Roskilde University

Computer Science & Informatics

Exploring Networking Technologies and Architecture Through Game Development

Arash Abedin
Student
IMT Department

Daniel Lesko
Student
IMT Department

Anders Lassen
Supervisor
IMT Department

Eugenio Maria Capuani
Student
IMT Department

Rune Barrett
Student
IMT Department

Dragos Illie
Student
IMT Department

Contents

Glossary of terms and abbreviations used

Throughout the report we make use of a number of terms that may otherwise be unfamiliar to the reader; These terms are listed below:

Local Player/Remote Player: These two terms denote a distinction in the NodeJS implementation and elsewhere, between the game player playing on the local machine on which the server is running, and other players. For each connected client, the local player's control inputs are handled locally on the machine and then broadcast to the server, while his actions are handled by the *PlayerController*. In contrast, a remote player's actions are simply received from the server so that they may be displayed in the local game instance. These functions are carried out by the *OtherPlayerController.* in Node.js version and *PlayerManager* in the Photon version.

PvP : Player versus Player.

A term often used to denote competitive games, whose primary goal is to compete against other human players.

CS : Client-Server.

See chapter 3

P2P : Peer-to-Peer.

See chapter 3

AAA : AAA or Triple-A is an informal classification often applied to games with the highest development budgets and levels of promotion. These games are often developed and published by multinational corporations as opposed to Independent developers.

Chapter 1

Introduction

In this project we're aiming to design, implement and deploy a cross platform online multiplayer game that enables its users to play together while using different platforms including mobile and PC (Windows, Mac and Linux).

The code for the project can be found at: `https://github.com/DREAD-Inc/GameProject2018`

The main focus areas of our project cover the following:

- Designing a game that has a certain degree of complexity.
- Designing the architecture of the game covering mainly the online networking areas and its performance.
- Improving the usability of the game while keeping the same level of user experience and complexity for the users on all platforms.
- Implementing and deploying the game while testing and optimizing the core areas of the game such as usability, networking performance and design.

1.1 Problem Formulation

The main aim of the project is to use the game we will be developing as a platform, in order to explore what it takes to create a multi-platform, multiplayer application with real-time communication. We will explore different approaches to development, software architecture and networking, in order to gain better insights into this. We envision that work might be needed in areas like networking, in order to ensure that players on an un-reliable mobile connection can enjoy the game. Another area we'd like to focus on is user experience. This is done both to explore what the challenges are in this scenario, and also in order to ensure that the controls and UI elements lead to a seamless transition between mobile and PC.

1.2 Exploratory Process

As none of us had very much experience with game design, development and especially game networking models from the start of the project, we chose to carry the development out as an exploratory process rather than a more traditional incremental and carefully planned project.

Because of this exploratory approach, developing the skeleton system of the game took much longer than expected. We spent a lot of time trying to construct a decent networking model with NodeJS, and slowly realized that our project was growing more complex and hard to manage. Mid-way through our development we had to re-focus our efforts: we switched from NodeJS to Photon in order to focus more on developing game features, and re-implementing the game with that framework. We have made some progress in that front, developing a new map and some more game concepts like teams, but sadly we were not able to fulfill our vision fully.

Further more, while we had planned and designed the controls to be easy to use on mobile, we did not get to implement the game on that platform. That said, the reader will find our discussion of UX and mobile platform in section 2.3, alongside UI mockups and a discussion of our experience designing the game with mobile platforms in mind..

Chapter 2

DREAD - The Game

This section will explain our initial plan for the gameplay.

2.1 Initial Gameplay Description

The players will be divided in two teams, that compete against each other.

Each team will have a base each end on the map, there will be various points that can be captured by players throughout the map.

Capturing points will allow the team to gather resources (e.g Gold), that can be used to purchase upgrades. Capturing and holding a majority of the control points of the map will also grant a team the opportunity to gain access to the enemy base so they can attack it. Gold can also be gained by killing other players.

Thus there are two main ways to destroy the enemy base:

- Either capture a majority of the points (with the help of lower level upgrades gained by passive income) and then launch an attack on the base.
- Or hoard Gold by killing players in order to unlock access to expensive items that will allow you to get into the base and destroy it quickly.

The goal of the opposing team is to defend their base and/or attack. Defense can be approached in two ways: they can either decide to meet the attackers on the field and deny them control of the points, or they can bunker down in their base and wait for the enemy to attack.

Our goal with this is to make the game more thoughtful and tactical, and leave the players with choice in how they want to play the game.

2.2 Development Environment & Game Engines

We chose a popular modern game engine partly for the reason that it will make it simpler for us to compile the game on our chosen platforms. Apart from that, it gives us the ability to include complex aspects of game development (such as physics) without spending too much time on it.

In particular we chose to use Unity [9] as it is a combined game engine / IDE that features a fast and simple work flow. The workflow in Unity builds upon a programming pattern known as an Entity Component System (see chapter 4), that makes it simple to make significant changes and reuse code for a large number of different objects in the scene.

2.3 User experience design

In this chapter, we are gonna talk about user experience design in cross-platform multiplayer experience. Our game will be playable on PC, Android and iOS, therefore we have to be careful about how we design the user experience in order to ensure a fair gameplay for everyone.

For our game, we wanted to make the UI for mobile devices as user friendly and minimalist as possible, without hindering mobile players compared to PC players. The ultimate goal for the mobile interface, is to create same player experience as on any other device. Therefore we have to consider how to position buttons and other important information on the screen, in order to not put players in a disadvantage with a lot of useless information, or out of proportion buttons and labels.

In the next sections, we will describe our present mock-ups for both mobile interface and PC interface. Both figure 2.2 and figure 2.3 depict a possible design for the User Interface based of a mental model, and are thus subject to change.

The mobile version is currently not implemented, and the PC version is missing some of the features described in figure 2.3

2.3.1 Usability and UX goals

The primary goal, when designing this game is to be very clear about the objective. We have these two top-level concerns - usability goals and user experience goals. First is concerned about meeting specific criteria such as efficiency or learnability while user experience goals are concerned with analyzing the quality of user experience, such as enjoyability, fun or aesthetically pleasing.

When talking about usability, we want to have our game fulfill most of the goals, which are the following:

- efficient to use (efficiency)
- effective to use (effectiveness)
- safe to use (safety)
- have good utility (utility)
- easy to learn (learnability)
- easy to remember how to use (memorability)

A very popular criterion for assessing whether a system is easy to learn is to apply the *ten-minute rule*. It proposes that novice users should be able to learn how to use a system in under 10 minutes. If not, the system fails [6]. In our case, if the user does not understand the primary objective and/or purpose of the game within first 10 minutes, then we failed to deliver an optimal product. It should be absolutely clear, even to a brand-new user, what the the goal of the application is (be it a game, or not) within first 10 of using an application.

User-experience goals are changing over time. The emergence of technologies such as VR/AR and mobile gaming being extremely popular, has brought much wider set of concerns.

- satisfying
- enjoyable
- fun
- entertaining
- helpful
- motivating
- aesthetically pleasing

- supportive of creativity
- rewarding
- emotionally fulfilling

There are many more goals to consider when talking about user experience obviously, but some suit our context more than others. Primary goals for our game would be fun, entertaining, enjoyable - it's a game. There is a relation between both, where we have to see the difference between unique systems and to assess proper primary goals. The relationship is shown in figure 2.1. [7]

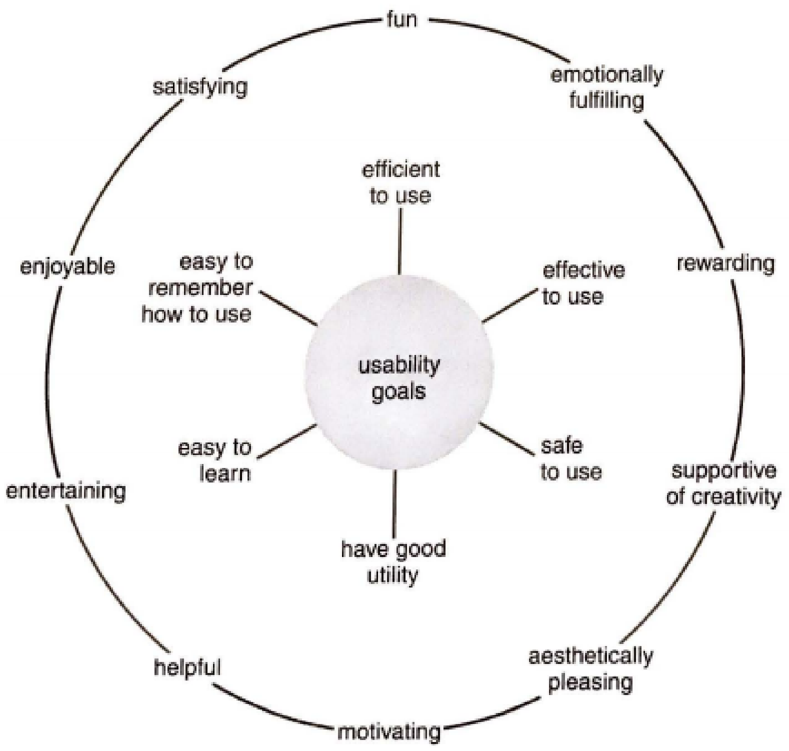


Figure 2.1: Usability and user experience goals

It is also important to assess risks when considering both usability and user experience goals, and we have to understand the trade-off between those two. Sometimes, some combinations will be incompatible and we will have to reconsider our primary goal of the application and adapt.

2.3.2 User interface for mobile (Android and iOS)

In figure 2.2 we can see our initial concept for mobile device interface. We tried to keep up with our standards and make it a competitive experience with PC players. The major difference in gameplay is how movement and shooting work. The idea was to create a balanced experience when playing this competitive PvP game.

The user controls player's movement using left virtual analog stick, moving it in any desired direction; when moving the stick a little further, it performs a dash in the intended direction. Shooting and aiming is merged into the right analog stick, to avoid having the player move their fingers across entire screen and creating confusion. Practically speaking this means that the player will shoot in whatever direction their are aiming towards, if the analog stick is pushed far enough.

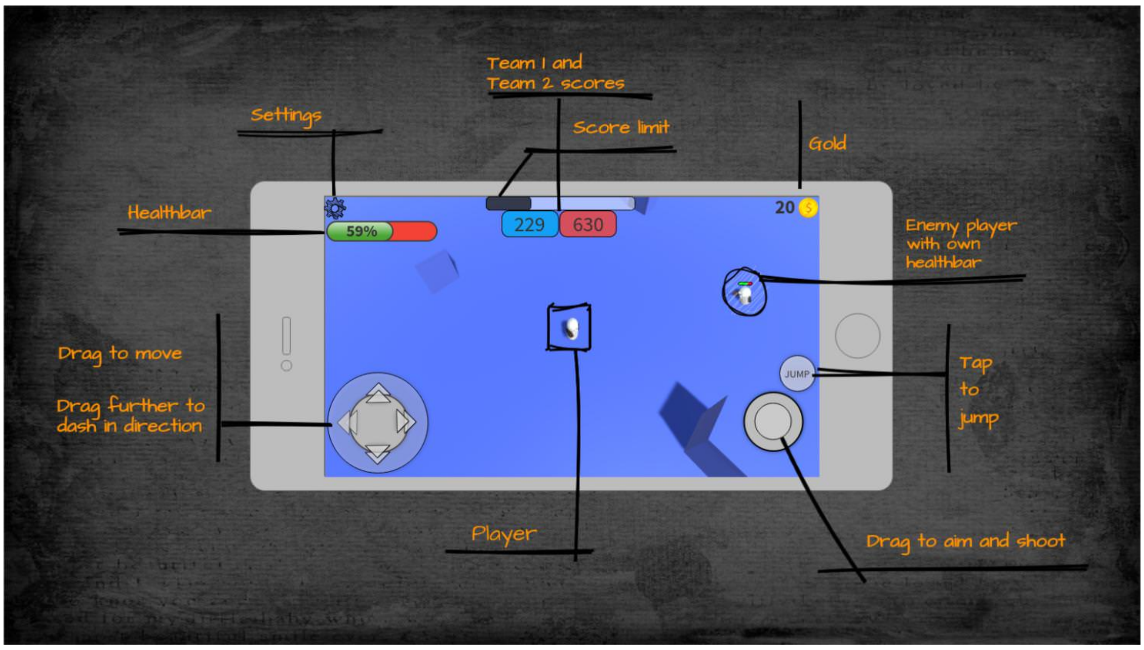


Figure 2.2: User interface for mobile devices

Other mobile control schemes

We have come to this control scheme after a number of iterations, and we have explored different ideas aimed at making the transition between PC and mobile UIs as seamless as possible. In the early stages we tried to use only one analog stick, which would control movement, while for shooting and aiming the user would tap a location. In this control scheme the user would tap in the general direction of an enemy to shoot, or press and hold for continuous fire. This control scheme would require a number of adjustments, such as the fact that all weapons should have an indication of where the projectile will land (for example with a long laser pointer) so that the user may have immediate feedback so it is easier to adjust their aim by pressing and holding.

This control scheme emulates more closely the way the mouse is used in the PC version, and it would make it arguably easier to move and shoot at the same time, but we ultimately abandoned the idea because it would not work very well on smaller devices. It would be quite nice on bigger screens like tablets or high-end phones, but on low-end, smaller ones it would get rather uncomfortable as the user would obstruct a lot more of the screen with his hand movements.

A related issue is that aiming this way would become less precise on a smaller screen.

While working on these alternative controls, we also realized that different control schemes put different constraints on the way we design the game. Using the control scheme described above would encourage the use of automatic or otherwise fast-firing weapons in order to maximize the number of hits, and weapons that follow or lock on enemies to minimize the amount of work the user has to do. When shooting is performed by tapping in the general direction of an enemy, it is more advantageous to saturate the area with projectiles in order to maximize the chances of hitting rather than using single-fire, high-damage weapons that leave the player vulnerable if they miss.

On the other hand, using the mouse on PC is a much more precise input method. Using this control scheme would lead to PC players having a distinct advantage in being able to aim quickly and precisely, while mobile players would be forced to get closer and rely on firepower to eliminate other players. We did not want such an imbalance in our game, so we decided to go back and re-design the mobile interface to give mobile players more precise control.

The UI pictured in 2.2 makes it so that the right analog stick is bound to a fixed position on the screen, which makes it more suitable for smaller devices.

With this layout we can better account for screen size, and aiming is more reliable since moving the virtual analog stick in a certain direction will always translate in the character aiming at a point in that direction. The user does not need to move his hands around (which may throw his aim off, especially if he is trying to move quickly). This also allows us to use a more varied arsenal of weapons as we have less restraints.

There is still a disparity in precision of input between mobile players and PC players, owned in part to the nature of the platforms they play on, and we need to account for that. For many years now, shooting games have tried to help players on platforms with less precise controls (such as game consoles like the Playstation or Xbox) by utilizing what is known as Aim Assist. As the name implies, Aim Assist is a collection of techniques that help the player aim and compensate for both latency and less precise controls.

In his talk at GDC 2013, Nick Weihs of Insomniac Games goes over many of the common Aim Assist techniques used in 3D first person shooter games [12].

On a high level, Aim Assist tweaks the camera controls to make it easier to perform small adjustment, or to ”snap” the player’s aim to an enemy when he aims close enough to the target. While a 3D, first-person environment makes this more complex to implement, a rudimentary aim-assist system could be implemented in the mobile version of the game to bridge the gap between the two platforms. As an example we could give characters an additional larger collider hitbox, which would be invisible to the player, but that would make it easier for the player to hit targets (especially moving ones). However, Aim Assist needs to be finely balanced to make the shooting experience enjoyable, but not make the effect too obvious for players. An excessive amount of aim assist may also lead to PC players being at a disadvantage, so it is something that needs to be taken into account.

2.3.3 User interface for PC

In 2.3 we can see our initial concept model for the PC version of the game, together with the default key bindings. The user interface is meant to be minimalistic to cover the least amount of screen space and to not clutter the screen with abundant data. The key-bindings are also intended for a fast-pace shooter environment. We have also considered no-mouse scenario, where players can use arrow keys in substitute of mouse (e.g. laptop users), although it may be less optimal.

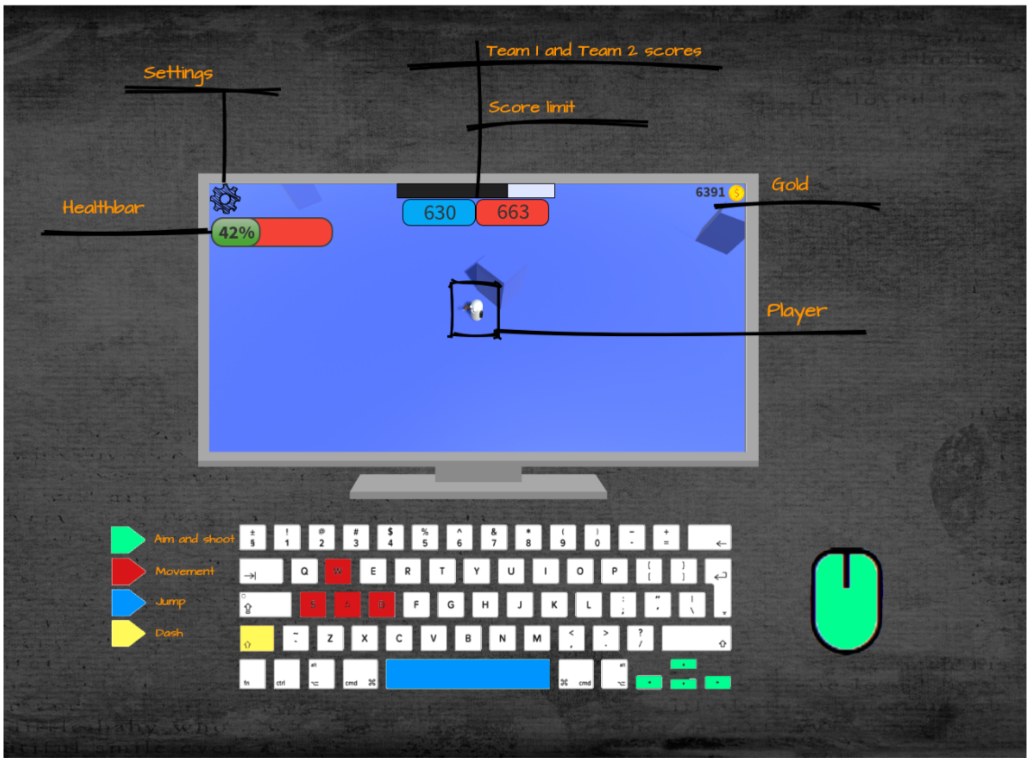


Figure 2.3: User interface for PC

Chapter 3

Networking

As it is more precisely described in chapter 2, the game we are developing has a number of players (around 10) playing on equal sized teams against each other. As such one of our priorities was to ensure a consistent and smooth gameplay on different platforms, keeping factors like network latency in mind.

In this chapter, we will describe various network architectures commonly used for games, and then go more in depth about Photon's networking architecture since it is a hybrid of different approaches.

Generally, there are two principal approaches for networking multiplayer games: *Client-Server (CS)* and *peer-to-peer (P2P) architectures.*

3.0.1 Client-Server architecture

In a Client-Server architecture the players (clients) all connect to a central, trusted server [10]. The server is responsible for relaying information between the clients, as well as arbitrating whenever the clients transmit conflicting data. The server's version of the game state is used to resolve conflicts between clients (for example if two clients disagree on where a third player is located).

There is a variant of the CS architecture called the *Mirror server architecture*, in which clients connect to various instances of the server located in various geographical locations. This is mostly done to accommodate large player numbers located in very distant parts of the world to reduce latency and group players from a similar geographic area. In this setup, upon launching the game the client will connect to the closest mirror. The Mirrors will act independently in normal gameplay, and synchronize with each other at a later point (*e.g* nightly).

There are various advantages and disadvantages that come with a client-server architecture: on the positives' side, it is easier to implement than a P2P architecture. Inter-client communication is much more streamlined, and its centralized nature makes programming easier. Furthermore server-based systems tend to be more reliable than P2P ones and less affected by client latency, especially in the Mirrored server configuration. This is because a major contributor to latency in such a setup is the physical distance from the server. In a multi-platform environment such as our game, where some clients may be on unreliable mobile networks, the use of a central server makes it easier to restore the game state if a client is disconnected.

As for the disadvantages of client-server systems, the main one is that they are more expensive. The server also has limited computing resources, which may pose a problem in games that deal with large player numbers (though this is not the case with our game)[10]. Finally, a central server is also a single point of failure, and in case of a crash it will render the game unplayable: this is generally unlikely, but it needs to be kept in mind if one wishes to ensure continued service.

3.0.2 Peer-to-peer architectures

A peer-to-peer architecture is not based on a central server, but instead on a network of clients that communicate with each other directly. Each client/peer manages a local instance of the game state that is updated as information comes in from other peers, and conflicting data is generally dealt with in two main ways: either by consensus (where data form different peers is compared to determine what happened) or by the use of a *Master Client*. The Master Client's game state is referred to, to determine whether an event has taken place in a similar way to what might happen in a CS architecture. The use of master client is arguably the more common approach between the two, as it makes conflict resolution easier.

A major advantage of P2P architectures is that they are extremely scalable, as resources available to the software scale with the number of clients. The downside is that P2P models are much more susceptible to latency: if a client cannot receive or send updates within a reasonable time, it will lag behind the network and disrupt the experience for that player. Another big advantage of P2P is that it eliminates the need for server infrastructure, making it cheaper and easier to develop multiplayer games on a low budget.

Within the master client approach, if the Master client does not have the resources to handle the game session (be it in terms of available bandwidth or processing power), it can ruin the experience for the other players.

Another problem that arises is that of host migration, where the game's host disconnects and a replacement must be found. Addressing this scenario requires more complex tracking of the game state in each client, as when a new host is found the game should resume from where it had stopped and loss of progress should be avoided. This translates in more frequent updates to each client's game state.

3.1 Networking Technologies & Frameworks Used in the Project

Below is an overview of the networking frameworks and technologies that we have used, or considered using, in our project:

3.1.1 Node.js & Socket.IO

Node.js is a popular server framework especially for web development. This is not in least due to its "one language to do it all" approach using JavaScript both on the server side and the client. Socket.IO is a high level JavaScript library that allows communication between a server and a client using Sockets. The Client/Server structure in Node.JS and Socket.IO can be seen in figure 5.1.

In combination, these two technologies make for an easy way to get a server up and running quickly. However - with respect to game development - it can become an issue when the complexity of the code increases. NodeJS/Socket.IO are not designed specifically for games, and therefore do not provide game specific tools for synchronizing large amounts of game objects. Since our aim for choosing high level technologies such as these are to make development faster and simpler, it might be better to choose a technology where this type of synchronization has been thought in during development of the server library.

3.1.2 Photon's networking architecture

Photon is a networking framework that has long been used in conjunction with Unity. It has been used in several AAA games and has been the main framework used by Codemasters since 2011 (Developing games for Nintendo, Sony and Microsoft consoles, as well as PCs. Notable names are the *Dirt* and *F1* franchises) [11]

Photon uses a cloud based server that has a free plan with 20 concurrent users, allowing for easy testing on different machines while developing. The cloud server runs a Master Server that handles game/room creation and allows for a lobby with access to games.

The Photon framework is mainly client server, but also offers facilities to introduce P2P elements in the networking. The Photon cloud acts as a central server to which players connect to play, but there is also a minor element of P2P with a Master Client approach.

The Photon cloud is composed of four main parts, a *Name server*, *Master servers*, *Game servers* and a number of *Hosted Rooms*.

The diagram in figure 3.1 from Photon's website ([2]) illustrates this division:

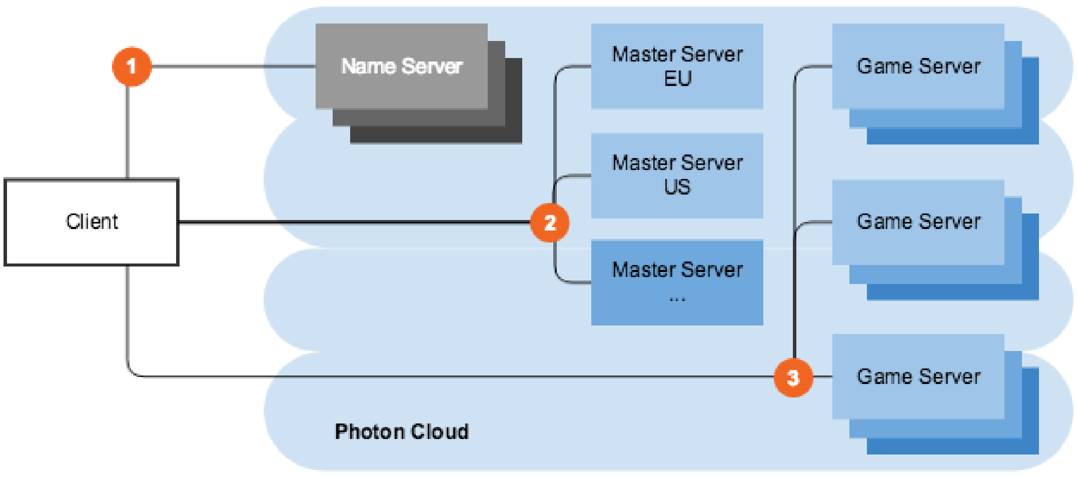


Figure 3.1: The Photon Cloud infrastructure

The practical use of the Photon Cloud will be more apparent in Chapter 6, but we will make a brief reference to our game in order to explain how this works. When a client connects to the game, it sends a request to the Name server, that determines the available regions and is then directed to an appropriate Master server. The Master server keeps track of all currently active game servers and how many players are in them. From there the client can either join an existing room in that Game server or create a new one. Game rooms are not represented in figure 3.1.

As we can see, the Master server setup is quite similar to the Mirrored server discussed above, with Master servers situated in different parts of the world to reduce latency. There is however one crucial difference between the two: Photon's Masters and Game servers do not host a persistent world in our case, and there is no need for the mirrors to update each other.

The P2P aspect of the Photon network comes in the form of a limited use of the Master Client approach in our project. Most data is sent and received from the server during normal gameplay, thanks to serializable objects and implementations of IPunObservable interface. All the game assets still reside on the client side. For this reason we select a Master Client to load and synchronize maps/scenes between clients. This ensures that there are no conflicts, and that all players are connected to the same scene.

3.1.3 Other worthy mentions

Unet (Unity's networking framework)

UNET is Unity's own networking stack for making multiplayer games. It provides a high-level API (HLAPI) that integrates tightly into Unity's Entity Component System, and provides a standardized way to network games. UNET also provides a low-level API (LLAPI), intended for more complex applications that require their own network code and infrastructure. Unet hasn't been updated in around three years however, and is in the process of being deprecated for a new Unity networking system developed in collaboration with *Multiplay*. This replacement has not been released yet.[4].

Unreal Engine

Unreal engine features an authoritative server framework that is used in popular games such as Fortnite, and is a competing game engine to Unity. It is similarly free to use for private individuals.

Chapter 4

The Entity Component System

This chapter will cover the history and essence of the Entity Component System, which is integral to Unity's work-flow

4.1 Overview

The Entity Component System (ECS) is a software development pattern used primarily in games that provides a large amount of flexibility. Many major publicly available game engines such as Unity, Unreal Engine and CryEngine have been built with ECS as their main pattern. [8] ECS seems to be very widely used in modern game programming.

There are lots of resources such as articles and talks with the purpose of explaining what ECS and its uses are. There are however not a lot of programming pattern books/peer reviewed articles that has ECS as a main concept. The reasoning for this could be that the description and wider use of ECS is relatively new. We have chosen to use this approach despite of that, as it seems to be a very efficient technique and as it is the chosen pattern for Unity.

4.1.1 The Problem

Traditionally, game development has been done using the object oriented concept, polymorphism, where classes extend and inherit from each other. Since games are often rather diverse and complex with many different entities sharing pieces of functionality, this can lead to some very deep and rigid hierarchies. In addition to this, games often change over time which makes it hard to plan out everything before starting development.

The idea and development of the Entity Component System started in the early 2000's as a solution to the above mentioned problem. An article from one of the developers, Adam Martin, describes his experience and the evolution of ECS [5], further referencing earlier work by Scott Bilas [1] .

The ECS pattern bears some resemblance to the Gang of Four [3] pattern named *Strategy*, in that both patterns take a part of an object's responsibility and delegates that to a different object. The main difference is that with the *Strategy* pattern, the delegated objects will usually be stateless (only containing algorithms/behavior) whereas in ECS the delegated objects (components) can keep track of the state and manage behavior at the same time.

4.1.2 Why ECS is good for Game Development

Suppose that the game includes a tree that behaves like normal trees, but this particular tree is also supposed to be able to shoot at players. In a polymorphism based system, the shooting functionality would have to come from some parent class that can shoot, or be duplicated in the tree class, even though all the other trees don't need this functionality. From a programming perspective this is considered bad practice, as it leads to either duplication or a very tight coupling.

There are in many different objects in games that will end up sharing some similar functionality such as physics calculations and rendering. These are very convenient to implement with ECS.

4.1.3 How ECS works

ECS works by a combination of three different concepts.

- **Component:** A component handles a single responsibility of an object (or an *Entity*) in the game. Typically this will resemble a C struct in that it has no functions and only stores data. One example of a typical component could be a position component. Some way to store an x, y and z coordinate (This component is called a *Transform* in Unity).
- **System:** A system is what gives the entities behaviour. It manipulate their data. One example could be a movement system which would manipulate the

coordinates stored in the *position* component, in order to move an object in the scene.

- **Entity:** The entities are the actual objects in the game. They can can be a characters, rocks or a shooting trees. Essentially, an Entity is no more than a list of components and their respective systems.

Abstracting component and system in Unity

In the Unity architecture, the separation between components and systems can be used in an abstract way. They can be implemented as one object without breaking any benefits of the pattern. Unity provides a component called a RigidBody for example. It contains the data needed for any kind of rigidbody physics (such as gravity, mass, friction etc.) as well as the algorithms and functions. This essentially makes it a combination of a component and a system.

All of our custom components are implemented in this way - without separating component and system - as this, in our use case, creates neatly structured code without having high coupling between any classes.

4.1.4 ECS in our project

To give the reader an example on how our game project uses ECS, we provided the ECS Diagram in figure 4.1 that demonstrates the entities, components and the relation between two of our main entities Player and Weapon. As is illustrated in the diagram, the main entities share similar components Rigidbody and Transform that provide properties and functionalies related to the physics and the position/rotation of the main entities.

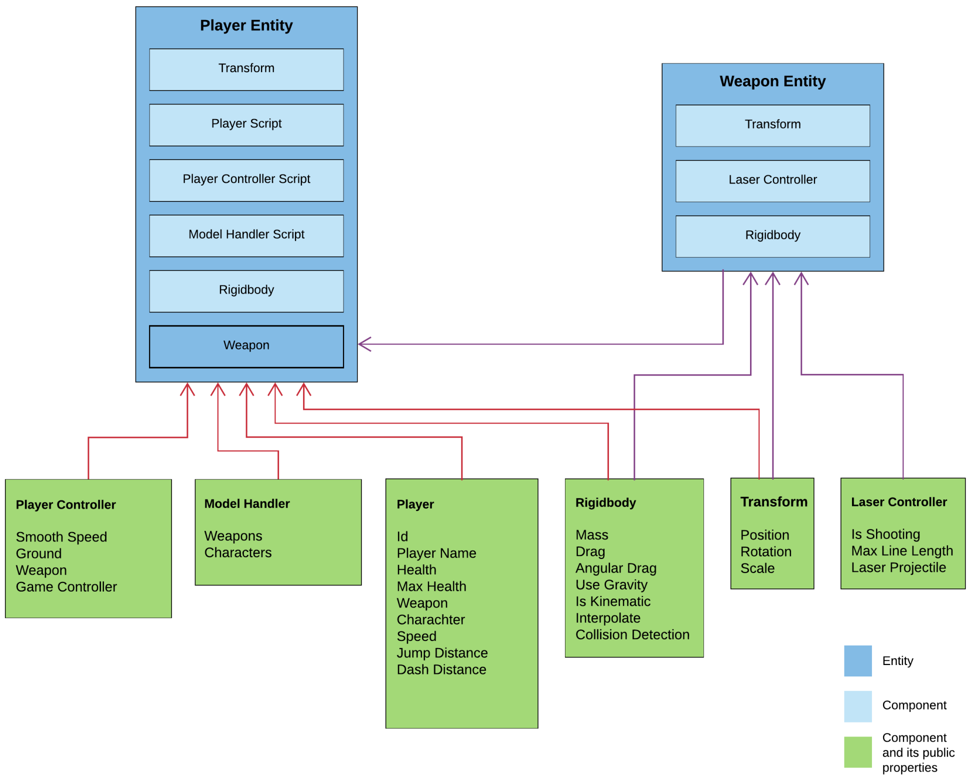


Figure 4.1: ECS diagram for the entities Player and Weapon

Chapter 5

Description of the Node.js Software Architecture

This chapter describes the atchitecture of the Node.js version of the game, by looking at the classes and processes that constitute the game.

5.1 Documentation of our Implementation

Figure (5.1) shows the main components of the Node.js implementation of the game.

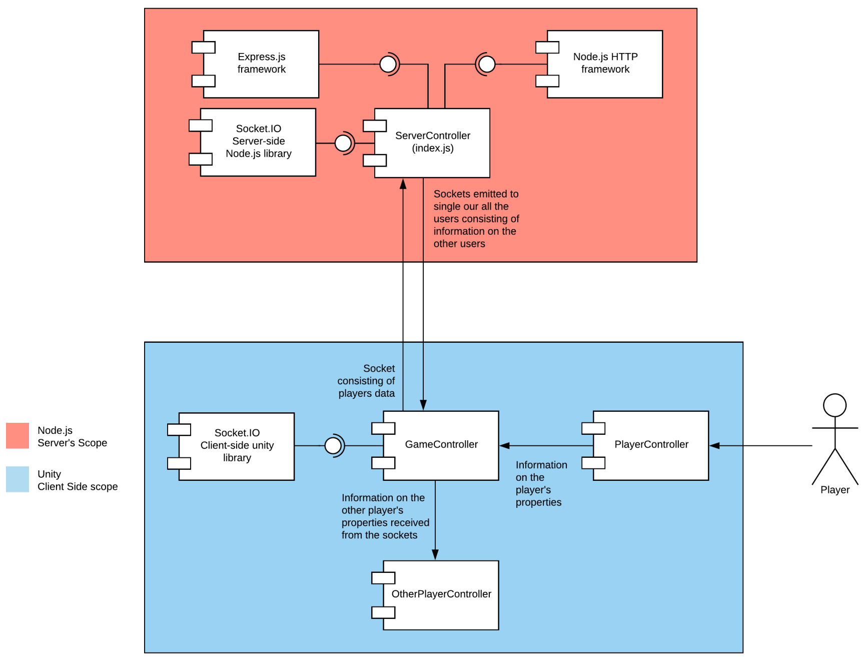

Express.js framework
Node.js HTTP framework
Socket.IO Server-side Node.js library
ServerController (index.js)
Sockets emitted to single our all the users consisting of information on the other users
Socket consisting of players data
Node.js Server's Scope
Unity Client Side scope
Socket.IO Client-side unity library
GameController
PlayerController
Player
Information on the player's properties
Information on the other player's properties received from the sockets
OtherPlayerController

Figure 5.1: A UML Component Diagram outlining the classes used in the software

As we can see the game works with a Client-Server network architecture, in which the server side handles player connection events and is responsible for propagating state changes to all other connected players.

The class diagram in figure 5.2 provides a more indepth view of the contents of these components. It only models the main elements of the software, and some classes have been left out. For example every weapon in the game extends the *Weapon* class and defines how that weapon should behave in-game. These sub-classes are not modeled.

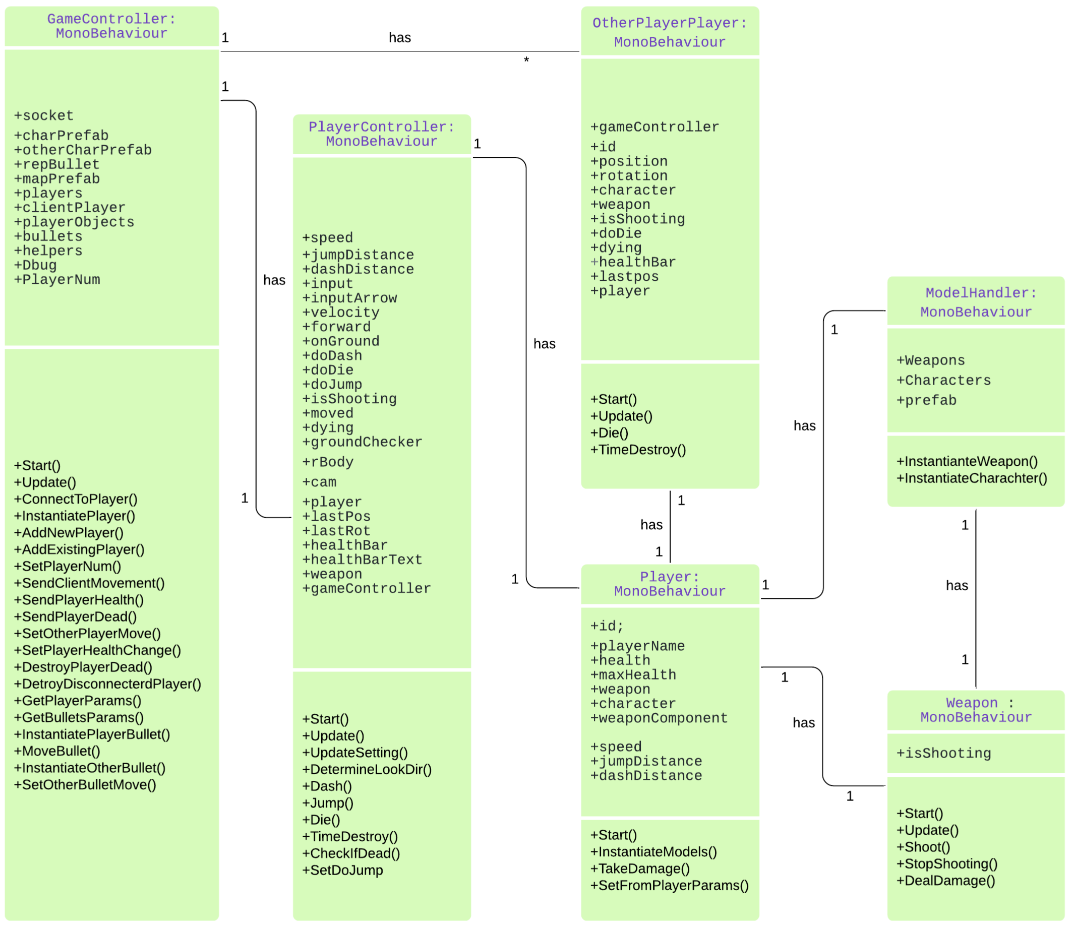


Figure 5.2: A UML Class diagram of the principal classes in the game

MonoBehaviour Class

The parent class MonoBehaviour[9] is the base class in the Unity library from which every Unity GameObject script derives. Below we're going to briefly describe some of the most used member functions from the MonoBehaviour class.

- Start(): This function is called once a script is started, before any of the Update methods are called.
- Update(): This function is called every frame.
- FixedUpdate(): This function has the same frequency as the physics system. It is called every fixed framerate frame and should be used instead of Update when dealing with Rigidbody [1].
- OnTriggerEnter(): This function is called when the GameObject collides with another GameObject. For instance this can be useful when we want to determine if a bullet collides with a player, so we can handle the damage dealt to that player.
- Destroy(): This function can be used to remove the game object. It's necessary when we want the player to be removed from the game upon death, to free the allocated memory.

Along with the functions that the parent class MonoBehaviour provides to our classes, it also provide some properties that are very useful. Bellow we describe some of them:

- GameObject: is the object that this entity is attached to. As an entity is always attached to a game object.
- Transform: The positional component attached to this GameObject. For instance by calling transform.position, we would get a 3d vector that refers to the position of the gameObject.

5.1.1 GameController

The *GameController* class functions as a high level entity that deals with creating an instance of the game map and of the various objects in it, while other controllers handle logic tied to specific game elements. the Game Controller also sends game state/player data to the server.

[1]A component added to an object which will put its motion under the control of Unity's physics engine

As we can see, the Game Controller has a wealth of functions, mainly dealing with instantiating other objects and handling communication with the server. In addition to sending player events to the server, the Game Controller also keeps track of what players are connected and their status (position, health etc.). This information is then relayed to the OtherPlayer Controller so that it may be displayed locally.

The bullet-related properties and functions inside the game controller are tied to a weapon in the game that we call the 'Reptile Gun'. They are in the Game controller since it need to be synchronized over the network. The Reptile gun will be discussed more thoroughly in chapter 7.1.

5.1.2 Player Controllers

The *PlayerController* class on the other hand, is responsible for handling player physics, movement and actions.

The *OtherPlayerController* its similar to the Player Controller, but it handles the movement and action data only from other players on the network so that they can be displayed. No input is processed here.

The `PlayerController` class, as we mentioned before, handles player movement and action: a lot of the player object proprieties are tied to the physics engine, and applied to the model via the use of a `rBody` object; attaching a `RigidBody` to a model makes it subject to physical forces (as opposed to static objects).

This class also handles the death and de-spawning of the player. In the event that the player's health reaches 0 or goes below it, the `doDie` flag is set to True. this value is checked every frame at run-time, and it triggers the `Die()` function if the boolean value is set to True.

The `Die()` function resets player parameters and de-spawns the player's character model from the map. There is another value in the class that concerns death, the `dying` variable. In order to explain its function, one needs to understand the difference between the `Update()` and `FixedUpdate()` methods: both are standard Unity methods that are called at every new frame: `Update()` is called at the very beginning of the frame, while `FixedUpdate()` is called later on; FixedUpdate is commonly used to compute physics, while Update is used for other functions that take priority.

The `dying` variable simply resets the Player's parameters before it de-spawns, this makes it so that the player character appears to fall over upon death, instead of immediately vanishing when the `Die()` function is called.

The *OtherPlayerController* class performs a similar function to the Player Controller described above, but while the Player Controller handles movement and interaction for the local player, the Otherplayer controller handles those same functions for players connected through the network.

As such, it is considerably shorter and mainly deals with receiving and sending data from/to the server, so that the actions of connected players may be displayed on the local machine.

The *Player* class is attached to the Player component, and holds key parameters about the player such as current health or the player's ID. This class has a lot less functionality than the Player Controller, and the values within it change from player to player. Its main task is to instantiate the player with the correct character model, and change the player's health when he/she takes damage.

5.1.3 Utility Classes

The *ModelHandler* handles loading of the different prefabs necessary for instantiating the 3D models used in the game. This is done by loading prefabs and combining them together to create a character in the game. A prefab in Unity is a combination of scripts and models that constitute some game entity. Game entities can then be instantiated using the description stored in the prefab.

5.2 Game Processes

This section describes some of the essential processes that make up the game.

5.2.1 Connection & Instantiation

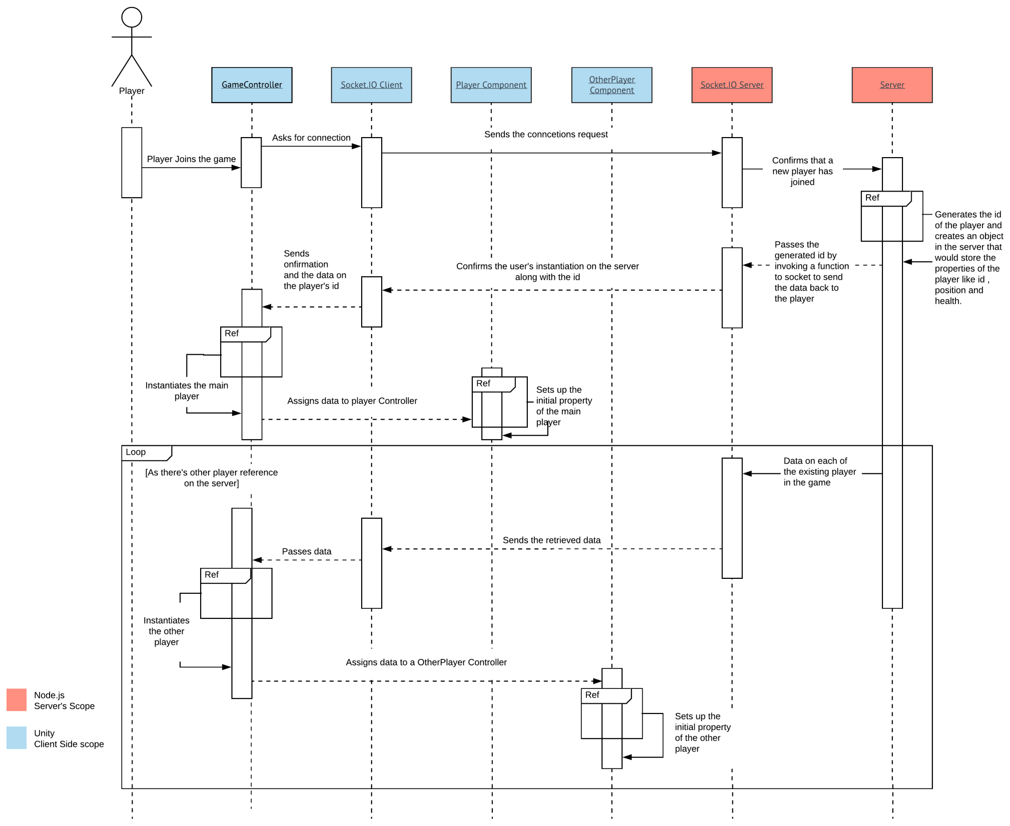

Player
GameController
Socket.IO Client
Player Component
OtherPlayer Component
Socket.IO Server
Server
Player Joins the game
Asks for connection
Sends the conncetions request
Confirms that a new player has joined
Ref
Generates the id of the player and creates an object in the server that would store the properties of the player like id , position and health.
Passes the generated id by invoking a function to socket to send the data back to the player
Confirms the user's instantiation on the server along with the id
Sends onfirmation and the data on the player's id
Ref
Instantiates the main player
Assigns data to player Controller
Ref
Sets up the initial property of the main player
Loop
[As there's other player reference on the server]
Data on each of the existing player in the game
Sends the retrieved data
Passes data
Ref
Instantiates the other player
Assigns data to a OtherPlayer Controller
Ref
Sets up the initial property of the other player
Node.js Server's Scope
Unity Client Side scope

Figure 5.3: A UML System Sequence Diagram showing how player connection and game initiation is handled

.

The diagram in figure 5.3 shows the player connection and initiation process, for both the local player as well as the remote player.

The first half of the diagram models how a local player joins the game.

When a player joins the game the Game Controller sends a request over the socket to the server. The server then adds the player to the list of currently connected users and generates a unique ID for him/her. The server will then keep track of the player's data with this ID. Alongside the ID, the server initializes the Player's position, model and health. This data is then relayed back to the client. The client's then binds the incoming information to a Player Controller, all subsequent player actions are handled by the Player controller.

The second half of the diagram, pictured as a loop, models how remote players are handled. Whenever other players join the game, after the process described above has taken place, the server broadcasts the player's data to the other connected players. This data (containing player position, rotation, and health status) is received by the connected clients and assigned to the OtherPlayer Controller, that is then used to display the actions of those players on the screen.

5.2.2 Movement

In figure 5.4 we can see how movement updates work between local player and other remote players.

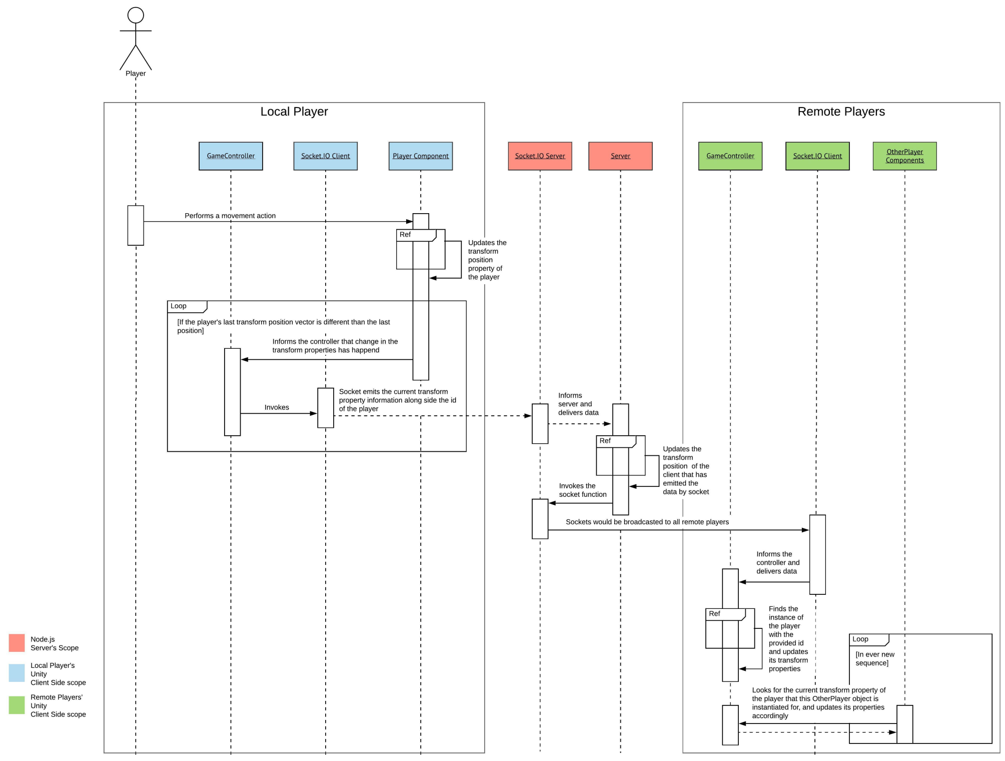

Player
Local Player
Remote Players
GameController
Socket.IO Client
Player Component
Socket.IO Server
Server
GameController
Socket.IO Client
OtherPlayer Components
Performs a movement action
Ref
Updates the transform position property of the player
Loop
[If the player's last transform position vector is different than the last position]
Informs the controller that change in the transform properties has happend
Socket emits the current transform property information along side the id of the player
Invokes
Informs server and delivers data
Ref
Updates the transform position of the client that has emitted the data by socket
Invokes the socket function
Sockets would be broadcasted to all remote players
Informs the controller and delivers data
Ref
Finds the instance of the player with the provided id and updates its transform properties
Loop
[In ever new sequence]
Looks for the current transform property of the player that this OtherPlayer object is instantiated for, and updates its properties accordingly
Node.js Server's Scope
Local Player's Unity Client Side scope
Remote Players' Unity Client Side scope

Figure 5.4: A UML System Sequence Diagram showing how movement is handled

.

When the local player performs any movement action, the *PlayerComponent* receives the input and updates the transform of the local player and if the position is different than previous one, then it informs the *GameController* that there is a change in properties. The *GameController* then invokes the socket.io client which then informs the server of the change in the local Transform.

The server updates the transform position data of the local player, then invokes the socket function and the data is then transmitted over the sockets to all remote players simultaneously. The Controller then finds the instance of player with provided id and updates the position properties.

The Game controller recieves data from the server and the OtherPlayer controller then uses this data to update the local instances.

5.2.3 Shooting & collisions

When the player presses the fire button, the boolean value `isShooting` (referenced in figure 5.2) is set to True. When this variable is set, the Player Controller invokes the weapon controller for the equipped weapon, which in turn determines the behaviour of the weapon and the projectile. The specific weapon controller then spawns the projectile into the game. As long as the projectile is traveling, it reports its position to the game controller so that the position may be broadcast to other players through the server. If the projectile collides with an object in the scene, it simply de-spawns. If, on the other hand, it collides with another player, it invokes the OtherPlayer Controller for the player with that same ID. The damage is then subtracted from the local instance of the OtherPlayer controller, and the new health value is then broadcast to the rest of the connected players. For the clients that were unaffected by the damage, they will simply receive this information on their local OtherPlayer controller and display it. Conversely the player that was damaged updates its own health value in the Player controller.

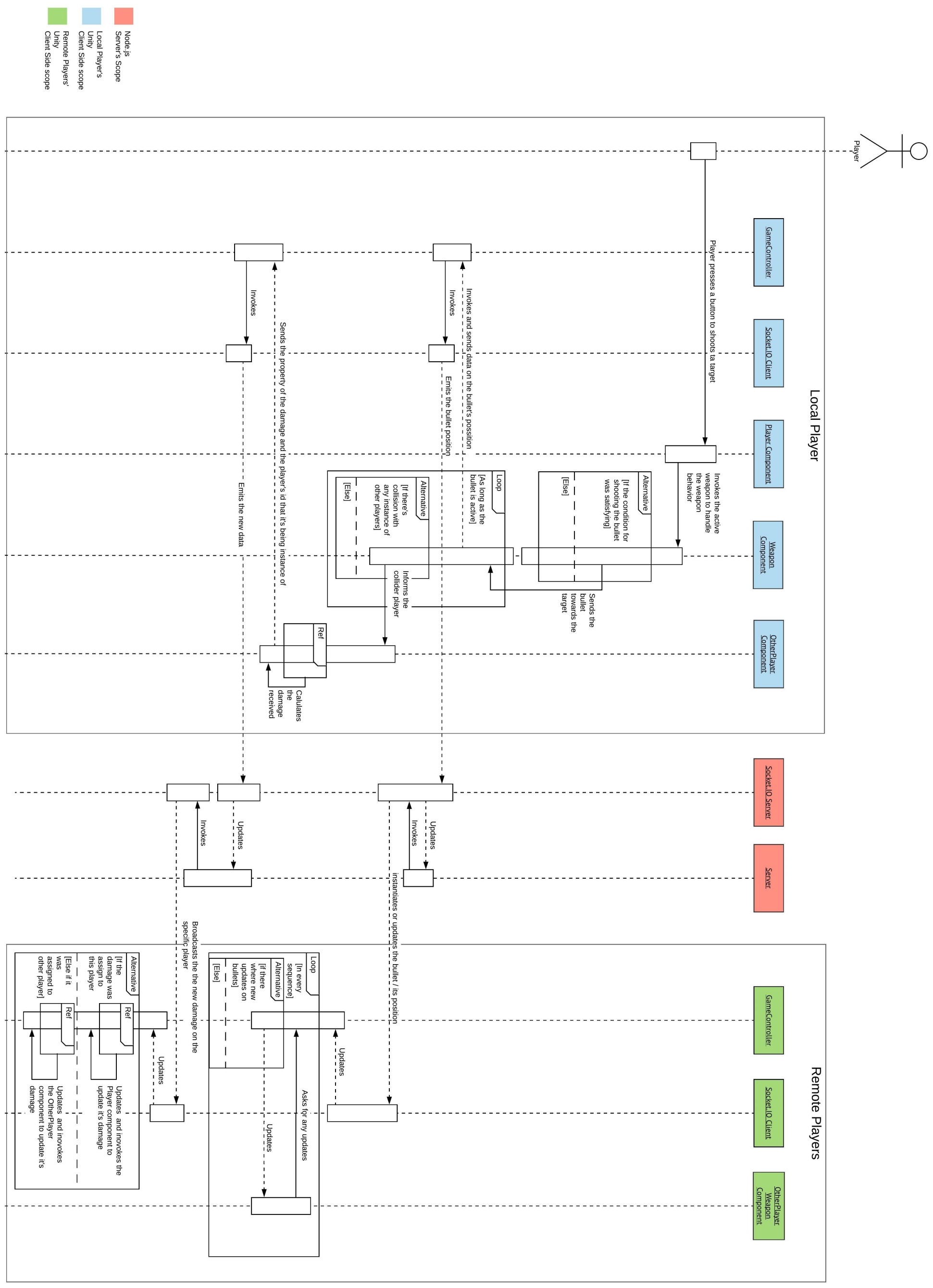

Node js Server's Scope
Local Player's Unity Client Side scope
Remote Players' Unity Client Side scope
Player
Local Player
GameController
Socket.IO Client
Player Component
Weapon Component
OtherPlayer Component
Player presses a button to shoots ta target
Invokes the active weapon to handle the weapon behavior
Alternative
[If the condition for shooting the bullet was satisfying]
[Else]
Sends the bullet towards the target
Loop
[As long as the bullet is active]
Alternative
[If there's collision with any instance of other players]
[Else]
Informs the collider player
Invokes and sends data on the bullet's position
Invokes
Emits the bullet position
Ref
Calculates the damage received
Sends the property of the damage and the player's id that it's being instance of
Invokes
Emits the new data
Socket.IO Server
Server
Updates
Invokes
Instantiates or updates the bullet / its position
Broadcasts the new damage on the specific player
Remote Players
GameController
Socket.IO Client
OtherPlayer Weapon Component
Loop
[In every sequence]
Alternative
[If there where new updates on bullets]
[Else]
Asks for any updates
Updates
Alternative
[If the damage was assign on this player
Ref
Updates and invokes the Player component to update it's damage
[Else if it was assigned to other player]
Ref
Updates and invokes the OtherPlayer component to update it's damage

Figure 5.5: A UML System Sequence Diagram showing how shooting and collision work

.

5.3 Implementation Details

In this section we describe some of the main areas of our code and technical implementation of this version.

5.3.1 Server Side node.js and Socket.io

We setup the nodejs.js and load the socket.io library by the following code.

```
  var express = require("express");
  var app = express();

  var server = require("http").createServer(app);
  var io = require("socket.io").listen(server);
  // We set the port that every client should connect to, to 3000.
  app.set("port", process.env.PORT || 3000);
```

Then the following variables are the main properties that are being used in order to handle the networking between the players.

```
  var clients = [];
  var id = 0;
  var OnlinePlayerNum = 0;
}
```

The following code handles each connection from each client that connects to the server.

Server Side Client Connection Handling

```
  io.on("connection", function(socket) {
  // we have omitted the network code that is handled here
  });
```

To provide examples on how the network handling code inside the upon function looks like, the following code snippets are how we handle the player instantiations and movements.

Data received from the connected client is sent to the rest of the clients connected to the server.

Server Side Player Instantiation

The following part of the code refers to the (Connection & Instantiation) diagram 5.3.

```
socket.on("USER_INITIATED", userData => {
  console.log(userData.weapon);
  clients.forEach(player => {
    socket.emit("GET_EXISTING_PLAYER", player);
  });
  //Collecting the instantiated user's data in our Clients array.
  clients.push(userData);
  socket.broadcast.emit("A_USER_INITIATED", userData);
});
```

In the function `socket.broadcast.emit("A_USER_INITIATED", userData)`, the socket emits the information to each of the players, except the player that has sent the data. The first argument is a title we give to the socket so that it can be caught on the receiver client side using `socket.on("title name")` function. And the second argument is the parameter we want to send to the receiver clients, to instantiate and update their remote players' properties.

Server Side Player Movement Synchronization

The following part of the code refers to the (movement) diagram 5.4.

```
socket.on("CLIENT_MOVE", function(movementData) {
  for (var i = 0; i < clients.length; i++)
    if (clients[i].id == movementData.id) {
      clients[i].position = movementData.position;
      clients[i].rotation = movementData.rotation;
      socket.broadcast.emit("OTHER_PLAYER_MOVED", movementData);
      return;
    }
});
```

In the function `socket.broadcast.emit("OTHER_PLAYER_MOVED", movementData)`, operates the same way as the other function `socket.broadcast.emit("A_USER_INITIATED", userData)` in the previous code snippet, but with different title and argument.

5.3.2 Client Side Implementation in Unity

The following code snippets include the main properties needed in the client side according to the game networking handling in the GameController class. A complete list of functions that handle the data received from the server through sockets as well as other part of the game logic, is available in the Class Diagram 5.2.

```
private Player clientPlayer;
private List<PlayerParams> players;
public List<GameObject> playerObjects;
public List<BulletParams> bullets;
```

The following code snippet is how we instantiate Socket.io client side in the GameController class.

```
private SocketIOComponent socket;
void Start()
    {
    socket = GetComponent<SocketIOComponent>();
    StartCoroutine(ConnectToServer());

    //Rest of the code
    }
    #region Connection
    IEnumerator ConnectToServer()
    {
        yield return new WaitForSeconds(0.5f);
        socket.Emit("USER_CONNECT");
    }
```

In the code we use the 0.5f seconds delay before connecting to the server, to ensure that the socket component has finished loading.

As we discussed how the server handles the user instantiation and movement in the prior section, we continue by describing how the client then handles the operation from its side.

Client Side Player Instantiation

The following code snippet is how a user instantiation is handled when the server emits information that a new player has joined.

```
private SocketIOComponent socket;
private GameObject otherCharPrefab;

void Start()
{
    socket.On("A_USER_INITIATED", AddNewPlayer);
    otherCharPrefab =
        (GameObject)Resources.Load("Prefabs/PlayerCharacters/OtherPlayer",
        typeof(GameObject));

    //Rest of the code
}
private void AddNewPlayer(SocketIOEvent evt)
{
    PlayerParams pp = PlayerParams.CreateFromJSON(evt.data.ToString());
    var newCharacter = Instantiate(otherCharPrefab, pp.getPosition(),
        Quaternion.Euler(0, -90, 0));
    newCharacter.GetComponent<Player>().SetFromPlayerParams(pp);
    players.Add(pp);
    playerObjects.Add(newCharacter);
}
```

The object `pp` of type PlayerParams, is the parameter we get from the server through the socket and the method `Instantiate()` is a method inherited from the MonoBehaviour class (that is discussed earlier in this report at section 5.1).

This is used to instantiate a GameObject in the game controller using the parameter pp. The parameters it takes in our case are: An existing object that we want to make a copy of, the initial position (Vector3) and rotation (Quaternion) of the object
.

Client Side Player Movement

In the following code snippets we describe how remote user movement is being handled when the client recieves data on a remote player's updated position. (The system sequence diagram in figure 5.4 is showing the interactions that are being handled in this part).

First, The target client that has done a movement would use the `socket.Emit()` in order to send its new position data to the server, then the server broadcasts the new data to all the other clients. The method `SendClientMovement()` is called when a movement is performed by the client. It requires the user's id, its position and rotation degree.

```
public void SendClientMovement(int id, Vector3 pos, Quaternion rot)
{
    var obj = new MovementObjJSON(id, pos, rot);
    socket.Emit("CLIENT_MOVE", JSONObject.Create(JsonUtility.ToJson(obj)));
}
```

As the server gets the data emitted from the moving client, it would broadcast the same data to all the other clients in order to update the position of the instance player of the moving remote player (discussed earlier in section 5.3.1).

The following code snippet is how the other clients get the data from server and handle the instance of remote player position updates.

```
 1 void Start()
 2 {
 3     socket.On("OTHER_PLAYER_MOVED", SetOtherPlayerMove);
 4 }
 5
 6 private void SetOtherPlayerMove(SocketIOEvent evt)
 7 {
 8     var move = JsonUtility.FromJson<MovementObjJSON>(evt.data.ToString());
 9     foreach (var p in players)
10         if (p.id == move.id)
11         {
12             p.position = move.position;
13             p.rotation = move.rotation;
14         }
15 }
```

In the above code snippet, in line 8, we re-convert the data that has been delivered in JSON format from the server, to a type of MovementObjJSON which is a created by C# and store that in the `move` variable. Then in line 9 we loop through the `players` list (that holds objects with information of the properties of the remote players). As the program finds the matching player inside the loop by its id, it updates the matching player instance with the transform data contained in the `move` object we recieved from the server.

Now that the object in the `players` list that holds the property of the client is updated. The instance of the remote player would map its position accordingly using the following code in the `OtherPlayerController`.

```
void Update()
{
    var pp = gameController.GetPlayerParams(id);
    if (pp == null) return;
    if (pp.position != lastpos)
    {
        transform.position = pp.position;
        transform.rotation = pp.rotation;
        lastpos = transform.position;
    }
}
```

Chapter 6

Photon Implementation

6.1 Introduction

Photon provides us with many useful features that helped us in our game-making process, including:

- A Central Server in the form of the Photon cloud described in section 3.1.2, which allowed for easier deployment and testing.
- A unified method of inter-client communication through the *PhotonNetworking* and *PhotonView* APIs. This was rather useful to us, as we found that every time we tried to add new functionality in the NodeJS implementation we had to amend our networking code, and make it more and more complex. If we found possible improvements to it we had to go back and change all these additional features.

  These APIs gave us the freedom to experiment without the worry of having to go back and change our networking code.
- A structured way to handle connection to the aforementioned Photon Cloud: this wasn't an immediate concern, but our NodeJS implementation was hard to scale.

The main concern that moved us in this direction was twofold: We recognized that our game was getting more and more complex (needlessly so), and we believed we were getting held back by having to invent a lot of logic that already exists.

Additionally, there were other concerns, such as the realization that, were we to port the game to mobile, we would have to ship the NodeJS binaries along with the game in order to make the server work.

While the process of making the back-end with NodeJS was a good learning experience, allowing us think about many lower-level optimization and such, we felt that it was detrimental to the development of the game itself.

6.2 Implementation Overview

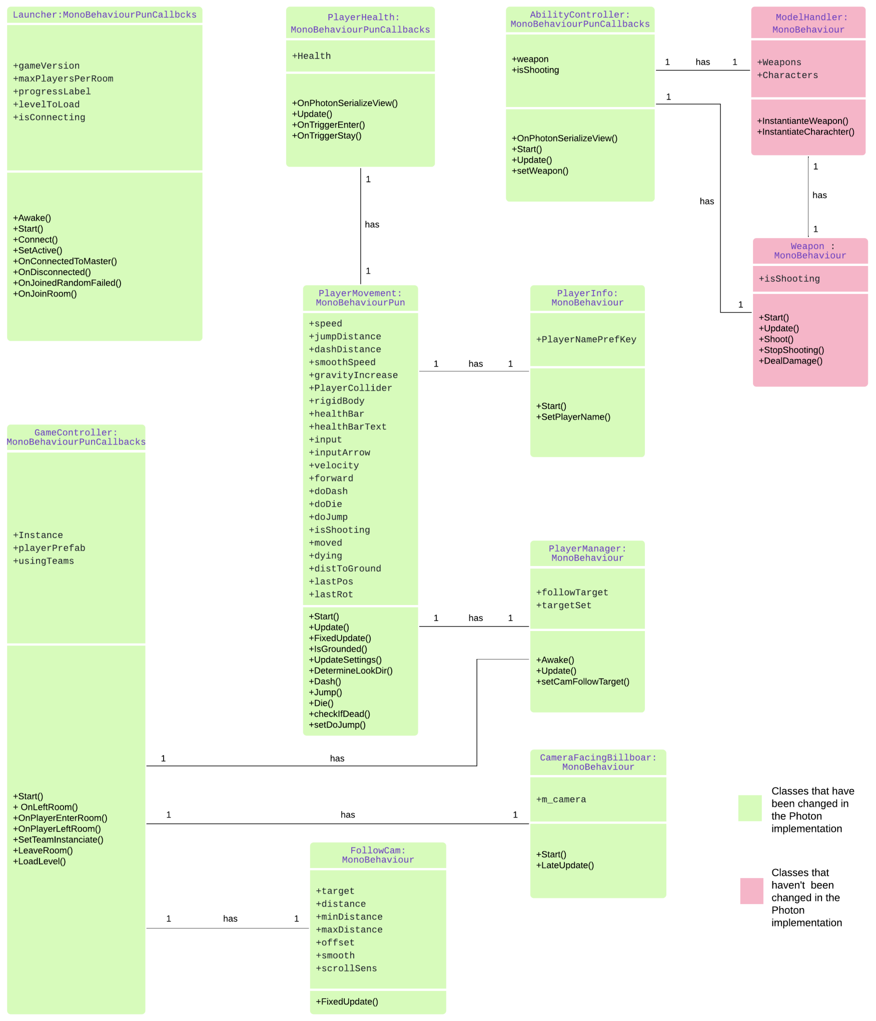


Figure 6.1: A UML Class diagram of the principal classes in the game after the Photon implementation

As we can see, in figure 6.1, the architecture of the software has changed substantially when compared to the Node.JS implementation: communication between clients has been streamlined thanks to the *Photon serializable view*, and some classes have been added to accommodate new functionality.

First up is the *Launcher* class, which is one of the new additions: this class incorporates some functionality that was previously present in the Game Controller, and mainly deals with connection and instantiation. The purpose of this class is to connect players to the game, as well as load new scenes when necessary. This class also manages the lobby - starting games and eventually other menu related features.

- The *GameController* class mainly assigns players to teams and selects a master client for the room.
- The *PlayerHealth* class keeps track of the local player's health, sending and receiving updates as necessary via PhototnSerializableView.
- The *FollowCam* class contains the main camera controls and parameters, which is bound to the local player via the *PlayerManager* class. This camera follows the player around during gameplay.
- movement and physics-based logic, while the *AbilityController* handles shooting and weapon instantiation.
- The *PlayerMovement* class, as the name implies, manages the Player
- The *ModelHandler* loads models from prefabs and has not been changed from the Node.js implementation.
- The *Weapon* class remain unchanged from the previous implementation.

As we can see, this Photon implementation moves a lot of functionality that was previously in the PlayerController and Player classes into more discrete classes for a better separation of concerns. Thanks to Photon, there is no longer a need for a separate OtherPlayer controller as we have made it dynamic using the *isLocal* boolean variable provided by the photon library. The communication between clients has also been streamlined which will be described here and in the discussion 8 chapter.

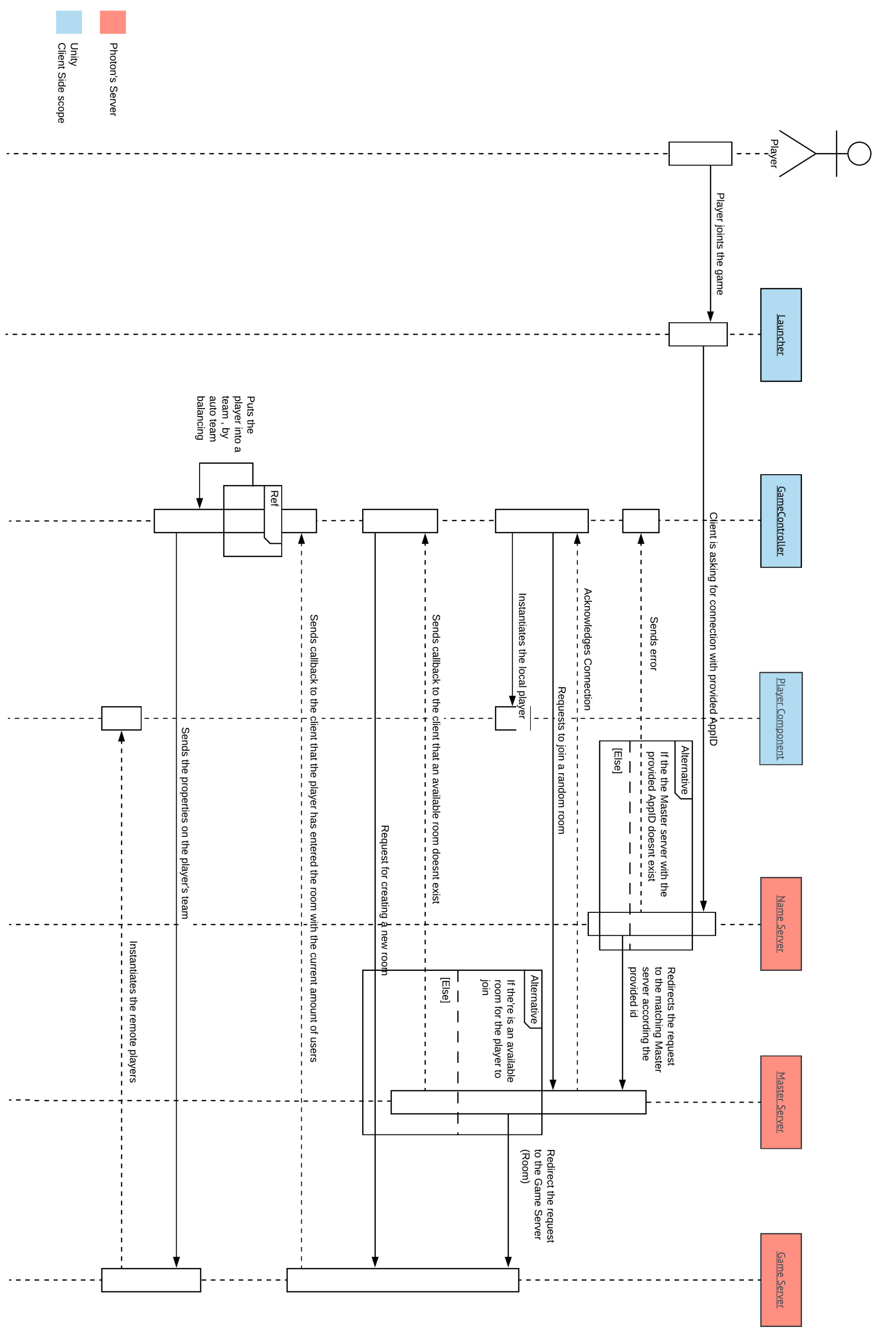

Photon's Server
Unity
Client Side scope
Player
Launcher
GameController
Player Component
Name Server
Master Server
Game Server
Player joints the game
Client is asking for connection with provided AppID
Alternative
If the the Master server with the provided AppID doesnt exist
[Else]
Sends error
Redirects the request to the matching Master server according the provided id
Acknowledges Connection
Requests to join a random room
Instantiates the local player
Alternative
If there're is an available room for the player to join
[Else]
Redirect the request to the Game Server (Room)
Sends callback to the client that an available room doesnt exist
Request for creating a new room
Sends callback to the client that the player has entered the room with the current amount of users
Ref
Puts the player into a team , by auto team balancing
Sends the properties on the player's team
Instantiates the remote players

Figure 6.2: A System Sequence Diagram depicting the Player connection in the Photon implementation

6.3 Game processes

6.3.1 Connection and Instantiation

In figure 6.2 we can see how players create and/or connect to a game within the Photon implementation.

As the player runs the game, he loads into the lobby scene. Starting the game calls the *Launcher* class, which is responsible for connecting , creating a room and serves as a menu for the player. The Launcher class attempts to establish a connection to the *Photon Name Server* with a given AppID, which identifies the game application. If the AppID exists within the name server, it routes the client to a Master server that contains information for all rooms for that particular game. The Master server acknowledges the client's connection, which then requests to join a random room. If an appropriate room is found (i.e. if there is a room with at least one free slot for the player to join), the player joins the room and thus can participate in an already existing session of the game. If, on the other hand, a suitable room cannot be found, the client creates a new room.

Regardless of whether the server was found or created, the server acknowledges that the player has entered the server and returns the number of connected clients. The client then proceeds to assign the local player to a team, by counting the number of players present on the server, and assigning each to either teams sequentially.

6.3.2 Movement

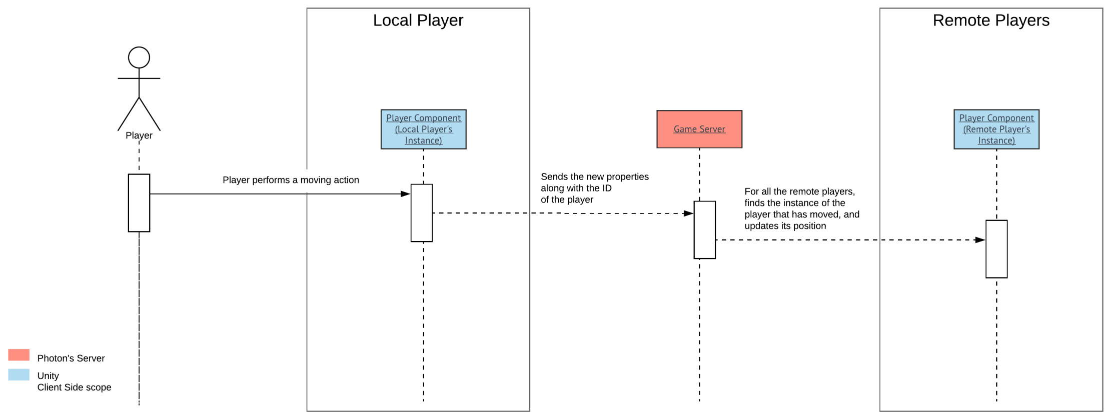
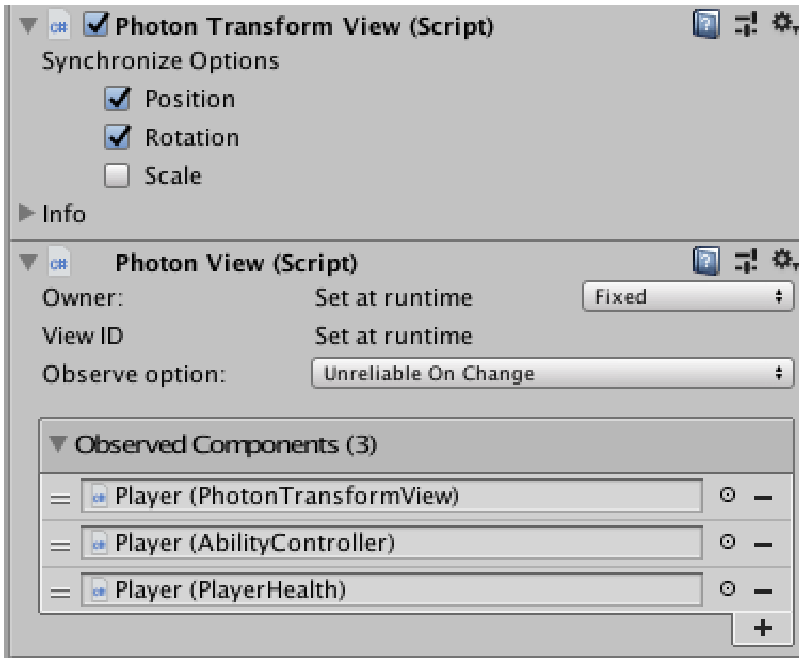

Local Player
Remote Players
Player
Player Component (Local Player's Instance)
Game Server
Player Component (Remote Player's Instance)
Player performs a moving action
Sends the new properties along with the ID of the player
For all the remote players, finds the instance of the player that has moved, and updates its position
Photon's Server
Unity Client Side scope

Figure 6.3: A UML System Sequence Diagram showing how movement works in the photon implementation

Because of the differences outlined in the previous sections, movement is greatly simplified in the Photon implementation. Movement is handled by the *PlayerMovement* class, which handles input from the local player and is also handles sending and receiving the position of other clients. This class does not implement the *OnPhotonSerializeView* method like the PlayerHealth or AbilityController classes. This is because the *PlayerMovement* script directly modifies the local players transform, and thus we can simply use the *PhotonView* class to synchronize the x, y and z coordinates from the transform as can be seen done in 6.4.


Photon Transform View (Script)

Synchronize Options

- [✓] Position
- [✓] Rotation
- [ ] Scale

Info

Photon View (Script)

Owner:	Set at runtime	Fixed
View ID	Set at runtime	
Observe option:	Unreliable On Change	

Observed Components (3)

- Player (PhotonTransformView)
- Player (AbilityController)
- Player (PlayerHealth)

Figure 6.4: A screenshot from the Unity editor

Once a player moves, the *PlayerMovement* class sends the updated Transform to the server alongside the ID of the local player, which then sends the transform to other clients. This information is received by each client's *PlayerManager* class, that determines what player has moved and updates the transform in the local instance.

6.3.3 Shooting and Collision

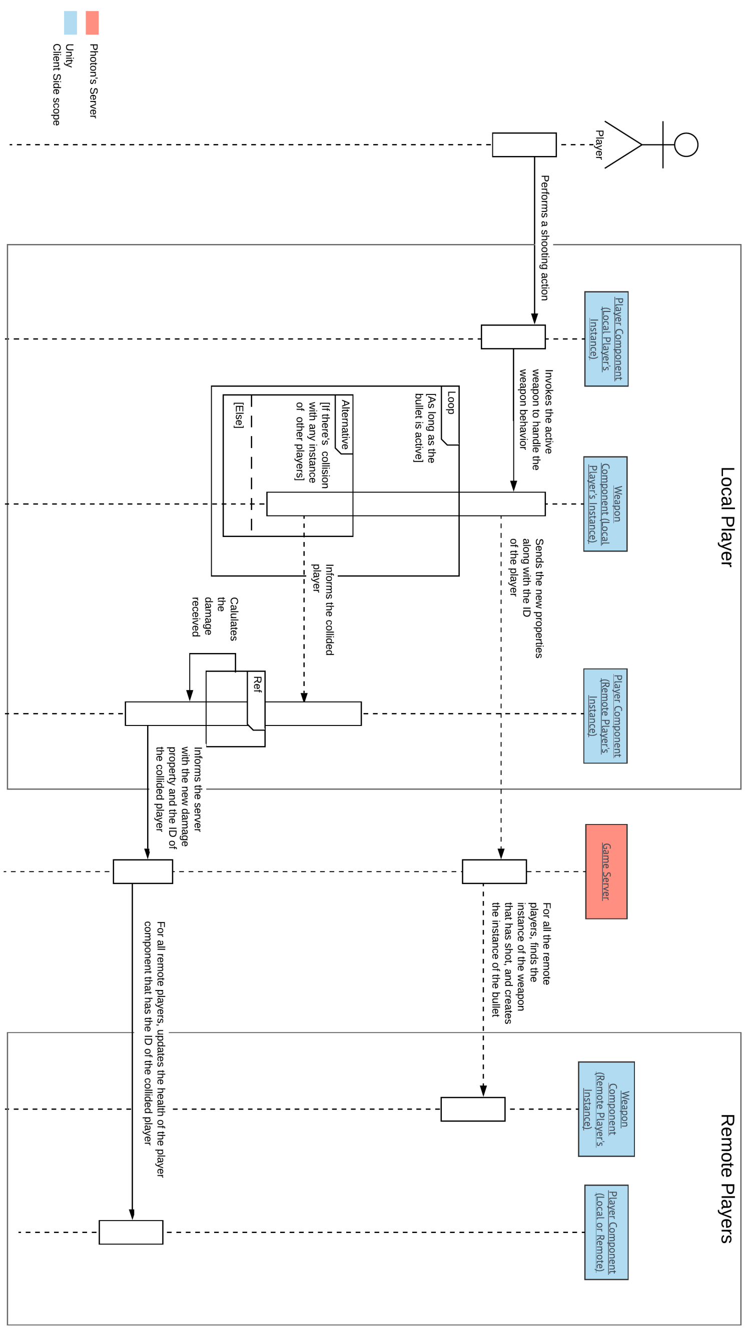

🟥 Photon's Server
🟦 Unity Client Side scope
Player
Performs a shooting action
Local Player
Player Component (Local Player's Instance)
Invokes the active weapon to handle the weapon behavior
Weapon Component (Local Player's Instance)
Loop
[As long as the bullet is active]
Alternative
[If there's collision with any instance of other players]
[Else]
Sends the new properties along with the ID of the player
Informs the collided player
Calulates the damage received
Ref
Player Component (Remote Player's Instance)
Informs the server with the new damage property and the ID of the collided player
Game Server
For all the remote players, finds the instance of the weapon that has shot, and creates the instance of the bullet
For all remote players, updates the health of the player component that has the ID of the collided player
Remote Players
Weapon Component (Remote Player's Instance)
Player Component (Local or Remote)

Figure 6.5: A System Sequence Diagram depicting the Player connection in the Photon implementation

The process of shooting is broadly similar to how it functioned in the NodeJS implementation, but with some differences.

When a player presses the fire button, the *AbilityController* sets the `isShooting` variable, sends it to the server and invokes the weapon controller of the weapon that is currently equipped. the weapon controller then spawns a projectile that sends its position to the server while it is active. When it collides with another player, it triggers a collision and the *PlayerHealth* class of the remote player subtracts a fixed value from the player's health. The updated health value is then sent to the server through a Photon stream on the next update.

6.4 Implementation Details

This sections goes into detail about the most relevant parts of the Photon implementation.

6.4.1 Initialization

The Photon library uses a settings file, *PhotonServerSettings.asset*, that contains necessary server settings such as the AppId (The developer key for the photon cloud). Once this AppId has been added to the settings file the application is ready to connect and use the online photon cloud server.

6.4.2 Connection & Joining Rooms

This section details the core implementations of the Photon solution. Many of these are the same features as in the Node.js version, but with the addition of a few utilities such as teams and a 3d lobby menu that the player can walk around in.

Connection

As Photon is a fairly high-level game networking library some common functionalities, such as connection, has been abstracted quite a bit. The following snippet shows the full connection process from our perspective.

```
public void Connect(string level)
{
    // if connected, join - else initiate connection to the server.
    if (PhotonNetwork.IsConnected)
    {
        // attempt joining a Random Room. If it fails, call
            OnJoinRandomFailed() to create new room.
        PhotonNetwork.JoinRandomRoom();
    }
    else //connect to Photon Server.
    {
        PhotonNetwork.GameVersion = gameVersion;
        PhotonNetwork.ConnectUsingSettings();
    }
}
```

First we use the `IsConnected` property from the PhotonNetwork object, to see if there is already a connection. If not, it calls the ConnectUsingSettings() function that connects to the photon server. If it is already connected it will attempt to join a room. See chapter 3 for more information on rooms and other Photon related concepts.

This connection process happens as soon as the application is started, and once it is done, a limited player character is spawned in the "lobby". This player character can move around and start a game, but cannot shoot.

Lobby, Interactables & Starting a game

At this point the player is in the lobby. There are some simple animated interactables in the current lobby that can be used for interacting in a different way from a traditional menu. For instance, they could open up an in-game store or a player inventory. In the current version of the game however, all they can do is to start a game. All interactables are deriving from a parent class called *InteractableController* that has the following function implemented.

```
void OnMouseDown()
{
    RaycastHit hit;
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    Debug.DrawRay(ray.origin, ray.direction * 100, Color.yellow);

    if (Physics.Raycast(ray, out hit))
    {
        Transform objectHit = hit.transform;
        if (objectHit.tag == "LobbyInteractable")
            Interact();
    }
}
```

This uses a ray cast (a virtual line with collision information) from the camera towards the direction of the clicked point. If an Interactable is is hit by this ray, its *Interact()* function will be called.

Every Interactable overwrites a parent function called *Interact()* which defined what should be done when it is clicked. The *Interact()* function from the *TeamDMPlay-Interactable* can be seen below.

```
public override void Interact()
{
    if (launcher == null)
    {
        print("Launcher reference not set on " + this.name);
        return;
    }
    launcher.Connect("test_teamDM");
}
```

This function implementation is very simple. It calls the connect function from the launcher class with a scene name as parameter, which will then load the player into a Team Deathmatch game.

6.4.3 Team Deathmatch and Freeplay

After the above process has been completed the player will enter a game of a certain game mode. At this very early stage in the development of our game, there is only two game modes, Team Deathmatch and Freeplay. The difference between these two modes is whether or not the player will join and fight against a team, or it will be every player against every other player.

Instantiation

At this point the player leaves the *Lobby* scene and enters one of the game scenes.

Instantiating something in the game (in our case our player prefab) using *PhotonNetwork.Instantiate* will automatically synchronize the content of the PhotonView component of the instantiated prefab for all clients in that respective room.

```
void Start()
{
    Instance = this;

    Debug.LogFormat("We are Instantiating LocalPlayer from {0}",
        SceneManager.GetActiveScene().name);
    // we're in a room. spawn a character for the local player. it gets
        synced by using PhotonNetwork.Instantiate

    if (!usingTeams)
    {
        //Spawn at random position
        PhotonNetwork.Instantiate(this.playerPrefab.name, new Vector3(0f, 5f,
            0f), Quaternion.identity, 0);
        PhotonNetwork.LocalPlayer.SetTeam(PunTeams.Team.none);
    }
    else
        SetTeamInstantiate(); // Spawn at a predefined team spawn location
}
```

If were not in a team game mode, we tell photon to synchronize our player character by spawning it with *PhotonNetwork.Instantiate*, and set it's team value to `none` allowing the player too shoot and take damage from any other player.

Teams

If the current game mode is team based (Currently this is only the case for Team Deathmatch), instead of directly calling `PhotonNetwork.Instantiate` in the start function shown above, we call `SetTeamInstantiate()` which at this point works as a simple way of making evenly numbered teams.

```
public void SetTeamInstantiate()
{
    if (PhotonNetwork.PlayerList.Length % 2 == 0)// even / uneven
        PhotonNetwork.LocalPlayer.SetTeam(PunTeams.Team.red);
    else
        PhotonNetwork.LocalPlayer.SetTeam(PunTeams.Team.blue);

    var spawn = spawnPoints[PhotonNetwork.PlayerList.Length].transform; //not
        -1 because player is added after this
    PhotonNetwork.Instantiate(this.playerPrefab.name, spawn.position,
        Quaternion.identity, 0);
}
```

This function uses the modulus operator to swap between the red and blue team based on whether there is an even or an uneven number of players in the room at the point of connection.

We have an array containing empty GameObjects with positions called SpawnPositions. See figure 6.6. This array contains predefined positional objects placed in each end of the map, where the teams base is supposed to be.

One side of the map holds the even numbered spawn points, and the other side holds the odd numbered ones. We select the spawn position object stored at the index corresponding to the current number of connected players. We then collect the position from that object, and spawn the player there using `PhotonNetwork.Instantiate`.

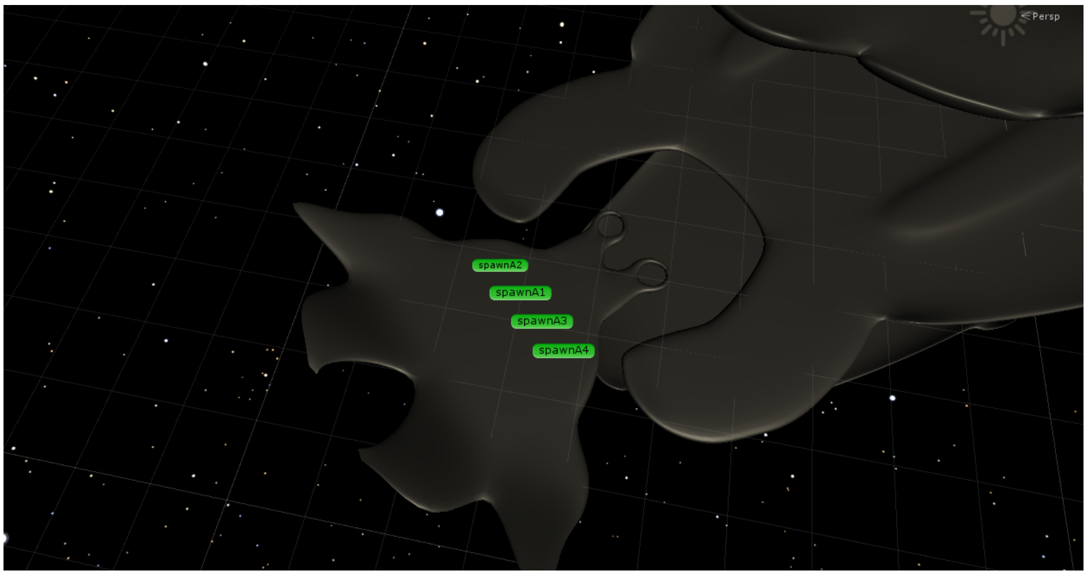


Figure 6.6: Spawn positions for team A

6.4.4 Movement

For moving the player we make use of the `AddForce` and `MovePosition` functions provided by Unity's Rigidbody component.

The MovePosition function recieves a Vector3 (a three dimensional vector) representing the desired new position. It then does a smooth transition over a number of frames using linear interpolation.

We create an input vector by collecting the numerical input values (which will be either 0 or 1) from the movement keys and store those values in their corresponding axis in a vector called 'input'. We multiply that vector with a speed value and the `Time.deltaTime` to make it independent from the frameRate.

```
//Physics are not calculated in sync with the normal update (where input
    should be collected), it should be handled in FixedUpdate
void FixedUpdate()
{
    if (!photonView.IsMine && PhotonNetwork.IsConnected)
        return;

    if (IsGrounded() && doJump) Jump();
    if (doDash) Dash();

    //Move the character
    rigidBody.MovePosition(rigidBody.position + input * speed *
        Time.fixedDeltaTime);

    //Increase gravity effect on the player
    rigidBody.AddForce(Vector3.down * gravityIncrease * rigidBody.mass);

    if (lastPos != transform.position || lastRot != transform.rotation)
    {
        lastPos = transform.position;
        lastRot = transform.rotation;
    }
}
```

We make sure that the player is connected before they are allowed to move. If they are, we retrieve the input and move their position. We then add a force in the downwards direction to simulate gravity using the Rigidbody's `AddForce` function. Lastly we store the current frame's position to see whether there has been any change in the following frames.

Look Direction

If the player is controlling with the mouse, we determine the direction the character is supposed to be looking at using a raycast from the camera and storing the point where the ray intersects with the map. We can then subtract the characters position from the stored intersection point, which gives us a vector corresponding to the direction from the player towards the intersection point.

If the player is using the arrow keys to control, we simply create a vector from the input value instead of the above. This gives the player eight fixed directions to move and look in.

Unity's transform component has a vector variable called `forward` which represents the forward direction of the character. Once we have a direction from either mouse or keyboard input, we can set this variable to rotate the character.

This however results in an instant rotation, which looks unrealistic. To avoid this we use the `Vector3.Lerp()` function which interpolates between two vectors based on a speed input. This rotates the character slowly over several frames.

```
private void DetermineLookDir()
{
    //if Mouse1 is pressed, use a rayCast from the cameta to get a position
        on the map for the char to look towards
    forward = Vector3.zero;
    var cam = Camera.main;
    Ray camRay = cam.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;
    if (Input.GetButton("Fire1") && Physics.Raycast(camRay, out hit))
    {
        forward = hit.point - transform.position;
        forward.y = 0;
    }
    else if (inputArrow != Vector3.zero) forward = inputArrow; //when Mouse1
        isnt pressed set the look direction to the key input
    else if (input != Vector3.zero) forward = input;

    //Update look direction
    if (forward != Vector3.zero) transform.forward =
        Vector3.Lerp(transform.forward, forward, Time.deltaTime *
        smoothSpeed);
}
```

Advanced Movement

In addition to moving around horizontally with the arrow keys, the user has the option to use the space key to jump upwards, and the shift key to dash quickly in their movement direction.

Both of these are implemented using the `RigidBody.AddForce()` function. We calculate an input vector corresponding to the direction and use this as input.

```
private void Dash()
{
    Vector3 dashVelocity = Vector3.Scale(input/*transform.forward*/,
        dashDistance * new Vector3((Mathf.Log(1f / (Time.deltaTime *
        rigidBody.drag + 1)) / -Time.deltaTime), 0, (Mathf.Log(1f /
        (Time.deltaTime * rigidBody.drag + 1)) / -Time.deltaTime)));
    rigidBody.AddForce(dashVelocity, ForceMode.VelocityChange);
    doDash = false;
}

private void Jump()
{
    rigidBody.AddForce(Vector3.up * Mathf.Sqrt(jumpDistance * -2f *
        Physics.gravity.y), ForceMode.VelocityChange);
    doJump = false;
}
```

Chapter 7

Implementation of shared game elements

In this chapter we are going to describe some of the other features in the game that are being implemented in a common way in both of the versions.

7.1 Reptile Weapon

Among the Weapons that are created in our game program we are going to explain the logic behind one of them called the Reptile Weapon as it's the weapon with the most functionality. This weapon can target anything and as it recognizes any object on its target (while holding the shoot button), it would send a bullet that follows the object until it collides that object.

As is shown in the class diagram 5.2, a `Weapon` type of object has the following functions: `Shoot()`, `StopShooting()` and `DealDamage()`. The method `DealDamage()` calculates the damage that it has dealt to the remote player. The system sequence diagrams 5.5 and 6.5 for Node.js and Photon versions respectively, demonstrate greater overview on how the shooting and damage operations are being handled, as well as the networking aspect of it.

The following code snippet is part of the logic on how the Reptile weapon handles the bullet shooting in the `ReptileController`.

```
1     void Start()
2     {
3       line = laserBeam.GetComponent<LineRenderer>();
4     }
5
6     void Update()
7     {
8         if (line.GetPosition(1).z < maxLineLength)
9             line.SetPosition(1, new Vector3(0, 0,
                  Mathf.Lerp(line.GetPosition(1).z, maxLineLength,
                  Time.deltaTime * 4)));
10    }
```

In the `Start()` method in line three, we instantiate the `line` object that is being used to detect an object. Then in the `Update()` function we generate a line towards the aiming position.

```
    private void ManageCollision(Collider other)
    {
        hasTriggered = true;
        var distance = Vector3.Distance(other.transform.position,
            transform.position);
        if (distance < maxLineLength)
            line.SetPosition(1, new Vector3(0, 0, distance));

        GlobeProjectile newGlobe = Instantiate(globe, firePoint.position,
            firePoint.rotation) as GlobeProjectile;
        newGlobe.fromMainPlayer = true;
        newGlobe.targetCharachter = other.gameObject;
        string id = transform.parent.GetComponent<Player>().id.ToString() +
            "rep" + globeId.ToString();
        newGlobe.id = id;
        globeId++;
        gameController.InstantiatePlayerBullet(id,"rep",firePoint.position,false);
    }
}
```

In the `ManageCollsion()` function, that is being called when the `line` object detects any collision with an object. We get the target of the collision and use it as one of the parameters to instantiate an object of the type `GlobeProjectile`, which is then being spawned from the weapon as a bullet that follows the object that it took as the parameter.

On line 14 we call the gameController's `InstantiatePlayerBullet` which handles the networking part of this for synchronization with the remote players.

7.1.1 GlobeProjectile Class

The following code snippets display a part of the code in the `GlobeProjectile` class, that handles following the character after the bullet is being sent out from the reptile weapon.

```
void FixedUpdate()
{
    if(fromMainPlayer){
        transform.position = Vector3.MoveTowards(transform.position,
            targetCharachter.transform.position, Time.deltaTime *
            smoothSpeed);
        gameController.MoveBullet(id, bulletType, transform.position,
            isExploded);
    }else{
        var bp = gameController.GetBulletParams(id);
        if (bp == null) return;
        if (bp.position != lastpos)
        {
            transform.position = bp.position;
            isExploded = bp.isExploded;
            lastpos = transform.position;
        }
    }
}
```

In the `FixedUpdate()` method in the `if / else` statement, the program checks whether this object should operate as if it is being handled as local instance of a bullet, or as an instance of a bullet that is sent out from a remote player. If it is being handled as the main instance, it would in each sequence get the current position of the character object it got as the property and moves toward that object using the `MoveTowards()` method.

The operation *Time.deltaTime * smoothSpeed* ensures that the movement would be independent of the frame rate because we're factoring in the Time.deltaTime (the time spent since the last frame). The `MoveTowards()` method is using linear interpolation to smooth out acceleration in any direction. The effect of liniar interpolation is that instead of moving directly from one position/rotation to another, intermediate values are created and the movement will be split over a number of frames.

The `else` statement is run if the bullet is a remote instance. It would update its position according to the parameters related to it self received from the server using the socket (This part is only being used in the node.js version, as in the Photon version, synchronization of transform properties of objects are streamlined by the use of Photon's `PhotonTransformView` class).

7.2 Camera Follow

We have a relatively simple camera follow script that has a fixed position at an angle above the player. The player can use the scroll wheel to zoom in and out, but has no other controls over the camera.

The player prefab (a prefab is a saved entity that can be spawned at runtime) has an invisible object above its head. This is the camera's `target`. We can change this target at runtime to make the camera look at something else in the case of an in-game event. Currently however, the rotation of the camera never changes and is thus simply set to a fixed value.

```
void FixedUpdate()
{
    if (!target) { return; }

    float num = Input.GetAxis("Mouse ScrollWheel");
    distance -= num * scrollSens;
    distance = Mathf.Clamp(distance, minDistance, maxDistance);

    Vector3 pos = target.position + offset;
    pos -= transform.forward * distance;

    transform.position = Vector3.Lerp(transform.position, pos, smooth * 0.5f
        * Time.deltaTime);
}
```

First we determine the distance from the player using either the default value or the input from the scroll wheel. We then move the camera to the targets position + a fixed offset using linear interpolation so the camera will smoothly change its position rather than ”teleporting”.

Chapter 8

Discussion

In this chapter we're going to assess the pros and cons related to each version of our game prototype described in the earlier chapters.

In chapter 9, the Conclusion, we will evaluate what approach would be more appropriate for our game system according to our plan.

8.1 Pros and cons of the Node.js and Socket.io implementation

In this section the pros and cons associated with our first version of the game (Node.js and Socket.io Client Server networking) would be evaluated and described through the following table

Pros	Cons
• The logic in the server is entirely being written and controlled by us. So a part of the game logic could be handled in the server. • Node.js uses JavaScript that is asynchronous in nature, ensuring that code execution does not stop. Due to this the program in the server won't unnecessarily stop the execution of code which is unrelated to the process that it is currently running. Practically speaking, this means that a client's request wouldn't impact the performance of the server handling other requests. • Node.js has a large ecosystem of open-source libraries, that can be used in our server-side development. • Socket.io is quite portable, since it provide libraries in many languages and environments. This is useful if we plan to build a new version of our game or port it to other game engines. • Socket.io is very easy to learn and use. It provides various high level functions that abstract the complexities of Socket based programs. • Socket.io is ideal for rapid development and deployment, and is thus quite useful when prototyping and testing a system.	• As Node.js uses JavaScript, there are some incompatibilities when passing data between the server and the Unity engine due to JavaScript's weak typing. • Our current implementation approach on this version requires us to go back and add/modify networking code on components in our program, when a new feature that has to be synchronized is implemented. This can be facilitated by a implementing a better, more generalized solution that would lead us to focus less on the core logic of our game and more on adding new features. • Node.js and Socket.io were not originally intended for games specifically, and the corpus of libraries/documentation to incorporate Socket.io into games is somewhat lacking. • As the server-side code is completely designed and created by us, we would need a hosting service to store the server-side program. This requires us to pay additional fees and investing more time on finding an appropriate and performant hosting server according to the number of current users.

Table 8.1: Pros and Cons of the node.js and socket.io implementation

8.1.1 Conclusion of Node.js and Socket.io implementation

Summing up the pros and cons discussed on the table nr 8.1, we can conclude that using Node.js and Socket.io in our game program gives us much more control over the logic of our server, thus we can handle many operations for the synchronization in the server-side code.

On the other hand however, integrating Node.js and Socket.io with the Unity C# client-side has the disadvantages of maintainability issues, complexity in the client-side code and incompatibility between Javascript and C# data structures.

8.2 Pros and Cons of the Photon implementation

Pros	Cons
• Photon is developed especially for game networking, targeting Unity as one of their supported game engines, thus it also provides a wealth of functionality for game networking and communication that makes the development process easier. There are no issues with incompatibility when transferring data between different data-types, and there is a tight integration with the Unity engine. • Photon provides servers in different locations based on our needs, which are free to use up to 20 concurrent users. In this case we would be able to run and test the game online from the beginning, rather than testing them locally. That gives us a more accurate overview on our game's performance. • Photon reduces the complexity of the program structure. • There is good documentation on how to implement game networking using Unity and Photon.	• We don't have the access required to manipulate the server logic and thus all of our game logic must be handled in the client side. • **Vendor lock-in**. Our application is dependent on a third-party (Photon) in order to function. • Although it's free to use for the first 20 concurrent users, it can become expensive once the number of users raises significantly.

Table 8.2: Pros and Cons of the node.js and socket.io implementation

8.2.1 Conclusion of Photon implementation

Summing up the results according to the pros and cons discussed on the table 8.2, we can conclude that using Photon for handling the networking area of our game program can be beneficial, due to the APIs and services that simplify the process of game development and would enable us to release the game faster. On the downside it limits us with a blackbox server logic and environment and make our game to be dependant on them.

Chapter 9

Conclusion

In the discussion above we have looked at the pros and cons of the two versions of the game. In this final conclusion, we'd like to sum up our experience with both and evaluate whether the switch to Photon was a good idea.

As it transpired from the discussion, the main disadvantage of the Node.js implementation was the growing complexity of our networking code, that forced us to go back and add more ad-hoc code every time a new feature needed to be network-aware. This was one of the major reasons that made us switch to Photon, but we believe that such a change should not have been necessary if we had had more time. Most of the issues we had with Node.js and Socket.io could be resolved by re-building the Node.js networking code to be more general and scalable.

This is to say that our switch to Photon was more due to circumstance that some flaw inherit in the technology, and from an academic stand point we would have benefited from building - or rather finishing - the network part ourselves. That said however, Photon proved useful to us when testing the application, and testing the final game would still be harder with the Node.js and Socket.io implementation.

As for Photon it greatly sped up our work, and simplified the implementation of both old and new networked features.

A running theme through this report is that the game was not finished, and that is a rather obvious point we could build upon. Another avenue of further research could be to include user feedback in our project form an early stage, improving on the UX perspective and perhaps giving us better insight in our game design process.

Since we spent a lot of time comparing Photon and Node.JS in this project, it could also prove useful to perform some empirical testing to asses the performance of each system with a finished game.

Bibliography

[1] Scott Bilas. A data-driven game object system. https://www.gamedevs.org/uploads/data-driven-game-object-system.pdf, 2002. Presented at the 2002 Games Developers Conference.

[2] Exit Games. Connection and authentication: Regions. https://doc.photonengine.com/en-us/pun/v2/connection-and-authentication/regions. Accessed: 2018-12-07.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] Brandi House. Evolving multiplayer games beyond unet. https://blogs.unity3d.com/2018/08/02/evolving-multiplayer-games-beyond-unet/. Accessed: 2018-12-12.

[5] Adam Martin. Entity systems are the future of mmog development. http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/. Accessed: 2018-10-02.

[6] Ted Nelson. Interactive systems and the design of virtuality. *Creative Computing*, 6(11):56–62, 1980.

[7] Jenny Preece, Yvonne Rogers, and Helen Sharp. *Interaction Design: Beyond Human-Computer Interaction.* Wiley, 2002.

[8] Unity. Entity component system. https://unity3d.com/learn/tutorials/topics/scripting/introduction-ecs. Accessed: 2018-10-02.

[9] Unity3D. The webpage of the unity game engine. https://unity3d.com/. Accessed: 2018-10-02.

[10] Steven Daniel Webb. *Referee-based architectures for massively multiplayer online games*. PhD thesis, Curtin University of Technology, Department of Computing, 2010.

[11] Rachel Weber. Codemasters to use exit's photon for online titles. `https://www.gamesindustry.biz/articles/2011-09-22-codemasters-to-use-exits-photon-for-online-titles`. Accessed: 2018-11-10.

[12] Nick Weihs. Techniques for building aim assist in console shooters. [video]. `https://www.gdcvault.com/play/1017942/Techniques-for-Building-Aim-Assist`, 2013. Accessed Dec. 14, 2018.