same time and in the same environment as other players, the actions of one player will affect other players, and the result of the whole match in general. Multiplayer games can be played locally or over the network. The term 'locally' is used when the players are all connected in the same network (local server); i.e., all the computers are connected to the same router (LAN) and do not need an internet connection. This way of achieving a multiplayer connection is not as popular nowadays as it was but is still used in some cases, such as computers in cybercafés, private tournaments or simply by friends playing in the same house. The second option is connection over the internet using networking technologies

where all the players or clients connect to a server or a host responsible for synchronizing them so they can play the same game simultaneously. Most of the multiplayer game genres – e.g. shooters, strategy, role-playing, and racing – provide a multiplayer option; in this project, however, we shall be focusing exclusively on the shooter genre.

Multiplayer games are widely popular, the reason behind this success being that a person usually finds a game more competitive and interesting when there are other people playing against them, instead of using artificial intelligence. Artificial intelligence or npc (non player character) is a sector in the game developing industry still in need of considerable work and vast improvement; players in single player games usually study the behavior of npcs and try to exploit it, specifically if an npc is poorly-designed, in which case it is easy for a player to learn how to counter its movements and the point is soon reached where the game is no longer challenging for the player. This is not the case with multiplayer games, where a player can predict some strategies will take place in the game but, in most cases, will not be able to predict the enemy player's moves with precision.

Human players are able to decide on their actions, learn from their mistakes, try different strategies, and operate as a team or individually if needed; this is the reason that artificial intelligence cannot be a suitable replacement until it reaches the same potential. Of course, this is not the only reason for the popularity behind multiplayer games: in multiplayer games, players can join the game with their friends and play as a team, which makes every game more enjoyable for multiplayer fans. The ranking systems contribute to the popularity of multiplayer games, also making them more challenging and motivating the players to compete against other players in their ranks, making them desirous of reaching the highest rank possible based on their skill level. Most successful multiplayer games organize tournaments where the top players and teams compete against each other to win prizes and recognition, encouraging fans and players to want to become successful in the game. This usually leads to more new players joining the game and the older players playing more often, striving to earn a higher rank.

Shooter Games

Shooter games are a sub-genre of action games where the players is often depicted as a soldier with weapons gadgets and other accessories at his disposal aimed to help him eliminate his enemies. Multiplayer shooter games currently have a variety of different game modes such as team deathmatch , deathmatch, domination, capture the flag , battle royal, and many more . KILLSHOT's game mode is team deathmatch, where two opposing teams compete against each other on a selected map, the players of one team trying to achieve the most eliminations of the players of the opposing team until one team is the clear winner.

The camera view of shooter games might vary at times, although the most popular cases are first person (FPS) and third person shooters (TPS). There are some more options, such as *top down* or a fixed camera overlooking the map but these are not as popular as the first two. A first person shooter is a shooter genre where the game camera represents the eyes of the player and is centered on his hands, which are usually holding his main weapons and accessories. The concept of the genre aims at giving the player the experience of real combat / war and depicts the situation as realistically as possible. A third person shooter, on the other hand, is very similar to a first person one with the main difference being that the camera is now located behind and slightly above the player, giving him a wider field of view and a clearer perspective of the players' location on the map. Both of these concepts were adopted for KILLSHOT, mainly a third person game with the camera placed behind the player but switching to first person view when the player is aiming. In this way, the player can have a wider field of view when navigating around the map but be more precise in their aim when encountering an enemy player.

Game Engines

Creating a game from scratch is an extremely difficult task due to the fact that, from the outset, a programmer needs to have a vast variety of tools at their disposal; in simpler terms, they need a way to project graphics onto the screen, simulate physics, implement code/logic, sound, animation, display UI elements and many more aspects involved in making a game as complete as possible. It is clear that if a developer wanted to make a game on his own without the use of external tools, they would first need to create each one of these components beforehand, a difficult and time-consuming task. Even for an experienced developer, it would take at least a couple of years before he could be able to move on to the actual development of the game.

To avoid this problem, developers and companies use Game Engines. A game engine is a software development environment, an editing program, created to provide it's user with all the necessary tools to create a game. Game engines usually provide a 3d world editing space (environment): a rendering engine, a physics engine, tools to implement code, animation, sound and many more game features. As we mentioned, creating a game engine is an extremely challenging task many times harder than creating an actual game and that is the reason behind why indie developers, smaller studios and even big companies often resolve to use a third party game engine to develop their titles. There are, however, some companies that choose to develop their own game engines mainly to be independent, avoiding the need to adapt to the changes that others might make to an engine, being able to develop their tools in such a way as to better assist their titles. It is logical that engines created by gaming studios are company property, are not free, and consequently for other developers to gain access to them they have to buy the engine or pay the subscription, always given that the gaming studio that developed the software is willing to put it up for sale. This is why indie game developers and gaming studios that cannot or are not willing to create or buy a game engine often resolve to using free game engines. The two most popular free game engines that are currently on the market are Unity3D and Unreal Engine, each with their own strengths and weaknesses. Of course, there are other free game engines available, but so far they cannot reach the capabilities of the first two mentioned.

Unity3D Engine

Between Unity3D and Unreal Engine, Unity3D was chosen as the game engine with which to develop KILLSHOT, mainly because Unity3D was easier for developers at an early stage. Also Unreal Engine only recently became free, while Unity3D had been a free engine for quite some time and consequently there was far more knowledge on how to use it. Its main advantages are as follows:

- It is a relatively easy game engine for beginners due to its simple and modular user interface and plug in tools.
- It currently has a vast amount of free or paid assets available on its store, the Asset Store, some of which created by Unity but most of which submissions from indie unity developers and modelers.
- It is compatible with all OS available including Windows, Mac, Linux, Android and IOS.
- It is compatible with more or less every new Console currently on the market including PlayStation 4, Xbox and Nintendo Switch.
- It supports most new software trends including Virtual Reality, Augmented Reality, Mixed Reality and Networking.
- It has a vast number of tutorials due to the huge number of developers and studios that use Unity3D for their own projects. An answer can easily be found online to many challenging problems since many have faced that problem in the past and shared their solution.
- Unity is well connected with many important companies, meaning that they are cooperating, or will cooperate in the future, to add more functionalities to the engine. For example, Unity has already published two new rendering pipelines and announced a new and improved physics

engine, visual scripting tools, landscape generating tools, networking system and many more innovations besides.

Unity3D is an immensely flexible and powerful tool used not only for creating games but also for seemingly endless other uses such as creating simulators, augmented reality applications, android applications or hologram projections. Its main benefit against other application development programs such as Android Studio is how easily it can implement the 3D element and make the application much more immersive.

This project was created in Unity3D version 2018.2.11f1. The project is vast and uses a large number of Unity components, so to avoid equally large documentation, we will suppose that everyone is familiar with Unity and its tools. To avoid confusion, we shall endeavor to explain how this project was created, and the way it works, with the least demonstration of code possible. A brief introduction to this game, and multiplayer games in general, is followed by an explanation as to how the login system works, how the accounts for each user were created, and the purpose they serve. In the third chapter, we shall describe how Unity's lobby asset works and the way it was modified to fit our project and provide a complete matchmaking service. In the fourth and final chapter, we shall describe the main gameplay and try to cover in depth how each and every aspect of the game was created, the purpose they serve and how these all connect.

Login scene

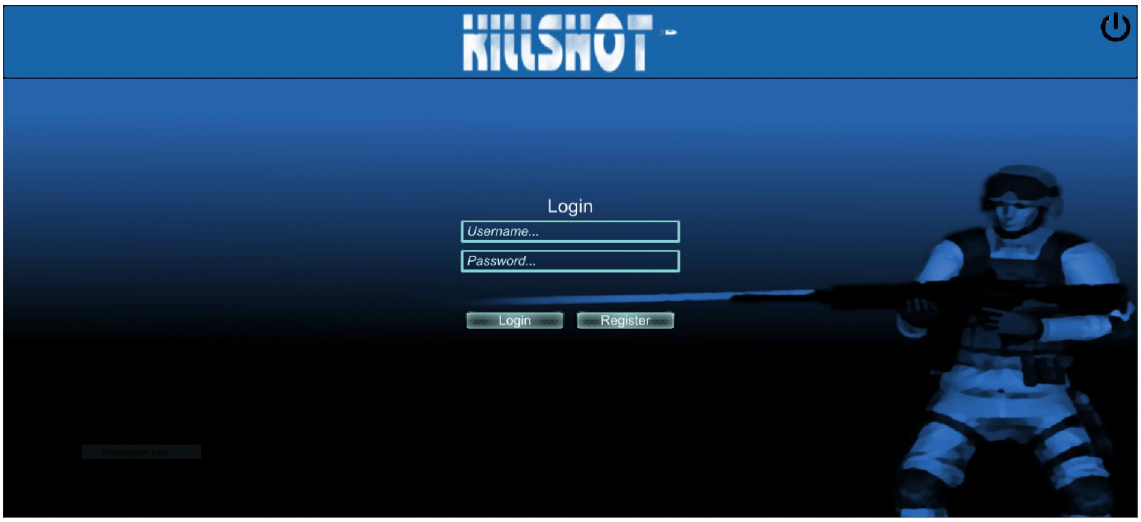
The main idea behind KILLSHOT was the development of a multiplayer-focused game, so the need to implement an account-based system was inevitable. Every multiplayer game has player accounts; some games may have an option to log in as a guest player, although this option cannot record the player's in-game statistics and score. When a user likes a game, he usually wants to create an account, in this way gaining access to an abundance of features such as keeping track of his records, his performance, how long he has been playing the game, or to be able to maintain a list of friends.

Big commercial games usually have large databases behind them, enabling the storage of all the player's accounts, as well as the details of every account, the player's progress and the in-game currency. These databases are mostly written in SQL and populated with large tables containing all the details needed; in our case, however, there is no need for such a vast database, although it would be relatively easy to build one using SQL in Unity3D.

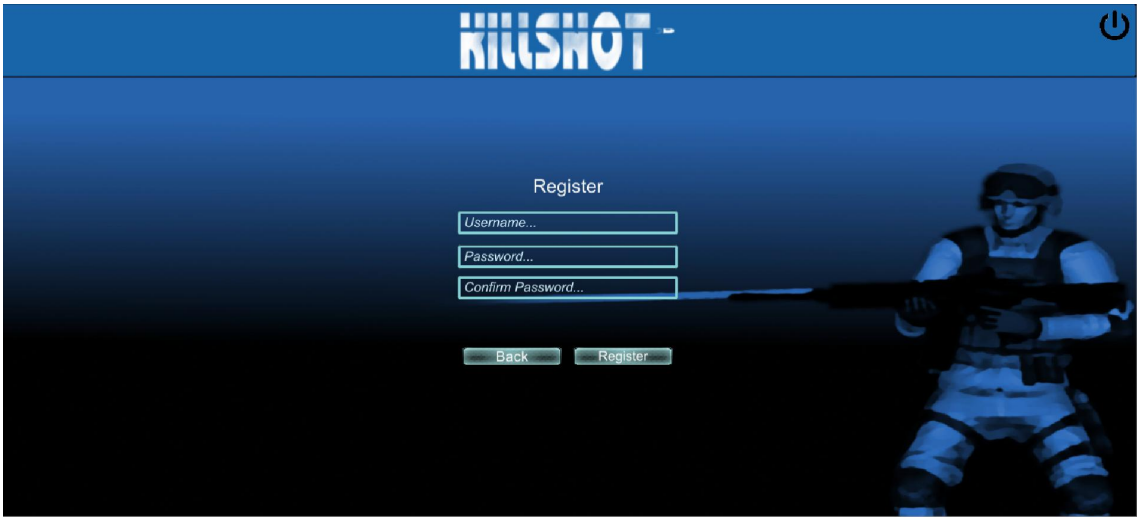
Database Asset

To implement a database without the use of SQL, a free Unity Asset was added to the project called free database control. This asset provides an internet-dependent online database and some built-in functions that are responsible for sending and retrieving the data to and from the database. This asset also includes a login and register form along with their script controllers, most of the code and the user interface being modified to meet the specifications of the game. The login script of the asset was modified to store the kills and deaths of the user in a string format that we shall explain in the Data Translator chapter below.

When the game begins, the login menu pops up and the user has to log in before being able to enter the main game content. If the player does not have an account, they must press the REGISTER button to be transferred to the Register menu where they have to enter their desired user name – the name is accepted only if no other user has claimed it for their account – and their password twice, just to make sure they have entered it correctly. If the player already has an account, they just have to enter their username and password in the login menu and press the LOGIN button. In each case after a successful register or login, the user will be transferred to the lobby menu. Between these two scenes, a preloader script, provided by the free database control asset, is employed to make sure that the game is ready to change scenes.



Login page



Register page

User Account Manager

The functions for sending and retrieving data from the asset were copied and transferred to a new script called User Account Manager. This script is intended to be the main controller for every aspect of the user's account. The functions for sending and retrieving data had to be modified to search the database based on the username of an account to avoid account confusion based on the fact that the username is unique. A login and a logout function were added; the first of these is responsible for storing the user's username after a successful login, and the second to erase those details after a user decides to log out. This script is very important for the game as it is used to synchronize the player's score after a kill or a death, and in this way they will be able to retrieve these values every time they access the game and log in, just as with an online SQL database. For example, when the player logs in, we will employ the 'get data' method to retrieve their score, and when the player kills another player in the game, we will employ the 'send data' method to add the new kill to his previous score.

```
1.  //this script manages the account of each player
2.  //it is responsible for login, logout ,sending and getting data
3.  public class UserAccountManager: MonoBehaviour
4.  {
5.      public static UserAccountManager instance;
6.      public static string playerUsername { get; protected set; }
7.      private static string playerPassword = "";
8.      public static bool IsLoggedIn { get;protected set;  }
9.      public static string LoggedInData { get; protected set; }
10.     public string LoggedInSceneName = "Lobby";
11.     public string LoggedOutSceneName = "LoginMenu";
12.     public delegate void OnDataReceivedCallback(string data);
13.     private void Awake()
14.     {
15.         if (instance != null)
16.         {
17.             Destroy(gameObject);
18.             return;
19.         }
20.         instance = this;
21.         DontDestroyOnLoad(this);
22.     }
23.     // logs the user out
24.     public void LogOut()
25.     {
26.         playerUsername = "";
27.         playerPassword = "";
28.         IsLoggedIn = false;
29.         SceneManager.LoadScene(LoggedOutSceneName);
30.     }
31.     //logs the user in, given the correct username and password
32.     public void LogIn(string username , string password)
33.     {
34.         playerUsername = username;
35.         playerPassword = password;
36.         IsLoggedIn = true;
37.         SceneManager.LoadScene("Lobby");
38.     }
39.     public string SetPlayerName()
40.     {
41.         return playerUsername;
42.     }
43.     //gets data based on username
44.     public void GetData(string playername,OnDataReceivedCallback onDataReceived)
45.     {
46.         if (IsLoggedIn)
47.         {
48.             StartCoroutine(sendGetDataRequest(playername, playerPassword, onDataRe
    ceived));
49.         }
50.     }
51.     public  IEnumerator sendGetDataRequest(string username,string password,OnDataR
    eceivedCallback onDataReceived)
52.     {
53.         //Send request to get the player's data string. Provides the username and
    password
54.         IEnumerator e = DCF.GetUserData(username, password);
55.         while (e.MoveNext())
56.         {
```

```
57.             yield return e.Current;
58.         }
59.         //The returned string from the request
60.         string response = e.Current as string;
61.         if (response == "Error")
62.         {
63.             playerUsername = "";
64.             playerPassword = "";
65.             Debug.Log("Error: Unknown Error. Please try again later.");
66.         }
67.         else
68.         {
69.             onDataReceived.Invoke(response);
70.         }
71.     }
72.     //sends data based on username
73.     public void SendData(string playername,string NewData)
74.     {
75.         if (IsLoggedIn)
76.         {
77.             StartCoroutine(sendSetDataRequest(playername, NewData));
78.         }
79.     }
80.     IEnumerator sendSetDataRequest(string playername,string data)
81.     {
82.         if (IsLoggedIn)
83.         {
84.             //Send request to set the player's data string. Provides the username,
     password and new data string
85.             IEnumerator e = DCF.SetUserData(playername, playerPassword, data);
86.             while (e.MoveNext())
87.             {
88.                 yield return e.Current;
89.             }
90.             string response = e.Current as string; // << The returned string from
    the request
91.             if (response == "Success")
92.             {
93.                 data = response;
94.             }
95.             else
96.             {
97.                 playerUsername = "";
98.                 playerPassword = "";
99.                 Debug.Log("Error: Unknown Error. Please try again later.");
100.             }
101.         }
102.         LoggedInData = data;
103.     }
104. }
```

User account manager script

Data Translator

The user's account stores the player's username, password (not visible to others), his total kills, and deaths throughout the games they have played. The database prefab used can store three values:

the username, the password and a string for data storage. The way this prefab works is quite limiting because we do not have a large variety of variables where we can store data that we want to keep track of, such as the hours played in the game, the average score of the player or, in our case, the kills and deaths. There is a way however, that we can achieve data storage, and that is through the data string provided by the database import mentioned earlier.

The way the system works is that we store all our data concatenated into a large string, and we have to use a dedicated script to unwrap it and wrap it back up again. For this reason, we created a data translator script that splits a string into sub-strings based on the symbol '/' and then we analyze each sub-string. The project stores the player's kills and deaths so we only need to mark the sub-strings with two more symbols at the start of each sub-string, the kills have the keyword/symbol "[KILLS]" in front of them, and deaths the keyword "[DEATHS]". Subsequently, through the player's networking script, each time they get a 'kill', we call the data translator, we split the data string looking for the keyword "[KILLS]". Then, we take the number after it, which indicates his previous kills, update it and reverse the process to store it back in the data string again; the same process is followed for deaths. This way we can easily store most of the statistics needed for our game with only the use of one string. Of course, we could add more statistics like the number of assists, though we would have to add another keyword such as "[ASSISTS]" and modify the data translator script to search for this keyword after it splits the data string into smaller sub-strings. Instead of synchronizing the game at the end of a round, we synchronize the statistics each time a player kills another player in game. After a kill, we employ the data translator for both players to assign the new values to their account after a small delay, in this way ensuring that, even if a player leaves before the round ends, the accounts will be always up to date.

```
1.  using System.Collections;
2.  using System.Collections.Generic;
3.  using UnityEngine;
4.
5.  //this script wraps and unwraps the data string of the user's account and translat
    es it to
6.  //his in game score. This script is going to be referenced from players when a kil
    l or death occurs
7.  public class UserAccountDataTranslator : MonoBehaviour
8.  {
9.      private static string KILLS_SYMBOL = "[KILLS]";
10.     private static string DEATHS_SYMBOL = "[DEATHS]";
11.
12.     //stores the players new scores into his data string
13.     public static string ValuesToData(int kills, int deaths)
14.     {
15.           return KILLS_SYMBOL + kills + "/" + DEATHS_SYMBOL + deaths;
16.     }
17.     //retrieves the player's stored kills from the data string
18.     public static int DataToKills(string data)
19.     {
20.         return int.Parse(DataToValue(data, KILLS_SYMBOL));
21.     }
22.     //retrieves the player's stored deaths from the data string
23.     public static int DataToDeaths(string data)
24.     {
25.         return int.Parse(DataToValue(data, DEATHS_SYMBOL));
26.     }
27.
28.     //splits the datat string to substring based on the "/"  character
29.     private static string DataToValue(string data, string symbol)
30.     {
31.         string[] pieces = data.Split('/');
```

```
32.         foreach (string piece in pieces)
33.         {
34.             if (piece.StartsWith(symbol))
35.             {
36.                 return piece.Substring(symbol.Length);
37.
38.             }
39.         }
40.         Debug.Log("error symbol not found ");
41.         return "";
42.     }
43. }
```

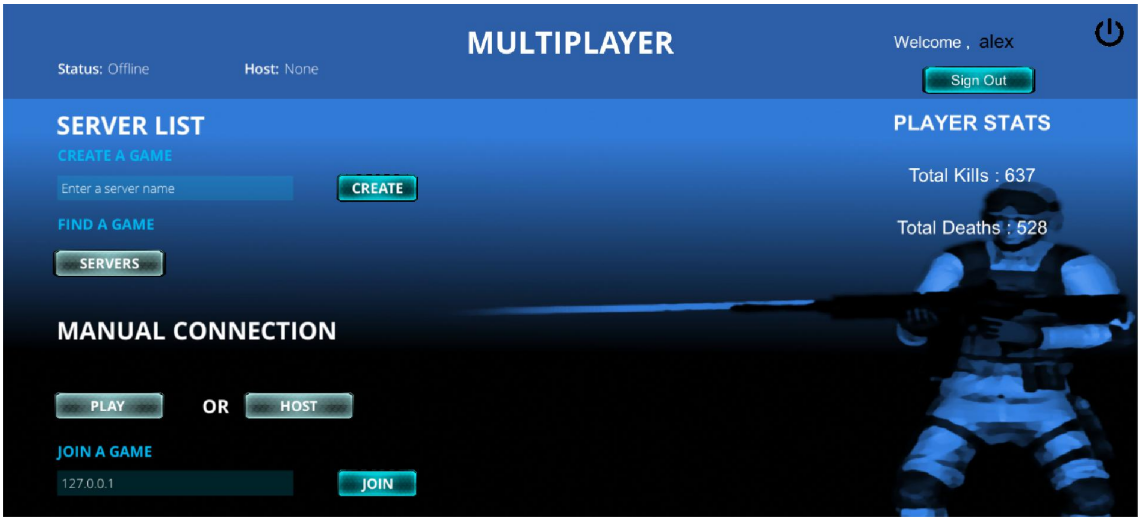
User account data translator script

Lobby scene

Lobby

Directly after the player has successfully logged in to his account, he is transferred to the lobby menu. The lobby was necessary to give the player the ability to choose the server room they want to enter; maybe they will choose a server with the less ping possible or a server room their friends have created for team play.

To implement a functional lobby, the free asset unity lobby was used from the asset store. This asset enables players in an online game to connect first to a lobby to organize gameplay details such as: game options, number of players, player color, desired team, etc. The User Interface (UI) is controlled by the Lobby Manager script, which also controls all the players' networked actions: in simpler terms, the lobby manager decides the rules of the game. The lobby has two functionalities: it can work with the online servers and in local/manual mode.

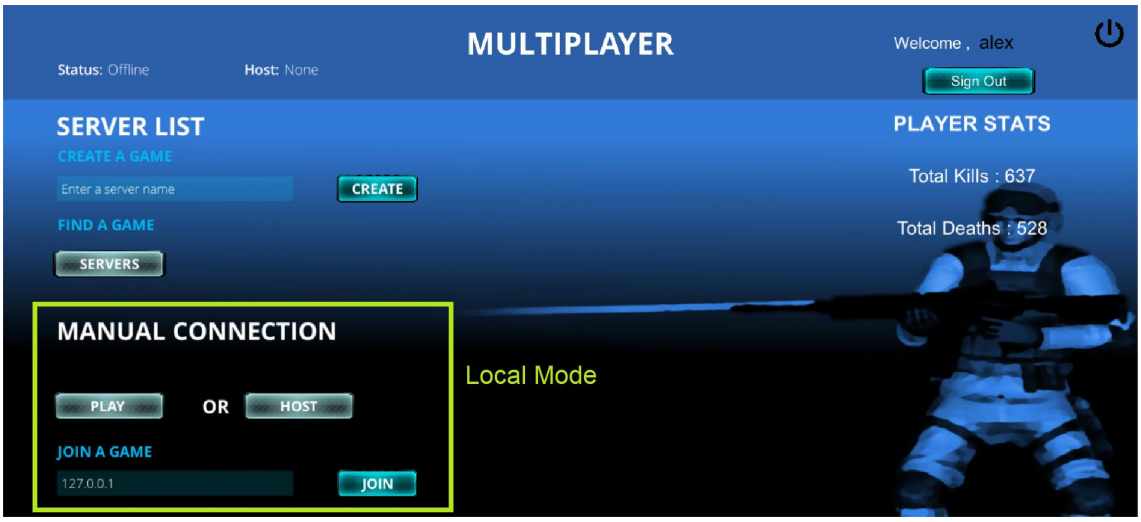


Local mode

In the local mode, the user can open multiple instances of the game on the same computer or open the game on different computers connected to the same network (router) and manually connect them in the localhost lobby.

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

This mode is used mainly for testing purposes when a game is under development and has not yet reached online status or when all the players are in the same network (LAN mode) and connecting to the network would actually be a lot slower. For example, if a group of friends decides to organize a tournament among them playing the game in the same house, the local/manual mode is the best choice because they can directly host and join the game without the use of an online server. That means their game will not be network dependent and, as a result, there will be no delay in their connection. They will be able to connect their computers through the router/modem using the IP of the network, and the host will synchronize the game much faster.

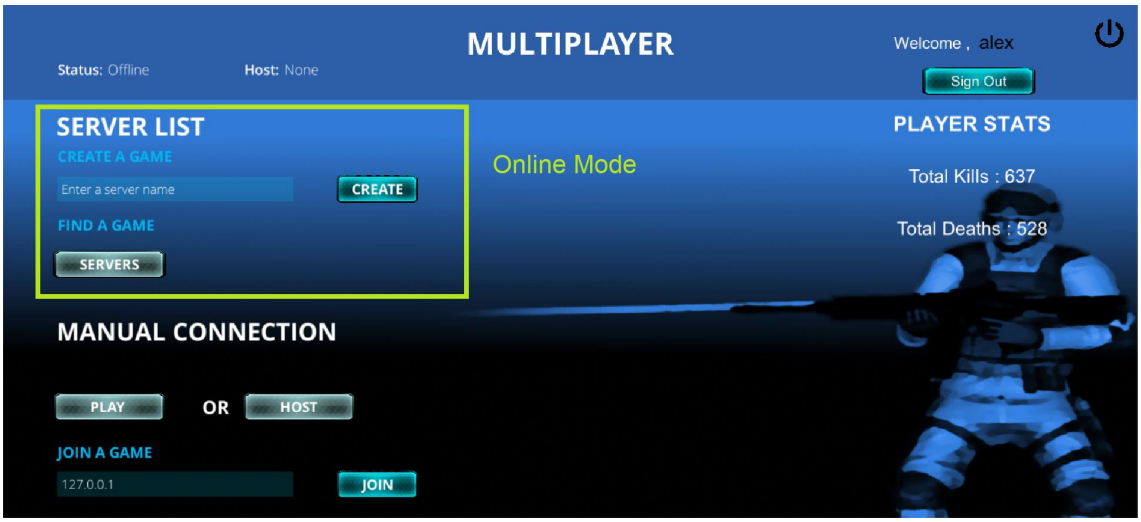


Lobby - local mode

Online mode

On the other hand, if the players are not located in the same network they will not be able to join the same localhost room, and will have to connect through a server that will keep them connected and synchronized. This brings us to the second option: the online mode, which uses the same principles as the localhost mode but this time, instead of creating a room locally, the user creates a server room that anyone with an internet connection using the game can see and join.

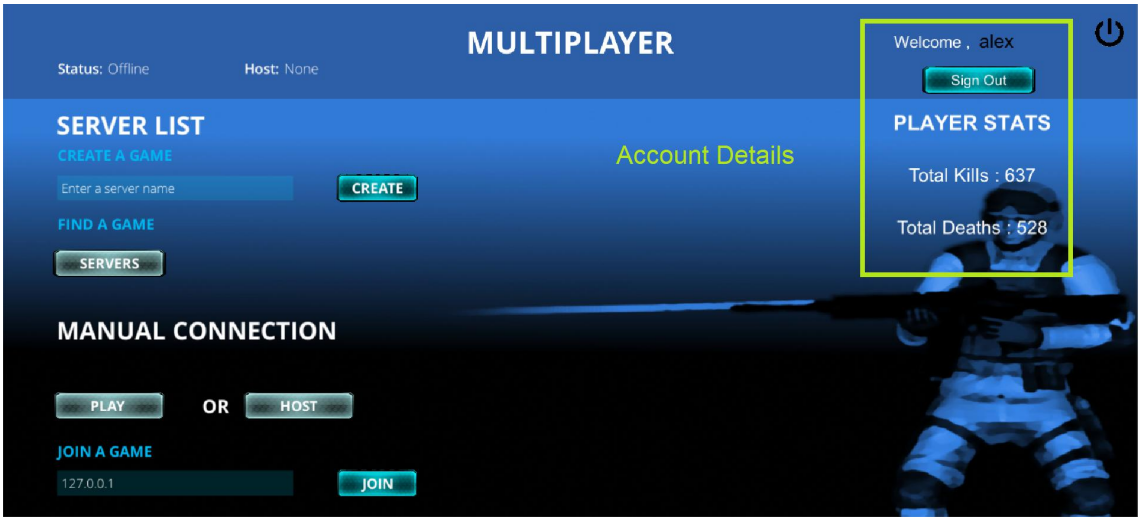
Before using the online UNET services, we had to enable them by going to the services panel and then clicking on the multiplayer option and selecting the ‘Go to dashboard’ option. The browser opens in the Unity page, which requires a login, after which we set the game’s multiplayer options such as the maximum players in a game room. When we have fully configured our project, we merely click on the SAVE option to enable the online services. It is important to mention that we can measure and monitor the project’s traffic from concurrent users on this page.



Lobby - online mode

Account

In the top right corner of the panel, we can find a welcome message followed by the username and a SIGN OUT button below it. Through a script added to the lobby asset, we display the username and, if they sign out, we log out the user and return to the login scene where they can login with a different account. Below the SIGN OUT button, we can locate the player's statistics, which consist of their total kills and deaths. This time, we check through another script the ID with which the user is logged in, and, based on his username, we retrieve his statistics and display them on the interface.



Lobby – player details

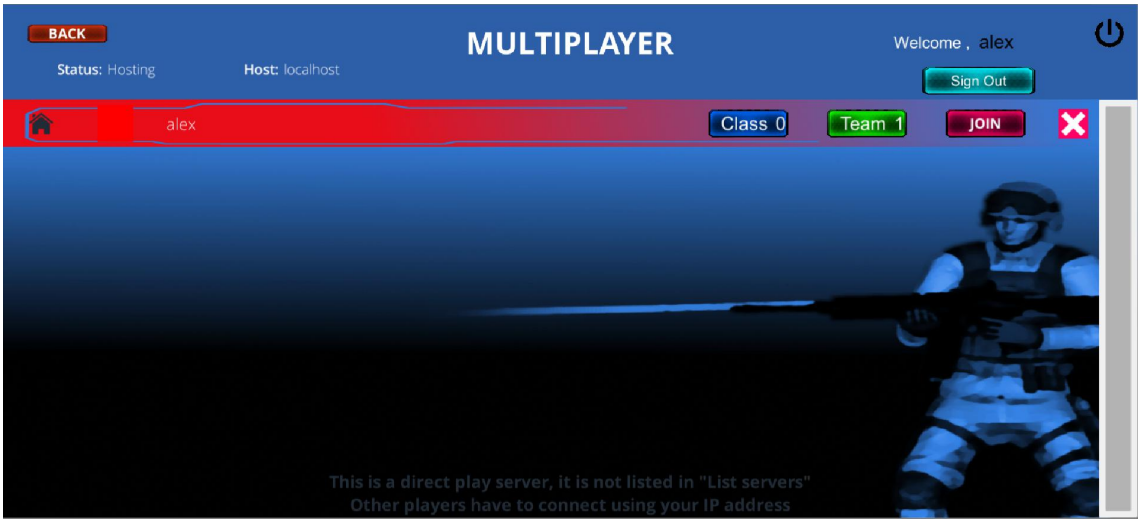
```
1.  public class UserAccountLobby : MonoBehaviour
2.  {
3.      public Text UsernameText;
4.      LobbyManager lobbyManager;
5.      // Use this for initialization
6.      void Start ()
7.      {
```

```
8.          //if the player is logged in we retrieve his account name
9.          //and display it in the users account details UI
10.         if (UserAccountManager.IsLoggedIn)
11.         {
12.             UsernameText.text = UserAccountManager.playerUsername;
13.         }
14.         lobbyManager = this.GetComponent<LobbyManager>();
15.     }
16.     //function for user log out
17.     public void LogOut()
18.     {
19.         if (UserAccountManager.IsLoggedIn)
20.         {
21.             UserAccountManager.instance.LogOut();
22.             Debug.Log("loged out");
23.             Destroy(lobbyManager);
24.             Destroy(gameObject);
25.         }
26.         else
27.         {
28.             Debug.Log("did not log out");
29.         }
30.     }
31. }
```

User account Lobby script

Lobby player

In both cases, when a player enters a lobby, a game object called lobby player is created; this represents the player in the room and is used to determine certain in-game player values such as the name of the player. In this case, the lobby player prefab was modified to enable the player to change his color (every player has a unique color), his class (there are 4 different player classes for the user to choose from), and his team (he can pick between Team 1 and 2). The player's name is also transferred but cannot be modified, the name taking its value from the player account when a user registers or logs in to the game.



Lobby player example

Every value the lobby player chooses will be applied on the game player the moment he enters the main game scene: the mini-map image and his name will be in the same color he picked, the player prefab created in the game will be based on the class selected, the color of the player clothes will be either white for Team 1 or green for Team 2. To achieve this customization, the scripts of the lobby asset were changed accordingly. The lobby player prefab was modified, with the addition of a button for class selection and a button for team selection.

Player Class

The class button increments a player class counter every time the user presses the button, and resets when it reaches the max limit. The user can select Class 0, 1, 2, or 3. Each class corresponds to a different player game object, which means that the player of each class can have a different character model with other animations and a particular inventory of weapons and gadgets. In this game, all player classes have the same model for the simple reason that it would have taken more time than was available to find three or four more player models, rig them, animate them through Mixamo, swap the models of every class, the animator controllers and adjust the weapons/gadgets on them. What is important is that we are not limited to one player model; should we so wish, we can change player models in the future. The system can support as many different classes as needed with minor modifications to the class number max limit but for the sake of simplicity it was limited to 4 classes. Below we explain in simple terms how this different player prefab selection was achieved.

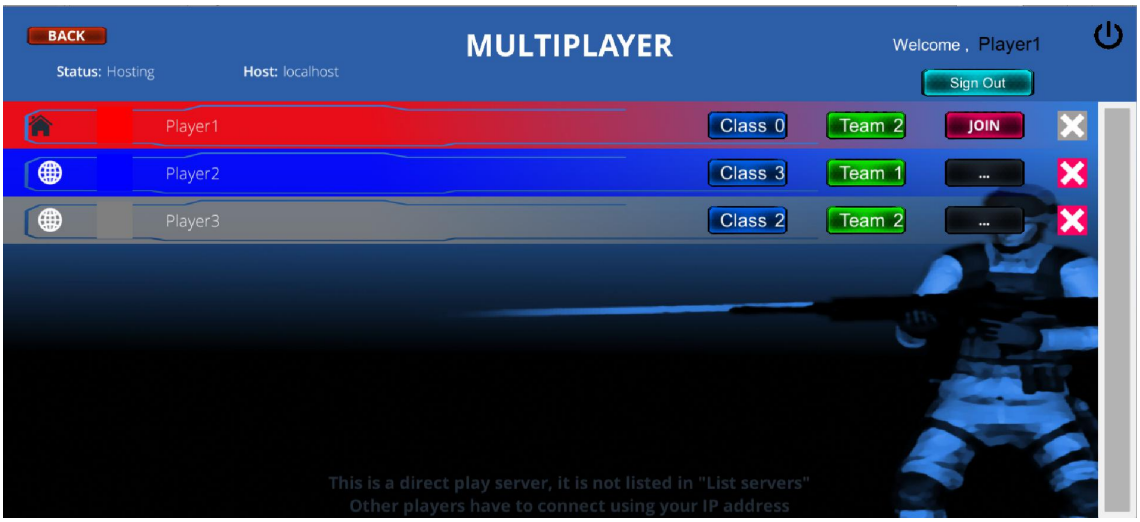
The functions that create the lobby player and the game player were overridden to enable custom player selection depending on the player class number. Using a dictionary, we store the player connection ID number and the player class number; every time a player changes class, by pressing the CLASS button, we update the class number on the dictionary, and in this way spawn the selected player based on the class number that was stored on the client's connection ID. This modification was needed because only the server/host has the ability to spawn players and consequently the clients needed a way to inform them with which class they want to spawn in the game.

Teams

When the user clicks the TEAM button, the value of the team number of the lobby player changes and can take the values 1 or 2, meaning they will either join as a player in Team 1 or Team 2. Again, we could have more than two teams but two were enough for a simple Team Death Match mode. The player's model will be colored according to the team they are in, with Team 1 soldiers wearing white and Team 2 soldiers wearing green. This was necessary to facilitate team differentiation, and the way it was achieved is that we assign a different texture image to the player's model according to their team. In reality, both teams use the same texture image that was manipulated in GIMP to create a white and a green variation.

Player color

By default, the lobby player has a COLOR button that is unique for every player and changes to an available color when clicked. The scripting of the player COLOR button was also modified to change the background color of the lobby player and make it clear which player is which. The color of each player will not only remain in the lobby to which it is transferred in game: the player's name on the scoreboard, the kill feed item, and above their head will be colored accordingly. This technique is mostly used in MOBA (strategy) games where 10 players compete in a 5 versus 5 match, but KILLSHOT is also designed to host up to 10 players, and thus being able to distinguish players based on color is useful.



Lobby player example 2

Lobby hook

Every time a button from the lobby player is clicked, a command is sent to the server to synchronize the value changes to all the clients, but this only happens inside the lobby when the players spawn. We therefore need a way to pass these values to the game players, otherwise they are going to begin with the default values. For this reason, a script to connect the lobby player and the in-game player was created called NetworkLobbyHook. This script extends the class LobbyHook used to connect these two game objects. Inside the script, we declare that we want the values ‘color’, ‘name’ and ‘team’ to be transferred from the lobby player to the game player. The class filed is not a value to pass through to the in-game player, it is referring to the game player object itself, there are 4 different prefabs of the soldier each with different items in his inventory, and we decide which prefab to spawn for each player based on the class that he selected.

After all the players have joined the lobby and set up their player preferences, the only thing left to do is to join the server. Each player has to click on the JOIN button, and, when they have all done so, a countdown timer starts and transfers them into the main game scene when it reaches the value zero.

Main game scene

The game and its functionality is controlled by the unity lobby and the network manager, both of which are unity components created to support multiplayer games in unity. At the start of a round, all players spawn randomly to one of their team’s spawn-points and, after 3 seconds, the details from the lobby player, such as the color and player name, are synchronized, too. When a game round ends, all players are transferred back to the lobby scene where they have the option to join a server room again and start a new game round or shut the game down.

Map design

The main gameplay depends solely on the players and the interactions between them. The rest of the map is a simple game scene with a small terrain representing a desert and some desert environment assets such as buildings, rocks, trees and boxes free of charge from the site DevAssets. The map design was not the focus of the project; the point was to design a solid game that can withstand any map design modifications in the future.

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

Desert map

Game logic

We can now conclude that understanding how the player works is very close to understanding how the whole game works. The (network) player is a Game Object with various scripts and other game objects attached to it each for a different purpose. At the start of the game, all the scripts attached to the player start running in parallel, communicating with each other, which makes the task of explaining the project rather difficult. In the following steps, were shall try separating the project into smaller sectors and give a brief explanation for each one in terms of the way they work and their purpose in the project.

Gameplay

The logic of the game is similar to modern multiplayer shooter video games and is an attempt to simulate a modern war scenario between two teams. The player of the game is represented by a soldier who can move in various ways and fire his weapons or use his gadgets in order to eliminate his enemies. The goal for a player is to score the most eliminations and have the least amount of fatalities possible. Of course, solo scores are important but it also important that the player helps their teammates, tries to function as a unit and utilizes in the best way the four disparate classes. A team with every player using the assault class is good for close combat but bad for ranged pick offs, and a team full of snipers is strong at long-range pick offs but significantly weak in close combat.

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

Gameplay preview

Game mode

The mode of the game is a commonly used mode called Team Death-Match. In this mode, two or more teams compete against each other until one of the teams is the clear winner or the time runs out. In KILLSHOT, there is no time limit; consequently, the end of the round depends only on the scores of each player in a team. There are two teams – Team 1 dressed in white and Team 2 in green. In some games, friendly fire is activated, which means that teammates can w and kill one another. In our case, friendly fire is disengaged and players can only inflict damage on and kill players in the enemy team, which leads to less confusion inside the team and makes the existence of teams an important factor in the game.

Each player spawns at a spawn-point in the scene; after all the players have entered the scene, their team name and team color are assigned. Each player has at his disposal weapons and gadgets based on the class they have selected. The players can fire their weapons and inflict damage on or kill a player in the enemy team, but cannot inflict damage on a teammate. After a player is killed, he re-spawns in full health at one of his team's spawn-points. The death and kills of the players are available on the scoreboard, the goal being for each player to have the best score possible and for his team to beat the enemy team. After a player reaches a specific amount of kills, the game ends and players return to the lobby scene. This can be changed in future to terminate a game round based on the team's total score and also store a win or lose for every team member.

Networking

Networking is probably the most important aspect of the project, as, in multiplayer games, each user is running an instance of the game or a copy of the game, and it is up to the server to synchronize all the game instances of the users to make it look as if they are all playing in the same instance of the game. In simpler terms, when a player moves in his instance of the game, the other players connected on the same server have to see the player do the exact same moves in their instance, and this needs to be done in real time. Timing is a key factor here and it is crucial that all the instances of the game are running in parallel and every action of a player is transferred exactly as it is to the other players the moment it is done. For example, if a player shoots his weapon, the action has to be synchronized across the network at exactly the same time; otherwise, if the action is transferred with a delay, the server is thrown out of sync, changing the original outcome of his action.

Unity has its own system of networking called UNET or Unity Networking which allows the development of multiplayer games best suited to small, multiplayer projects such as this one, a 5 vs 5 player game. This system will soon be depreciated and replaced with a new and better networking system, but for this project UNET will be sufficient. There is also another way to implement networking in a unity game, and that is through the engine called Photon, a free plugin that we can use which provides better servers and a lighter system allowing the development of larger scale multiplayer games such as open world role-playing games (MMO RPG). This project is implemented entirely in UNET but the transition to Photon is easy, the only changes needed being the bits of code used for networking; every other gameplay factor or script will remain intact.

UNET

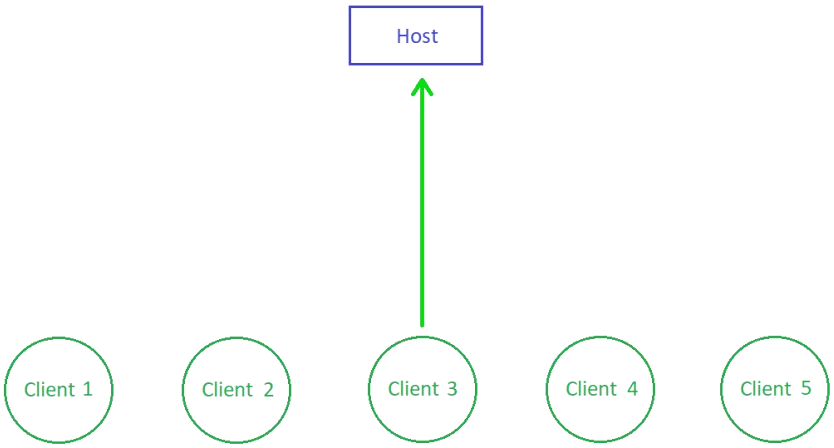
UNET works in the following way: it requires a game object – the server, usually named the network manager – and each object that needs networking elements must have a network ID, with every other object behaving in a normal non-multiplayer way. In UNET, the server will be the first user to host a room, and the rest of the players that entered this room will be the clients of the server.

While playing the game, the network manager will monitor the behavior and movement of every networked object and transmit those changes to all the instances of the game in real time so that each user will have the same version of the game running on his computer. In this project, the network manager will be created directly from the lobby plug in mentioned earlier, using the network manager component that unity offers. The soldier and the projectiles will have a network ID component from unity to indicate their location on the scene, and their actions need to be synchronized across the server. We do not need to add any networking components to the other objects in the scene because the rest are static objects used to design the map that remain the same throughout the game, such as buildings and rocks.

CMD-RPC

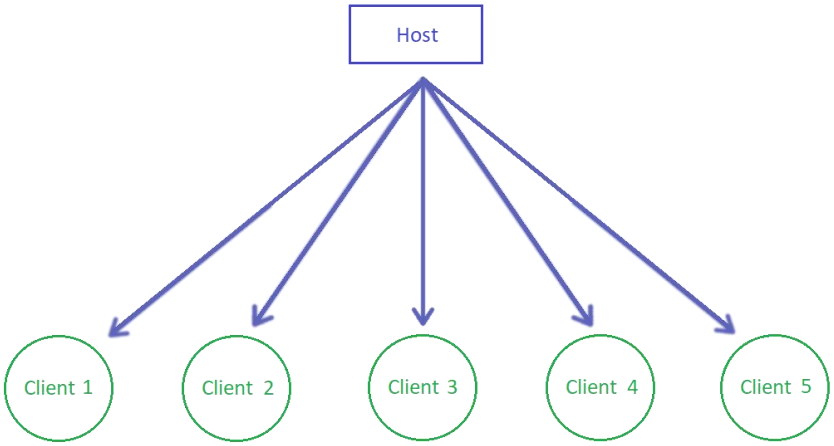
The terms Cmd – Command – and Rpc – Remote Procedure Calls – are widely used in UNET and by extension in this project. Cmds and Rpcs are Unity functions specifically used when we need to synchronize an action across the network. To understand these principles a little better, let us suppose that a client decides to fire his weapon; there is a specific methodology followed to transfer his action among all players. To begin with, when the client fires their weapon a Cmd is sent from the client to the host/server to inform them that the client fired their weapon, and this action is immediately displayed in the client's scene. The Cmd alone is, of course, not enough: this action only takes place in the client's copy of the game, and the other players including the host will not see this action in their copies of the game. The host/server is the one that controls all the networked actions, while a client does not have the authority to make changes to the game – their actions have to go through the host first and the host will then have to inform the other clients of their action. Which brings us to the next part of the methodology: the Rpc. An Rpc is a method sent from the server back to all the other clients, so that every client will display the same action taking place, regardless of who made it. Usually the client calls a Cmd, which in turn calls an Rpc that has the actual shooting code in it, and, in this way, the same piece of code runs simultaneously in all clients. To recap the way synchronization in UNET works:

- The client calls a Cmd (Command) and messages the server.



Client 3 send CMD to host with its action

- The Cmd is sent to the server and the server then calls the proper Rpc (Remote Procedure Call).
- The Rpc containing the actual code is then sent to every client in the game.



Host sends RPC to all clients informing them of client 3's action

- The same piece of code runs in all clients' versions of the game.

This way, in our example, after the Rpcs are executed in every client's version of the game, all the players will see that the initial client has fired his weapon. The same principle is used for every action that needs networking authority such as switching weapons/scopes/attachments or deducting a player's health points after being hit by a projectile.

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

Most of the problems that occur when games are being converted from single-player to multiplayer are due to lack of understanding of this principle. In single player games, there are several ways to implement a mechanism; on the other hand, multiplayer games developed using UNET have a limited amount of ways to implement it. Small details such as switching scopes and attachments or even transmitting sound effects have to use Cmds and Rpcs to achieve synchronization and avoid networking issues. However, we have to keep in mind that constantly calling commands on the network is a risky move that requires a great deal of processing power and bandwidth. We need to optimize the game in a way that only uses commands over the network when it is absolutely necessary for the game, and sometimes even make compromises to have functionality and low ping over astonishing effects and details.

Player networking

The Unity component Network ID was added to the player with the Local Authority option checked to indicate that our player is both a Network Object and a local player; otherwise, the server would not be able to update the player's location or actions to other clients when the player moves.

To manage all the player actions in a networked game, the Player Network script was created; in this way, we can keep the number of scripts with networking authority to a minimum. We want to keep the game lightweight and not overload the server with unnecessary information, otherwise there is a great chance that the clients will lose the connection to the server and, as a result, the server will kick every client back to the lobby scene. This script is the one that controls and monitors the actions of all the other scripts; for example, it receives the input of the user and changes the state of the soldier accordingly. This script handles most of the player's networked actions. When a non-networking script wants to send a command to the game server, it will go through this script to achieve synchronization. A simple example is when the player fires their weapon. In reality, the shooter script of the player contacts this script to inform it that the player fired their weapon and that it is now time to send that command to the server, otherwise only the player shooting would see their weapon firing a bullet.

This brings us to a point where we have to decide which actions of the player need to be synchronized and which do not. Every action that does not require synchronization, such as the crosshair position, can be in a simple monoBehaviour script, while all the others that need to be synchronized will be in a networked script or use the functionality of one.

```
1.  //the main networking script of the player responsible to synchronize across the n
    etwork most of the player's actions
2.  [RequireComponent(typeof(Player))]
3.  [RequireComponent(typeof(AudioSource))]
4.  public class PlayerNetWork : NetworkBehaviour
5.  {
6.      [SyncVar] public string playerName;
7.      [SyncVar] public int playerTeamNumber;
8.      [SyncVar] public Color playerColor = Color.red;
9.      private AudioSource playerAudioSource;
10.     [SerializeField] AudioClip[] playerClips;
11.     public Texture[] teamTextures;
12.     Player player;
13.     PlayerMove playerMove;
14.     PlayerAnimation playerAnimation;
15.     WeaponController weaponController;
16.     WeaponRealoader weaponReloader;
17.     ScopeContoller scopeController;
18.     AttachmentController weaponAttachmentController;
19.     public PlayerNetworkUI ui;
20.     [SerializeField] GameObject minimapQuadImage;
```

```
21.     [SerializeField]Transform customiseMenuCameraLookTarget;
22.     SkinnedMeshRenderer renderer;
23.     [SerializeField] GameObject playerUiPrefab;
24.     private GameObject playerUiInstance;
25.     private PlayerName playerNameCanvas;
26.     public bool CanEnterCar;
27.     public bool IsDriving = false;
28.     public PlayerHealth playerHealth;
29.     NetworkState state;
30.     NetworkState lastSentState;
31.     //SERVER ONLY
32.     NetworkState lastSentRpcState;
33.     NetworkState lastReceivedState;
34.     List<NetworkState> predictedStates;
35.
36.     //a subclass used to transfer the player's input to a network state
37.     [System.Serializable]
38.     public partial class NetworkState : InputController.InputState
39.     {
40.         public float AimTargetX;
41.         public float AimTargetY;
42.         public float AimTargetZ;
43.         public float TimeStamp;
44.     }
45.     //a function used to connect the in game player with the user account and retr
    ieve the username
46.     void SetPlayerAccount()
47.     {
48.         if (UserAccountManager.IsLoggedIn)
49.         {
50.             Cmd_SetPlayerNameToServer(UserAccountManager.instance.SetPlayerName());

51.             Cmd_SetPlayerTeam(playerTeamNumber);
52.         }
53.     }
54.     //a function used to return all player back to the lobby when the round ends
55.     public void ReturnToLobby()
56.     {
57.         FindObjectOfType<NetworkLobbyManager>().ServerReturnToLobby();
58.         Cursor.lockState = CursorLockMode.None;
59.         Cursor.visible = true;
60.     }
61.     // Use this for initialization
62.     void Start()
63.     {
64.         //assigning all the neccessary references to the network player
65.         player = GetComponent<Player>();
66.         playerMove = GetComponent<PlayerMove>();
67.         playerAnimation = GetComponent<PlayerAnimation>();
68.         predictedStates = new List<NetworkState>();
69.         weaponController = player.GetComponentInChildren<WeaponController>();
70.         scopeController = weaponController.ActiveWeapon.GetComponentInChildren<Sco
    peContoller>();
71.         weaponAttachmentController = weaponController.ActiveWeapon.GetComponentInC
    hildren<AttachmentController>();
72.         weaponReloader = player.GetComponentInChildren<WeaponRealoader>();
73.         playerHealth = player.GetComponent<PlayerHealth>();
74.         playerNameCanvas = GetComponentInChildren<PlayerName>();
75.         playerAudioSource = GetComponent<AudioSource>();
```

```
76.         renderer = player.GetComponentInChildren<SkinnedMeshRenderer>();
77.         state = new NetworkState();
78.
79.         if (isLocalPlayer)
80.         {
81.             //declaring player instance as the local player
82.             player.SetAsLocalPlayer();
83.             //create PlayerUi
84.             playerUiInstance = Instantiate(playerUiPrefab);
85.             playerUiInstance.name = playerUiPrefab.name;
86.             minimapQuadImage.GetComponent<Renderer>().material.color = Color.cyan;

87.             //configure playerUI
88.              ui = playerUiInstance.GetComponent<PlayerNetworkUI>();
89.             if (ui != null)
90.             {
91.                 ui.SetPlayer(GetComponent<Player>());
92.             }
93.             playerNameCanvas.SetPlayerName(player);
94.             StartCoroutine(SyncPlayerAtGameStart());
95.         }
96.         UpdateState();
97.     }
98.     private void OnDisable()
99.     {
100.         Destroy(playerUiInstance);
101.     }
102.     //used to collect the players input and decide his next actions like moving
    and firing his weapon
103.     private NetworkState CollectInput()
104.     {
105.         var state = new NetworkState
106.         {
107.             Horizontal = GameManager.Instance.InputController.Horizontal,
108.             Vertical = GameManager.Instance.InputController.Vertical,
109.             IsWalking = GameManager.Instance.InputController.IsWalking,
110.             IsSprinting = GameManager.Instance.InputController.IsSprinting,
111.             IsCrouched = GameManager.Instance.InputController.IsCrouched,
112.             IsJumping = GameManager.Instance.InputController.IsJumping,
113.             Reload = GameManager.Instance.InputController.Reload,
114.             Fire1 = GameManager.Instance.InputController.Fire1,
115.             Fire2 = GameManager.Instance.InputController.Fire2,
116.             IsAiming = GameManager.Instance.InputController.IsAiming,
117.             AimAngle = GameManager.Instance.InputController.AimAngle,
118.             IsProne = GameManager.Instance.InputController.IsProne,
119.             TimeStamp = Time.time
120.         };
121.         if (state.Fire1)
122.         {
123.             Vector3 shootingSolution = player.WeaponController.GetImpactPoint();

124.             state.AimTargetX = shootingSolution.x;
125.             state.AimTargetY = shootingSolution.y;
126.             state.AimTargetZ = shootingSolution.z;
127.         }
128.         return state;
129.     }
130.     //used to sync the players details after 1,5 seconds
131.     IEnumerator SyncPlayerAtGameStart()
```

```
132.     {
133.         yield return new WaitForSeconds(1.5f);
134.         SetPlayerAccount();
135.     }
136.     // Update is called once per frame
137.     void Update()
138.     {
139.         //if a player reaches 10 kills the round ends and the players return to
    the lobby
140.         if (player.PlayerHealth.KillCount > 9)
141.         {
142.            //End of round,players return to lobby after 8 seconds
143.             Invoke("ReturnToLobby", 8);
144.         }
145.         if (isLocalPlayer)
146.         {
147.             scopeController = weaponController.ActiveWeapon.GetComponentInChildr
    en<ScopeContoller>();
148.             weaponAttachmentController = weaponController.ActiveWeapon.GetCompon
    entInChildren<AttachmentController>();
149.             //command to swap player's weapons
150.             if (Input.GetKeyDown(KeyCode.Alpha1))
151.             {
152.                 Cmd_EquipWeapon(0);
153.             }
154.             if (Input.GetKeyDown(KeyCode.Alpha2))
155.             {
156.                 Cmd_EquipWeapon(1);
157.             }
158.             //collect moving info
159.             state = CollectInput();
160.             //move player
161.             playerMove.SetInputController(state);
162.             playerMove.Move(state.Horizontal, state.Vertical);
163.             //we set the player's camera target transform as player's transform

164.             CustomCamera02.aimTarget = this.transform;
165.             CustomCamera02.localPlayerTransform = this.transform;
166.             //we assign the minimap's target transform as the player's transform

167.             Minimap.localPlayerTransform = this.transform;
168.         }
169.         if (lastReceivedState == null)
170.         {
171.             return;
172.         }
173.         UpdateState();
174.     }
175.     //changes player color through the  texture image of the players material,an
    d the player's icon color
176.     void ChangePlayerTexture()
177.     {
178.         if ( player.Team==1)
179.         {
180.             renderer.material.mainTexture = teamTextures[0];
181.             minimapQuadImage.GetComponent<Renderer>().material.color = Color.whi
    te;
182.         }
183.         else if (  player.Team == 2)
```

```
184.         {
185.             renderer.material.mainTexture = teamTextures[1];
186.             minimapQuadImage.GetComponent<Renderer>().material.color = Color.gre
    en;
187.         }
188.         if (isLocalPlayer)
189.         {
190.             minimapQuadImage.GetComponent<Renderer>().material.color = playerCol
    or;
191.         }
192.     }
193.     //used to update the network state of the player ,like the animation state
194.     void UpdateState()
195.     {
196.         if (lastReceivedState == null)
197.         {
198.             return;
199.         }
200.         if (isLocalPlayer && !isServer)
201.         {
202.             //remove all states if there are any
203.             predictedStates.RemoveAll(x => x.TimeStamp < lastReceivedState.TimeS
    tamp);
204.
205.
206.         }
207.         if (!isLocalPlayer)
208.         {
209.             //updateing animation states of player
210.             playerAnimation.Horizontal = lastReceivedState.Horizontal;
211.             playerAnimation.Vertical = lastReceivedState.Vertical;
212.             playerAnimation.IsWalking = lastReceivedState.IsWalking;
213.             playerAnimation.IsSprinting = lastReceivedState.IsSprinting;
214.             playerAnimation.IsCrouched = lastReceivedState.IsCrouched;
215.             playerAnimation.IsJumping = lastReceivedState.IsJumping;
216.             playerAnimation.IsAiming = lastReceivedState.IsAiming;
217.             playerAnimation.AimAngle = lastReceivedState.AimAngle;
218.             playerAnimation.IsProne = lastReceivedState.IsProne;
219.             Vector3 shootingSolution = new Vector3(lastReceivedState.AimTargetX,
     lastReceivedState.AimTargetY, lastReceivedState.AimTargetZ);
220.             playerMove.SetInputController(lastReceivedState);
221.             player.SetInputState(lastReceivedState);
222.             if (shootingSolution != Vector3.zero)
223.             {
224.                 player.WeaponController.ActiveWeapon.SetAimpoint(shootingSolutio
    n);
225.             }
226.         }
227.     }
228.     //update the player's state based on the previous one
229.     void FixedUpdate()
230.     {
231.         if (isLocalPlayer)
232.         {
233.             if (isInputStateDirty(state, lastSentState))
234.             {
235.                 lastSentState = state;
236.                 Cmd_HandleInput(SerializeState(lastSentState));
237.                 predictedStates.Add(state);
```

```
238.
239.                 }
240.             }
241.             if (isServer && lastReceivedState != null)
242.             {
243.                 NetworkState stateSolution = new NetworkState
244.                 {
245.                     Horizontal = lastReceivedState.Horizontal,
246.                     Vertical = lastReceivedState.Vertical,
247.                     IsWalking = lastReceivedState.IsWalking,
248.                     IsSprinting = lastReceivedState.IsSprinting,
249.                     IsCrouched = lastReceivedState.IsCrouched,
250.                     IsJumping = lastReceivedState.IsJumping,
251.                     Fire1 = lastReceivedState.Fire1,
252.                     Fire2 = lastReceivedState.Fire2,
253.                     Reload = lastReceivedState.Reload,
254.                     IsAiming = lastReceivedState.IsAiming,
255.                     AimAngle = lastReceivedState.AimAngle,
256.                     IsProne = lastReceivedState.IsProne,
257.                     TimeStamp = lastReceivedState.TimeStamp
258.                 };
259.                 if (isInputStateDirty(stateSolution, lastSentRpcState))
260.                 {
261.                     lastSentRpcState = stateSolution;
262.                     Rpc_HandleStateSolution(SerializeState(lastSentRpcState));
263.                 }
264.             }
265.         }
266.         //Commands (Cmd) and Remote precedure calls (Rpc)
267.         //used to set the player's team
268.         [Command]
269.         public void Cmd_SetPlayerTeam(int team)
270.         {
271.             Rpc_SetPlayerTeam(team);
272.         }
273.         [ClientRpc]
274.         void Rpc_SetPlayerTeam(int Team)
275.         {
276.             player.Team = Team;
277.             playerTeamNumber = Team;
278.             ChangePlayerTexture();
279.         }
280.         //used to sync audio across all players
281.         [Command]
282.         public void Cmd_PlayAudioClip(int clipNumber)
283.         {
284.             Rpc_PlayAudioClip(clipNumber);
285.         }
286.         [ClientRpc]
287.         public void Rpc_PlayAudioClip(int clipNumber)
288.         {
289.             playerAudioSource.PlayOneShot(playerClips[clipNumber]);
290.         }
291.         //used to synchronise the player's names
292.         [Command]
293.         void Cmd_SetPlayerNameToServer(String Name)
294.         {
295.             Rpc_ChangePlayerName (Name);
296.         }
```

```
297.     [ClientRpc]
298.     void Rpc_ChangePlayerName(string Name)
299.     {
300.         player.name = Name;
301.     }
302.     //weapon commands
303.     //used to synchronise the weapon change of the player across the network
304.     [Command]
305.     public void Cmd_EquipWeapon(int weaponIndex)
306.     {
307.         Rpc_EquipWeapon (weaponIndex);
308.     }
309.     [ClientRpc]
310.     void Rpc_EquipWeapon(int weaponIndex)
311.     {
312.         weaponController.EquipWeapon (weaponIndex);
313.         weaponReloader= weaponController.ActiveWeapon.Reloader;
314.     }
315.     //used to synchronise the weapon's scope change of the player across the net
    work
316.     [Command]
317.     public void Cmd_EquipScope(int scopeIndex)
318.     {
319.         Rpc_EquipScope(scopeIndex);
320.     }
321.     [ClientRpc]
322.     void Rpc_EquipScope(int scopeIndex)
323.     {
324.             scopeController = weaponController.ActiveWeapon.GetComponentInChildr
    en<ScopeContoller>();
325.             scopeController.EquipScope(scopeIndex);
326.     }
327.     //weapon attachments
328.     //used to synchronise the weapon's attachment change of the player across th
    e network
329.     [Command]
330.     public void Cmd_EquipAttachment(int attachmentIndex)
331.     {
332.         Rpc_EquipAttachment(attachmentIndex);
333.     }
334.     [ClientRpc]
335.     void Rpc_EquipAttachment(int attachmentIndex)
336.     {
337.             weaponAttachmentController = weaponController.ActiveWeapon.GetCompon
    entInChildren<AttachmentController>();
338.             weaponAttachmentController.EquipAttachment(attachmentIndex);
339.      }
340.     //fire
341.     //used to synchronise the firing a the player's weapon
342.     [Command]
343.     public void Cmd_ActiveWeaponFire()
344.     {
345.         Rpc_ActiveWeaponFire();
346.     }
347.     [ClientRpc]
348.     void Rpc_ActiveWeaponFire()
349.     {
350.         player.WeaponController.ActiveWeapon.Fire();
351.     }
```

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

```
352.     //used to synchronise the weapon's reaload across the network
353.     [Command]
354.     public void Cmd_ReloadWeapon()
355.     {
356.         Rpc_ReloadWeapon();
357.     }
358.     [ClientRpc]
359.     void Rpc_ReloadWeapon()
360.     {
361.         weaponReloader.Reload();
362.     }
363.     //used to synchronise the player's input
364.     [Command]
365.     void Cmd_HandleInput(byte[] data )
366.     {
367.         lastReceivedState = DeserializeState (data);
368.     }
369.     //remote client scan
370.     [ClientRpc]
371.     void Rpc_HandleStateSolution(byte[] data)
372.     {
373.         lastReceivedState = DeserializeState (data);
374.     }
375.
376.     bool isInputStateDirty(NetworkState a , NetworkState b)
377.     {
378.         if (b == null)
379.         {
380.             return true;
381.         }
382.         return  a.Horizontal != b.Horizontal ||
383.                 a.Vertical != b.Vertical ||
384.                 a.IsWalking != b.IsWalking ||
385.                 a.IsSprinting != b.IsSprinting ||
386.                 a.IsCrouched != b.IsCrouched ||
387.                 a.IsJumping != b.IsJumping ||
388.                 a.Fire1 != b.Fire1 ||
389.                 a.Fire2 != b.Fire2 ||
390.                 a.Reload != b.Reload ||
391.                 a.IsAiming != b.IsAiming ||
392.                 a.AimAngle!=b.AimAngle||
393.                 a.IsProne!=b.IsProne
394.                ;
395.     }
396.     //used to seriliaze and deserialize data in binary format
397.     private BinaryFormatter bf = new BinaryFormatter ();
398.     private byte[] SerializeState(NetworkState state)
399.     {
400.         using (MemoryStream stream = new MemoryStream ())
401.         {
402.             bf.Serialize (stream, state);
403.             return stream.ToArray ();
404.         }
405.     }
406.     private NetworkState DeserializeState(byte[] bytes)
407.     {
408.         using (MemoryStream stream = new MemoryStream (bytes))
409.         {
410.             return (NetworkState)bf.Deserialize (stream);
```

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

```
411.         }
412.     }
413. }
```

Player network script

Input controller

Input controller is a script created to handle the user's input and translate it to the in-game player's actions. The script maps all the keyboard and mouse keys used in the game and, based on the key pressed, changes the player's state or activates a function of the game; for example, the "W" and "S" keys are assigned to change the vertical moving state of the player and the Tab key is assigned to activate the scoreboard of the game. Having a script to control the in-game actions of the player is much more efficient than having various pieces of code inside other scripts. In this way, the code has greater reusability, should we decide to build the game for other platforms, too, such as PlayStation or Android, in which case we only need to make modifications to the input controller script to adjust the controls.

```
1.  // this scripct collects the player's input and translates it to in game actions o
    f the soldier
2.  // it maps evry key of the keyboard and mouse to an avction including menu toggles

3.  public class InputController : MonoBehaviour
4.  {
5.      //NETWORK input state subclass
6.      //this is used to update the animation state to all clients
7.      [System.Serializable]
8.      public class InputState
9.      {
10.         public float Horizontal;
11.         public float Vertical;
12.         public bool IsWalking;
13.         public bool IsSprinting;
14.         public bool IsCrouched;
15.         public bool IsJumping;
16.         public bool Fire1;
17.         public bool Fire2;
18.         public bool Reload;
19.         public bool IsAiming;
20.         public float AimAngle;
21.         public bool IsProne;
22.     }
23.     public float Horizontal { get { return State.Horizontal; } }
24.     public float Vertical { get { return State.Vertical; } }
25.     public bool IsWalking { get { return State.IsWalking; } }
26.     public bool IsSprinting { get { return State.IsSprinting; } }
27.     public bool IsCrouched { get { return State.IsCrouched; } }
28.     public bool IsJumping{ get { return State.IsJumping; } }
29.     public bool Fire1 { get { return State.Fire1; } }
30.     public bool Fire2 { get { return State.Fire2; } }
31.     public bool Reload { get { return State.Reload; } }
32.     public bool IsAiming{ get { return State.IsAiming; } }
33.     public float AimAngle { get { return State.AimAngle; } }
34.     public bool IsProne{ get { return State.IsProne; } }
35.     public Vector2 MouseInput;
36.
```

```
37.     public bool MouseWheelUp;
38.     public bool MouseWheelDown;
39.     public bool ScopeSwitchToggled;
40.     public bool WeaponAttachmentSwitchToggled;
41.     public bool Escape;
42.
43.     //UI Menus
44.     public bool ScoreboardToggled;
45.     public bool CustomiseMenuToggled;
46.     public bool InteractWithObject;
47.     public bool TeleportToggled;
48.     float aimInput;
49.     public InputState State;
50.     private void Awake()
51.     {
52.         DontDestroyOnLoad(this.gameObject);
53.     }
54.     void Start()
55.     {
56.         State = new InputState ();
57.     }
58.     // in the update we map every key to an in game action or an animation state

59.     private void Update()
60.     {
61.         //movement and mouse inputs
62.         State.Horizontal = Input.GetAxis("Horizontal");
63.         State.Vertical = Input.GetAxis("Vertical");
64.         //inputs to toggle between movement modes
65.         State.IsWalking = Input.GetKey (KeyCode.X);
66.         State.IsSprinting = Input.GetKey (KeyCode.LeftShift);
67.         if (Input.GetKeyDown(KeyCode.C))
68.         {
69.             State.IsCrouched = !State.IsCrouched;
70.         }
71.         State.IsJumping = Input.GetKeyDown (KeyCode.Space);
72.         //weapon inputs
73.         State.Fire1 = Input.GetButton ("Fire1");
74.         State.Fire2 = Input.GetButton ("Fire2");
75.         State.Reload = Input.GetKeyDown (KeyCode.R);
76.         State.IsAiming = Input.GetButton ("Fire2");
77.         SetRotation (Input.GetAxis("Mouse Y")*4);
78.         State.AimAngle = GetAngle ();
79.         if(Input.GetKeyDown (KeyCode.Z))
80.         {
81.             State.IsProne = !State.IsProne;
82.         }
83.         MouseInput = new Vector2(Input.GetAxisRaw("Mouse X"), Input.GetAxisRaw("Mo
    use Y"));
84.         Escape = Input.GetKeyDown (KeyCode.H);
85.         MouseWheelUp = Input.GetAxis ("Mouse ScrollWheel")>0;
86.         MouseWheelDown = Input.GetAxis ("Mouse ScrollWheel")<0;
87.         ScopeSwitchToggled = Input.GetKeyDown (KeyCode.Alpha3);
88.         WeaponAttachmentSwitchToggled = Input.GetKeyDown(KeyCode.Alpha4);
89.         ScoreboardToggled = Input.GetKey (KeyCode.Tab);
90.         if (Input.GetKeyDown(KeyCode.O))
91.         {
92.             CustomiseMenuToggled = !CustomiseMenuToggled;
93.         }
```

```
94.         InteractWithObject = Input.GetKey(KeyCode.E);
95.         TeleportToggled = Input.GetKeyDown (KeyCode.T);
96.     }
97.     //the section below is resposible to rotate the upper body of the soldier to e
    nable vertical aiming
98.     float minAngle=-20.0f;
99.     float maxAngle=20.0f;
100.     public void SetRotation(float amount)
101.     {
102.         float clampedAngle = GetClampedAngle (amount);
103.         transform.eulerAngles = new Vector3 (clampedAngle, transform.eulerAngles.
    y ,transform.eulerAngles.z);
104.     }
105.     public float GetAngle()
106.     {
107.         return CheckAngle (transform.eulerAngles.x);
108.     }
109.     public float CheckAngle(float value)
110.     {
111.         float angle = value - 180;
112.         if (angle > 0)
113.         {
114.             return angle - 180;
115.         }
116.         return angle + 180;
117.     }
118.     private float GetClampedAngle(float amount)
119.     {
120.         float newAngle = CheckAngle (transform.eulerAngles.x - amount);
121.         float clampedAngle = Mathf.Clamp (newAngle,minAngle ,maxAngle);
122.         return clampedAngle;
123.     }
124. }
```

Input controller script

Main camera

In the game scene, a scripted camera is located, the camera being assigned to each player the moment they enter the game; to be specific, the player will assign the camera through their main networked script. This method is necessary to avoid confusion when multiple players join the scene because every new client will try to use the previous client's camera. We could simply attach a camera inside every player's game object but then the camera would not have the same fluid movement it has and would be more static.

This camera is the main camera of the game scene, and will follow the player. When the player is not aiming, the camera will stay behind the player in a fixed position while he moves or rotates; in other words, this will be the third person mode of the game. If the player is aiming, the camera will be transferred behind the scope of the current weapon and will also follow the weapon's position and rotation with a slight delay to make the movement more realistic. This mode is the first person mode. In this way, we get a third person game that switches to first person when the player needs to aim and be more accurate. In addition, the camera has a collision detector to avoid going inside other objects; namely, if the player is standing close to a wall the camera will detect the wall and move forward as many units as are needed so as not to render the wall from inside, which would cause graphical issues. Furthermore, in this camera we will display every UI element of the game, such as the weapon panel, the scoreboard or the crosshair.

First person view

```
1.  public class CustomCamera02 : MonoBehaviour
2.  {
3.      //this is the scipt controlling the playe'rs main camera
4.      public static Transform aimTarget;
5.      public static Transform localPlayerTransform;
6.      //a subclass used to set the main camera's values to positions it behind the p
    layer model
7.      [System.Serializable]
8.      public class CameraRig
9.      {
10.         public Vector3 CameraOffset;
11.         public float Damping;
12.         public float CrouchHeight;
13.     }
14.     public float cameraSpeed = 150.7f;
15.     /// <summary>
16.     /// default camera values 0,2,-5 , damiping 6
17.     /// </summary>
18.     [SerializeField]CameraRig defaultCamera;
19.     [SerializeField]CameraRig aimCamera;
20.     [SerializeField]CameraRig fpsCamera;
21.     Transform scopeViewTransform;
22.
23.     //a functions called form the player network script to attach each camera to t
    he proper player
24.     void SetTarget (Transform t)
25.     {
26.         aimTarget = t;
27.         localPlayerTransform = t;
28.     }
29.     //a function called when a player joins the game
30.     void HandleOnLocalPlayerJoined()
31.     {
32.         if (aimTarget.transform.Find ("AimingPivot")) {
33.             //Debug.Log ("Aiming pivot found and assigned");
34.             aimTarget = aimTarget.transform.Find ("AimingPivot");
35.         }
```

```
36.         else
37.         {
38.             Debug.Log ("No Aiming pivot found ");
39.             aimTarget = localPlayerTransform;
40.         }
41.     }
42.     // Update is called once per frame
43.     void LateUpdate ()
44.     {
45.         if (!isActiveAndEnabled)
46.         {
47.             Debug.Log ("Camera disabled");
48.         }
49.         if (aimTarget && localPlayerTransform)
50.         {
51.             HandleOnLocalPlayerJoined ();
52.             CameraRig cameraRig = defaultCamera;
53.             //when player is aiming
54.             if (GameManager.Instance.InputController.IsAiming && localPlayerTransf
    orm.GetComponentInChildren<ScopeView> () != null && localPlayerTransform.GetCompon
    entInChildren<ScopeView> ().isActiveAndEnabled )
55.             {
56.                 if (!GameManager.Instance.InputController.IsSprinting && !GameMana
    ger.Instance.InputController.IsProne &&  GameManager.Instance.InputController.Vert
    ical < 0.3)
57.                 {
58.                     scopeViewTransform = localPlayerTransform.GetComponentInChildr
    en<ScopeView>().transform;
59.                     Vector3 smoothScopePos = Vector3.Lerp(transform.position, scop
    eViewTransform.position, cameraSpeed);
60.                     transform.position = smoothScopePos;
61.                     transform.rotation = scopeViewTransform.rotation;
62.                 }
63.                 else
64.                 {
65.                     float targetHeight = cameraRig.CameraOffset.y + (GameManager.I
    nstance.LocalPlayer.IsLocalPlayer && GameManager.Instance.InputController.IsCrouch
    ed ? cameraRig.CrouchHeight : 0);
66.                     Vector3 targetPosition = aimTarget.position + localPlayerTrans
    form.forward * cameraRig.CameraOffset.z + localPlayerTransform.up * targetHeight +
     localPlayerTransform.right * cameraRig.CameraOffset.x;
67.                     Vector3 collisionDestination = aimTarget.position + localPlaye
    rTransform.transform.up * targetHeight - localPlayerTransform.transform.forward *
    0.5f;
68.                     HandleCameraCollision(collisionDestination, ref targetPosition)
    ;
69.                     transform.position = Vector3.Lerp(transform.position, targetPo
    sition, cameraRig.Damping * Time.deltaTime);
70.                     transform.rotation = Quaternion.Lerp(transform.rotation, aimTa
    rget.rotation, cameraRig.Damping * Time.deltaTime);
71.                 }
72.             }
73.             else
74.             {
75.                 //third person view
76.                 float targetHeight = cameraRig.CameraOffset.y + (GameManager.Insta
    nce.LocalPlayer.IsLocalPlayer && GameManager.Instance.InputController.IsCrouched?
    cameraRig.CrouchHeight : 0);
```

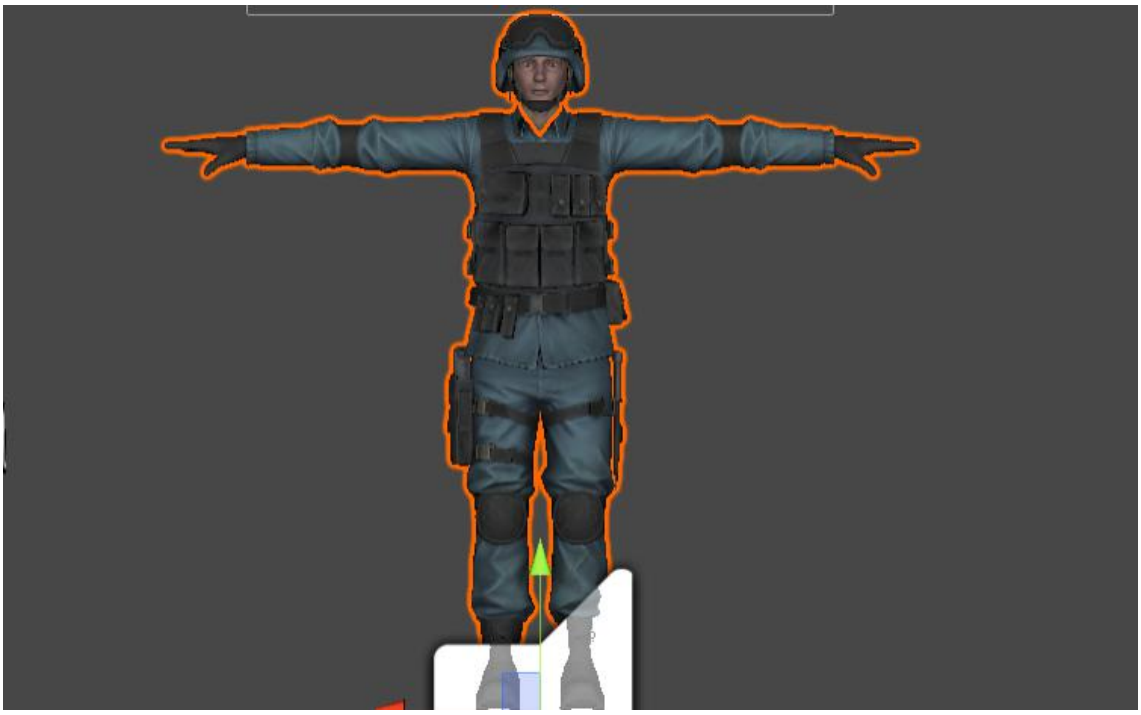
```
77.                 Vector3 targetPosition = aimTarget.position + localPlayerTransform.
    forward *cameraRig.CameraOffset.z+localPlayerTransform.up *targetHeight+ localPlay
    erTransform.right * cameraRig.CameraOffset.x;
78.                 Vector3 collisionDestination = aimTarget.position+localPlayerTrans
    form.transform.up*targetHeight-localPlayerTransform.transform.forward*0.5f;
79.                 HandleCameraCollision(collisionDestination,ref targetPosition);
80.                 transform.position = Vector3.Lerp (transform.position, targetPosit
    ion, cameraRig.Damping * Time.deltaTime);
81.                 transform.rotation = Quaternion.Lerp (transform.rotation, aimTarge
    t.rotation, cameraRig.Damping* Time.deltaTime);
82.             }
83.         }
84.     }
85.     //this function is used to avoid the camera collision with other objects
86.     //instead of going through the various meshes in the scene the camera moves fo
    rward using this function
87.     private void HandleCameraCollision(Vector3 toTarget,  ref Vector3 fromTarget)

88.     {
89.         RaycastHit hit;
90.         //if there is a collision move the camera to the collision point
91.         if (Physics.Linecast (toTarget,fromTarget, out hit))
92.         {
93.             Vector3 hitPoint = new Vector3 (hit.point.x + hit.normal.x * 0.2f, hit.
    point.y,hit.point.z+hit.normal.z*.2f);
94.             fromTarget = new Vector3 (hitPoint.x, fromTarget.y, hitPoint.z);
95.         }
96.     }
97. }
```

Player camera script

Player model and animations

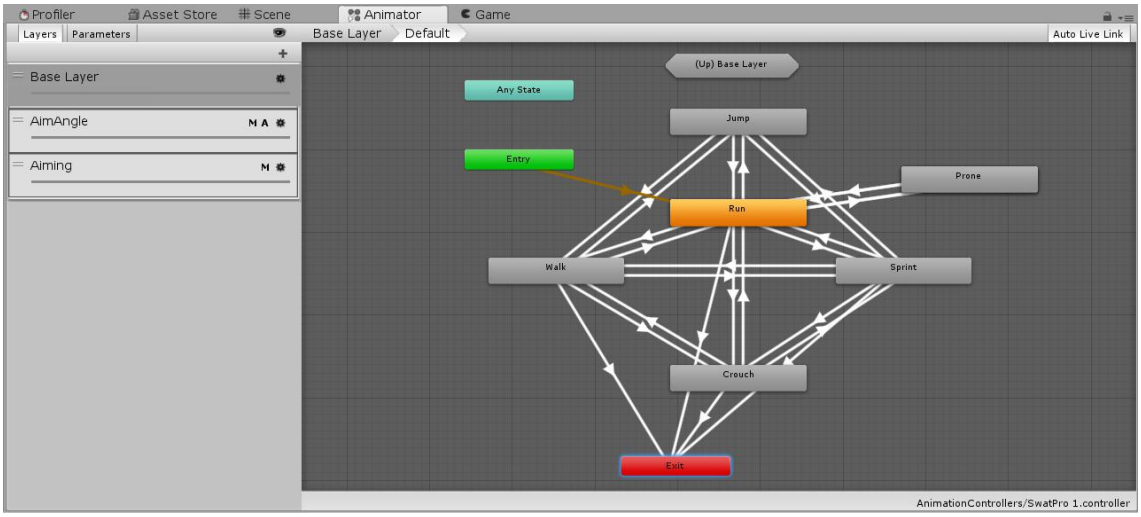
The player uses a capsule collider as its main collider and a character controller for movement. The model of the player and its animations were downloaded from Mixamo, a site from adobe with free character models and animations for indie developers. The selected character is the SWAT soldier and most of the animations come from the pro rifle pack. The model of the player is under the ‘Mesh’ Empty Object located inside the player’s Game Object.



Soldier model

Animator controller

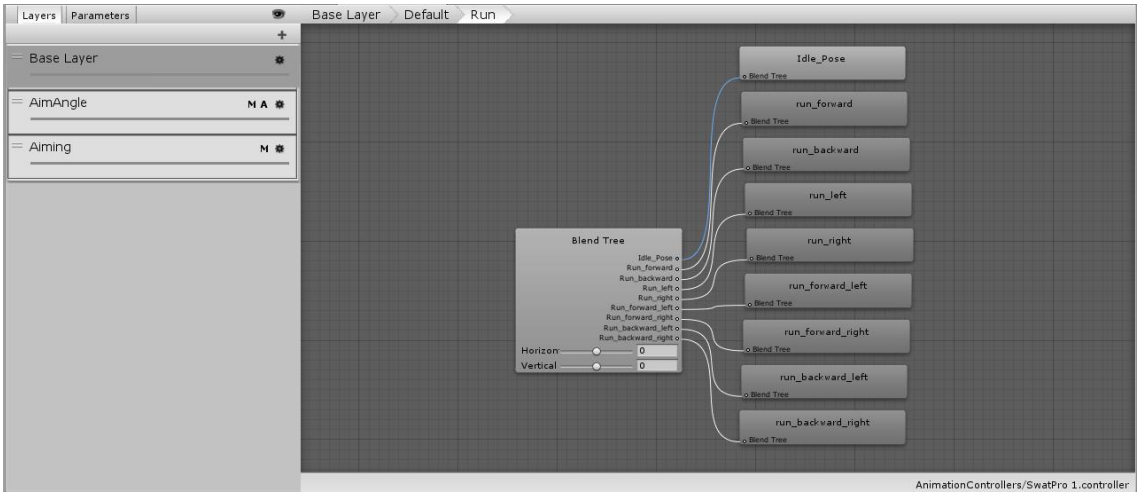
The animations of the player are controlled from the player's animator controller (name SwatPro) which consists of player states such as walking and running, which instead of containing a single animation clip are using blend trees. States in the animator controller resemble the states in a deterministic automaton, and the current state of the animator will change based on the values of the animator's parameters.



Animator Controller – Base layer

Blend trees are used to blend multiple animations together. In this project, we blend animations based on the player's input; for example, if the user presses the "W" and "A" keys to move forward

and left, the animator will blend/mix the forward and left animations and create a left diagonal animation.

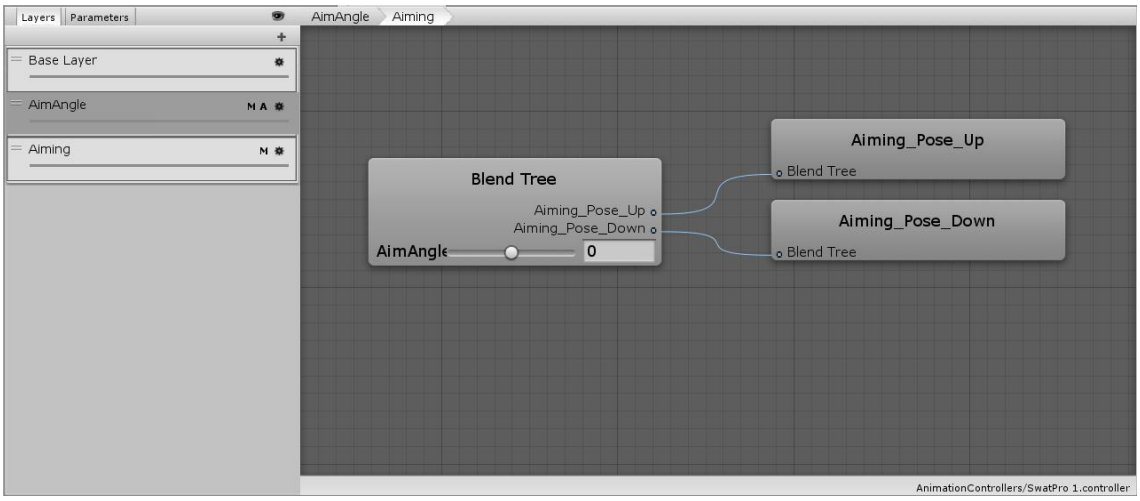


Run state – Blend tree example

Parameters are variables, the same type of variables used in programming integer, float, boolean, etc, used in the animator controller to change the state of the player's animations or any other object that has animations, such as npcs. The animator controller has the following parameters:

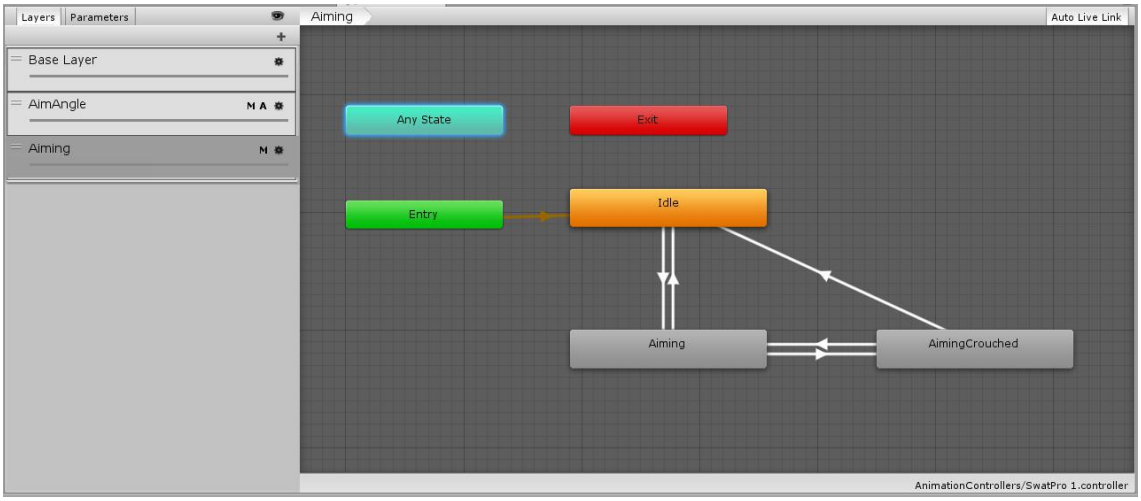
Float: Horizontal, Vertical, and AimAngle, responsible for the movement and aiming animations. The horizontal and vertical parameters take as input the float of the pressed movement keys (Horizontal: A, D Vertical: W, S), and based on these two values the animator blends the moving animations to make the soldiers feet move/walk in the chosen direction. The animation blending of the horizontal and vertical values affects every moving animation of the soldier – walk, run, sprint, etc. In addition to typical forward and backward movement, there are also animations for the diagonal directions, in this way creating a smoother motion effect in all eight directions.

The AimAngle parameter is the float that takes as input the value of the vertical movement of the user's mouse and affects the aiming animation of the soldier by rotating a bone in his spine (extending to the whole upper body above that spinal bone) that enables him to aim up or down based on the angle calculated from the mouse value. For example, if the user moves his mouse forwards, the parent bone on the spine of the soldier will be rotated in such a way to make his upper body face/aim downwards. The proper mechanism for aiming in shooter games is using inverse kinematics but for this project the rotation of the spine will suffice.



Animator Controller – Aim Angle layer

Boolean: IsWalking, IsSprinting, IsCrouched, IsAiming, IsJumping, IsProne are responsible for changing the animation states based on the player's input. The names of these parameters explain the animation of the player. For example, if the player is moving and the user presses the left shift key to sprint, the IsSprinting parameter will become true and the animation state will change from default (Run) to the Sprint state, and the running animation of the soldier will switch to the sprint animation. The soldier in this game has the ability to run, sprint, walk, walk crouched, lay on the ground (prone) and move slowly or jump.



Animator Controller – Aiming Layer

We can easily assume that the animator has the following states controlled by the values of the previously mentioned parameters: Run (Default State), Sprint, Walk, Crouch, Jump, Prone.

Each of these states is, in fact, a blend tree, blending all the animations of each state depending on the player's input; for example, if the player is crouched but moving, his feet will blend the animations depending on the player's direction, providing us with a more realistic movement.

Usually in UNET projects, most developers use a network animator component to synchronize the animations of each player effortlessly across the network. In this project, however, every change in animations and synchronization is made through code. In the player network script, we declare player states such as IsRunning and use these values, which change based on the

player's input, to change the animation state with the same names. Thus, we have the same result, with all animations transferred across the network over which we always have control.

```
1.  // this script is used to controll the soldier's animation states
2.  public class PlayerAnimation : MonoBehaviour
3.  {
4.      Animator animator;
5.      // all animation states available
6.      public float Horizontal;
7.      public float Vertical;
8.      public bool IsWalking;
9.      public bool IsSprinting;
10.     public bool IsCrouched;
11.     public bool IsJumping;
12.     public bool IsAiming;
13.     public float AimAngle;
14.     public bool IsInCover;
15.     public bool IsProne;
16.     public bool IsDriving;
17.
18.     private PlayerAim m_PlayerAim;
19.     private PlayerAim PlayerAim
20.     {
21.         get
22.         {
23.             if (m_PlayerAim == null)
24.             {
25.                 m_PlayerAim = GameManager.Instance.LocalPlayer.playerAim;
26.             }
27.             return m_PlayerAim;
28.         }
29.     }
30.     // singleton of player script
31.     private Player m_Player;
32.     private Player Player
33.     {
34.         get
35.         {
36.             if (m_Player == null)
37.             {
38.                 m_Player = GetComponent<Player> ();
39.             }
40.             return m_Player;
41.         }
42.     }
43.     // collects the player's input and assigns the values to the animator controll
    er parameters
44.     void GetLocalPlayerInput()
45.     {
46.         Horizontal  = Player.InputState.Horizontal ;
47.         Vertical = Player.InputState.Vertical;
48.         IsWalking = Player.InputState.IsWalking;
49.         IsSprinting = Player.InputState.IsSprinting;
50.         IsCrouched = Player.InputState.IsCrouched;
51.         IsJumping = Player.InputState.IsJumping;
52.         IsAiming = Player.InputState.IsAiming;
53.         AimAngle = Player.InputState.AimAngle;
54.         IsProne = Player.InputState.IsProne;
```

```
55.     }
56.     // Use this for initialization
57.     void Awake ()
58.     {
59.         animator = GetComponentInChildren<Animator> ();
60.     }
61.     // Update is called once per frame
62.     void Update ()
63.     {
64.         if (GameManager.Instance.IsPaused)
65.         {
66.             return;
67.         }
68.         if (Player.IsLocalPlayer)
69.         {
70.             GetLocalPlayerInput ();
71.         }
72.         // updates the animation states in the animator based on the values of eac
    h parameter
73.         animator.SetFloat ("Horizontal", Horizontal);
74.         animator.SetFloat ("Vertical", Vertical);
75.         animator.SetBool ("IsWalking",IsWalking);
76.         animator.SetBool ("IsSprinting", IsSprinting);
77.         animator.SetBool ("IsCrouched", IsCrouched);
78.         animator.SetBool ("IsJumping",IsJumping);
79.         animator.SetBool ("IsAiming",IsAiming);
80.         animator.SetFloat ("AimAngle", AimAngle );
81.         animator.SetBool ("IsInCover", IsInCover);
82.         animator.SetBool ("IsProne", IsProne);
83.         animator.SetBool("IsDriving", IsDriving);
84.     }
85. }
```

Player animator script

Movement

So far, there are only movement animations, as the player is not actually moving. To implement movement of the player, the script player script was added. This script is responsible for translating the input of the user to move the player and adjust their speed based on their state.

The player's default state is run and has a specific speed declared; if the state of the player is changed to sprint, for example, as a result, the soldier's speed will increase. The player moves based on the vertical and horizontal input (W, A, D, S keys), and the speed takes its value from the current player state as well as the soldier animation. We notice that so far the player only moves along 2 axes: the Z axis (forward-backward) and the X axis (right-left), while the Y axis (up-down) is determined by the jump input. If the users presses JUMP, we now move the player on the 3rd axis, too, which results in the player jumping in the air. We declared a float to represent gravity in the game and, every time the player is in the air, we use the gravity value to move him down on the Y axis. In conclusion, the players can move around the map, change their states, which will change their moving animations and speed, jump in the air and come back down again after a short time, based on gravity's pull.

Movement – Soldier running example

```
1.  // this script  controlls the player's movement in the scene
2.  public class PlayerMove : MonoBehaviour
3.  {
4.      InputController.InputState playerInput;
5.      public float verticalVelocity;
6.      public float gravity = 30.0f;
7.      float jumpForce=6.50f;
8.      private Vector3 MoveDirection = Vector3.zero;
9.      // singleton reference of the  player script
10.     private Player m_Player;
11.     public Player Player
12.     {
13.         get
14.         {
15.             if (m_Player == null)
16.             {
17.                 m_Player = GetComponent<Player> ();
18.             }
19.             return m_Player;
20.         }
21.     }
22.     // singleton reference of player's character controller
23.     private CharacterController m_moveController;
24.     public CharacterController MoveController
25.     {
26.         get
27.         {
28.             if (m_moveController == null)
29.             {
30.                 m_moveController = gameObject.GetComponent<CharacterController>();

31.             }
32.             return m_moveController;
33.         }
34.     }
35.     void Awake()
36.     {
```

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

```
37.         playerInput = GameManager.Instance.InputController.State;
38.     }
39.     // sets up the player's motion state
40.     public void SetInputController(InputController.InputState state)
41.     {
42.         playerInput = state;
43.     }
44.     // moves the player in the scene
45.     public void Move()
46.     {
47.         if (playerInput == null)
48.         {
49.             playerInput = GameManager.Instance.InputController.State;
50.             if (playerInput == null)
51.             {
52.                 return;
53.             }
54.         }
55.         if (Player.PlayerHealth.IsAlive)
56.         {
57.             Move(playerInput.Horizontal, playerInput.Vertical);
58.         }
59.     }
60.     // moves the player based on the horizontal and vertical input values (WASD ke
    ys)
61.     public void Move(float horizontal , float vertical)
62.     {
63.         if (!Player.PlayerHealth.IsAlive)
64.         {
65.             return;
66.         }
67.         // we adjust the speed based on the motion state
68.         float moveSpeed = Player.Settings.RunSpeed;
69.         if(playerInput.IsWalking)
70.         {
71.         moveSpeed = Player.Settings.WalkSpeed;
72.         }
73.         if(playerInput.IsSprinting)
74.         {
75.         moveSpeed = Player.Settings.SprintSpeed;
76.         }
77.         if(playerInput.IsCrouched)
78.         {
79.         moveSpeed = Player.Settings.CrouchSpeed;
80.         }
81.         if(playerInput.IsProne)
82.         {
83.         moveSpeed = Player.Settings.ProneSpeed;
84.         }
85.         // move player or jump if he is standong on the ground
86.         if (MoveController.isGrounded)
87.         {
88.             MoveDirection = new Vector3(horizontal , 0, vertical );
89.             MoveDirection = transform.TransformDirection(MoveDirection);
90.             MoveDirection *= moveSpeed;
91.             if (playerInput.IsJumping)
92.             {
93.                 MoveDirection.y = jumpForce;
94.             }
```

```
95.         }
96.         else
97.         {
98.             //move player if he is currently above ground after jumping
99.             MoveDirection = new Vector3(horizontal, MoveDirection.y, vertical );
100.             MoveDirection = transform.TransformDirection(MoveDirection);
101.             MoveDirection.x *= moveSpeed;
102.             MoveDirection.z *= moveSpeed;
103.         }
104.         // implement gravity to  make the player land on the ground after a fall
    or a jump
105.         MoveDirection.y -= gravity * Time.deltaTime;
106.         if (Player.PlayerHealth.IsAlive)
107.         {
108.             MoveController.Move(MoveDirection * Time.deltaTime);
109.         }
110.     }
111.     // Update is called once per frame
112.     void FixedUpdate ()
113.     {
114.         if(!Player.IsLocalPlayer)
115.         {
116.             return;
117.         }
118.         if(!Player.PlayerHealth.IsAlive || GameManager.Instance.IsPaused)
119.         {
120.             return;
121.         }
122.         if (!GameManager.Instance.IsNetworkGame)
123.         {
124.             Move ();
125.         }
126.     }
127. }
```

Player movement script

Health

The health script is another networked script used in this game. It is separated from the player network script for the simple reason that the player network script would be too long. It is also recommended to separate scripts based on their use. The health script will keep track of the player health, and whether or not he is alive, while also deducting health points should the player have damage inflicted. This script is also inheriting its functionality form the destructible script, which is the master script for all the scripts that might need health points and can be killed or destroyed like npcs or breakable items, this is a good practice in order to have reusable code for future improvements.

Each player has 100 health points indicating his current health, which are displayed in a UI element directly above his weapon panel. Each time a player is shot, the UI health bar shrinks to the left and the health text changes to indicate their new current health status. If the player's health points reach zero, then the player dies, a death is added to the player's score, a kill is added to the enemy player's score, and the player re-spawns after some time. In addition the player's ragdoll is activated making his whole body drop on the floor. A ragdoll is a Unity3d component which applies physics in every major part of the soldier's body simulating a human with no control over his limbs and no strength to stand still used to represent an elimination of player or a player fainting in most games nowadays. When the player re-spawns, his health points reset back to 100, which also

includes the players UI text and Image length, the ragdoll is deactivated and the animator is re activated to make the soldier appear alive once again. In addition, when we add the kills and deaths to a player score, we do not just add them to their in-game score but also add them to their account's total score using the previously mentioned script User Account Manager, which applies the total score change based on the player's name. It is important to note that once again we use the Cmd and Rpc methods to precisely deduct health points from a player and synchronize this in our network along with the death of a player. It is not rational to assume that the health points of a player are deducted equally in every player's version of the game by default; indeed, in most cases it is not.

```
1.  [RequireComponent(typeof(Collider))]
2.  public class Destructible : NetworkBehaviour
3.  {
4.      private PlayerNetWork playerNetwork;
5.      public const float maxHealth = 100.0f;
6.      [SyncVar] public int KillCount = 0;
7.      [SyncVar] public int DeathCount = 0;
8.      public int TotalKillCount = 0;
9.      public int TotalDeathCount = 0;
10.     NetworkStartPosition[] TeamSpawnPoints;
11.     [SerializeField] Ragdoll playerRagdoll;
12.     float damageTaken;
13.     //syncronized variable across the network indicating if the player is alive or
     not
14.     [SyncVar] public bool m_isDead = false;
15.     public bool isDead
16.     {
17.         get { return m_isDead; }
18.         protected set { m_isDead = value; }
19.     }
20.     //syncronized variable across the network containing the player's current heal
    th points
21.     [SyncVar] public float currentHealth = maxHealth;
22.     private void Start()
23.     {
24.         playerNetwork = this.GetComponent<PlayerNetWork>();
25.         TeamSpawnPoints = FindObjectsOfType<NetworkStartPosition>();
26.     }
27.     //a reference of the player's Ui healthbar in order to change its size when th
    e player's
28.     //health points change
29.     [SerializeField] public RectTransform healthbar;
30.     public bool IsAlive
31.     {
32.         get
33.         {
34.             return currentHealth > 0;
35.         }
36.     }
37.     private void EnableComponents() { }
38.     private void DisableComponents() { }
39.     public virtual void Die()
40.     {
41.         if (isServer)
42.         {
43.             Rpc_Die();
44.         }
45.     }
```

```
46.     //remote procedure call send to clients when the player dies
47.     [ClientRpc]
48.     void Rpc_Die()
49.     {
50.         isDead = true;
51.         playerRagdoll.EnableRagdoll(true);
52.         StartCoroutine(RespawnPlayer());
53.     }
54.     // function that selects the spawn point of the player
55.     void ChooseSpawnPoint(int team)
56.     {
57.             if (TeamSpawnPoints != null && TeamSpawnPoints.Length > 0)
58.             {
59.                 foreach (NetworkStartPosition spawnpoint in TeamSpawnPoints)
60.                 {
61.                     if (spawnpoint.tag == team.ToString())
62.                     {
63.                         this.transform.position = spawnpoint.transform.position;
64.                         this.transform.rotation = spawnpoint.transform.rotation;
65.                     }
66.                 }
67.             }
68.     }
69.     private IEnumerator RespawnPlayer()
70.     {
71.         yield return new WaitForSeconds(5f);
72.         ResetAfterDeath();
73.     }
74.     //command that inflicts damage to the player throught the server
75.     [Command]
76.     public void Cmd_TakeDamage(float amount, GameObject enemyPlayerGO)
77.     {
78.             Rpc_TakeDamage(amount, enemyPlayerGO);
79.     }
80.     //remote procedure call sent to all clients when a player loses heal points
81.     [ClientRpc]
82.     public virtual void Rpc_TakeDamage(float amount ,GameObject enemyPlayerGO)
83.     {
84.         if (isDead)
85.         {
86.             return;
87.         }
88.         if (!IsAlive)
89.         {
90.             return;
91.         }
92.         //source that attacked the player
93.         if (enemyPlayerGO.GetComponentInChildren<Player>()==null)
94.         {
95.             return;
96.         }
97.         //decalre enemy player if attacked by one
98.         Player enemyPlayer = enemyPlayerGO.GetComponentInChildren<Player>();
99.         PlayerNetWork enemyPlayerNetwork = enemyPlayer.GetComponent<PlayerNetWork>
    ();
100.          //check if players are in the same team
101.          PlayerNetWork thisPlayer = this.GetComponent<PlayerNetWork>();
102.          if (thisPlayer.playerTeamNumber == enemyPlayer.GetComponent<PlayerNetWor
    k>().playerTeamNumber)
```

```
103.         {
104.             return;
105.         }
106.         //player is enemy so apply damage
107.         currentHealth -= amount;
108.         if(currentHealth<=0)
109.         {
110.             currentHealth = 0;
111.             Die ();
112.             //increase enemy player kills
113.             ++enemyPlayer.PlayerHealth.KillCount;
114.             enemyPlayer.PlayerHealth.TotalKillCount++;
115.             //increase this player's deaths
116.             ++DeathCount;
117.             TotalDeathCount++;
118.             GameManager.Instance.onPlayerKilled.Invoke( this.transform.name.ToSt
    ring(),enemyPlayer.name.ToString(),enemyPlayerNetwork.playerColor);
119.             SyncScore(this.name);
120.             enemyPlayerNetwork.playerHealth.SyncScore(enemyPlayerNetwork.name);

121.         }
122.     }
123.     //reseting values like health points when player is ready to respawn
124.     public void ResetAfterDeath()
125.     {
126.         currentHealth = maxHealth;
127.         playerRagdoll.EnableRagdoll(false);
128.         playerNetwork = this.GetComponent<PlayerNetWork>();
129.         TeamSpawnPoints = FindObjectsOfType<NetworkStartPosition>();
130.         if (TeamSpawnPoints != null )
131.         {
132.            // Debug.Log("Spawnpoints found");
133.             ChooseSpawnPoint(playerNetwork.playerTeamNumber);
134.         }
135.         isDead = false;
136.     }
137.     //command from server to respawn the player
138.     [Command]
139.     void Cmd_Respawn()
140.     {
141.         Rpc_Respawn ();
142.     }
143.     // remote procedure call showin the clients that the player just respawned
144.     [ClientRpc]
145.     void Rpc_Respawn()
146.     {
147.         this.transform.position = Vector3.zero;
148.         ResetAfterDeath();
149.     }
150.     IEnumerator SyncPlayerScore()
151.     {
152.         while (true)
153.         {
154.             yield return new WaitForSeconds(5f);
155.         }
156.     }
157.     //DATA/SCORE SYNCING REGION
158.     void SyncScore(string name )
159.     {
```

```
160.         //Debug.Log(this.name +" trying to get data");
161.         if (UserAccountManager.IsLoggedIn)
162.         {
163.             UserAccountManager.instance.GetData(name,OnDataReceived);
164.         }
165.     }
166.     //Update statistics in dtatbase when data is received
167.     void OnDataReceived(string data)
168.     {
169.         if (TotalKillCount == 0 && TotalDeathCount == 0)
170.         {
171.             return;
172.         }
173.         int kills = UserAccountDataTranslator.DataToKills(data);
174.         int deaths = UserAccountDataTranslator.DataToDeaths(data);
175.         int newKills =TotalKillCount + kills;
176.         int newDeaths =TotalDeathCount + deaths;
177.         string newData = UserAccountDataTranslator.ValuesToData(newKills, newDea
    ths);
178.         //syncying the new  sent data
179.         TotalKillCount = 0;
180.         TotalDeathCount = 0;
181.         UserAccountManager.instance.SendData(this.name,newData);
182.     }
183. }
```

Destructible script inherited from player health script

Weapon Controller

Every player contains an empty object called Weapons, which is essentially a placeholder for his items; in other words, his inventory. The player weapons are located in this placeholder, with each class having one main weapon and one secondary weapon / gadget. At the start of the game, we deactivate all the weapons in the weapon list (we can add as many weapons in the list as we want, but we have restricted the number to two for the sake of simplicity) and activate the first weapon on the list. Each time the player scrolls the mouse wheel or presses the keys “1” or “2”, we switch weapons by deactivating the current weapon and activating the desired one. The script responsible for switching weapons is called Weapon Controller and, every time the soldier switches weapons, a command is sent to the server through the Player Network script so that all the clients can see that he switched his weapon. The same principle is used to synchronize the changes to all clients when the soldier switches his weapon’s scope or attachment.

Switching weapons

Once again, the switching of weapons is an action that must be transferred across the network; otherwise, when a client switches his weapons, he would be the only one to see, the other clients would not notice, and this would result in networking issues. We can easily assume that the solution to the problem is once again the use of Cmds and Rpcs. If a client wants to switch weapons, he sends a Cmd to the server and the server calls the proper Rpc to all the clients, the Rpc itself contacts the weapon controller and calls the equip function for the desired weapon. As a result, the weapon in the client’s hand is replaced with the desired one in every player’s version of the game.

```
1.  //this script controlls all the weapons/gadgets in the player's inventory
2.  public class WeaponController : MonoBehaviour
3.  {
```

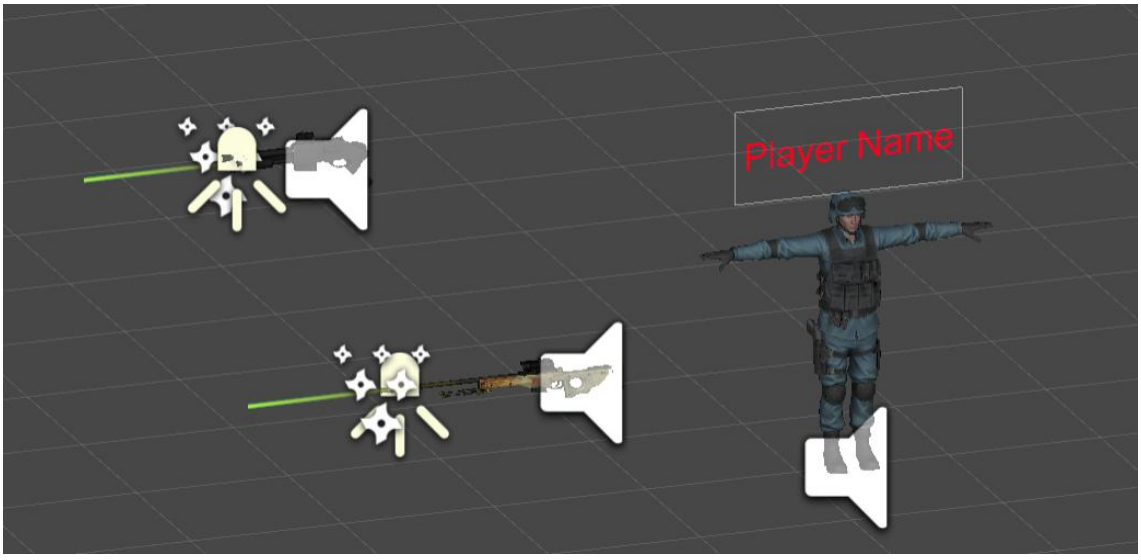
```
4.      protected  PlayerNetWork playerNetwork;
5.      public GameObject cam;
6.      [SerializeField]float weaponSwitchTime;
7.      Shooter[] weapons;
8.      [HideInInspector]
9.      public bool CanFire;
10.     public int currentWeaponIndex;
11.     Transform weaponHolster;
12.     //declare the active weapon
13.     Shooter m_ActiveWeapon;
14.     public Shooter ActiveWeapon
15.     {
16.         get
17.         {
18.             return m_ActiveWeapon;
19.         }
20.     }
21.     //Equips the first weapon in the players inventory
22.     void Awake()
23.     {
24.         CanFire = true;
25.         weaponHolster = transform.Find("Weapons");
26.         weapons =weaponHolster.GetComponentsInChildren<Shooter>();
27.         if (weapons.Length > 0)
28.         {
29.             EquipWeapon(0);
30.         }
31.     }
32.     //calculates the impact point of the projectile
33.     public Vector3 GetImpactPoint()
34.     {
35.         if (Camera.main == null)
36.         {
37.             return  transform.position + transform.forward * 50;
38.         }
39.         Ray ray = Camera.main.ViewportPointToRay (new Vector3 (0.5f, 0.5f, 0.0f));

40.         RaycastHit hit;
41.         //ray getPoint has to be same value as crosshairs z transform
42.         if(Physics.Raycast(ray,out hit))
43.         {
44.             return hit.point;
45.         }
46.         return transform.position + transform.forward * 50;
47.     }
48.     //deactivates the selected weapon
49.     void DeactivateWeapon()
50.     {
51.         for (int i = 0; i < weapons.Length; i++)
52.         {
53.             weapons [i].gameObject.SetActive (false);
54.             weapons [i].transform.SetParent (weaponHolster);
55.         }
56.     }
57.     //used to switch the weapon based on the scroll wheel motion
58.     internal void  SwitchWeapon(int direction)
59.     {
60.         playerNetwork = GetComponentInParent<PlayerNetWork>();
61.         if (playerNetwork != null && !playerNetwork.isLocalPlayer)
```

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

```
62.         { return; }
63.         CanFire = false;
64.         currentWeaponIndex += direction;
65.         if(currentWeaponIndex > weapons.Length-1 )
66.         {
67.             currentWeaponIndex = 0;
68.         }
69.         if(currentWeaponIndex <0)
70.         {
71.             currentWeaponIndex = weapons.Length - 1;
72.         }
73.         playerNetwork.Cmd_EquipWeapon(currentWeaponIndex);
74.     }
75.     //used to equip the desired weapon on the player
76.     internal void EquipWeapon(int weaponNumber)
77.     {
78.         DeactivateWeapon ();
79.         CanFire = true;
80.         m_ActiveWeapon = weapons [weaponNumber];
81.         m_ActiveWeapon.Equip ();
82.         weapons [weaponNumber].gameObject.SetActive (true);
83.     }
84.     // Use this for initialization
85.     void Start ()
86.     {
87.         playerNetwork = GetComponentInParent<PlayerNetWork>();
88.     }
89. }
```

Weapon controller script



Solder – Primary weapon – Secondary weapon

Weapons

Every weapon game object has two scripts attached to it: the Shooter script mainly responsible for shooting the weapon's projectiles and the Reloader script responsible for the weapon's ammo and the reloading mechanisms.

Weapon Shooter

The shooter script is the one that places the active weapon in the hand of the soldier when he switches his weapon. This script has references to the muzzle, the projectile, the muzzle effect, crosshair position, rate of fire, the UI images and the sound clip number of the weapon. Specifically:

- **Muzzle:** The position from which all projectiles are fired; it is located at the open end of each weapon's gun barrel.
- **Projectile:** The object that is fired from this weapon/gadget. There are so far 3 projectiles in the game: a bullet, a grenade and a grappling hook.
- **Muzzle Effect:** A particle system, modified to look like a burst effect from the muzzle, which plays every time the players fires his weapon, lending the weapon a more realistic look when firing.
- **Crosshair position:** The position where the UI's crosshair should be placed in each weapon. If the player is not aiming, we will display the crosshair in that position. This position is located 30 units ahead of the muzzle position to indicate where the bullet will be sent to after being shot.
- **Rate of fire:** This is a float that differs according to each weapon and which is used to determine how fast the weapon can shoot a projectile; the smaller this number, the faster the weapon can fire.
- **UI images:** Every weapon has stored images that are used to display the weapon in the player's user interface; in this way, the player can be sure of which weapon he has selected just by looking at the weapon image.
- **Sound clip:** The player has a list of sound clips stored which includes the sounds of the weapons; using the selected number, the weapon can choose the correct sound clip from the list and play it every time we fire.

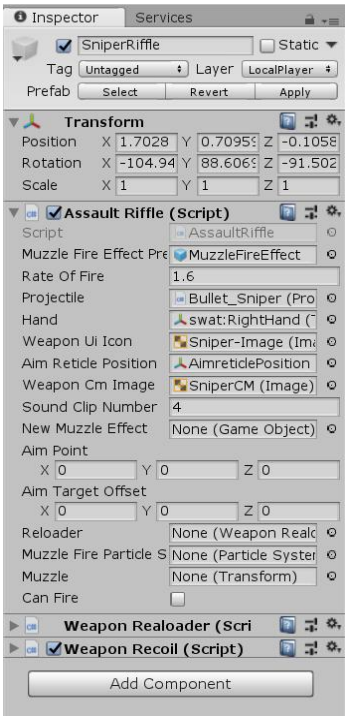
Each time the user fires the weapon, a projectile (unique to every weapon) is fired from the muzzle position. At the same time, the muzzle particle effect will play once, as well as the right sound clip. As mentioned earlier, the rate the weapon shoots projectiles depends on each weapon's particular rate of fire specification.

```
1.  public class Shooter : MonoBehaviour
2.  {
3.      //references essential for the weapon to operate
4.      [SerializeField] GameObject muzzleFireEffectPrefab;
5.      [SerializeField]float rateOfFire;
6.      [SerializeField]Projectile projectile;
7.      [SerializeField]Transform hand;
8.      [SerializeField]public Image weaponUiIcon;
9.      [SerializeField]public Transform aimReticlePosition;
10.     [SerializeField] public Image weaponCmImage;
11.     [SerializeField] public int SoundClipNumber;
12.     public GameObject newMuzzleEffect;
13.     Player  player;
14.     public Vector3 aimPoint;
15.     public Vector3 AimTargetOffset;
16.     public WeaponRealoader Reloader;
17.     public ParticleSystem muzzleFireParticleSystem;
18.     private PlayerNetWork playerNetWork;
```

```
19.     private WeaponRecoil m_WeaponRecoil;
20.     public WeaponRecoil WeaponRecoil
21.     {
22.         get
23.         {
24.             if(m_WeaponRecoil==null)
25.             {
26.                 m_WeaponRecoil = GetComponent<WeaponRecoil>();
27.             }
28.             return m_WeaponRecoil;
29.         }
30.     }
31.     float nextFireAllowed;
32.     public Transform muzzle;
33.     public bool canFire;
34.     ObjectPool objectPooler;
35.     public void SetAimpoint(Vector3 target)
36.     {
37.         aimPoint = target;
38.     }
39.     // places the active weapon/ gadget to the soldiers hand
40.     public void Equip()
41.     {
42.         transform.SetParent (hand);
43.         transform.localPosition = Vector3.zero;
44.         transform.localRotation = Quaternion.identity;
45.     }
46.     //setting all the references of the weapon at start
47.     void Awake()
48.     {
49.         muzzle = transform.Find("Model/Muzzle");
50.         player = GetComponentInParent<Player> ();
51.         Reloader = GetComponent<WeaponRealoader> ();
52.         playerNetWork = player.GetComponent<PlayerNetWork>();
53.         muzzleFireParticleSystem = muzzle.GetComponent<ParticleSystem> ();
54.         newMuzzleEffect = (GameObject)Instantiate(muzzleFireEffectPrefab, muzzle.p
    osition, muzzle.rotation);
55.         objectPooler = ObjectPool.Instance;
56.         objectPooler.CreatePool(this.name, projectile.gameObject, Reloader.clipSiz
    e);
57.     }
58.     //reloads the weapon and playes the audio clip for reload
59.     public void Reload()
60.     {
61.         if (Reloader == null)
62.         {
63.             return;
64.         }
65.         if (player.IsLocalPlayer)
66.         {
67.             playerNetWork.Cmd_ReloadWeapon();
68.         }
69.         playerNetWork.Cmd_PlayAudioClip(7);
70.     }
71.     // activates the particle system created to display a muzzle flash in the weap
    on's muzzle when firing a projectile
72.     public void PlayMuzzleFlashParticleSystem()
73.     {
74.         if (muzzleFireParticleSystem == null)
```

```
75.         {
76.             return;
77.         }
78.         muzzleFireParticleSystem.gameObject.SetActive(true);
79.         muzzleFireParticleSystem.Play ();
80.     }
81.     //fires the next projectile form the weapon's muzzle if allowed and plays the
    audio clip of the gunsound
82.     public virtual void Fire()
83.     {
84.         canFire = false;
85.         if (player.InputState.IsProne || player.InputState.IsSprinting )
86.         {
87.             return;
88.         }
89.         if (Time.time < nextFireAllowed)
90.         {
91.             return;
92.         }
93.         if (player.IsLocalPlayer && Reloader != null)
94.         {
95.             if (Reloader.IsReloading)
96.             {
97.                 return;
98.             }
99.             if(Reloader.RoundsRemainingInClip==0)
100.             {
101.                 return;
102.             }
103.             Reloader.TakeFromClip (1);
104.         }
105.         nextFireAllowed = Time.time + rateOfFire;
106.         Projectile newProjectile= objectPooler.SpawnFromPool(this.name, muzzle.t
    ransform.position,muzzle.rotation).GetComponent<Projectile>();
107.         newProjectile.SetPlayer(player);
108.         //place and play the muzzle particle effect of the weapon
109.         newMuzzleEffect.transform.position = muzzle.position;
110.         newMuzzleEffect.GetComponent<ParticleSystem>().Play();
111.         if (newMuzzleEffect == null)
112.         {
113.             Debug.Log("Empty particle system prefab ");
114.         }
115.         if (newProjectile.GetComponent<Grenade>() != null && newProjectile.GetCo
    mponent<Rigidbody>()!=null)
116.         {
117.             Rigidbody newGrenadeRb = newProjectile.GetComponent<Rigidbody>();
118.             newGrenadeRb.AddForce(muzzle.forward* 800.0f);
119.         }
120.         if (this.WeaponRecoil)
121.         {
122.             this.WeaponRecoil.Activate ();
123.         }
124.         playerNetWork.Cmd_PlayAudioClip(SoundClipNumber);
125.         canFire = true;
126.     }
127.
128. }
```

Weapon shooter script - handling every weapon's main functions



Sniper rifle - Inspector view

Weapon reload

There is a reloading system that works with every weapon in the game, and each weapon object contains the re-loader script that manages the amount of ammo (projectiles) each player has at his disposal. Every weapon has the following attributes to manage its ammo usage:

1) **Weapon type:** An enumerator indicating the type of the current weapon or gadget, such as sniper, shotgun, etc.

2) **Max ammo:** A number indicating the maximum number of projectiles the player can "carry" for each weapon.

3) **Clip size:** Another number, this time used to indicate the maximum number of projectiles a weapon can carry in its magazine

4) **Reloading time:** Indicating the time needed for each weapon to perform a full reload.

The initial amount of ammo for every weapon is based on the max ammo amount. Each time a player fires a projectile, we deduct the amount from the clip and later from the total amount of ammo. The player can only fire projectiles based on the amount of ammo remaining in the clip/magazine; if there is no more ammo left in the clip but the player still has ammo left for this weapon in his inventory, he has to reload his weapon to be able to fire the rest of the projectiles. Every time the player reloads his weapon, we add to the rounds remaining in the clip amount the amount of ammo missing from the clip.

```
1.  public class WeaponReloader : MonoBehaviour {
2.
3.      [SerializeField]int maxAmmo;
4.      [SerializeField]float reloadTime;
```

```
5.      [SerializeField]public int clipSize;
6.      [SerializeField]Container inventory;
7.      [SerializeField]EWeaponType weaponType;
8.      public int shotsFiredInClip;
9.      bool isReloading;
10.     System.Guid containerItemId;
11.
12.     public event System.Action OnAmmoChanged;
13.     //calculate the rounds remaining in the magazine
14.     public int RoundsRemainingInClip
15.     {
16.         get
17.         {
18.             return clipSize - shotsFiredInClip;
19.         }
20.     }
21.     //ammo remaining in the players inventory
22.     public int RoundsRemainingInInventory
23.     {
24.         get
25.         {
26.             return inventory.GetAmmountRemaining(containerItemId);
27.         }
28.     }
29.     //check if the player is already reloading
30.     public bool IsReloading
31.     {
32.         get
33.         {
34.             return isReloading;
35.         }
36.     }
37.     void Awake()
38.     {
39.         inventory.OnContainerReady += () => {
40.             containerItemId = inventory.Add (weaponType.ToString(),maxAmmo);
41.         };
42.     }
43.     //reload weapon/gadget
44.     public void Reload()
45.     {
46.         if(isReloading)
47.         {
48.             return;
49.         }
50.         isReloading = true;
51.         //add time dealy for reload based on weapon type
52.         GameManager.Instance.Timer.Add (()=>{
53.             ExecuteReload(inventory.TakeFromContainer (containerItemId, clipSize -
    RoundsRemainingInClip));
54.         },reloadTime);
55.
56.     }
57.     //finish reload
58.     private void ExecuteReload(int amount)
59.     {
60.         isReloading = false;
61.         shotsFiredInClip-=amount;
62.         HandleOnAmmoChanged ();
```

```
63.     }
64.     //take bullets from clip
65.     public void TakeFromClip(int amount)
66.     {
67.         shotsFiredInClip += amount;
68.         HandleOnAmmoChanged ();
69.     }
70.     public void HandleOnAmmoChanged()
71.     {
72.         if (OnAmmoChanged != null)
73.         {
74.             OnAmmoChanged ();
75.         }
76.     }
77. }
```

Weapon reloader script

Weapon components

Each Weapon has the following components: Model, Muzzle, Scope container, Attachment container. Model, as the name indicates, contains the model of each weapon, and inside each model an empty game object called Muzzle is placed in the weapon's muzzle position. The muzzle is the position on the weapon from where all the projectiles are fired. The next two components have more or less the same functionality. Every weapon contains a list of scopes to help the player aim when he presses the right click, and a list of attachments used regularly in modern first person shooter games. There are 4 different scopes to choose from: two of the scopes are regular ones of the red dot type, but the other two are actually magnification scopes using a special shader in the scope lens to magnify the image they see.

Scope example - scope with magnification effect

Scope example - Red dot scope

Every weapon contains 3 attachments: one red laser, one green laser and a flashlight. The flashlight is actually a yellow unity spotlight that can reach a length of 30 units. The lasers are line renderers with a script attached to them that makes them expand forwards until they hit an object. The texture of the line render is a laser texture tinted red and green for each laser.

Attachment example - Red and green laser

Projectiles

Every weapon or gadget can fire a projectile, which in this game come in 3 different variations: bullet, grenade and hook. The projectile system is designed in such a way as to be able to support as many different types of projectiles needed for the game, but the variations were limited in this project to 3 projectiles for the sake of simplicity. Each of these variations inherits from the class projectile but has different implementations of the code.

Projectile Base Class

Each projectile has the following attributes, and by extension so do the variations of the projectiles:

- **Speed:** A value representing the speed each projectile will move after being shot. The projectiles usually travel forward from the weapon's muzzle position at high speed without being subject to the forces of gravity, with the exception of grenades, which use gravity and physics to hit the ground after a delay of some seconds.
- **Damage:** A value representing the amount of damage the projectile will inflict on a player if it successfully hits him.
- **Time to live:** A float used to monitor how much time a projectile will stay active if it does not collide with any objects while moving. If a projectile does not hit any objects after being fired, we want to deactivate it so it does not take any more processing power.
- **Time alive:** We use this value to calculate how much time a projectile was kept active; if this number reaches the value of the previous variable, we deactivate the projectile and reset the time alive value.

In conclusion, after being fired from a weapon, each projectile will travel forward until it hits a game object. If it hits an object, and that object is a player, it will deduct the amount of damage it inflicts from the player's health points. If the projectile hits a regular object, it does not inflict any damage and becomes inactive immediately. On the other hand, if the projectile does not hit any objects while moving, it will become inactive when the time it has been active reaches the maximum limit we have declared for the projectile.

```
1.  [RequireComponent(typeof(Rigidbody))]
2.  public class Projectile : NetworkBehaviour
3.  {   //essential references for projectile
4.      //like tha player who sent the shot
5.      protected Player player;
6.      [SerializeField]protected  float projectileSpeed;
7.      [SerializeField]protected float timeToLive;
8.      protected float timeAlive;
9.      protected bool isProjectileAlive = true;
10.     [SerializeField]public float damage;
11.     [SerializeField]public Transform bulletHole;
12.     protected Transform projectileTransform;
13.     protected Vector3 projectilePosition;
14.     protected RaycastHit hitWanted;
15.     protected Vector3 destination;
16.     protected Transform hitNormalTransform;
17.
18.     //we set the player that shot the projectile to our player in case we need to
    assigna kill to him
19.     public virtual  void SetPlayer(Player _player)
20.     {
21.         player = _player;
22.     }
23.     void Start()
24.     {
25.         projectileTransform = transform;
26.         projectilePosition = projectileTransform.position;
27.     }
28.     //we deactivate the projectile when it reaches its destination
29.     protected virtual void OnDestinationReached()
30.     {
31.         DestroyProjectile();
32.         return;
33.     }
34.     //moves the projectile forward and checks for possible collision
```

```
35.     protected virtual void MoveProjectile()
36.     {
37.         if (IsDestinationReached())
38.         {
39.             OnDestinationReached();
40.             return;
41.         }
42.         transform.Translate(Vector3.forward * projectileSpeed * Time.deltaTime);
43.         if (destination != Vector3.zero)
44.         {
45.             return;
46.         }
47.         RaycastHit hit;
48.         if (Physics.Raycast(transform.position, transform.forward, out hit, 5f))
49.         {
50.             hitWanted = hit;
51.             CheckDestructible(hit);
52.         }
53.     }
54.     //moves the projectile or deactivates it if its reached its maximum life spawn

55.     [ServerCallback]
56.     void Update()
57.     {
58.         timeAlive += Time.deltaTime;
59.         if ((timeAlive > timeToLive)||!isProjectileAlive)
60.         {
61.             DestroyProjectile();
62.         }
63.         MoveProjectile();
64.     }
65.     // checks if the object the projectile is about to hit is of type destructible

66.     protected virtual void CheckDestructible(RaycastHit hitInfo)
67.     {
68.         var destructible = hitInfo.transform.GetComponentInParent<Destructible>();

69.         if(destructible == null)
70.         {
71.             destination = hitInfo.point+ hitInfo.normal *0.001f;
72.         //TO INSTANTIATE BULLETHOLE-was disabled to lessen the network bandwith
73.         //  hitNormalTransform = (Transform)Instantiate (bulletHole,destination, Q
    uaternion.LookRotation(hitInfo.normal)*Quaternion.Euler(0,180,0));
74.         //  hitNormalTransform.SetParent (hitInfo.transform);
75.         }
76.         else if (destructible.tag == "Player" && isProjectileAlive)
77.         {
78.             ApplyDamageToPlayer(destructible);
79.         }
80.     }
81.     // the function called when the projectile reaches a destination / hits an obj
    ect
82.     protected virtual bool IsDestinationReached()
83.     {
84.         if(destination == Vector3.zero)
85.         {
86.             return false;
87.         }
88.         Vector3 directionToDestination = destination - transform.position;
```

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

```
89.         float dot = Vector3.Dot (directionToDestination, transform.forward);
90.         if (dot < 0)
91.         {
92.             return true;
93.         }
94.         return false;
95.     }
96.     //applies damage to the object with the destructible script-- in this project only players have one
97.     protected virtual void ApplyDamageToPlayer(Destructible destructible)
98.     {
99.         destructible.Cmd_TakeDamage(damage, player.gameObject);
100.         isProjectileAlive = false;
101.         return;
102.     }
103.     // we deactivate the projectile
104.     protected void DestroyProjectile()
105.     {
106.         gameObject.SetActive(false);
107.     }
108.     // we reset the projectile when we need to use it again
109.     protected void OnEnable()
110.     {
111.         ResetProjectile();
112.     }
113.     //resets the projectile's attibutes
114.     public virtual void ResetProjectile()
115.     {
116.         isProjectileAlive = true;
117.         timeAlive = 0;
118.         destination = Vector3.zero;
119.     }
120. }
```

Projectile script – inherited from all projectile classes

Bullet

The bullets used the exact code of the projectile class; in future, however, the code will be transferred to the bullet and the master class projectile will become an empty script used as an interface for projectiles. Originally, the bullet had the ability to create a bullet hole on the point of impact with an object, but this was later removed because it used too much networking bandwidth to create this effect.

Bullet

Grenade

A grenade uses rigidbody, a Unity component to apply physics to the object. After activating the grenade, we apply force to it to make it move, instead of moving it linearly as in the case of the other two projectiles. The grenade has a **particle system** that displays an explosion, a **countdown timer,** and an **explosion radius**. Each time we fire a grenade, the countdown timer starts. When it reaches zero, the grenade explodes in a radius specified from the radius variable we mentioned earlier. Every object that is inside that radius and contains a rigidbody is moved by the explosion and damage is inflicted on every player. At the time of the explosion, the particle effect will play, making the explosion much more realistic.

Grenade model

```
1.  public class Grenade : Projectile
2.  {
3.      float countdown;
4.      bool hasExploded = false;
5.      public GameObject explosionEffect;
6.      public float radius = 5.0f;
7.      public float explosionForce = 700;
8.
9.      // Use this for initialization
10.     void Start ()
11.     {
12.         countdown = timeToLive;
13.     }
14.     // Update is called once per frame
15.     void Update ()
16.     {
17.         countdown -= Time.deltaTime;
18.         if (countdown <= 0.0f && !hasExploded)
19.         {
20.             Explode();
21.         }
22.     }
```

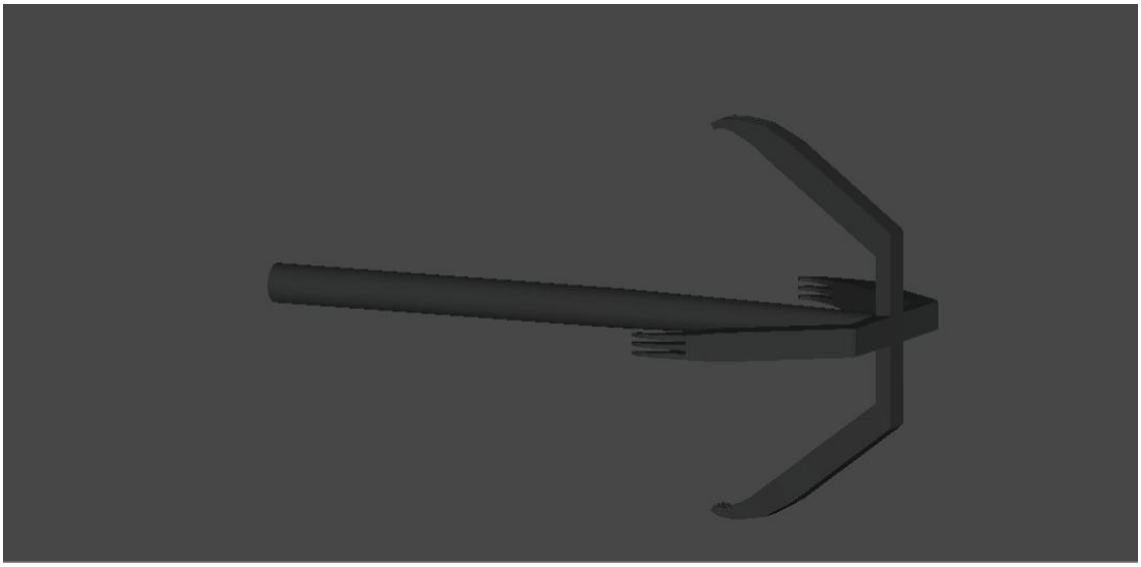
```
23.     //used to synchronise the explosion effect across the network
24.     [Command]
25.     public void Cmd_Explode()
26.     {
27.         Rpc_Explode();
28.     }
29.     [ClientRpc]
30.     void Rpc_Explode()
31.     {
32.         Explode();
33.     }
34.     //the function responsible for the explosion of the grenade
35.     void Explode()
36.     {
37.         //show effect
38.         Instantiate(explosionEffect, transform.position, transform.rotation);
39.         //damage nearby players /destructibles
40.         Collider[] collidersToDamage = Physics.OverlapSphere(transform.position, r
    adius);
41.         foreach (Collider nearbyCollider in collidersToDamage)
42.         {
43.             Destructible destructible = nearbyCollider.GetComponent<Destructible>()
    ;
44.             if (destructible != null)
45.             {
46.                 ApplyDamageToPlayer(destructible);
47.             }
48.         }
49.         //Move nearby objects
50.         Collider[] collidersToMove = Physics.OverlapSphere(transform.position, rad
    ius);
51.         foreach (Collider nearbyCollider in collidersToMove)
52.         {
53.             Rigidbody rb = nearbyCollider.GetComponent<Rigidbody>();
54.             if (rb != null )
55.             {
56.                 rb.AddExplosionForce(explosionForce, transform.position, radius);

57.             }
58.         }
59.         DestroyProjectile();
60.        countdown = timeToLive;
61.     }
62.     //reset the time to live when grenade is ready for use
63.     private void OnEnable()
64.     {
65.         countdown = timeToLive;
66.
67.     }
68. }
```

Grenade script – inherits functionality from projectile script

Explosion effect

Grappling hook



Grappling hook model

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

The hook has zero damage, because it was not created as a weapon but more as a mobility gadget for the player. Of course, if we so desire, we can make it inflict damage simply by changing that value. The way the grappling hook works at the start is just like the bullet: after being fired, it will travel forward from the weapon's muzzle position, every frame based on its speed.

To make the grappling hook appear more realistic, a line renderer with a rope texture was added, displaying a rope textured line from the hook's end to the muzzle of the weapon, expanding in length as the hook moves forward. In contrast to what happens with the bullet, we do not deactivate the hook when it hits a target but instead begin pulling the player towards the hook's position with enough pulling force to make the player travel there fast. When the player is within 2 units of the hook, we destroy the rope and deactivate the grappling hook. This mechanism gives us the effect of a grappling hook gadget and in addition gives the player class the ability to climb and reach destinations that other players cannot, such as climbing on the roof of a tall building.

Hook effect – Hook traveling, expanding the rope along the way

```
1.  public class GrapplingHook :Projectile
2.  {
3.      protected float PlayerToHookDistance;
4.      Transform muzzleTransform;
5.      private float hookPullForce=60.0f;
6.      private float maxWireLength = 110.0f;
7.      [SerializeField]Transform hookEndTransform;
8.      // creates a virtual rope from the projectile to the weapon's muzzle using a l
    ine renderer
9.      void CreateGrapplingHookRope()
```

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

```
10.     {
11.         LineRenderer grapplingHookRope = GetComponent<LineRenderer>();
12.         grapplingHookRope.SetPosition(0, muzzleTransform.position);
13.         grapplingHookRope.SetPosition(1, hookEndTransform.position);
14.     }
15.     // Use this for initialization
16.     void Start ()
17.     {
18.         muzzleTransform = player.WeaponController.ActiveWeapon.muzzle;
19.     }
20.     // Update is called once per frame
21.     void Update ()
22.     {
23.         SetUpPlayerAndHook();
24.         if (PlayerToHookDistance > maxWireLength)
25.         {
26.             DestroyProjectile();
27.         }
28.         // moves the projectile in every frame
29.         MoveProjectile();
30.         // created a line renderer used as rope
31.         CreateGrapplingHookRope();
32.         if ((hitWanted.point + hitWanted.normal * 2) == player.transform.position)

33.         {
34.             DestroyProjectile();
35.         }
36.     }
37.     //synchronises the hook when it reaches a destination
38.     //when the hook hits an object basedon the srver we call the function to drag
    the player along
39.     [Command]
40.     public void Cmd_HookReachedDestination()
41.     {
42.         Rpc_HookReachedDestination();
43.     }
44.     [Client]
45.     public void Rpc_HookReachedDestination()
46.     {
47.         OnDestinationReached();
48.     }
49.     // used to pull the player to the hook when the hook reaches a destination
50.     protected override void OnDestinationReached()
51.     {
52.         Vector3 playerToHookDirection = destination - muzzleTransform.position;
53.         PlayerToHookDistance = playerToHookDirection.magnitude;
54.         Vector3 normalizedPlayerToHookDirection = playerToHookDirection / PlayerTo
    HookDistance;
55.         player.transform.Translate(normalizedPlayerToHookDirection * Time.deltaTim
    e * hookPullForce);
56.         player.transform.eulerAngles = new Vector3(0, 0, 0);
57.         if (PlayerToHookDistance <= 2)
58.         {
59.             DestroyProjectile();
60.         }
61.     }
62.     //sets the  direction of the hook and the distance between it and the player
63.     private void SetUpPlayerAndHook()
64.     {
```

```
65.         Vector3 playerToHookDirection = transform.position - muzzleTransform.posit
    ion;
66.         PlayerToHookDistance = playerToHookDirection.magnitude;
67.     }
68. }
```

Grappling hook script – inherits functionality from projectile script

Object pooling

In an earlier version of the project, every time the player fired his weapon we created a new projectile based on the weapon and destroyed it when we did not need it anymore. Although this method worked, it was frankly not the most effective approach to the problem. We need to avoid constantly creating and destroying objects since this process slows downs unity and especially a networking game. This issue might not be visible in a small project but in a large project with more than one player in a scene who play simultaneously and whose weapons can fire more than 30 rounds in under a minute this way of doing things can significantly slow down the game and even cause a server disconnection. To solve this problem, a method called object pooling, widely used in unity, was implemented. Object pooling is a method that, instead of creating and deleting objects, creates a pool (group) of objects when the game loads and deactivates them, using a queue to keep track of the objects. Each time we need to use a specific object such as a projectile, we activate one of the inactive objects, use it and deactivate it when we no longer need it. It could be said that, in a way, instead of creating and disposing objects and processor power with no control, we recycle the objects while keeping the computing power to a minimum. This method is much more effective and can be used in many cases, not only with projectiles, such as UI elements, health ammo kits – basically, wherever we need more than one object that we will constantly use. In this project, object pooling was used to keep the creation and destruction of the projectiles to a minimum. The system operates as follows:

1) After the game loads, we create an object pool for every weapon the player has in his inventory: this means 2 object pools for every player (primary and secondary weapon/gadget).
2) For every pool, we create a number of projectiles based on the weapon's clip size. For example, the sniper rifle can have 10 bullets in its magazine, so we just create a pool called sniper rifle that contains 10 sniper rifle bullets (projectiles).
3) After the pools for every player are created, we deactivate every object in the pools.
4) When a player fires his weapon, we take the first inactive projectile from the pool queue of his currently active weapon, place it in the weapons muzzle and activate it.
5) The scripting of the projectile remains the same. If it is a bullet, for example, it will travel forward from the muzzle position until it hits a target and apply damage if the target it hits is a player.
6) When the projectile used is no longer needed, instead of destroying it we deactivate it and place it at the end of the queue.
7) The process is repeated even if we reach the last element of the original queue – by adding the inactive objects at the end, we can continue recycling our objects for as long as we want.

To achieve this, a script called Object Pool was created and added to each player class. Each time we need a bullet, we call a function of this script to activate the next object in the queue and place it in the muzzle position. Whenever the active projectile is no longer needed because it has served its purpose, we deactivate it and place it at the end of the queue. Moreover, to avoid coding issues each time a projectile is activated, we reset its values so that it will operate as a brand new projectile; otherwise, the grenades, for example, would explode only the first time that we use them.

The model of the player is under the “Mesh” empty object inside the player, immediately followed by another empty game object termed “Weapons”. This is in essence a folder inside the player object that contains all player weapons and gadgets, and we shall be looking at this in greater detail further below.

```
1.  public class ObjectPool : MonoBehaviour
2.  {
3.
4.      // object pooling
5.      // subclass determining the attributes of a pool
6.      public class Pool
7.      {
8.          public string tag;
9.          public GameObject prefab;
10.         public int size;
11.     }
12.     //singleton
13.     //we create one pool of projectiles from every weapon/gadget in players invent
    ory
14.     //so far 2 pools per player, the amount of projectiles created is determined
15.     //from each weapon's clip size
16.     public static ObjectPool Instance;
17.     private void Awake()
18.     {
19.         Instance = this;
20.         poolDictionary = new Dictionary<string, Queue<GameObject>>();
21.     }
22.     // storing the pools using a dictionary
23.     public Dictionary<string, Queue<GameObject>> poolDictionary;
24.     public List<Pool> pools;
25.     // setting up the pools
26.     void SetupPools()
27.     {
28.         foreach (Pool pool in pools)
29.         {
30.             Queue<GameObject> objectPool = new Queue<GameObject>();
31.             for (int i = 0; i <= pool.size; i++)
32.             {
33.                 GameObject obj = Instantiate(pool.prefab);
34.                 obj.SetActive(false);
35.                 objectPool.Enqueue(obj);
36.             }
37.             poolDictionary.Add(pool.tag, objectPool);
38.         }
39.     }
40.     // create an object pool based on the prefabs the size needed and also assigni
    ng a name to each pool
41.     public void CreatePool(string poolName,GameObject poolPrefab,int poolSize)
42.     {
43.             Queue<GameObject> objectPoolQueue = new Queue<GameObject>();
44.             for (int i = 0; i <= poolSize; i++)
45.             {
46.                 GameObject obj = Instantiate(poolPrefab);
47.                 obj.SetActive(false);
48.                 objectPoolQueue.Enqueue(obj);
49.             }
50.           poolDictionary.Add(poolName, objectPoolQueue);
51.     }
```

```
52.     // used to spawn an object from a pool at the requested position
53.     public GameObject SpawnFromPool(string tag, Vector3 position,Quaternion rotati
    on)
54.     {
55.         if (!poolDictionary.ContainsKey(tag))
56.         {
57.             Debug.LogWarning("Pool with tag " + tag + " deos not exist");
58.             return null;
59.         }
60.         GameObject objectToSpawn = poolDictionary[tag].Dequeue();
61.         objectToSpawn.SetActive(true);
62.         objectToSpawn.transform.position = position;
63.         objectToSpawn.transform.rotation = rotation;
64.         poolDictionary[tag].Enqueue(objectToSpawn);
65.         return objectToSpawn;
66.     }
67. }
```

Object pool script

Sound

Sound is another perfect example of how things that we consider simple can cause serious issues in a networked game. While the project was still in single player mode, there was no issue with the sound: the soldier's footstep sounds, the weapon's gunshot sounds and the reloading sound worked perfectly. However, once the game was converted to multiplayer mode, most of the sounds were distorted and could not be transferred to other clients. The sound could not be played directly from the weapon as the weapons (child objects) did not recognize the network id component of the player (parent object). Adding a network id to each weapon would only burden the bandwidth but make the possibility of a disconnection higher. One solution would be to pass the sound clip through a Cmd in the same way as we would do for a normal variable but UNET does not support this. The solution once again was the use of Rpcs and Cmds, but instead of passing the whole sound clip as a variable all the sound clips were added to a list in the soldier's game object and we only passed the number of the sound clip we wanted to play. Every time a script needs to play a sound, that script will call the Cmd method from the player network script using the proper sound clip number. For example, when we shoot a projectile through the shooter script, it will use a reference of the player network script to call the Cmd created to play sound clips using the proper sound clip number that matches the selected weapon. Thus, the sound is transmitted to all clients with no distortion using the least possible amount of bandwidth.

Picking up ammo

A picking up mechanism was also added to this project. This is modular and can be used to pick up various items, although in our case it was not used for anything but ammo pick up, that means that every ammo box has an ammo pick up script inheriting from the pickup script. Approximately in the center of the map, six crates are located. These represent the ammo boxes for the six different types of weapons or gadgets in this game so far. Each contains a specific amount of ammo for the selected weapon type and has a re-spawn time; each of these values can easily be modified in the inspector of the crate's game object should we wish to add more ammo or use it for a newly added weapon.

Every time a player's collider touches that of one of the crates, the crate is deactivated, making it appear as if the crate was picked up by the player, and the ammo is added to the inventory of the player. If the player picks up an ammo crate for the weapon he is currently carrying, we can

notice the ammo being immediately added to his maximum carried ammo. If the player cannot carry any more ammo, then no ammo is added even if he picks up the correct crate. We can also assume that if, for example, the player picks up 30 bullets but has space only for 10, then only ten bullets will be added to his maximum ammo. After the crate is picked up, in other words deactivated, its timer begins counting and the re-spawning time we set up in its inspector is reactivated.

Ammo crates preview

```
1.  // this script is used to pick up ammo boxes and add the ammo to the proper weapon
     based on the weapon/ gadget type
2.  // it extends the pick up item script
3.  public class AmmoPickup : PickupItem
4.  {
5.      [SerializeField]EWeaponType weaponType;
6.      [SerializeField]float respawnTime;
7.      [SerializeField]int amount;
8.      public override void OnPickup(Transform item)
9.      {
10.         base.OnPickup (item);
11.
12.         var playerInventory = item.GetComponentInChildren<Container> ();
13. //we deactivate the ammo box for some seconds, and re activate it after
14.         GameManager.Instance.Respawner.Despawn (gameObject, respawnTime);
15.         playerInventory.Put (weaponType.ToString(),amount);
16.         //we add the ammo picked up to the amount of projectiles the player is car
    rying
17.         //for that specific weapon,it we pick up more than we can carry we only pi
    ck up
18.         //enough ammo to reach the max amount able to be carried
19.         item.GetComponent<Player> ().PlayerShoot.ActiveWeapon.Reloader.HandleOnAmm
    oChanged ();
20.     }
21. }
```

Ammo pickup script – inheriting from pickup script

User interface

In this section, we shall explain how the User interface or UI elements of the game work. To display UI in a game, Unity uses a canvas in the form of an empty panel that can host 2D images and other elements such as text. The game mainly uses 3 UI canvases: one located inside the soldier, one saved as a unity prefab that is created for each player when the game begins and one world space canvas located above the soldier's head displaying his account name. In a later development stage of the game, every UI element will be transferred to the prefab we mentioned, but they have been kept separate until now to avoid causing unexpected errors in the gameplay. Below we list every UI element used in this project and later we shall explain these in greater depth in terms of their use and the way in which each functions.

1. Mini-map
2. Icon image
3. Scoreboard
4. Scoreboard Item
5. Player name
6. Health bar
7. Weapon Panel
8. Crosshair
9. Kill feed item

Player canvas

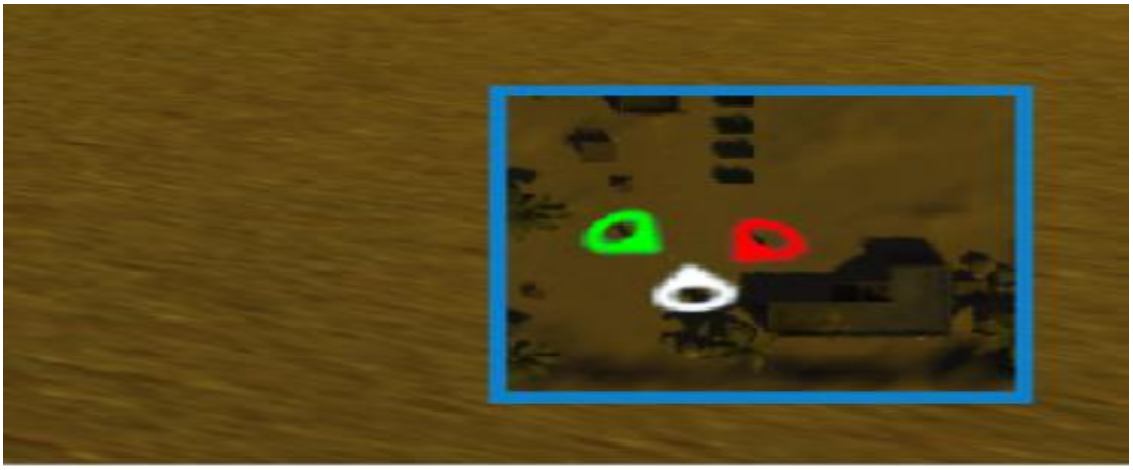
Mini-map and icon image

Inside the player object, we can also find an object PlayerNetworkCanvas; that is to say, the player canvas used to display the mini-map and scoreboard UI, directly after which we meet a camera termed Mini-map Camera and a quad Image termed MinimapQuadImage. The mini-map Camera is a camera placed above the player that observes them in game and sends the image it records in the (raw) image inside the player canvas mentioned earlier to display a top view render of the scene above the player; in other words, a mini-map, as titled in many video games. The quad previously mentioned is a simple 2D image between the mini-map camera (icon image) and the player, so when we look at the mini-map we see this icon and understand where our player is located in the game and the direction he is facing. In simpler terms, each player has a camera placed some units above them, rendering the soldier, his icon image and the map from a top down view. This rendering is then transferred to an image in their UI panel, resulting in a mini-map with the icon representing the position and rotation of the player.

```
1.  // this is the script that creates a minimap at the right bottom corner of the scr
    een
2.  public class Minimap : MonoBehaviour
3.  {
4.      //essential references for minimap like the local player
5.      //and the raw image texture in the canvas where we display
6.      //the image recorded from the minimap camera above the player
7.      public static Transform localPlayerTransform;
8.      Player player;
9.      RenderTexture minimapRenderTexture;
10.     Camera minimapCamera;
11.     [SerializeField] RawImage minimapRawImage;
12.     public Transform playerTransform;
13.     // we assign the minimap to our local player to avoid camera confusion over th
    e network
```

```
14.     void SetTarget(Transform t)
15.     {
16.         localPlayerTransform = t;
17.     }
18.     // Use this for initialization
19.     void Start ()
20.     {
21.         player = GetComponentInParent<Player>();
22.         minimapCamera = GetComponent<Camera>();
23.         // we have to create a new texture (in the ui) for each player to display
    the minimap on it
24.         // otherwise all the players will use the same texture which creates confu
    sion in the network
25.         minimapRenderTexture = new RenderTexture(150, 150, 16, RenderTextureFormat.
    ARGB32);
26.         minimapRenderTexture.Create();
27.         if (minimapCamera != null)
28.         {
29.             minimapCamera.targetTexture = minimapRenderTexture;
30.         }
31.         if (minimapRawImage != null)
32.         {
33.             minimapRawImage.texture = minimapRenderTexture;
34.         }
35.     }
36.     // update the position and rotation fo the minimap camera
37.     void LateUpdate ()
38.     {
39.         if (localPlayerTransform)
40.         {
41.                 Vector3 newPosition = localPlayerTransform.position;
42.                 newPosition.y = transform.position.y;
43.                 transform.position = newPosition;
44.                 transform.rotation = Quaternion.Euler(90f, 0f, 0f);
45.         }
46.     }
47. }
```

Minimap script



Mini-map of player

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

Scoreboard and scoreboard item

Inside the PlayerNetworkCanvas, in addition to the mini-map there is also the scoreboard**.** The scoreboard parent object is a simple 2d image used as a panel that contains the rest of the scoreboard. The image contains a title, two texts used as team titles and two transforms with a vertical layout group, one for each team. The transforms are used as placeholders for the scoreboard items separated into two lists, one for each team. The scoreboard item is a 2D unity prefab created from a single image and 3 texts inside the image. The first text is used to display the player's name in the player's selected color, the second to display their total amount of kills and the third their total amount of deaths. The scoreboard items are created every time the user activates the scoreboard and later destroyed, the reason being that we need to keep track of the player's score at all times. Consequently, we refresh his score continuously. The scoreboard works as follows: The scoreboard is deactivated and is activated only when the player presses the Tab key to enable the panel. When the scoreboard is activated, the scoreboard items are created and then placed in one of the two lists based on the player's team according to the vertical layout group, with the next item from the same team placed below the first one etc. When the scoreboard is deactivated, we destroy every item created and repeat the process.

Scoreboard of the game

```
1.  public class ScoreBoard : MonoBehaviour
2.  {
3.      [SerializeField] GameObject ScoreboardItemPlayer;
4.      // we create 2 score lists one for each team
```

```
5.      [SerializeField] Transform ScoreboardPlayerListTeam1;
6.      [SerializeField] Transform ScoreboardPlayerListTeam2;
7.      Image ScoreBoardImage;
8.      public bool IsScoreBoardEnabled = false;
9.      // Use this for initialization
10.     void Start ()
11.     {
12.         ScoreBoardImage = this.transform.GetComponent<Image>();
13.         DeactivateScoreBoard();
14.     }
15.     // activates the scoreboard
16.     public void ActivateScoreBoard()
17.     {
18.         SetPlayerScoreBoard();
19.         ScoreBoardImage.enabled = true;
20.         this.enabled = true;
21.         for (int i = 0; i < transform.childCount; i++)
22.         {
23.             transform.GetChild(i).gameObject.SetActive(true);
24.         }
25.         IsScoreBoardEnabled = true;
26.     }
27.     // deactivates the scoreboard
28.     public void DeactivateScoreBoard()
29.     {
30.         ScoreBoardImage.enabled = false;
31.         this.enabled = false;
32.         for (int i = 0; i < transform.childCount; i++)
33.         {
34.             transform.GetChild(i).gameObject.SetActive(false);
35.         }
36.         IsScoreBoardEnabled = false;
37.     }
38.     //sets the scoreboard by clearing it and recrating it based on the new score v
    alues for each player in each team
39.     public void SetPlayerScoreBoard()
40.     {
41.         ClearScoreBoard();
42.         //get array of players
43.         Player[] players = FindObjectsOfType<Player>();
44.         //loop through the array
45.         foreach (Player player in players)
46.         {
47.             PlayerNetWork playerNetwork = player.GetComponent<PlayerNetWork>();
48.             Transform ScoreboardPlayerList;
49.             // we check the team of each player to update their stats someone gets
     a kill
50.             if (playerNetwork.playerTeamNumber == 1)
51.             {
52.                 ScoreboardPlayerList = ScoreboardPlayerListTeam1;
53.             }
54.             else
55.             {
56.                 ScoreboardPlayerList = ScoreboardPlayerListTeam2;
57.             }
58.             // we instantiate the kill feed itam and place it on the board
59.                 GameObject itemGO = (GameObject)Instantiate(ScoreboardItemPlayer,
    ScoreboardPlayerList);
```

```
60.                 ScoreBoardItemPlayer item = itemGO.GetComponent<ScoreBoardItemPlay
    er>();
61.                 if (item != null)
62.                 {
63.                 //we set up the kill feed item's text and color based on the two p
    layers involved
64.                     item.SetUp(player.name, player.PlayerHealth.KillCount, player.
    PlayerHealth.DeathCount,playerNetwork.playerColor);
65.                 }
66.         }
67.     }
68.     //clears the scoreboard by destroying every child object in it
69.     public void ClearScoreBoard()
70.     {
71.         foreach (Transform child in ScoreboardPlayerListTeam1)
72.         {
73.             Destroy(child.gameObject);
74.         }
75.         foreach (Transform child in ScoreboardPlayerListTeam2)
76.         {
77.             Destroy(child.gameObject);
78.         }
79.     }
80. }
```

Scoreboard script - keeping track of in game records for all players

Player name canvas

Another Component inside the player object is the NamePlate, which contains a small canvas in world space mode with a simple text located above the player's head. When the player enters the game and becomes synchronized, the NamePlate will display the player's name in the player color passed from the lobby player. The script responsible for this functionality – PlayerName, – also disables the nameplate text if the player is not alive and re enables it when the player has re-spawned.

UI Prefab canvas

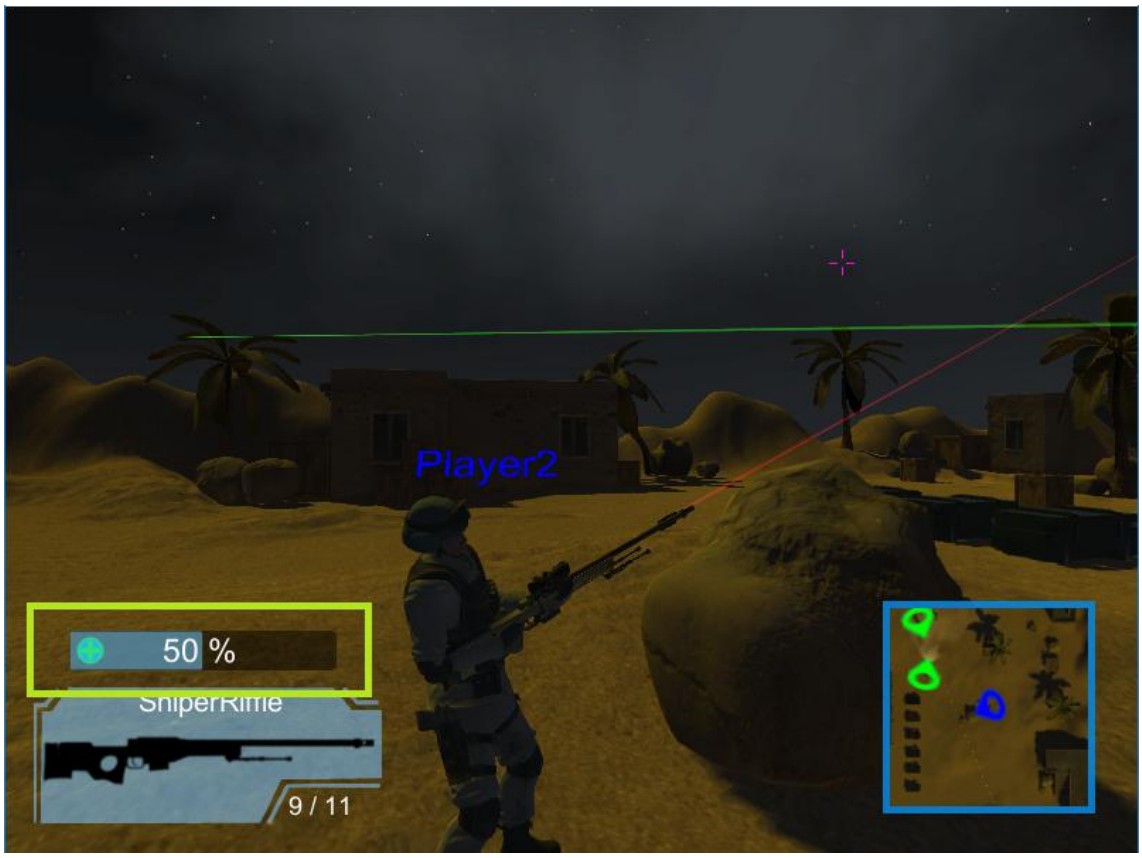
Health bar

A health bar displaying the player's health frequently used in shooter games was created using the following process:

1. We created a background image
2. We added a foreground image docked on the left of the background image
3. A text was added to the background image displaying the remaining health points of the player in text.
4. An image displaying a health icon was added to the background image, adding to the visual appeal of the health bar.

When the player has full health (100 health points), the foreground image has the same length as the background image, but as each player accepts damage the foreground shrinks in width while remaining docked on the left of the background image. In simpler terms, the foreground image is used as a slider image that becomes smaller in width when the player is damaged, in this way

simulating the effect of his health declining. In addition, the health text refreshes every frame displaying the number of health points left for the player.

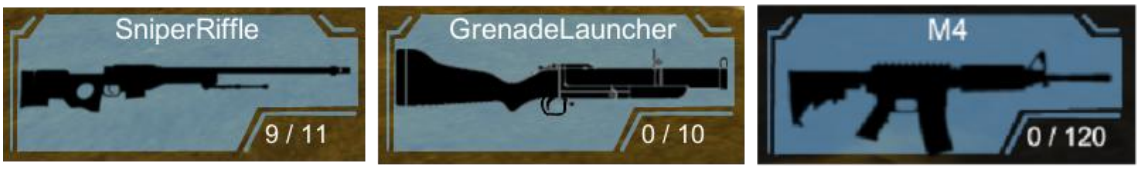


Health-bar of player example

Weapon/Gadget panel

On the left side of the screen below the health bar, the weapon/gadget panel is located. The weapon panel contains the following UI Elements:

1. **Weapon/Gadget Image:** Every weapon game object contains an image saved as a prefab, which displays the weapon in a form of an icon from a side view. We display that image on the weapon panel to inform the user of the current weapon they are holding.
2. **Weapon/Gadget Name:** The weapon name is self-explanatory; it is a text UI element that takes its value from the name of the weapon currently used by the player.
3. **Weapon/Gadget Ammo:** The weapon ammo functions in a similar way to the weapon name; it is a text that gets its value from the amount of ammo in the weapon's magazine (clip), and the character and maximum amount of ammo currently carried by the player for the current weapon. The character "/" is added to the text between these values to separate the ammo in clip from the total ammo.
4. **Background Image:** A simple background image was added behind all UI elements mentioned earlier to give the panel a more modern look.



Weapon / Gadget panel examples

Crosshair

A crosshair is a UI element widely used in shooting games. In most cases, this has the shape of a cross with an empty space in the center indicating that the player has to aim using that empty space as the point of impact of his projectiles. In some games, there is a tendency to stylize the crosshair by giving it a more circular or diamond shape but the concept and functionality remains the same. The crosshair is used when the player is in third person mode to help them aim without the use of a scope.

It is understandable that aiming in first person mode using a scope will be more accurate and easier for a player but occasionally in a game there is no time to switch to first person mode. Moreover, in this way, the player has the ability to fire projectiles more accurately when moving. The crosshair is a dynamic cross that spreads outwards from the center while the player is constantly firing an automatic rifle and returns to its initial form after some seconds when they stop. This functionality was added to display the effect of losing accuracy and make the constant firing of a weapon less effective; otherwise, most players would abuse the mechanism and avoid using bold action rifles like the sniper.

Kill feed item

When a player kills another player, an event is triggered which creates on every player's screen a UI element called Kill feed item, which is a rectangle in the player's color carrying the event's message. Let us suppose that Player1 is in orange and kills Player2; at that moment, an orange rectangle will appear on the top left corner of the screen of every player with the message "Player1 killed Player2".

```
1.  // this class is responsible to create a ui kill feed item whena kill occurs in th
    e scene
2.  public class KillFeedItem : MonoBehaviour
3.  {
4.      public  Text killFeedText;
5.      private Image background;
6.      // Use this for initialization
7.      void Start ()
8.      {
9.          killFeedText = GetComponentInChildren<Text>();
10.     }
11.     private void Awake()
12.     {
13.         background = transform.GetComponent<Image>();
14.     }
15.     // creates the ui item
16.     public void SetupkillFeedItem(string player, string source,Color playerColor)

17.     {
18.         //function called from scoreboard when one player kills another one
19.         // setting the prefabs text as fro example player01 killed player02
```

```
20.         killFeedText.text = "<b>" + source + "</b>" + " Killed " + player;
21.         background = transform.GetComponent<Image>();
22.         background.color = playerColor;
23.     }
24. }
```

Kill feed item example

```
1.  public class PlayerNetworkUI : MonoBehaviour
2.  {
3.      //this script is used to handle the player's ui prefab
4.      public Player player;
5.      private WeaponRealoader weaponReloader;
6.      [SerializeField] RectTransform healthBarFront;
7.      [SerializeField] Text healthText;
8.      //weapons
9.      [SerializeField] Image weaponImage;
10.     [SerializeField] Text weaponNameText;
11.     [SerializeField] Text ammoText;
12.     [SerializeField] Image weaponCmImage;
13.     [SerializeField] Image scopeCmImage;
14.     [SerializeField] GameObject CmPanel;
15.     [SerializeField] RawImage PlayerViewRawImage;
16.     //player
17.     Image playerViewImage;
18.     //aim
```

```
19.     [SerializeField] Transform ReticleTransform;
20.     [SerializeField]GameObject reticleGO;
21.     //kill feed
22.     [SerializeField]Transform killFeedPanelTransform;
23.     [SerializeField] GameObject killFeedItemPrefab;
24.     //crosshair
25.     Transform crossTop;
26.     Transform crossBottom;
27.     Transform crossLeft;
28.     Transform crossRight;
29.     float reticleStartPoint;
30.     public float crosshairSpeed = 10.0f;
31.     public bool IsCustomisePanelEnabled = false;
32.     // Use this for initialization
33.     void Start ()
34.     {
35.         //kill feed item
36.         GameManager.Instance.onPlayerKilled += SetupKillFeedItem;
37.         //aim - setting the crosshair game object
38.         crossTop = ReticleTransform.Find("Cross/Top").transform;
39.         crossBottom = ReticleTransform.Find("Cross/Bottom").transform;
40.         crossLeft = ReticleTransform.Find("Cross/Left").transform;
41.         crossRight = ReticleTransform.Find("Cross/Right").transform;
42.         reticleStartPoint = crossTop.localPosition.y;
43.         //checking if this is our local player to set the Ui on his camera
44.         if (player.IsLocalPlayer)
45.         {
46.             SetUpPlayerViewPanel();
47.         }
48.         DisableCmPanel();
49.     }
50.     // Update is called once per frame
51.     void Update ()
52.     {
53.         SetHealthUi((float)player.PlayerHealth.currentHealth/100.0f);
54.         //updating the weapon Ui
55.         SetWeaponPanel();
56.         //updating crosshairs movement
57.         SetCrosshair();
58.         ApplyScale(player.WeaponController.ActiveWeapon.WeaponRecoil.recoilScale*2
    0);
59.         if (IsCustomisePanelEnabled)
60.         {
61.             DisableCmPanel();
62.         }
63.         if (GameManager.Instance.InputController.CustomiseMenuToggled && !IsCustom
    isePanelEnabled)
64.         {
65.             EnableCmPanel();
66.         }
67.     }
68.     //function used to set the ui prefab to the proper player
69.     public void SetPlayer(Player _player)
70.     {
71.         player = _player;
72.     }
73.     //function for team text- not used currently
74.     void SetTeamText()
75.     {
```

```
76.      //   teamText.text = "Team "+player.Team.ToString();
77.     }
78.     //used to scale the player's haelth bar size and health points text
79.     void SetHealthUi(float _amount )
80.     {
81.         healthBarFront.localScale = new Vector3( _amount, 1f,1f);
82.         float HPpct = _amount * 100;
83.         healthText.text = HPpct.ToString() + " % ";
84.     }
85.     //used to set up the ui weapon panel displaying the current weapon : name, ima
    ge  and remaining ammo
86.     void SetWeaponPanel()
87.     {
88.         weaponReloader = player.GetComponentInChildren<WeaponRealoader>();
89.         ammoText.text = weaponReloader.RoundsRemainingInClip.ToString() + " / " +
    weaponReloader.RoundsRemainingInInventory.ToString();
90.         weaponNameText.text = player.WeaponController.ActiveWeapon.name.ToString();

91.         weaponImage.sprite = player.WeaponController.ActiveWeapon.weaponUiIcon.spr
    ite;
92.         weaponCmImage.sprite = player.WeaponController.ActiveWeapon.weaponCmImage.
    sprite;
93.         scopeCmImage.sprite = player.WeaponController.ActiveWeapon.GetComponentInC
    hildren<Scope>().scopeCmImage.sprite;
94.     }
95.     //used to set the crosshair while in  third person view
96.     void SetCrosshair()
97.     {
98.         if (GameManager.Instance.InputController.IsAiming)
99.         {
100.             DisableReticle();
101.         }
102.         else
103.         {
104.             EnableReticle();
105.             Vector3 screenPosition = Camera.main.WorldToScreenPoint(player.Weapo
    nController.ActiveWeapon.aimReticlePosition.position);
106.             ReticleTransform.position = Vector3.Lerp(ReticleTransform.position,
    screenPosition, crosshairSpeed * Time.deltaTime);
107.         }
108.     }
109.     //disables crosshair
110.     void DisableReticle()
111.     {
112.         reticleGO.SetActive(false);
113.     }
114.     //enables crosshair
115.     void EnableReticle()
116.     {
117.         reticleGO.SetActive(true);
118.     }
119.     //changes the distance betwwen the crosshair's ui lines to display the effec
    t of recoil /accuracy decreased
120.     public void ApplyScale(float scale)
121.     {
122.         crossTop.localPosition = new Vector3(0, reticleStartPoint + scale, 0);
123.         crossBottom.localPosition = new Vector3(0, -reticleStartPoint - scale, 0)
    ;
```

```
124.         crossLeft.localPosition = new Vector3(-reticleStartPoint - scale, 0, 0);
125.         crossRight.localPosition = new Vector3(+reticleStartPoint + scale, 0, 0);
126.     }
127.     //enables the customise menu panels
128.     void EnableCmPanel()
129.     {
130.         CmPanel.SetActive(true);
131.         IsCustomisePanelEnabled = true;
132.     }
133.     //disables the customise menu panels
134.     void DisableCmPanel()
135.     {
136.         CmPanel.SetActive(false);
137.         IsCustomisePanelEnabled = false;
138.     }
139.     //sets up the kill feed item every time a player kills an enemy player
140.     public void SetupKillFeedItem(string player ,string  source,Color playerColo
    r)
141.     {
142.        GameObject killFeedItemGO = (GameObject) Instantiate(killFeedItemPrefab,
    killFeedPanelTransform);
143.         killFeedItemGO.GetComponent<KillFeedItem>().SetupkillFeedItem(player,sou
    rce,playerColor);
144.         Destroy(killFeedItemGO, 4.0f);
145.     }
146.     public void  SetUpPlayerViewPanel()
147.     {
148.     }
149. }
```

Player network Ui prefab script

Conclusion

In conclusion, KILLSHOT is an indie multiplayer shooter game created to fulfil a Master Thesis. The concept behind it was to endeavor to implement as many mechanisms as possible with the fewest possible complications and bugs. Of course, it is not a fully-fledged game that could be seen as a serious threat to Triple-A companies, but, as a personal project, it has reached and surpassed the original goal set. The most important factor is that it was developed in such a way as to be able to support a great deal of different additions with the fewest possible necessary changes and reworks. Some of the possible additions will be discussed in the section below.

Future improvements

The project has plenty of mechanisms and has been developed to support substantial additional content without the need for a fundamental reconstruction. That leads us to believe that the only limit to how far the project can go depends on how far we want to take it. With this in mind, we shall mention content and mechanisms that can be added to enrich the project.

- The aiming mechanism can be replaced by inverse kinematics, a widely known method of aiming in shooter games that is more accurate and more realistic.

- We can add new models of soldiers to each different class. The project can already support the importation of new models and animations and the replacement of the old ones in each class.
- We can add more weapons and gadgets. Weapons are easy to replicate as we only need to change the model of a weapon and values such as changing its rate of fire or its damage capability in the inspector. Gadgets, on the other hand, will be trickier as these need inspiration and to fit the project, as well as being easy to implement.
- We can add new projectiles. By adding these, we give the player new ways to interact with the game and make it more interesting. Some commonly used projectiles in shooter games are smoke grenades or flashbangs, a special kind of grenade that blind the enemy for a short period.
- We could give the host the option to choose the map in which the players are going to spawn from a variety of maps instead of only one, since, after some time, it becomes boring to play in the same map. In addition, we could add the element of weather effects or day and night mode to the maps.
- Other gameplay details that could be added include the use of health kits and ammo kits for players to be able to restore their own and their teammate's health points and ammunition. The ability to revive a teammate would also be an interesting addition.
- An aspect that could be reworked to improve the project is the replacement of UNET with PUN (Photon Unity Networking). This would be a substantial improvement to the project, since UNET will depreciate over the next two years and, in terms of overall performance, PUN is considered a better solution. Using UNET, to make sure we avoid any unexpected disconnections, we would have to pay Unity a very large monthly feed to be able to send bigger packages of data to the connected clients to keep the game synchronized. By using PUN, we would achieve the same quality of work as with Unity's paid service just by using PUN's free services.

Bibliography

For the development of this project, 2 series of tutorial videos were followed :

1)The Third Person Shooter (Multiplayer) from Stevie Rof :

https://www.youtube.com/watch?v=-dVUGLQzORQ&list=PLJfktYG5YLJGK-ROk1Gj1_i8AtnIyeKTs

2) Making a Multiplayer FPS in Unity from Brackeys :

https://www.youtube.com/watch?v=UK57qdq_lak&list=PLPV2KyIb3jR5PhGqsO7G4PsbEC_Al-kPZ

Both of these series have numerous thorough videos that helped in the development of this project, although not everything was used because much was ultimately not needed, e.g. the implementation of npcs. Of course, using only the instructions from these videos was, in itself, insufficient; the rest of the project was developed through personal research in specific sectors, and personal development and testing.

Additional links:

https://docs.unity3d.com/Manual/UNet.html

https://answers.unity.com/questions/1364438/unity-lobby-multiplayer-multiple-player-prefabs.html

https://forum.unity.com/threads/how-to-set-individual-playerprefab-form-client-in-the-networkmanger.348337/#post-2256378

USER MANUAL

Open Project

KILLSHOT : Διαδικτυακό παιχνίδι πολλών παικτών πρώτου και τρίτου προσώπου σχεδιασμένο σε Unity3D

1) Locate the Killshot Builds folder, open the folder and double click the file with the .exe ending (Killshot.exe). Should windows ask for online permission, grant it; it is essential that the game has access to the internet/network.

2) Select your desired resolution and whether you want the game in full screen mode or not, then press PLAY.

Log in /Register

KILLSHOT is a multiplayer-based game, so the use of player accounts is unavoidable, with guest entry unavailable in this project; this means that initially every new user has to set up a simple account.

3) Login menu pops up. If you already have an account, enter your details on the login form and press PLAY. If you do not have an account, press Register and you will be transferred to the Register panel, enter a username and a password twice, to make sure you entered it correctly, and press ENTER.

Lobby

4) You are now located in the Lobby menu. Your account details are located on the right of your screen; there you can find your in-game records, a logout and an exit game button. You have 3 options to join a room: 1) you can join an online server room from the server list, 2) you can connect to a room directly by typing the correct network IP address, and 3) You can create/enter a room using localhost mode, which means a computer in the same network as you, or the same computer with multiple instances of the game running, can join the room.

Create/Join a Room

5) **Use online servers**: If you want to create a server room, enter a server name under the Server List section and press CREATE. If you want to join an existing one, press the button named SERVERS and select your preferred server by name.

6) **Use manual connection:** If you want to create/host a room, press PLAY under the manual connection section. You can also create a dedicated server, but you will not be able to play, only the clients. If you want to join a room, all you have to do is enter the IP of the pc hosting the room: type it under the JOIN A GAME section and press JOIN.

Ready up player

Once you are in a room, you have the choice of customizing your player before entering the game; the player name is already set up based on your username.

7) Select COLOR by pressing the colored square – the color for the player will change to the next available color and will be your player color in the main game.

8) Select player class by pressing the CLASS button. You have 4 available classes: Class 0,1,2, and 3. Each class has different weapons that the others do not have access to, and the system will be able to support different player models in the future.

9) Select your team by pressing the TEAM button; this mode is Team Death Match, which means you have 2 available teams to join and you cannot hurt your teammates in the game, only the enemy team players.

10) Once you have your player set, press JOIN; you cannot modify your player anymore. When all players have pressed JOIN, a countdown timer starts, indicating the beginning of the game, and the lobby will transfer to the main game and spawn the players when it reaches the value zero.

Play the game

11) You are now in the game and have control over your soldier. Once you understand the gameplay and basic controls of the player, you are good to go:

W: Move Forward

S: Move Back

A: Move Left

D: Move Right

Left click: Shoot bullet

Right click: Aim weapon

Mouse wheel Scroll: Change weapon

1: Primary weapon

2: Secondary weapon

3: Change scope

4: Change weapon attachment

Left Shift: Sprint

C: Crouch

Spacebar: Jump

Z: Prone

X: Walk Slow

Tab: Scoreboard

H: Enable cursor

12) Once you become familiar with the controls, you are ready to play. The rules are simple: play as part of your team and try to score as many kills and avoid as many deaths as possible until the match is over.