Bachelor's thesis

Information and Communications Technology | Game Development

2019

Mika Anttonen

ONLINE MULTIPLAYER ON MOBILE GAME

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology | Game Development

2019 | number of pages: 55, number of pages in appendices: 3

Supervisor: Principal Lecturer Mika Luimula, Adj.Prof

Mika Anttonen

ONLINE MULTIPLAYER ON MOBILE GAME

In the present gaming world, multiplayer games are the rulers in business. The greatest gaming companies make more money from selling services inside online games than from selling new games ever. Unity's decision to completely overhaul their game engine's network functions will bring a significant change on how smaller developers will be able to develop their games.

The goal of the thesis was to compare different approaches towards developing multiplayer games and to use one of those approaches add online multiplayer functions on a mobile game. The game is an updated and ported version of an unpublished 2D space shooter made by Markus Leinonen and Iisko Lappalainen and it is called X-Craft. The original version was created as a computer game using the Pascal programming language.

Developing multiplayer functions depends greatly on which game engine is used. After researching different options, Unity was chosen as the game engine for X-Craft. In August 2018, Unity announced they will renew the engine's networking tools. Due to the announcement, Photon Unity Networking was chosen to be used for developing multiplayer functions.

The thesis discusses planning a multiplayer game followed by comparisons between a few popular game engines. It also delves into the future of developing a multiplayer game with Unity based on blogs and articles available during the time of the writing of the thesis. The asset used in the development of X-Craft is called Photon Unity Networking is introduced on the level it has been used.

The updated version was made using Unity and C# programming language. The development of the game was the responsibility of the author with the clients being part of planning and testing of the game. The results are going to be a part of the final version of X-Craft.

The game was tested in different locations and with different devices. Even though the results on playability were not ideal, the results on the multiplayer functionalities exceeded expectations and gave a good direction on how the development of X-Craft should continue. The development will continue according to the plans made based on the result of testing.

KEYWORDS:

multiplayer, Unity, C#, mobile game, game development, Photon Unity Networking

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka | Peliteknologia

2019 | 55 sivua, 3 liitesivua

Ohjaaja: yliopettaja Mika Luimula, dosentti

Mika Anttonen

VERKKOMONINPELI MOBIILIPELISSÄ

Click here to enter text.

Nykyisessä videopelien maailmassa moninpeli on markkinoiden johtavassa asemassa. Suuret peliyhtiöt saavat enemmän rahaa verkkopelien sisällä myytävistä palveluista kuin ne ovat saaneet uusien pelien myynnistä koskaan. Unityn päätös tehdä pelimoottorinsa verkkotoiminnot kokonaan uudestaan tuovat mukanaan suuren muutoksen siihen miten pienemmät pelien kehittäjät voivat kehittää pelinsä.

Opinnäytetyön tarkoituksena oli tutkia eri lähestymistapoja moninpelin kehittämisessä ja lisätä verkkomoninpeliin tarvittavat funktiot mobiilipeliin jollain kyseisistä lähestymistavoista. Peli on päivitetty versio ennen julkaisemattomasta tietokoneelle tehdystä 2D space shooterista nimeltä X-Craft. Alkuperäisen version tekivät Markus Leinonen ja Iisko Lappalainen. Alkuperäisen version koodi on kirjoitettu Pascal-ohjelmointikielellä.

Pelimoottori vaikuttaa suuresti moninpelin toteutukseen. Eri vaihtoehtojen tutkimisen jälkeen X-Craftin pelimoottoriksi valittiin Unity. Unity ilmoitti elokuussa 2018 pelimoottorin verkkotyökalujen uudistuksesta. Ilmoituksen johdosta, moninpelin kehittämiseen päätettiin käyttää Photon Unity Networking nimistä työkalua.

Päivitetty versio tehdään Unityllä ja C# -ohjelmointikielellä. Toimeksiantajat ovat olleet mukana pelin suunnittelussa ja testauksessa. Tulokset tulevat olemaan osa X-Craftin lopullista versiota.

Peliä testattiin eri alueilla ja erilaisilla laitteilla. Vaikka pelattavuuden kannalta tulokset eivät näyttäneet ideaalisilta, moninpelin tulokset ylittivät vähimmäisodotukset ja antoivat hyvin suuntaa millä tavalla X-Craftin kehitystä kannattaa jatkaa. Pelin kehitys tulee jatkumaan tulevaisuudessa testien pohjalta tehtyjen suunnitelmia myötäillen.

ASIASANAT:

moninpeli, Unity, C#, mobiilipeli, pelinkehitys, Photon Unity Networking

CONTENTS

APPENDICES

FIGURES

TABLES

LIST OF ABBREVIATIONS (OR) SYMBOLS

API	Application Programming Interface
CCU	Concurrent Users
CPU	Central Processing Unit
EA	Electronic Arts
ECS	Entity Component System
FPS	First Person Shooter
GEQ	Game Experience Questionnaire
HLAPI	High Level Application Programming Interface
ID	Identifier
IP	Internet Protocol
LAN	Local Area Network
LLAPI	Low Level Application Programming Interface
LTS	Long Term Support
MMORPG	Massively Multiplayer Online Role-Playing Game
OS	Operating System
PC	Personal Computer
PUN	Photon Unity Networking, might refer to the asset or the service in general
PVP	Player Versus Player
RPC	Remote Procedure Call
VR	Virtual Reality
UNet	Unity Networking

1 INTRODUCTION

Playing with friends has become an essential part of gaming. It has become so important that EA's Chief Financial Officer Blake Jorgensen told that one reason behind cancelling the anticipated Star Wars game from Visceral Games was that it "continued to look like a much more linear game [which] people don't like as much today as they did five years ago or ten years ago"(Jorgensen 2017). Many saw this as a statement on EA focusing on multiplayer games in the future. The trend can also be seen on streaming site twitch.tv where most streams on the front page show people playing multiplayer games such as Fortnite or PlayerUnknown's Battlegrounds. This thesis focuses on bringing that possibility on a mobile game(VG24/7.2017).

X-Craft is an unpublished game made originally by Markus Leinonen and Iisko Lappalainen, who were also the clients of this thesis, during 1990s. X-Craft is a 2D dungeon shooter where the player flies a spaceship and tries to destroy enemies' spaceships while surviving themselves. This thesis discusses the process of making the necessary functions for making this an online multiplayer game using Unity.

The clients had shown interest in working with the author after previous experiences with him. After considering a few options, developing multiplayer functionalities on mobile devices was picked as the topic of the thesis as multiplayer mode will be one of the main components of X-Craft. Other options would have required a lot more progress before development on them before they could start. Multiplayer functions were, however, something that was supposed to be worked on soon. At the end of the thesis, X-Craft would have working gameplay over network, simple matchmaking, two game modes and scoreboard that the players could check during a match. If time allowed, work on some sort of league or prestige system would be implemented.

This thesis is divided roughly in two parts. The first few chapters concentrate on the theoretical part. Chapter 1 discusses online multiplayer games in general. The chapter starts with the definition of online multiplaying followed by how multiplayer functionality in games is designed. Finally, the chapter compares different game engines and how they approach developing multiplayer games.

Chapter 2 focuses on Unity. Unity is currently renewing their networking solution. This chapter discusses the new solution based on information that has been published on official websites and forums.

Chapter 3introduces an asset called Photon Unity Networking. Basics of the asset are discussed, and comparisons are made between Photon and Unity's old solution, UNet. Some of the components will be discussed with examples from X-Craft.

Chapter 4 discusses the differences between the original version and the new updated version of X-Craft. It also introduces the game more, so the reader understands better what is needed during the development.

After Chapter 4, come the chapters that show the practical work itself. They show what happens before, during and after a match and how these things were achieved. Everything that needs more work than just changing a few settings in Unity is here. The way Photon Unity Networking is used is also explained in detail during these chapters. Each of these chapters starts with the requirements set by the client considering the chapter's topic.

Chapter 7 revolves around testing. It shows how the tests were planned, what happened during them and finally the results. The tests will include testing playability, usability and servers. These results will be used to further development and to help deciding what kind of servers to use in the future.

The last chapter concludes everything that has happened during making multiplayer functions. It discusses the results of final tests of development and what will happen in the future. The chapter also includes feedback from the original developers and possible exclusions from final product.

2 ONLINE MULTIPLAYER

According to Oxford Living Dictionaries, multiplayer is an adjective “denoting or relating to a video game designed for or involving several players”(Oxford Living Dictionaries). In other words, it is a video game that can be played by multiple players simultaneously. In the past, this was commonly achieved by having multiple players play on the same device but with different controls or by having players play in turns. While these settings have not completely disappeared, they are usually replaced or accompanied by the possibility of playing the game online. When playing online, each player has their own device with the game on which they will interact with the game world and each other. This, and technological progress in general, has given developers a possibility to make other than turn-based games for mobile devices.

2.1 Planning an online multiplayer game in general

Planning in developing online games is very important, maybe even more important than with single player games. In single player games, the developer has to only think what happens on the player’s computer. If there is anything happening outside of the player’s computer, it is most likely saving the game on cloud or leaderboards comparing statistics of players who have played the game. The game will still work offline with everything happening locally. Of course, there are some exceptions, for example SimCity in 2013, that require Internet connection all the time.

A developer firstly has to consider is what game engine to use. This is of course an integral part of developing games already, but it also changes how multiplayer development happens. Due to the clients’ wishes, X-Craft is being developed on Unity. This subject is discussed more in the next chapter.

The second really significant issue is whose servers the developers are using. The servers can either be their own or rented. Some server providers have made plugins for different game engines which eases the development a lot. A plugin or plug-in is “a small computer program that makes a larger one work faster or have more features”(Cambridge Dictionary). When selecting a server provider, a developer should consider how fast the servers need to be, the pricing and locations of servers. During the development, there was a decision to change the server provider due to Unity’s decision

on deprecating their old network components. This created a great amount of extra work when much of old code had to be edited and new code had to be written.

After making the previous decisions, it is time to plan the game itself. When planning a multiplayer game, developers need to decide how to split the game between matchmaking and gameplay, and how they will work together and independently. According to video game site and wiki Giant Bomb, matchmaking is “a technique which allows players or teams to be matched with others for the purpose of playing an online multiplayer game”. It also includes deciding the rules of that play session and possibly social activities. The simplest examples of matchmaking include a list of play sessions the player can join and possibly a ‘Quick Match’ option which connects the player to the first available play session(Giant Bomb 2019.)

Matchmaking is a significant part in making multiplayer game enjoyable. Beginners do not generally like to play against skilled veterans and veterans do not want to carry a team full of beginners who do not know how to play. This is usually solved with some kind of level or prestige system. Counter Strike and Rainbow Six Siege have solved this by collecting stats from past matches player has played and give them a rank based on their success. After giving them their initial rank, the players can rise or drop in ranks depending on their success in future sessions. Players also have possibility to play casual matches from where their success does not contribute to their rank.(Counter Strike Fandom 2019.)

Many players also prefer playing with their friends instead of strangers. This behaviour can be observed both in individual and team-based games. Some games have a possibility to form a team with friends and then search for a match. There are also many games that allow players to make their own private matches where others enter by either accepting an invitation or with a password. Some computer games also allow the player to connect with their friends by entering IP address. This is very common in games where players work towards a goal together. Private matches are usually considered casual matches and they don’t contribute on players’ ranks.(Véron et al. 2014.)

Gameplay is what some people call the game itself. In gameplay, the player controls their character and interacts with the game world. Gameplay uses the rules set in matchmaking to control some events in the game.

2.2 Differences between game engines

For a comparison among game engines, three popular game engines were chosen that can be used to develop multiplayer games. Those game engines are Unreal Engine 4, Unity and CryEngine V. The latest version of Unity that can be used to develop multiplayer games without using external programs or plugins as described in this text is Unity 2018.2. Currently, the newest version of Unreal Engine 4 is 4.22.1 and the newest CryEngine V is 5.5.2. Articles, forums and official documents were used to collect information and opinions from as many perspectives as possible. The comparisons between the game engines were made according to costs, offered services and in general. The information that was considered important during the start of the development has been gathered in Table 1 at the end of the chapter.

All of the aforementioned game engines use different ways to receive money from the developer. All three are available for testing for free. Unity's costs are the most complicated with their levels of licenses. Depending on the revenue the game produces annually, Unity has three different licenses. If the revenue is under $100,000 the developer can use a Personal license which is free. Revenue under $200,000 requires Plus which is either $35/month or $25/month with a prepaid period of one year. The last license called Pro does not have revenue limits and costs $125/month. In addition to these, Unity will also charge if the game uses their servers for multiplayer. All licenses have a certain limit of free players before they will charge for the servers. Personal allows 20, Plus 50 and Professional 200 concurrent users. Concurrent users(CCU) means the number of users allowed to connect to the application at the same time. The cost after the limit is $0.49/GB for the data that goes through their servers. (Unity 2019.)

Both Unreal Engine 4 and CryEngine V cover their costs with royalties. Both take 5% of gross revenue of the game after a certain level has been reached. Before these levels, both are free. For Unreal Engine 4, the limit is $3,000/product/calendar quarter. For CryEngine V, it is $5,000/product/year. This makes testing games with larger groups easier as the developer does not have to pay as long as the game does not produce any revenue. Unlike Unity, both of these engines depend on developers own servers for multiplayer, so the server costs are up to the server provider the developer decides to use.(Epic Games 2019, Crytek 2019.)

All three game engines have their own web stores where the developer can buy assets made by the company itself or other developers, though as of March 2019 CryEngine's store is still a beta version (Crytek 2019). These assets include both graphical and technical assets. Unity's and Unreal Engine's web stores have assets that can be used to develop multiplayer games but CryEngine's store is currently lacking them.

All three game engines support Windows, Linux, PlayStation 4, Xbox One and some VR gears. Unity and Unreal Engine 4 also support iOS and Android. Unity is the only one of the three that supports newest Nintendo devices. As mobile support is currently in development for CryEngine V, it was not considered as a possibility for developing X-Craft but was still used in these comparisons due to its popularity. As for programming language, Unreal Engine 4 is the strictest as it only supports C++. Unity supports C# and JavaScript and CryEngine V supports C++, C# and Lua. All of them allow the developers to download and edit the editor's source code and all of them are written in C++. Unreal Engine 4 and Unity support 32-bit and 64-bit builds but CryEngine will stop supporting 32-bit builds with CryEngine 5.6. Google has announced that after August 1st, 2019 "new apps or app updates that include native code are required to provide 64-bit versions in addition to 32-bit versions when publishing to Google Play" (Google 2019).

Thinkwik wrote an article comparing the three during April of 2018. Pluralsight made the same comparisons in 2015 with earlier version of CryEngine and attendance of Source 2. They concluded that CryEngine is the hardest to learn with all its features. Unity was considered the best game engine for developing mobile games. CryEngine and Unreal Engine were considered superior to Unity in creating high detailed graphics. As X-Craft is a game with really simple graphics, this did not affect the decision making. Different forums that were used to discuss the differences often came to similar conclusions with opinions on developing multiplayer games differing. Unity was also often considered the easiest to use. (Thinkwik 2018, Pluralsight 2015, Reddit, Android Authority 2018)

After reading the article from Thinkwik and considering the information above, the original developers and the author agreed to start working with Unity to develop X-Craft. Unity was also familiar to some degree with the author and the clients which made the decision easier. The game would be developed as far as possible with Unity and other software would be used only to support the development.

Table 1 shows the information that was used to decide the game engine used for developing X-Craft. The opinions from journalists and other developers did not take into

account what platform the game would be played on. Instead, the opinions concentrated on if the game engine would be suitable for a game looking and behaving like X-Craft. Considering the information collected and clients' opinion, Unity was chosen as the game engine used to develop X-Craft. The main deciding factors for the decision were support for multiple platforms, ease of porting the game for other platforms, and familiarity with the game engine.

Table 1. Deciding factors on choosing game engine.

	Unity	Unreal Engine 4	CryEngine V
Price	Free for testing with under 20 CCU, after release $25+/month depending on user amount and amount of data	5% of revenue after $3000/product/ calendar quarter	5% of revenue after $5000/product/year
Support for mobile	Android, iOS	Android, iOS	Doesn't support
Programming language	C#, JavaScript	C++	C++, C#, Lua
32-bit/64-bit support	Yes	Yes	Only 64-bit
Journalists opinion on suitability for games like X-Craft	Recommended	Possible	Too heavy, a lot of unnecessary tools and components
Other developers' opinion on suitability for games like X-Craft	Recommended for beginners	Recommended if developer is already familiar with game engine	Doesn't offer anything new compared to Unreal Engine
Developer's familiarity with game engine	Multiple years	Only basics	None
Porting a game for iOS/Android	Easy	Might require a complete redo	Not possible

3 UNITY NETWORKING 2019 AND FORWARD

At the start of August of 2018, Unity sent an email to Unity users that use Unity services concerning deprecation of UNet. UNet consists of High and Low Level API (HLAPI and LLAPI), relay services and matchmaking services. At the same time Unity announced their future plans for replacing these multiplayer services. The services would be developed mainly with connected games in mind which Unity has worked on with Google. Unity has yet to reveal prices for these services or if they will have free version for small player amounts that was used in the first bigger test of X-Craft before the change to Photon.

Reasoning behind these actions was feedback Unity had gotten from game developers using Unity. Feedback was summarized on a blog published on 2nd of August. Feedback included developers wanting “more-scalable and transparent core-networking and fully supported server-authoritative games”. In the same blog, they told how they planned to achieve these goals and about the estimated schedule of making these changes available to developers.(House 2018.)

3.1 Schedule of the new multiplayer solution

According to the original schedule, UNet would no longer ship with Unity after 2018.4(also known as 2018 LTS where LTS stands for Long Term Support). APIs would receive critical fixes for two years after 2018.4 release and relay and matchmaking services would operate for at least three more years after the release. However, unlike they said, some components of UNet had already been removed in Unity 2018.3.

The first Unity versions using the new services would be released in fall or December of 2018. On November Unity has released an example of FPS project using these services. The version released as of writing this sentence uses Unity 2018.3 beta 6 and works on both Windows and Linux. The project is free to download though it is not finished yet.(Unity 2018.)

Unity released an updated schedule in April of 2019 which is shown in Figure 1. HLAPI would be deprecated as planned but other components would be delayed. LLAPI would be shipped until 2019.4 and would be supported for two years after that. Relay and

matchmaker services' support would be extended until 2022. The schedule for the new solution also received some updates. Game server hosting and communication services have already been released. The new network transport layer and server runtime and integrated matchmaker service are still in alpha phase and would remain there until late 2019.(Unity 2019.)

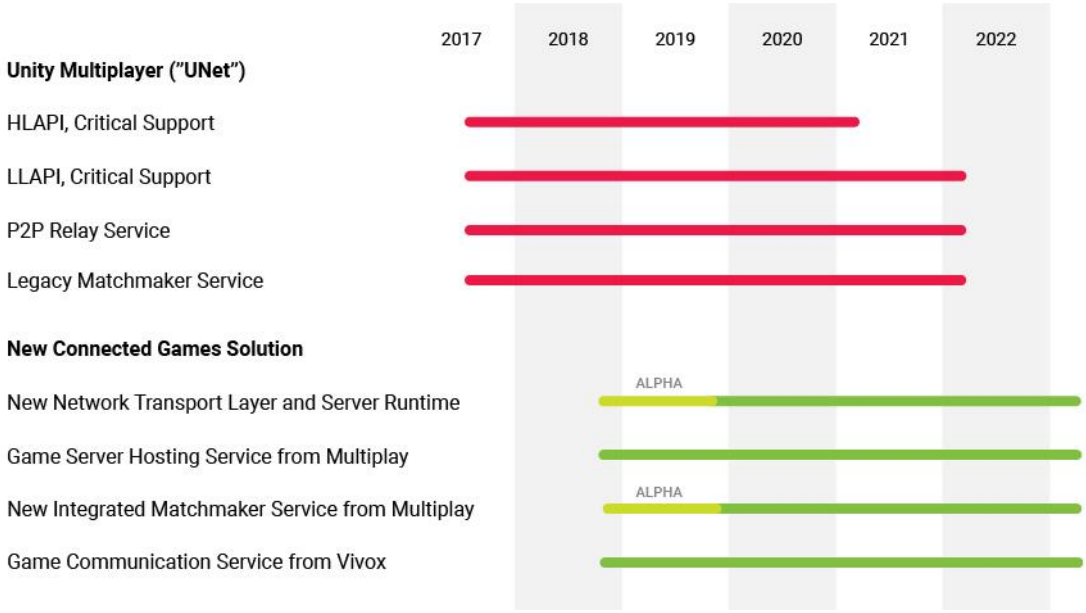

Figure 1. Schedule of Unity's new multiplayer features.

3.2 Connected Games

In a blog written on 21st of July, the vice president of Cloud Service for Unity Technologies Suhail Dutta, defines connected games as "games that connect players to other players as well as players to developers". This would be shown as player vs player multiplayer and content updates. He later divides connected games into four segments: dynamic single player, turned based multiplayer, real time multiplayer and persistent game spaces.(Dutta 2018.)

Dynamic single player games are mainly single player games with some online features like chat and leaderboards. In these games, fast internet connection isn't necessary, and some can be played offline. Most mobile single player games fall into this category.

Turned based multiplayer games include all turned based games that require more than one player. The speed requirement of internet connection depends on the amount of data between players, servers and developers. Fast connection is recommended for

better gaming experience. These games include games like Sid Meier's Civilization series and board games.

Real time multiplayer games and persistent game spaces are both games that require continuous internet connection as everything needs to be as in sync as possible. The difference between these two categories is that real time multiplayer games are played in sessions or rounds, but persistent game spaces stay 'alive' even if players disconnect. Real time multiplayer games include most current eSports games like Rainbow Six Siege, StarCraft and Overwatch. The game used for this thesis, X-Craft, is also in this category. Unity has mentioned in multiple blogs and forums that they initially focus on real time multiplayer games. Persistent game spaces include MMORPGs like World of Warcraft and ArcheAge.

Developing and maintaining connected games require infrastructure, knowledge required to use it as well as possible and resources to operate and support it. Unity's target is to provide these for all Unity developers requiring it for their games.

3.3 Change the topic

Unity released a second post in September on their developer blog to open up their plans more. The blog concentrated on what are the main things they are keeping in mind while planning and developing the new tools. The blog was divided in four parts where each part discussed different subject and long-term and short-term, which should be available Fall 2018, plans for them.(House 2018)

The first chapter discusses networking with Unity in general. The writer mentions a few ongoing-initiatives that should help developers building games. These initiatives include Job System, Burst Compiler and Entity Component System (later referred to as ECS). More info on these initiatives can be found on Unity's websites but in short Job System's purpose is to allow games to use multiple or all CPU cores on their device, Burst Compiler is a compiler specialized for Unity's game code and ECS code writing method that focuses on solving problems with data and behavior included in the game. These initiatives should make Unity able to support any games developers design regardless of amount of data that is either handled on players' devices or transferred through network. The main things developers at Unity will keep in mind while building the new

networking solution are transparency, modularity, archetype specificity and performance and scalability.

Second chapter is about servers. In the future, Unity will use dedicated game servers as default. In the past, server activity was host based which created problems especially on slower devices, some of which were seen during the first bigger test of X-Craft. As main reasons behind this choice writer lists consistency, scalability, security and fast iteration. This is where Multiplay comes in. Multiplay is a company that has hosted gaming events for years and is also known for providing game services. Unity acquired Multiplay's Game Hosting Division in late 2017. Unity will use Multiplay's technology and experience on servers to help users optimize their server activity. They also released information on alpha release, announcing that it would include Linux server support, a package for Server Query Protocol and stats and logs to show what is happening on the servers.(Unity Technologies 2017, Multiplay 2017.)

In the past, Unity's default matchmaking has been really simple. Player chooses whether they want to join LAN or online game and in case of online game whether they want to create or join a room. Everything else had to be done by the developer. Unity has promised change to this in the future solution called Open Match. Open Match is an Open Source project, so anybody can take a look at it and contribute in developing it if they want. Open Match will be managed by Unity and it will be integrated with Multiplay's technology by default. It should also help developers make their own rules for games by giving players premade customizable match logic. It should make creating rules and different match types a lot easier for the developer and should save them a huge amount of time.

The last chapter discusses a matter that are very important to indie developers, server cost and modularity. As ways of reducing cost, the mention five things minimizing consumption profile, server time used and networking bandwidth, running servers on Linux OS and blending bare-metal with cloud bursting once the game is able to benefit from a constant baseline of bare metal servers. The first three things lower the cost in very simple way, the less data servers need to handle, the less money it costs. Running servers on Linux helps saving on license costs that would come from using some other operating systems. According to TechTarget cloud bursting is ”an application deployment model in which an application runs in a private cloud or data center and bursts into a public cloud when the demand for computing capacity spikes” (TechTarget 2017).

4 PHOTON UNITY NETWORKING

After Unity informed their customers about the deprecation of their old multiplayer components including UNet, there was a decision made to change the server provider. For the evaluation, the author went through forums, links found on those forums and basic tutorials for the ones that were considered the most suitable. Following a week of evaluating other server providers that work with Unity, the clients and developer decided to continue with Photon. A developer of ENet CSharp has later made tests with some of the most popular ones including but not limited to ENet, Photon and UNet(GitHub 2018). ENet was easily the most efficient but as it did not support mobile devices during the start of multiplayer development of X-Craft it was not an option. Photon is one the most popular server providers for Unity that according to different forums is easy to use(Unity Forum, Reddit). Their plugin for general networking operations was called Photon Unity Networking. On August 24th, Photon released a new version of the plugin called PUN 2. Since the lobby and most of the gameplay was already built at that point using the old plugin, which is now known as Photon Unity Networking Classic, and there were no tutorials or guides on how to convert to the new plugin, the development would continue with the old plugin(from now on referred to as PUN which was its official abbreviation).

Like Unity, Photon offers a free version with limited users. After that however the pricing changes. Instead of pricing based on amounts of data transported, Photon uses CCU to price their services. This is counted from the peaks of every region your application is used in(EU, US etc.) during that month. Some price plans offered by Photon include CCU Burst. CCU Burst allows the developer to upgrade their plan after the first time CCU cap is reached. This offer is only available for 48 hours after reaching the cap first time. During those 48 hours there is no CCU cap, so the developer does not have to worry about losing players. After that time period the CCU cap will return to normal.

4.1 Using PUN and differences to UNet – Editor

To start using the PUN, the developer has to follow some steps before actually starting to network objects. First off is of course downloading and importing the plugin. The plugin can be found from asset store with Photon Unity Networking Classic – Free or Photon PUN+ Classic depending on what the game needs. In X-Craft the free version was used.

After importing the asset, PUN asks for appID. To get the appID, the developer needs to register to Photon's website and create a project. After creating a project on the website, developer can copy the ID from the website. Developer can also do this later and add the ID on PhotonServerSettings on Unity. PUN Wizard also gives other useful links including documentation and forums. More on the settings in PhotonServerSettings can be found on Photon's website. The only setting changed for X-Craft was the region used. In X-Craft Best Region was selected as hosting method and regions were limited to EU for testing. This was done because some testers were connected to different region and they could not play with other testers.

While there are some components that have similar variables in them, most of them work differently. In this chapter PUN's components used in X-Craft inside editor will be looked into and see how they differ from UNet's similar ones.

First component is Photon View. Every component that has network features has one. It is similar to UNet's Network Identity in that both are included in every networked object and both show the owner of the object. However, Photon View also has a part called "Observed Components". Observed Components is an array that includes all the networked components of that gameobject. This includes both scripts and components included in PUN. Figure 2 shows a screen capture of Photon View as it used on player's spaceship in X-Craft.

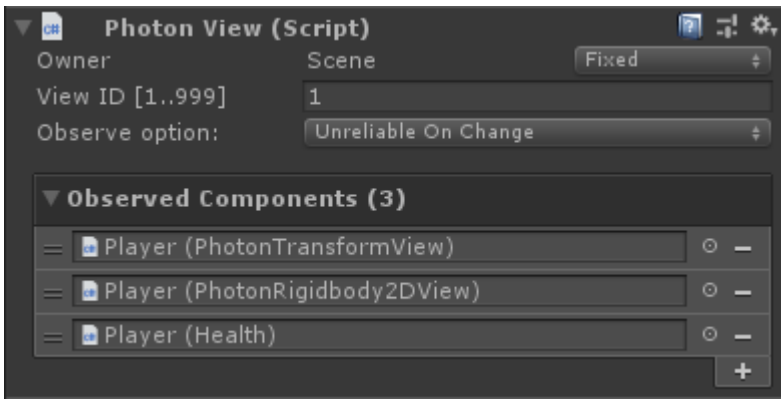

Figure 2. Player's Photon View.

Next one is Photon Transform View. This mostly resembles Network Transform. Photon Transform View is similar to Network Transform in that both are used to change settings considering gameobject's transform in network with Photon's component having a bit more options. Both components are missing something that the other one has. Network

Transform has the option to network gameobject's rigidbody instead of transform. This adds all the information of the gameobject's rigidbody on top of transform information. In PUN this needs to be done with separate component. Instead of this Photon Transform View includes a very important part for some games that Network Transform is missing and that is gameobject's scale. Currently X-Craft does not have any objects that need to have their scale updated over network but in some games, this is a critical part. In UNet scaling over network needs to be programmed by the developer. Another great thing that Photon Transform View offers is the ability to see the transform of the object on the server compared to client. This helps in adjusting interpolate and extrapolate settings and shows how much lag there is. Figure 3 shows player's Photon Transform View as it appears currently on X-Craft.

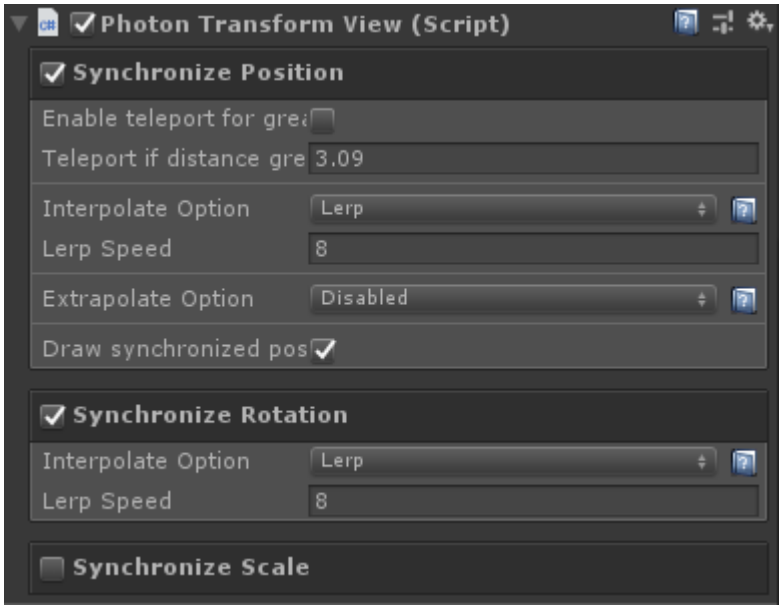

Figure 3. Photon Transform View.

Last PUN component used in X-Craft is called Photon Rigidbody 2D View. This component also has a 3D version of it with the same name without "2D". This component adds rigidbody 2D to the array of networked components. Photon Rigidbody 2D View has only two variables, will the gameobject's velocity be synchronized and will gameobject's angular velocity be synchronized. As mentioned before, UNet includes this in Network Transform and does not have a separate component for it. Below in Figure 4 is an example of Photon Rigidbody 2D View on how it is used in X-Craft.

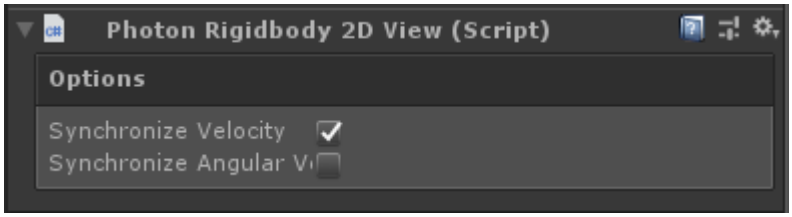

Figure 4. Photon Rigidbody 2D View.

Photon also has other components that were not used in X-Craft including but not limited to Photon Animator View that handles networking animations, and Photon Handler where developer can decide how often information of the object will be sent over the network.

4.2 PUN in programming

Photon has its own premade methods that can be used and modified to make your own code. Some of the methods Photon uses are similar to Unity's methods and all of them are named in a way that tells what they do or when they are called. For example, void OnJoinedRoom() is called whenever a player joins a room and can be easily modified by developers.

Photon also has Remote Procedure Calls or RPCs for short just like UNet. RPC is "a protocol that provides the high-level communications paradigm used in the operating system" and it is used to carry "data between communicating programs"(IBM). While UNet separates RPCs to Commands and ClientRpc calls depending on who calls them and who runs them, in Photon the developer can decide who runs the method while they are called. RPCs in Photon need an attribute PunRPC. RPCs are also called differently on Photon than on UNet. In UNet, it is enough that the RPC is called like a normal method. In Photon, RPCs are called with PhotonView.RPC(). PhotonView.RPC() always includes at least two parameters, the name of the RPC called, and who will execute it and with what timeframe. The RPC called is added as the first parameter in string form without its own parameters or parenthesis. So, RPC_LoadedGameScene(PhotonPlayer photonPlayer) becomes "RPC_LoadedGameScene".

Who executes and with what timeframe has seven possibilities that belong to enum PhotonTargets. The first three options have the RPC called immediately on targets when received and the RPC will not affect players who join the room later. They are All, Others and MasterClient. As the name suggests, RPC called with target All will be executed on

all clients currently in the room including the client who calls it. Others is the same except the method will not be executed on client who calls it. MasterClient means that it will be executed only on master client. Fourth and fifth options are AllBuffered and OthersBuffered. The targets are the same as in All and Others, but the calls are buffered so they will be executed also on new clients when they join unless the client who called the RPC has left the room before. The sixth option is AllViaServer. AllViaServer is the same as All except it is sent through the server instead of straight from client to client. The benefit, according to Photon, is that the RPCs are sent to all the players in the same order. So, if many clients send different RPCs, they will be executed in the same order for all despite possible lag. The last option is AllBufferedViaServer which is combination of AllViaServer and AllBuffered.

After the first two parameters, come the parameters needed for the RPC itself. As long as PUN can serialize the parameter used it can be used. The current version of X-Craft rarely uses RPCs with parameters so there will not be many examples. In old versions sharing players' health inside room was handled through RPC that had multiple parameters. The RPC was written in the code RPC_NewHealth(int health, PhotonPlayer photonPlayer) where the first parameter was the new health of the player's ship and second parameter points to the player whose health was changed. This RPC was called as PhotonView.RPC("RPC_NewHealth", PhotonTargets.All, health, photonPlayer).

When a script is created on UNet that uses network methods and variables, those classes need to be derived from NetworkBehaviour instead of MonoBehaviour. In Photon the class they are derived from needs to be changed too but instead of NetworkBehaviour they should be derived from PunBehaviour class included in Photon namespace. This provides .photonView and all the callbacks and events PUN can call for the developer to use or override as they want. Another useful but not necessary thing to do is to implement IPunObservable interface which makes sharing variables over network really easy. IPunObservable defines a method called OnPhotonSerializeView. Inside this method the developer adds all the variables that require sharing. Method is divided in two parts with if statements. First part happens when client is sending data and can determined by using if(stream.isWriting == true). stream is a parameter included in OnPhotonSerializeView. After the if statement shared variables are written in form stream.SendNext(variableName); where variableName is replaced with whatever the variable is called currently in the code. After writing all the shared variables in this form will come the second part where those variables are read. This part will be determined

by if or else if statement with parameter stream.isReading. Inside this statement are written the same variables in the same order in the form of this.variableName = (variableType)stream.ReceiveNext();. Variable type is required for the script to receive the variable in the right form or type. Below is a clip from a script used in every object that causes damage in the game other than player's ship. The clip shows how the name of the client who shot the bullet is shared.

```
void IPunObservable.OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
    if(stream.isWriting)
    {
        stream.SendNext(shooter);
    }
    else if(stream.isReading)
    {
        this.shooter = (string)stream.ReceiveNext();
    }
}
```

Figure 5. Sharing shooter of projectile in Damage script.

After doing all this the only thing left to do is to go to editor and drag the script to the array in Photon View of the object with the script as shown in Figure 1 of Chapter 4.1. Once this is done, sharing variables is done. There is no need for the developer to make any complicated code to share the variable like with UNet.

5 X-CRAFT

X-Craft is a 2D PvP dungeon shooter originally made for PC. The original version was made in 1990s by Markus Leinonen and Iisko Lappalainen. The game was never published and only reached beta testing. Due to scheduling problems, the project was put on ice until 2018. In spring of 2018, the original developers contacted Turku University of Applied Sciences looking for a student who would be interested in making a remake of the game as a work placement or final thesis.

In the original there was only one game mode, last man standing. Both players had set amount of lives. Once the lives of one player were depleted, the round ended and, if more than one map was selected before the match, the next round started. When all selected maps have been played through, the game ends and winner is announced. Figure 6 shows the map selecting screen in the original version. Players were able to select the number of maps with a chance to select same map multiple times or randomize the selected maps with amount chosen stays the same. The left side of the screen was supposed to show the highlighted map, but the development never reached that part.

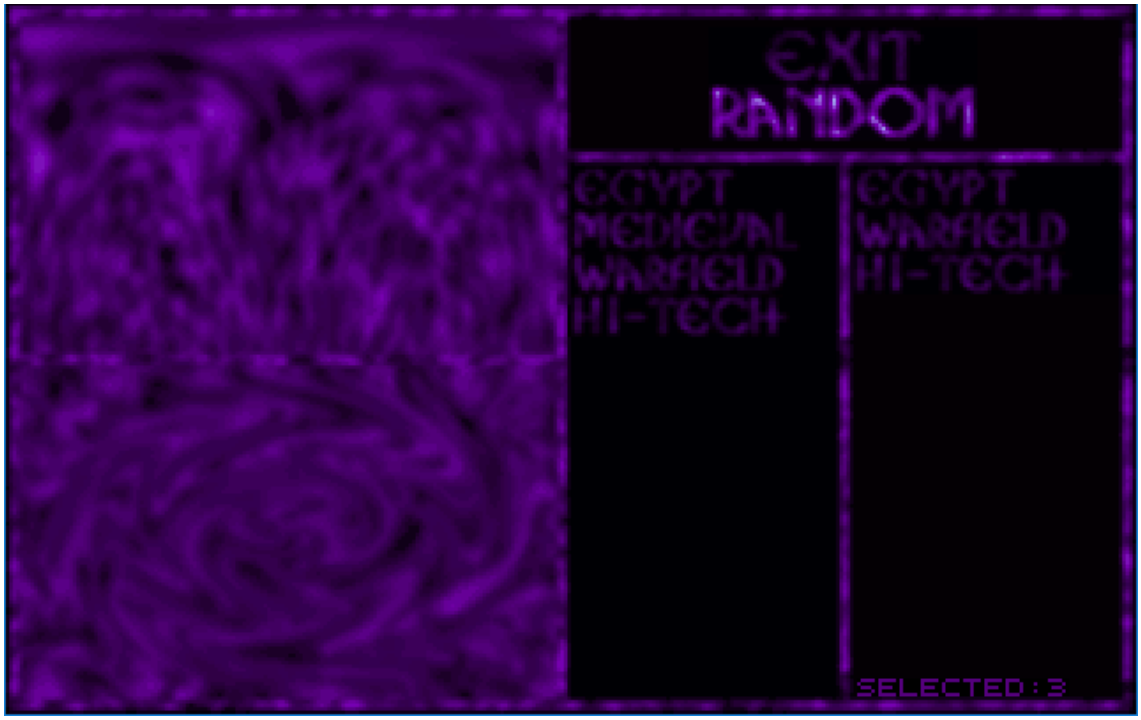

Figure 6. Original menu for selecting maps.

Multiplayer in the original version was made to work on one PC in split screen mode. The game allowed only two players to take part and they had to use the same keyboard. Players were able to choose from multiple weapons that they could use to destroy their opponent or environment. At the start of a round, the players spawned on landing pads that could be used to repair the ship and reload or change weapons. In Figure 7, it can be seen that the top player has just destroyed part of the map from where the pixels are still flying. The bottom player is still on the landing pad. The info on the side of each players windows show a minimap, small images of the chosen weapons and their ammo bars, health bar, remaining lives and credits which could be used to buy weapons.

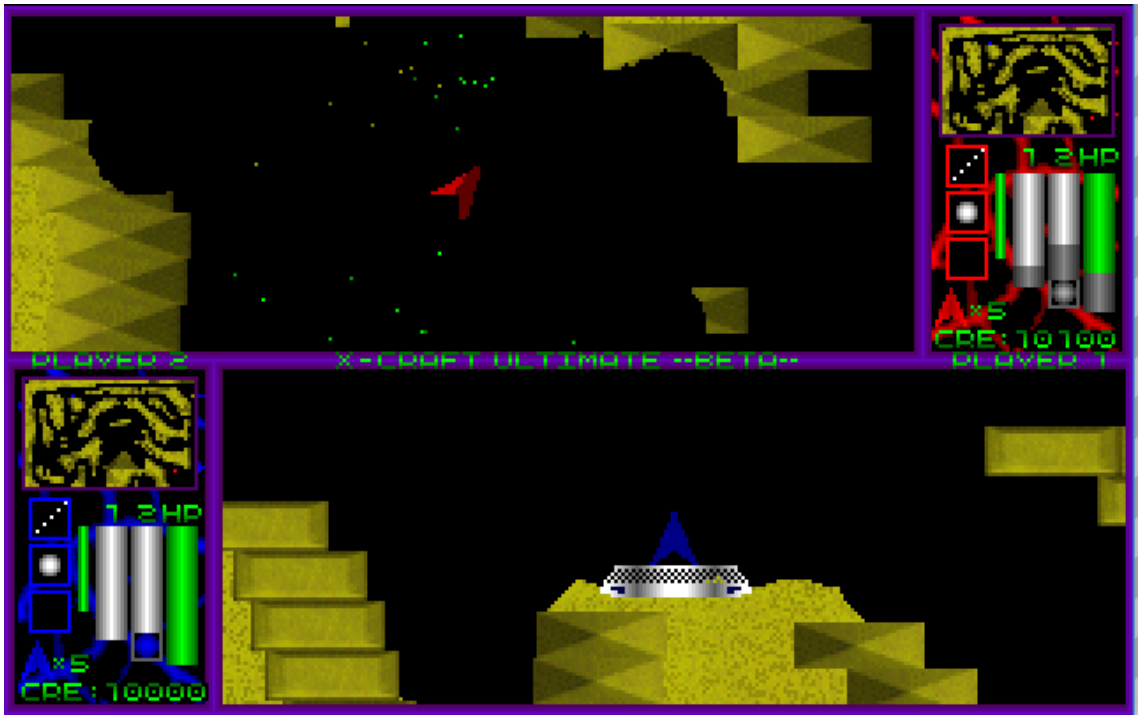

Figure 7. Screenshot of the original version.

The new version is made for mobile devices. Currently the game is only for Androids but in the future iOS support might be possible. Because of obvious restrictions of mobile devices, the game must be played on two or more separate devices. The maximum number of players per game has not been decided as of writing this thesis but for testing it has been set to 4.

X-Craft currently has two different game modes, deathmatch and single player. Since there are currently only two modes, the choice between them will be made by ticking a checkbox while creating a room. This will be later shown in Figure 9 and Snippet 3. Single

player mode is meant for helping new players get in the game easier. Single player rooms will not appear on the room list in lobby. It is useful for practising flying and shooting with spaceship. The player starts in the same map as in deathmatch. This will help them to also get used to the map. Around the map are some dummy spaceships. These spaceships have the same stats as the player's spaceship, but they do not have anything to control them and they are not affected by gravity. They have the same starting health points as the player and will react to collisions the same way. Instead of respawning after their health hits zero, they will be destroyed and will not come back until the level is restarted. Player is able to restart the level with an extra button in pause menu added to single player mode called "Reload Level".

Deathmatch is for people who want to compete with either against friends or strangers and is currently considered the main game mode of the game. The host of the room decides the needed kills for winning the match while making the room. This number is saved as a custom room property and will be loaded during the start of the match. When one of the players reaches the target, a scoreboard will be shown with players' names followed by their stats. When finished checking the stats, player will be returned to lobby.

6 MATCHMAKING

In this chapter the matchmaking in X-Craft and changing some settings on the player's client will be discussed. Topics of discussion will be about connecting to the network, how the rooms are created and joined to, changing both PUN's ready-made properties and custom properties for players and rooms and starting a game. Some of the code may use UI elements added in Unity Editor. Explaining the interactions with these elements will be kept to minimum if they do not contain critical information considering multiplayer. While developing the matchmaking, tutorials on YouTube channel First Gear Games were followed. The results of the tutorials served as the foundation of the matchmaking which was later expanded on.

6.1 Requirements and planning

The initial plans for matchmaking in this thesis involved player ranks only on a planning level. Ranking system would not be considered high priority and would only be made if there was extra time. This makes programming matchmaking a lot easier since searching a room would rely only players' input. Player would be given a list of rooms from which to choose. Both private and public matches would be available. Since X-Craft is a mobile game entering IP address to find a room is not an option. Inviting people would need a database where players' information including their friend list or something else where to pick the ID's of people the player wants to invite. This was put on hold for now but might be used in the future when the game is otherwise running. For now, X-Craft would use password to enter private rooms. Since X-Craft does not currently have chat function, sharing the password is left to the player in other manners. In the UNet version, this phase was not reached before the announcement regarding the deprecation, so it only exists in the Photon version.

Originally X-Craft was supposed to have two different multiplayer modes with different rules when this thesis reaches its ending. This was later changed so that the game would have one multiplayer mode commonly known as ”Deathmatch”, where the host of the room would be able to decide the number of kills needed for the win and a single player mode where players could practice flying and shooting targets. Adding more modes later is considered a possibility but not a requirement for this thesis.

6.2 Lobby

When the game is started, or the player returns to lobby there are some things that happen without the player noticing. These things are shown in Snippet 1. The lines under `Start` check if the device is connected to the server and connects in case it is not. Using `PhotonNetwork.ConnectUsingSettings("1.0")` makes sure that players that have different versions of the game will not be connected to the same lobby.

Snippet 1. Connecting to master server.

```
void Start(){

    if (!PhotonNetwork.connected)

        PhotonNetwork.ConnectUsingSettings("1.0");

}
```

When the game is started the first time on a device, the player is presented with a screen shown on Figure 8 that allows them to choose their own personal settings for the game including their username. This screen can be later be accessed from the main lobby. The only one of these settings that uses network components is the username. The changes in the settings call for different methods in script `ChangeName`.

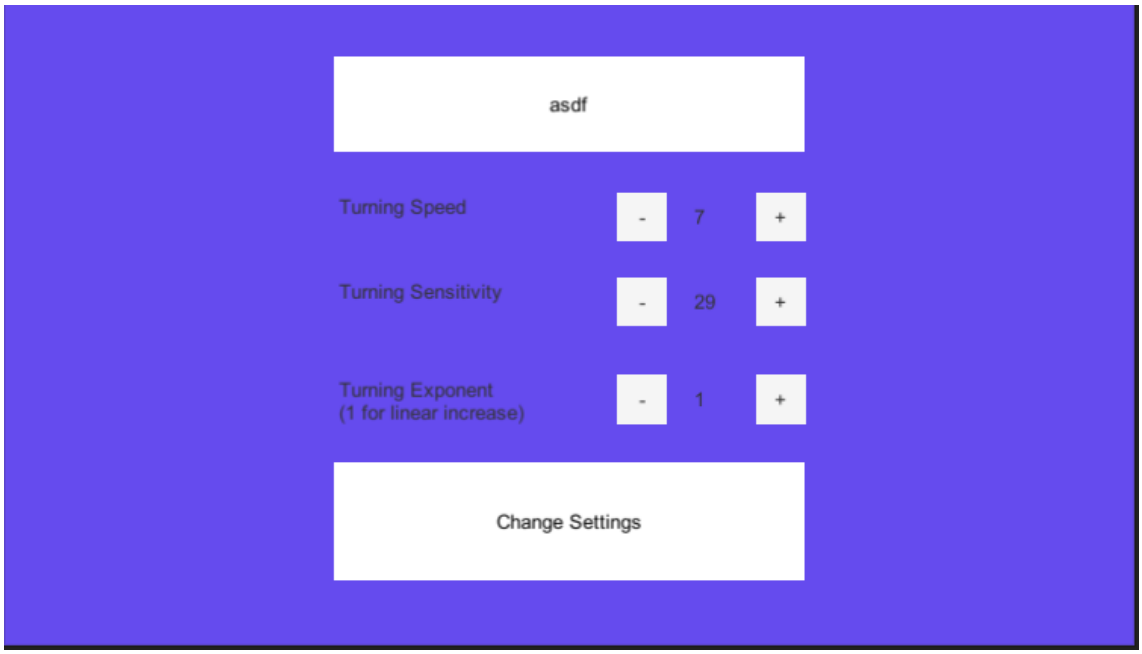

Figure 8. Player settings.

As mentioned before, the only variable in `ChangeName` that has effects in networks is the username. As X-Craft does not currently have any database where to storage player information, the username will be saved on local device and will not be discussed on this thesis. Snippet 2 shows how the name is taken from the box and set as the username. The username is saved in two places in PUN, `PhotonNetwork.playerName` and `PhotonNetwork.player.NickName`. Both are default properties for players. Assigning the name to `PhotonNetwork.playerName` will also change `PhotonNetwork.player.NickName`.

Snippet 2. Changing player name.

```
PhotonNetwork.playerName = PlayerName.text + " ";
```

Figure 9 shows the main lobby with one existing room called qwer. The UI elements on the left are used to change the properties of a room that the player creates. Joining a room happens with tapping the button with the room's name on it.

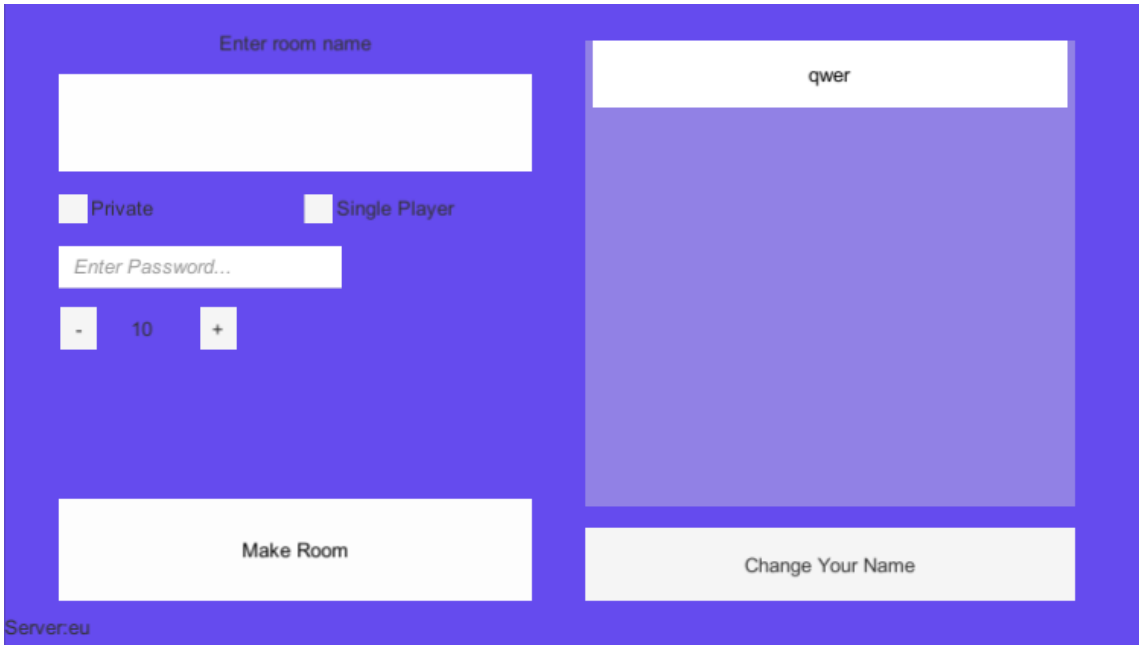

Figure 9. Main lobby.

Snippet 3 explains how a new room is created. Creating the room happens in a script called `CreateRoom`. First a variable of `RoomOptions` is created. `RoomOptions` includes all the properties of the room no matter if they are default or custom. The if statement checks if the checkbox `SinglePlayer` is checked. If it is checked, the room

settings will be set to fit single player game. If it is not checked, the room is visible and on default can be joined by anyone. Maximum number of players is currently four. Next, custom properties will be set for the room. There are three custom properties currently `isPrivate`, `Password` and `targetKills`. All of these need to be included even if the room is open for anyone to join. A custom property for a room can be set with `roomOptions.CustomRoomProperties.Add()`. The first parameter is the key which the property is referred with when needed and the second parameter is the value of the property. The custom property referred to with `"targetKills"` is later used in gameplay. By using `roomOptions.CustomRoomPropertiesForLobby` properties can be set so that they can be used in the lobby. The last method `PhotonNetwork.CreateRoom()` creates the room. The only parameter always necessary for creating room is the first parameter that is the name of the room being created. The second parameter sets the properties of the room and the third which lobby the room is created in.

Snippet 3. Creating a room.

```
RoomOptions roomOptions = new RoomOptions();

if (SinglePlayer.isOn){

  roomOptions = new RoomOptions() { IsVisible = false, IsOpen = false,
MaxPlayers = 1 };

}

else{

  roomOptions = new RoomOptions() { IsVisible = true, IsOpen = true,
MaxPlayers = 4 };

}

roomOptions.CustomRoomProperties = new Hashtable();

roomOptions.CustomRoomProperties.Add("Private", isPrivate);

roomOptions.CustomRoomProperties.Add("Password", Password.text);

roomOptions.CustomRoomProperties.Add("targetKills",
Int32.Parse(TargetKills.text));
```

```
roomOptions.CustomRoomPropertiesForLobby = new string[] { "Private" };

PhotonNetwork.CreateRoom(RoomName.text, roomOptions,
TypedLobby.Default);
```

Showing the rooms requires the most programming from all the scripts in matchmaking in the current version of X-Craft. Snippets 4, 5 and 6 show methods involved in showing the rooms on the list. All of these scripts belong to `RoomLayoutGroup` script. The prefab that is used for buttons representing the rooms have some code themselves which will be discussed later. For showing the room X-Craft uses some premade methods included in PUN. Snippet 4 shows the first of these methods called `OnReceivedRoomListUpdate()`. `OnReceivedRoomListUpdate()` is called when roomlist of the lobby is updated on the server. The method first collects all the rooms from the server and saves them in an array. All of these rooms are then handled in a method called `RoomReceived()` which is discussed in Snippet 5. The last thing `OnReceivedRoomListUpdate()` does is call the method `RemoveOldRooms()` to remove the rooms that has stopped existing or are full.

Snippet 4. Receiving an updated room list.

```
private void OnReceivedRoomListUpdate(){
  RoomInfo[] rooms = PhotonNetwork.GetRoomList();
  foreach(RoomInfo room in rooms){
    RoomReceived(room);
  }
  RemoveOldRooms();
}
```

Snippet 5 shows how a button for a room is created. `RoomReceived` uses `room` as a parameter that is given to it from `OnReceivedRoomListUpdate`. The method first uses the room's name to get an index of the room in the list of buttons. If the room does not have a button yet, the index will be -1. In this case, it is marked as visible and if the maximum limit of players of the room has not been reached yet, the button will be created using a prefab. The button will then be assigned as a child object of the object that has the `RoomLayoutGroup` script. That object is currently the background of the list, which

can be seen on right in Figure 9 as the area with lighter color than the rest of the background. The component `RoomListing` will be added to a list of components that represent the buttons used for joining room. The last thing to do with just new buttons for room is to give it a new index. The next statement happens to all buttons and is used to make sure the buttons are in order if one or more rooms have no need for a button anymore. The code will get the index of the button from the list containing all buttons, update its name and mark it as updated. If the room button for the room is not needed anymore, the script will not enter this part.

Snippet 5. Updating the room buttons.

```
private void RoomReceived(RoomInfo room){

  int index = RoomListingButtons.FindIndex(x => x.RoomName ==
           room.Name);

  if(index ==-1){

    if(room.IsVisible && room.PlayerCount < room.MaxPlayers){

      GameObject roomListingObj = Instantiate(RoomListingPrefab);

      roomListingObj.transform.SetParent(transform, false);

      RoomListing roomListing =
           roomListingObj.GetComponent<RoomListing>();

      RoomListingButtons.Add(roomListing);

      index = (RoomListingButtons.Count - 1);

  }}

  if(index != -1){

    RoomListing roomListing = RoomListingButtons[index];

    roomListing.SetRoomNameText(room.Name);

    roomListing.Updated = true;

}}
```

The last method in updating the room list is deleting the buttons not needed anymore. This is shown in Snippet 6. The method starts by creating a list for `RoomListing` scripts

that need to be removed. The script will go through the list used in Snippet 5 which includes all the `RoomListing` existing currently. If the room has not been marked as updated, it will be added to the list of buttons that are to be removed. If it has been marked as updated, it will be marked as not being updated anymore for future uses. Next the script will go through the list `removeRooms`. It gets the object the current `roomListing` has been attached to, remove the `roomListing` from the list of all `roomListing` scripts and finally destroy the object which includes the button itself.

Snippet 6. Removing the unneeded buttons for rooms.

```
private void RemoveOldRooms(){
  List<RoomListing> removeRooms = new List<RoomListing>();
  foreach(RoomListing roomListing in RoomListingButtons){
    if (!roomListing.Updated)
      removeRooms.Add(roomListing);
    else
      roomListing.Updated = false;
  }
  foreach(RoomListing roomListing in removeRooms){
    GameObject roomListingObj = roomListing.gameObject;
    RoomListingButtons.Remove(roomListing);
    Destroy(roomListingObj);
}}
```

Next is the script in all the buttons, `roomListing`. The way it is handled in the list was already discussed. The only property that has not been discussed and involves network activities is the how it will be used to join a room. This is done with one line of code as shown in Snippet 7. It calls for a method in `lobbyCanvas` and sends the selected room's name as parameter. In that method, the script calls for a premade function `PhotonNetwork.Joinroom()`. `PhotonNetwork.Joinroom()` includes one parameter which is the room's name.

Snippet 7. Joining a room.

```
button.onClick.AddListener(() =>
lobbyCanvas.OnClickJoinRoom(RoomNameText.text));
```

6.3 Inside a room

After creating or joining a room, the player will be shown the view in Figure 10. The view currently has only three functions on it, show all the players on the list on right, a button for starting a game at the top and a button for leaving the room at the bottom left. The same view will be shown regardless if a player has chosen single player mode or not. Start delayed refers to a match that where the game cannot be joined later. Leaving the room will move the player back to screen showed in Figure 9.

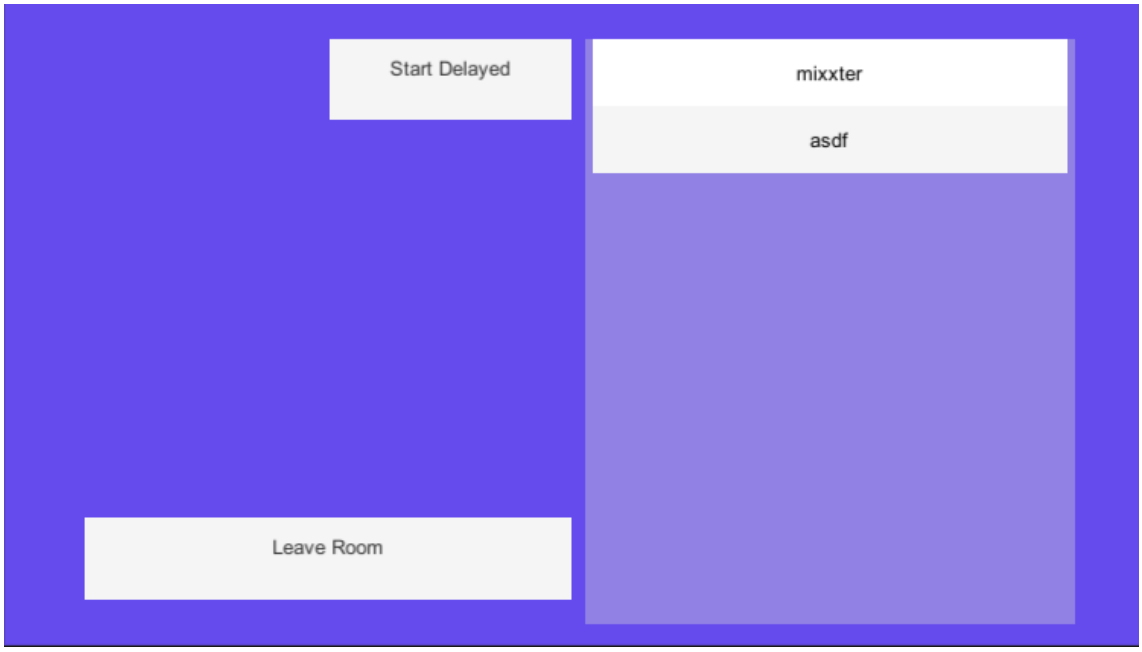

Figure 10. View inside a room.

Adding and removing a player from the list is quite similar to the way rooms are added and removed from the list in lobby. Instead of updating the whole list every time something changes on the list, adding or removing a player will be done by extending methods that are called by PUN by default when a player enters or leaves a room. Snippet 8 shows how a player is added to the list. When a player enters a room, PUN calls for a method called `OnPhotonPlayerConnected(PhotonPlayer`

`photonPlayer)`. In X-Craft, this method is used to call a method `PlayerJoinedRoom(PhotonPlayer photonPlayer)`. Both methods have the joining player as a parameter. Just in case there is an error and the method has been called by accident, the script checks if the player exists. If they exist, the script continues. The script creates a button for the player from a prefab, sets it as a child object of the list showing players in the room, gets the `PlayerListing` in the object, adds the name of the player to button with `playerListing.ApplyPhotonPlayer(photonPlayer)` and adds the `PlayerListing` to a list containing all the scripts of the same kind. Just like with rooms with the exception of property showing if the button has been updated missing.

Snippet 8. Adding a player to the list.

```
private void PlayerJoinedRoom(PhotonPlayer photonPlayer){

  if (photonPlayer == null)

    return;

  GameObject playerListingObj = Instantiate(PlayerListingPrefab);

  playerListingObj.transform.SetParent(transform, false);

  PlayerListing playerListing =
           playerListingObj.GetComponent<PlayerListing>();

  playerListing.ApplyPhotonPlayer(photonPlayer);

  PlayerListings.Add(playerListing);

}
```

Snippet 9 shows how a player is deleted from the list when they leave a room. Again there is premade method that is called when a player leaves the room and it is called `OnPhotonPlayerDisconnected(PhotonPlayer photonPlayer)`. The method calls our own method `PlayerLeftRoom(PhotonPlayer photonPlayer)`. The parameter is the player leaving the room. The script will get the index of the `PlayerListing` of the leaving player the same way as room's index was taken when updating the list of rooms. This time, the index is used for error checking. If the player has already been removed, the index is -1 and there is no need to remove the button. If it is not -1, the script will find the object where the current `PlayerListing` has been

attached from the list and destroy it. Then the `PlayerListing` will removed from the list.

Snippet 9. Removing the button of a leaving player.

```
private void PlayerLeftRoom(PhotonPlayer photonPlayer){

  int index = PlayerListings.FindIndex(x => x.PhotonPlayer ==
           photonPlayer);

  if(index != -1){

    Destroy(PlayerListings[index].gameObject);

    PlayerListings.RemoveAt(index);

}}
```

If a player taps the button used for leaving a room, a method shown in Snippet 10 will be called. The method simply calls a premade method for leaving a room.

Snippet 10 Leaving a room.

```
public void OnClickLeaveRoom(){

  PhotonNetwork.LeaveRoom();

}
```

Snippet 11 shows last method currently used inside the room, starting the game. The game can only be started by master client of the room. Master client is generally the player who has created the room. If that player has left the room, the next player on the list will become the new master client. Checking if the player who tapped the button to start the game is right in the start of the method. The next lines will be used to remove the room from the list shown in main lobby. `PhotonNetwork.room.IsOpen = false` makes it so that the room cannot be entered and `PhotonNetwork.room.IsVisible = false` makes it so it cannot be seen in the lobby. `if (!SinglePlayer)` checks if the player who created the room selected single player mode while making the room. This decides what scene the script will load.

Snippet 11. Starting a match.

```
public void OnClickStartDelayed(){
  if (!PhotonNetwork.isMasterClient)
    return;
  PhotonNetwork.room.IsOpen = false;
  PhotonNetwork.room.IsVisible = false;
  if (!SinglePlayer)
    PhotonNetwork.LoadLevel(1);
  else
    PhotonNetwork.LoadLevel(2);
}
```

7 DURING A MATCH

In chapter 4 using PUN in general was discussed. In this chapter the text will go more in depth on how it was used in X-Craft both in programming and Unity Editor. This will include snippets from the code, references to components shown in chapter 4 and reasoning behind some of the choices made during development. Code or variables in Editor that are not required to understand multiplayer aspects of X-Craft, like input or bullet amount, will not be shown or discussed. Since PUN has made sharing variables really easy code snippets shown are really short.

7.1 Requirements and planning

Since X-Craft is mobile game, the amount of data that will be send through the server during matches must be limited. Limitations are set by limitations of hardware and prices of data transmission on mobile devices. While the editor can be used as one player during testing and its tools for data gathering, it does not show a realistic image on how the game will perform on real usage. If the amount of data transmitted is too high, it might scare potential players away. High data amount would make X-Craft lose the edge that mobile games have compared to computer and console games. People would not be able to play it where they want as the data transmission would cost too much on some operators and the game would use too much battery.

With these restrictions in mind the developer decided to limit the variables sent through the server to the minimum needed for playing. The variables included the information on objects' location and speed, projectiles' shooters, players' current health, kills and deaths. Since PUN uses prefabs to spawn objects over network a lot of variables don't need transmission and are already included in the game files.

The initial plan, that was followed pretty accurately during development, had location and visually perceivable actions in high priority that should be finished first. After that would come sharing stats, some of which would be involved in high priority actions. Ending and target of the match was low priority and would be dealt when everything else is ready.

7.2 Development

At the start of multiplayer development, X-Craft already had most of the functions in single player environment. The ship was able to fly, shoot, get damaged and repair itself. Now all of this was supposed to be performed online. There were also some new functions including counting points and leaving room or game that needed to be made. Before the announcement of deprecation of UNet, the gameplay side of X-Craft did not have much in it. Projectiles and spaceships were flying, and projectiles were getting destroyed when they collided with something. However, the variables were not being shared and there were problems with the position were projectiles were spawned. Fortunately, PUN makes most of these things really easy.

7.2.1 Gameplay in Unity Editor

As mentioned on chapter 4, every object that needs to perform actions over network require some components on Editor. All of these components are described in the same chapter. In the current version of X-Craft there are only three such objects, the ship, bullet and bomb. With bullet and bomb the components needed were the same. Both needed Photon View and Photon Rigidbody 2D View. On Photon Rigidbody 2D View Synchronize Velocity checkbox was checked. The initial velocity and direction were determined during the moment when the object was initialized. However, since the velocity and direction also change during the existence of these objects, they need to be updated regularly. Photon Rigidbody 2D View does this without programming needed from the developer. On Photon View, the Observed Components include Photon Rigidbody 2D View and Damage script attached to the object. Damage script is discussed later in this chapter.

Prefab of the ship that the player controls is called Player. Player object had the same components as bullet and bomb but also included Photon Transform View. This was needed for respawns. Since Photon Rigidbody 2D View is only capable of sending information on objects velocity and angular velocity, moving player by changing values of its transform or changing rotation of player are left unnoticed. Photon Rigidbody 2D View was included to make the movement smoother on other players' screen. As with bullet and bomb, Synchronize Velocity was checked on Photon Rigidbody 2D View. As shown on Figure 3, Photon Transform View is synchronizing position and rotation. Both

position and rotation had the same interpolate settings with Lerp as the method and Lerp Speed as 8. The game has this far only been tested with three players so these settings may differ in the final version of the game. Draw synchronized position does not make any difference in gameplay but helps with finding the optimal settings so it was used. Player's Photon View included components mentioned above and Health script in Observed Components field.

7.2.2 Gameplay scripts

During gameplay, X-Craft has six scripts that perform network activities, `Health`, `ShootingScript`, `Damage`, `RealPlayerController`, `Scoreboard` and `PauseScripts`. The scripts will be discussed in this order. Snippets 11~14 are part of `Health` script. `Health` has methods involved in players' stats and collision between two players or players and a landing pad. As the name suggests, `ShootingScript` has code involved with shooting. It also handles cameras. `Damage` handles collisions excluding the ones handled by `Health`. It changes players' current health points on these collisions. The variable showing the shooter of a bullet is included in `Damage`. `RealPlayerController` handles the movement of player and some things performed at the start of the game. `Scoreboard` collects kills and deaths of every player and compares them to the target of current game. At the end of the game `Scoreboard` activates Scoreboard canvas and shows players their stats. `PauseScripts` has methods for leaving the game or reloading the level in single player. Since all of these scripts, except `PauseScripts`, use methods from PUN, they inherit class `PunBehaviour` from namespace `Photon`. Scripts `Health` and `Damage` share variables between players so they also inherit interface `IPunObservable`.

`Health` currently has four variables it shares through network, `currentHP`, `deaths`, `killAmount` and `lastDamager`. `deaths` and `killAmount` are also used in `Scoreboard` script and they will be handled a bit different from the other two. PUN has a way of attaching a variable to a player's network profile instead of the gameobject the player is controlling called custom properties. Custom properties use PUN's own `Hashtable` class instead of the default one in C#. This requires the developer to either initialize the variables of that class by referring it through the namespace or by overriding the class when determining the namespaces that are referred in the script. This is done

by adding `using Hashtable = ExitGames.Client.Photon.Hashtable;` to the list of referations at the start of the script. After this a `Hashtable` called `newProperties` can be created simply by writing `Hashtable newProperties = new Hashtable();`. Snippet 12 shows how `newProperties` is used to initialize the custom properties used in `Health`. This part of code is from the `void Start()` of the script and it initializes both variables as 0. `"deaths"` and `"killAmount"` are the keys used to find the variable from the `Hashtable` and refer to variables with same name.

Snippet 12. Initializing custom properties for player.

```
newProperties.Add("deaths", 0);

newProperties.Add("killAmount", 0);

PhotonNetwork.player.SetCustomProperties(newProperties);
```

The next part where custom properties are used in `Health` is when someone dies, and a kill is awarded to the last damager of the player. This method is called with `PhotonView.RPC("RPC_AddKill", PhotonTargets.All, ship.GetComponent<Health>().lastDamager)`. In Snippet 13 the first line of the body adds one kill to the killers variable `killAmount`. A local `Hashtable` called `hash` is created to update the custom properties. The `killAmount` is added to the `Hashtable` with a key that has the same name as earlier in the script. `hash` is then used to update the custom properties.

Snippet 13. Updating the amount of kill to the killing player.

```
[PunRPC]

void RPC_AddKill(string lastDamager)

{

  GameObject.Find(lastDamager).GetComponent<Health>().killAmount += 1;

  Hashtable hash = new Hashtable();

  hash.Add("killAmount", killAmount);

  PhotonNetwork.player.SetCustomProperties(hash);

}
```

Snippet 14 shows a method called `UpdateProperties()` that is used to update the custom properties of players. `Hashtable hash` is only used for the update so that it does not get confused with `newProperties`. The current local values of `killAmount` and `deaths` are added to the corresponding properties and update those properties.

Snippet 14. Updating properties.

```
void UpdateProperties()

{

    Hashtable hash = new Hashtable();

    hash.Add("killAmount", killAmount);

    hash.Add("deaths", deaths);

    PhotonNetwork.player.SetCustomProperties(hash);

}
```

Snippet 15 shows the last part where `Health` shares variables through network is through the `OnPhotonSerializeView()` method. There all the variables mentioned earlier are sent regularly through the network for all the players in the match. In the current version of X-Craft, this is the only time where attention needs to be payed to in which order the variables are written as they need to be sent and received in the same order.

Snippet 15. Transmitting variables in Health.

```
void IPunObservable.OnPhotonSerializeView(PhotonStream stream,
PhotonMessageInfo info){

    if (stream.isWriting){

        stream.SendNext(this.currentHP);

        stream.SendNext(this.lastDamager);

        stream.SendNext(this.deaths);

        stream.SendNext(this.killAmount);

    }
```

```
    else if(stream.isReading){

        this.currentHP = (int)stream.ReceiveNext();

        this.lastDamager = (string)stream.ReceiveNext();

        this.deaths = (int)stream.ReceiveNext();

        this.killAmount = (int)stream.ReceiveNext();

    }}
```

`ShootingScript` is really simple on network properties. It turns off other players' cameras for the local player and spawns the projectiles over network. Snippet 16 shows how the cameras are turned off. `photonView.isMine` checks whether the Photon View component attached to the current gameobject is local player's. The other condition makes it so that the body of the if statement is only performed in the multiplayer game. The first line of the body deactivates the third child object of the gameobject which in this case is the camera. `return` takes care that the rest of the `void Update()` is not performed.

Snippet 16. Turning off other players cameras.

```
if (!photonView.isMine && SceneManager.GetActiveScene().name !=
"SinglePlayer"){

    gameObject.transform.GetChild(2).gameObject.SetActive(false);

    return;

}
```

Instantiating objects on PUN is also simple. However, there are two things that need to be done on editor before this can be done. First the object about to instantiate must have Photon View attached to it. Second the prefab must be saved inside a folder called "Resources". Resources must be the root folder but can hold subfolder inside it that has the prefabs as it was done in X-Craft. Snippet 17 shows machine gun bullets being instantiated. `bullet` is the name used to refer to that bullet later in the code where some of its variables are changed. The first parameter of `PhotonNetwork.Instantiate()` is the path to the prefab file inside the Resources root file. `projectilePrefabPrimary` has been set on editor. While instantiating a

bomb this is the only value to be changed. It is changed to `projectilePrefabSecondary`. The second parameter is where the bullet will spawn which in this case is a gun that is a bit ahead of the player's ship to make sure that the bullet will not collide during spawn. Third parameter dictates the rotation of the bullet during spawn. The last parameter is the group where the Photon View belongs to. It is a necessary parameter for the method, but it does not change anything in the gameplay currently.

Snippet 17. Instantiating machine gun bullet.

```
GameObject bullet = PhotonNetwork.Instantiate(Path.Combine("Prefabs",
projectilePrefabPrimary.name),
GetComponentInParent<ShootingScript>().gun.transform.position,
Quaternion.identity, 0);
```

`Damage` is a script attached to every projectile. The only things it does over network are destroying the projectiles and updating the shooter of that projectile. The shooter of the bullet is shared with `OnPhotonSerializeView()` the same way as other variables on Snippet 15. Snippet 18 shows the part of code where a projectile is destroyed over network. Since the method that destroys objects over network can only be performed from the master client, the code needs to have something to check if the gameobject is owned by them. The gameobject must also have Photon View attached to it and it has to be created with `PhotonNetwork.Instantiate()`. If these conditions are not met the object will only be destroyed locally. If they are met, `PhotonNetwork.Destroy()` is called to destroy for everyone in the match. `PhotonNetwork.Destroy()` accepts two different parameters. One of them is a reference to the object itself and the other one is reference to the Photon View attached to that object. According to PUN's API there is no difference between them.

Snippet 18. Destroying projectile.

```
if(obj.GetComponent<PhotonView>() != null &&
PhotonNetwork.isMasterClient == true)
    PhotonNetwork.Destroy(gameObject.GetPhotonView());
else
    Destroy(obj);
```

There are currently two networking activities that are performed in `RealPlayerController`. The first part that happens only at the start of the multiplayer games. PUN currently does not perform some things that are needed when leaving a match. This has required for us to write some code that works around those problems. One of the problems is that it does not destroy the old instance of the player for the leaving player. Because of this when the player starts a new game after returning to lobby, they will have two ships and neither of them will work correct. This is fixed by destroying the other instance. Snippet 19 collects all the gameobjects with tag “Player” to an array. The program then checks if any of those objects have a duplicate of it by comparing the owner of their Photon Views. This is done until all the duplicates are destroyed.

Snippet 19. Destroying dublicate players.

```
playerAmount = GameObject.FindGameObjectsWithTag("Player").Length;

GameObject[] players = new GameObject[playerAmount];

players = GameObject.FindGameObjectsWithTag("Player");

for(int i = 0; i < players.Length - 1; i++){

    for(int j = i+1; j< players.Length; j++){

        if(players[j].GetComponent<PhotonView>().owner ==
           players[i].GetComponent<PhotonView>().owner){

            Destroy(players[j]);

            j = i + 1;

        }}}
```

Snippet 20 shows how the second part updates `Text` object called `gameInfo` to show some statistics of the local player. The first line shows the name of the server player is currently playing on. The second line shows the version of the game that can be changed from the Player Settings in Unity. The lines from three to five show some stats from `Health` script. The last line shows the roundtrip time from the local device to the server.

Snippet 20. Showing player stats.

```
gameInfo.text = "Server:" + PhotonNetwork.CloudRegion.ToString() +
"\nVersion:" + Application.version + "\nKills: " +
GetComponentInParent<Health>().killAmount + "\nDeaths: " +
GetComponentInParent<Health>().deaths + "\nLast Damager: " +
GetComponentInParent<Health>().lastDamager + "\nPing: " +
PhotonNetwork.GetPing();
```

`Scoreboard` currently runs only code that is involved in networking one way or another. At the start of the match it gets the current room's target that must be reached to win the match from the room's custom properties. Setting room's custom properties are explained chapter 6 in the part where a room is created. Since there is currently only one game mode for multiplayer there is no need to write to check which property is picked and what it is compared to. Getting room's custom property works pretty much the same way as getting player's custom property. In `Scoreboard` the property referred to with the key `targetKills` is saved to an integer with the same name. This is shown in Snippet 21. The integer `targetKills` has already been declared earlier. `room` is premade variable of type `Room` in `PhotonNetwork` class and it refers to the current room where the player is in during the execution of the command. `"targetKills"` is the key referring to the custom property declared by the master client while making the room. This code will get that custom property, cast it to type `int` and save it `targetKills` to be used later.

Snippet 21. Getting a custom room property.

```
targetKills = (int)PhotonNetwork.room.CustomProperties["targetKills"];
```

Comparing the current situation to the target is shown in Snippet 22. `PhotonNetwork` has an array called `playerList` which includes every `PhotonPlayer` in the current match. `Scoreboard` uses a foreach loop to go through each `PhotonPlayer` and get their custom properties. Currently these properties from these players are stored in a string called `scores` with the players' `NickName`. Each players' kill amount is then compared to `targetKills` and if one them has reached the required amount, a canvas showing a visual scoreboard is shown and the winner is announced.

Snippet 22. Comparing players' points to target.

```
foreach(PhotonPlayer p in PhotonNetwork.playerList){
```

```
    scores += p.NickName + " Kills:" + p.CustomProperties["killAmount"]
           + " Deaths:" + p.CustomProperties["deaths"] + "\n";

    if((int)p.CustomProperties["killAmount"] == targetKills){

        scoreBoardCanvas.SetActive(true);

        winner.text = "Winner Of The Match: " + p.NickName;

        Debug.Log(p.NickName + " is the winner");

    }}
```

The last script used during a match is called `PauseScripts`. `PauseScripts` has the code used in the pause menu which is opened by pressing Back button on the player's device while in a match. Two of the methods declared in this script perform network activities. The first method called `LeaveGame()` is shown in Snippet 23. This is a good example how self-explanatory some of the code in PUN is. `LeaveGame()` is used when the player wants to leave the current match but does not want to close the application. The first line inside the method makes the local player disconnect from the current room and connect back to the lobby. The second line loads the main menu scene which opens up when the game is started. It is similar `LoadScene()` but it can also be used to load the scene for everyone in the room if the player is master client.

Snippet 23. Leaving the current game.

```
public void LeaveGame()

{

    PhotonNetwork.LeaveRoom();

    PhotonNetwork.LoadLevel(0);

}
```

Snippet 24 shows the last method in the current version of X-Craft that performs network actions during a match, `QuitGame()`. This method disconnects the player from Photon server and closes the application. The device automatically disconnects from the server when the application is closed but by doing it manually the information about it is sent to other players faster and will not be slowed down by the other actions performed while the application closes.

Snippet 24. Closing the game

```
public void QuitGame()

{

    PhotonNetwork.Disconnect();

    Application.Quit();

}
```

8 TESTING

This chapter discusses the tests for the version of X-Craft used for this thesis that were hold from April 13th till April 15th. It discusses the planning, execution and results of the tests including the difficulties faced during planning and execution.

8.1 Planning

Planning for the testing started in early March. The original plan had the developers of the current and original version of X-Craft as testers. The test was meant to measure the success of the development by measuring stability, transmission speed and amount of data transmitted over the network. Each tester would use their own devices for the test. There would also be one computer connected to the game to gather information on how the game performed. The test was supposed to happen in late March or early April. However, due to conflicting schedules of developers these plans were cancelled and would be followed later in the development.

After the first plan failed, the author and clients decided to do a usability test. The highest priority of the development this far had been making the game easy to approach and use. The new plan involved new testers who had not played the game yet. The testers would still test the game with their own devices if possible. Most testers live in Finland, but one tester lives in the Netherlands where the used servers are located. Evaluation of the test would be done by using Game Experience Questionnaire(later referred to as GEQ) made by Eindhoven University of Technology(Eindhoven University of Technology 2013). GEQ is a commonly used questionnaire that is meant to measure how much testers enjoy different aspects of a game. As X-Craft does not currently have other social activities than playing with others through the server, Social Presence Module from GEQ was not used. In addition to GEQ, some open-ended questions were added. The questions involved device used for test, past experiences with games, possible bugs faced during the test and free feedback. The questionnaire can be seen in Appendix 1. After receiving approval from the original developers, the tests were ready to start.

8.2 Execution

The tests started by sharing the game with testers. For this, Google Developer Console was used. Google Developer Console had been earlier used to share builds of the game for the developers for individual testing and was deemed as a good method. In the Developer Console, the developer can choose the testers by adding the email addresses they use for Google Play Store to the list of testers. Like earlier in the development, internal testing track was used. After adding the email addresses of the testers, the testers received a link to a website that informed them about the test and asked for their permission for the tests. After giving permission, they were able to download the game from Play Store. So far everything went well.

The first test session included two outside testers and one joining later. The developer attended the session as overseer. Before testing the multiplayer aspects of the game, the testers would be allowed to practice for a moment in single player mode. During this practice session, it became apparent that the controls were hard to get into and would affect the results of the test too much. Thankfully, the questions for the original testing plan had been made already. The tests continued following those questions with the new testers but without a computer. From the abandoned questions, device, bug reports and free feedback would be used for reporting. Due to missing computer, the amount of data transmitted could not be measured. Since the code shown in Snippet 19 was still in use, some information on transmission speed could be received by using roundtrip time.

After changing the questions, the third outside tester joined the test with their own practice session. Following the practice session, the real test was ready to start. The testers joined a single room in which they played a couple of matches. Between the sessions, the master client of the room was changed to see if master client's device affected the results. The testers were occasionally asked about the roundtrip time their device showed.

Another test was conducted later in the weekend with the tester from the Netherlands and one developer. This session followed the changed plan from the previous day. The tester was during the time at home outside Amsterdam but agreed to play X-Craft the next day at work for a while so we could receive the roundtrip time in optimal location. The feedback on controllers followed the trend from the previous day.

8.3 Results

Devices used for the test ranged from a few years old OnePlus 3 to a new OnePlus 6T. All devices were either middle or high-performance phones with low-performance missing. There were no differences found between the performance of devices while playing. Having some low-performance phones could have affected the results but testing that would have required buying a new phone.

During the test session, X-Craft worked quite well. There was one crash reported during testing sessions. While starting a match, the screen on one player's device turned black and the game did not respond anymore. The other testers in that session were able to continue as intended. The problem has been reported to the developer, but the reason has not been found yet. Addition to this, one of the devices disconnected once from the server during a session. However, the tester said that the device has had a lot of connection problems in the past so at this moment the connection problem cannot be reported as a problem in X-Craft or Photon.

As expected, roundtrip times differed a lot depending if the tester was in Finland or the Netherlands. In Finland, the roundtrip times were commonly inside the range of 45 to 60. In the Netherlands, they were from 25 to 35. The number of players in the match did not seem to affect the time with the number that attended the test. The highest values of roundtrip time were reached when the number of projectiles on screen was high. This required tens of projectiles on the screen at the same time, which means that number of players might have bigger impact directly when it reaches 20 that has been talked during development. The roundtrip times were considered good for an unoptimized version but should be lowered later when the development continues. One method that might reduce the impact of bullets on roundtrip time is to increase the interval between the times their locations are transmitted. The projectiles are currently only affected by physics to change their location during their flight so sending the initial speed and location might be enough for them to follow the right flight pattern. There are plans to add projectiles with different flight patterns later which would need something different to reduce the impact.

As mentioned above, the controls were a problem for everyone, and it could be seen in the free feedback. Other feedback received mentioned that the game has potential but would require more work. Tutorial, which has been discussed with the original developers, also came up. Tutorial would also help the players with coming familiar with

the game. Making one could not fit in the development schedule yet. The graphics also received critique, but they have been considered as low priority this far.

9 CONCLUSION

The purpose of this thesis was to compare different ways of developing multiplayer games and use the gained knowledge to build a multiplayer framework for X-Craft that can be later expanded. The framework was supposed to include simple matchmaking with possibility to set rules for a match, possibility for a private match, two multiplayer game modes, a simple scoreboard to show during match and, if there was enough time, a prestige or league system. These goals were achieved even with Unity's announcement about deprecation of UNet forcing a restart of multiplayer development apart from prestige or league system and two multiplayer game modes. As mentioned in Chapter 6, a decision was made during development to change the two multiplayer game modes to one multiplayer game mode and a single player mode. This new goal was achieved. The current version was capable of performing all the tasks needed for the portion of multiplayer that has been planned this far. The original developers have been satisfied with what has been achieved so far.

The testing period has given the developers information on what they should do next both with multiplayer and game design in general. Depending on changes between PUN and PUN 2 the framework can be used to some extent even after changing the asset. Outside of the planned content, knowledge on releasing a game on Google Play Store was also gained and forwarded to the clients. Important information on Unity's future on multiplayer was also discovered.

As the thesis discusses, several changes have happened with the technologies used for the project. Unity is in the middle of changing their multiplayer tools and Photon has released a new asset to replace the one currently used in X-Craft. A sudden announcement from Unity made the first couple of months of developing and testing multiplayer functions irrelevant and forced a complete restart of multiplayer development. If the development of the multiplayer part of the game was started now, PUN 2 would likely be used. Unity's multiplayer tools would not be used at any point which would save around two months of development time. If Unity's current schedule for their new multiplayer tools succeeds, starting the project late in 2019 or in 2020 might have involve using them.

The next steps of development will revolve around making the game as enjoyable as possible with the current framework. The controlling system will likely be changed on

most parts and a tutorial scene will be made. Along with these plans, optimization of the existing game will start decreasing the roundtrip time between players' device and server. A new developer has joined the team already and will take the position of the main developer. The new developer has already ported most of the game to work with iOS. The author has delivered a summary of the contents of the game to the new developer and suggestions for the future. The author will continue to help with testing and development depending on their schedule.

REFERENCES

ANDROID AUTHORITY, 2018, Which is better? Unity vs Unreal Engine for Android game development. Available: https://www.androidauthority.com/unity-vs-unreal-engine-android-game-development-842045/ [25.3.2019]

CAMBRIDGE DICTIONARY, Definition of plug-in. Available: https://dictionary-.cambridge.org/dictionary/english/plug-in

COUNTER STRIKE FANDOM, 2019-last update Matchmaking. Available: https://counter-strike.fandom.com/wiki-/Matchmaking#Skill_groups [26.3.2019]

DONALDSON A., 2017. ”EA: Visceral Star Wars game canned because players don’t like linear games as much as they used to”. Available: https://www.vg247.com/2017/11/29/ea-visceral-star-wars-game-canned-because-players-dont-like-linear-games-as-much-as-they-used-to/ [26.5.2018]

DUTTA S., 2018. Bringing connected games within reach with Google Cloud published on Unity Blog. Available: https://blogs.unity3d.com/2018/06/21/bringing-connected-games-within-reach-with-google-cloud/ [24.11.2018]

EINDHOVEN UNIVERSITY OF TECHNOLOGY, 2013 Game Experience Questionnaire. Available: https://pure.tue.nl/ws/portalfiles/portal/21666907/Game_Experience_Questionnaire_-English.pdfOXFORD LIVING DICTIONARY, Definition of multiplayer. Available: https://en.oxford-dictionaries.com/definition/multiplayer

EXIT GAMES. Photon Unity Networking API. Available: https://doc-api.photonengine-.com/en/pun/v1/ [26.5.2019]

EXIT GAMES. PUN Basic Tutorials. Available: https://doc.photonengine.com/en-us/pun/v1-/demos-and-tutorials/pun-basics-tutorial/intro [26.5.2019]

FIRST GEAR GAMES. 2018-last update, Unity – Photon Networking. Available: https://www-.youtube.com/playlist?list=PLkx8oFug638qVMIrtqOnwmqnW6o8WDgQ1

GITHUB, 2018. Benchmark Results. Available: https://github.com/nxrighthere/BenchmarkNet-/wiki/Benchmark-Results [27.03.2019]

GOOGLE, 2019. Get your apps ready for the 64-bit requirement. Available: https://android-developers.googleblog.com/2019/01/get-your-apps-ready-for-64-bit.html [25.3.2019]

GOOGLE CLOUD PLATFORM. 2019-last update, Open match. Available: https://github.com-/GoogleCloudPlatform/open-match [15.2.2019]

HOUSE B., 2018. Evolving multiplayer games beyond UNet. Available: https://blogs.unity3d.com/2018/08/02/evolving-multiplayer-games-beyond-unet/[20.11.2018]

HOUSE B., 2018. Multiplayer Connected Games: First steps forward. Available: https://blogs-.unity3d.com/2018/09/12/multiplayer-connected-games-first-steps-forward/ [27.11.2018]

IBM. IBM Knowledge Center, Remote Procedure Call Available: https://www.ibm.com-/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.progcomc/ch8_rpc.htm

MULTIPLAY, 2018. Unity Technologies Acquires Game Hosting Division of Multiplay from GAME Digital Plc. Available: https://multiplay.com/2017/11/28/unity-technologies-acquires-game-hosting-division-multiplay-game-digital-plc/ [28.11.2018]

PLURALSIGHT, 2015. Unity, Source 2, Unreal Engine 4, or CryENGINE - Which Game Engine Should I Choose? Available: https://www.pluralsight.com/blog/film-games/unity-udk-cryengine-game-engine-choose [15.3.2019]

REDDIT, 2018-last comment, Multiplayer options in Unity. Available: https://www.reddit.com-/r/gamedev/comments/9vp2ye/multiplayer_options_in_unity/ [27.03.2019]

REDDIT, 2018-last comment, Unity networking advice. Total noob here. Available: https://www.reddit.com/r/gamedev/comments/9uztlg/unity_networking_advice_total_noob_here/e9a2944/ [27.03.2019]

REDDIT, 2018-last comment, Unity vs. Unreal for Networked Multiplayer. Available: https://www.reddit.com/r/gamedev/comments/87ascc/unity_vs_unreal_for_networked_multiplayer/ [20.03.2019]

TECHTARGET, 2017. Cloud bursting. Available: https://searchcloudcomputing.techtarget.com-/definition/cloud-bursting [26.5.2019]

THINKWIK, 2018. CryEngine vs Unreal vs Unity: Select the Best Game Engine. Available: https://medium.com/@thinkwik/cryengine-vs-unreal-vs-unity-select-the-best-game-engine-eaca64c60e3e [20.4.2018]

UNITY TECHNOLOGIES, 2019-last update, FPS Sample. Available: https://github.com/Unity-Technologies/FPSSample [28.2.2019]

UNITY TECHNOLOGIES, 2019-last update, UNet Deprecation FAQ. Available: https://support.unity3d.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ?_ga=2.91264858.1553078388.1545030351-1564194144.1537864922 [23.4.2019]

UNITY TECHNOLOGIES, 2019. Unity Pro. Available: https://store.unity.com/products/unity-pro?gclid=EAIaIQobChMI1PfXnumu4QIVg8CyCh2ULwkWEAAYASAAEgIaavD_BwE [1.4.2019]

UNITY TECHNOLOGIES, 2018. Unity Technologies Acquires Game Hosting Division of Multiplay from GAME Digital, PLC Available: https://unity3d.com/company/public-relations/news/unity-technologies-acquires-game-hosting-division-multiplay-game [28.11.2018]

VÉRON, M., MARIN, O. and MONNET, S., 2014. Matchmaking in multi-player on-line games: Studying user traces to improve the user experience. Available: https://www.researchgate.net/publication/260086948_Matchmaking_in_multi-player_on-line_games_Studying_user_traces_to_improve_the_user_experience [20.5.2019]

Appendix

1. Feedback form in testing.

X-Craft usability test questionnaire– April 2019

Player information

What mobile device did you use?

Previous experience with games (how often do you play, what platform do you play on etc.)?

What country and city did you play in?

GEQ

In the following questions, answer the questions on how well they described your feelings. Mark your answer with X.

In-Game GEQ

	Not at all	Slightly	Moderately	Fairly	Extremely
I felt successful					
I felt bored					
I found it impressive					
I forgot everything around me					
I felt frustrated					
I found it tiresome					
I felt irritable					
I felt skilful					

I felt completely absorbed					
I felt content					
I felt challenged					
I had to put a lot effort into it					
I felt good					

Post-Game GEQ

	Not at all	Slightly	Moderately	Fairly	Extremely
I felt revived					
I felt bad					
I found it hard to get back to reality					
I felt guilty					
It felt like a victory					
I found it a waste of time					
I felt energised					
I felt satisfied					
I felt disoriented					
I felt exhausted					
I felt that I could have done more useful things					
I felt powerful					
I felt weary					
I felt regret					
I felt ashamed					
I felt proud					
I had a sense that I had returned from a journey					

Post-Game Questions

Did you experience any problems while playing the game?

Did you change the player settings and if so, what settings felt the most comfortable?

Free feedback: