jamk.fi

Development of online game prototype with Unity engine

Multiplayer solutions

Timo Jetsonen

Bachelor's thesis
May 2016
Technology, Communication and Transport
Degree Programme in Software Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Jetsonen, Timo	Bachelor's thesis	May 2016
		Language of publication: English
	Number of pages: 100	Permission for web publication: x

Title of publication

Development of online game prototype with Unity engine

Multiplayer solutions

Degree programme

Software Engineering

Supervisor(s)

Nelimarkka, Paavo and Hämäläinen Raija

Assigned by

-

Abstract

The objective was to create networking functionalities for a game prototype called Quantum Knight, by using two distinct implementation methods with the Unity Engine and comparing them against each other.

The implementation methods under investigation were Unity Engine's own Unity Networking and one of Unity's most popular third party networking plugins: Photon Unity Networking. The goal was to achieve at least nearly similar operational functionality with both implementation variants and to demonstrate hands-on how the functionalities were actualized. The project itself, the thesis subjects not included, was carried out with Timo Holopainen and it was not assigned by an outside employer.

As a result, two separate properly functioning lobby systems and gameplay mechanic implementations were accomplished. Firstly, they were evaluated by means of ten distinct criteria and given points based on the author's experience, after which the acquired statistical network data was analyzed. Due to tool constraints, the only comparable and trustworthy attribute obtainable from both Unity Profiler and Photon Stats Gui was the round trip time.

The point distribution resulted in a tie, indicating that both implementation methods excel in different key areas, while Unity Networking had overall higher round trip times. This presumably originated from the fact how the Unity servers are globally situated, or that the UNet packages being sent are considerably larger in size. Ultimately, the Photon implementation was chosen by the team to be in the final build of the prototype. The choice resulted directly from the reality of Photon's servers being currently more reliable along with its final outcome being more refined.

Keywords/tags (subjects)

Unity, Multiplayer, C#, Plugin, UNet, Photon, Prototype

Miscellaneous

Number of pages in Appendices: 48

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Jetsonen, Timo	Opinnäytetyö, AMK	Toukokuu 2016
	Sivumäärä 100	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi

Development of an online game prototype with Unity engine

Multiplayer solutions

Tutkinto-ohjelma

Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)

Nelimarkka, Paavo ja Hämäläinen, Raija

Toimeksiantaja(t)

-

Tiivistelmä

Työn tavoitteena oli toteuttaa verkkopeliominaisuudet Quantum Knight-peliprototyyppiä varten Unity-pelimoottorilla kahta erilaista toteutustapaa käyttäen sekä verrata niitä toisiinsa. Tarkasteltavana olevat toteutustavat olivat Unity-pelimoottorin oma Unity Networking sekä yksi Unityn suosituimmista kolmannen osapuolen verkko-ominaisuuksiin keskittyvistä liitännäisistä: Photon Unity Networking. Päämääränä oli saavuttaa lähestulkoon samankaltainen toiminnollisuus molemmilla toteutustavoilla sekä havainnollistaa, kuinka kyseiset toiminnollisuudet todellisuudessa saatiin aikaan. Projekti toteutettiin ilman ulkopuolista toimeksiantajaa.

Työn konkreettisena tuloksena saatiin aikaan kaksi erillistä, asianmukaisesti toimivaa moninpeliaulajärjestelmää sekä pelimekaanillista toteutusta. Näitä verrattiin toisiinsa kymmeneen erilaiseen arvosteluperusteeseen pohjautuen ja niiden pohjalta pisteyttäen, minkä jälkeen hankitut tilastolliset tiedot analysoitiin. Mittaustyökalujen, eli Unity Profilerin sekä Photon Stats Guin, rajoittuneisuudesta johtuen ainut verrattavissa oleva sekä luotettava vertailukohde oli edestakainen viive (englanniksi round trip time).

Vertailussa annetut pisteet jakautuivat lopputuloksena tasan molempien toteutustapojen välille, mikä puolestaan viittaa siihen, että molemmat toteutustavat kunnostautuvat eri osa-alueilla. UNet-toteutuksessa puolestaan mitattiin kaiken kaikkiaan korkeammat edestakaiset viiveet. Tämän todettiin johtuvat todennäköisesti siitä, miten Unityn serverit on globaalisti sijoitettu tai vaihtoehtoisesti siitä, että sen lähettämät paketit ovat kooltaan suurempia. Photon toteutus valittiin loppujen lopuksi prototyypin viimeiseen versioon sen viimeistellymmän lopputuloksen sekä vakaampien serverien vuoksi.

Avainsanat (asiasanat)

Unity, Moninpeli, C#, Liitännäinen, UNet, Photon, Prototyyppi

Muut tiedot

Liitteiden sivumäärä: 48

Contents

Figures

Tables

1 Introduction

As the years have passed by, ever since the development of Tennis for Two in 1958, multiplayer games have become more and more popular. As opposed to single player activities, which set the players against preprogrammed challenges or artificial intelligence controlled opponents, multiplayer games allow the players to interact with other individuals in partnership, competition or rivalry, providing them with social interaction. Typically, multiplayer games require players to share the resources of a single player system or to utilize networking technology in order to play together over greater distances. (Multiplayer video game 2016.)

While networking technologies have evolved and grown more advanced, so have multiplayer games. Since the rapid increase in the availability of the Internet in the 1990s, followed by the improvements in connection speeds in early 2000s, the "local only" features of multiplayer games have been steadily receding into the minority. Through their Internet connections, players are now able to socialize with even thousands of other individuals from around the world on the same server as they are in, in MMOGs, massively multiplayer online games. As socialization is an important aspect in modern gaming, the implementation of online features is something to be considered during the development of every major release, as well as with smaller independent games. (Online game 2016.)

In this thesis, two distinct multiplayer solutions have been investigated in order to implement online functionality into a game prototype by using Unity Engine. The implementation methods under investigation were Unity Engine's own Unity Networking and one of Unity's most popular third party networking plugins: Photon Unity Networking. The objective was to achieve at least nearly similar, appropriately operational functionality with both implementation variants in order to compare them against each other and to demonstrate hands-on how the functionalities were actualized. The development of the prototype was carried out with Timo Holopainen, who, for his part, focused on comparing object oriented programming with entity based model in Unity Engine, as the author concentrated on said networking functionalities. The project was not assigned by an outside employer.

To begin with, the structure, basic functionalities and main operating principles of Unity Engine are familiarized with in the theory section, followed by the elementary background knowledge concerning both implementation methods. The second half introduces the prototype project along with illustrative descriptions on how the actual network functionalities were accomplished, ultimately leading to the comparison of these two methods. In addition to the author's previous experience on using the Unity Engine for over a year, the basis of knowledge was mainly accumulated by researching and analyzing the online documentations of Unity Network and Photon.

Due to the fact that the author has been an enthusiastic gamer for over 20 years and has an interest in working in the game industry in the future, the driving motivation behind this project was to improve the author's own workmanship in this field, as well as get acquainted with unfamiliar technology. That being said, the author did not have prior experience whatsoever, concerning multiplayer functionality programming in any environment, nor with Unity Networking or Photon.

2 Unity

2.1 What is Unity?

Unity is a cross-platform game engine developed by Unity Technologies and used to develop 2D or 3D video games for PCs, consoles, mobile devices or web browsers. Popular especially among indie game developers, Unity boasts with over 4 million registered users around the world (Figure 1). In the year 2016 it dominates as the most popular global game development software. (Fast Facts 2016; Unity (game engine) 2016.)

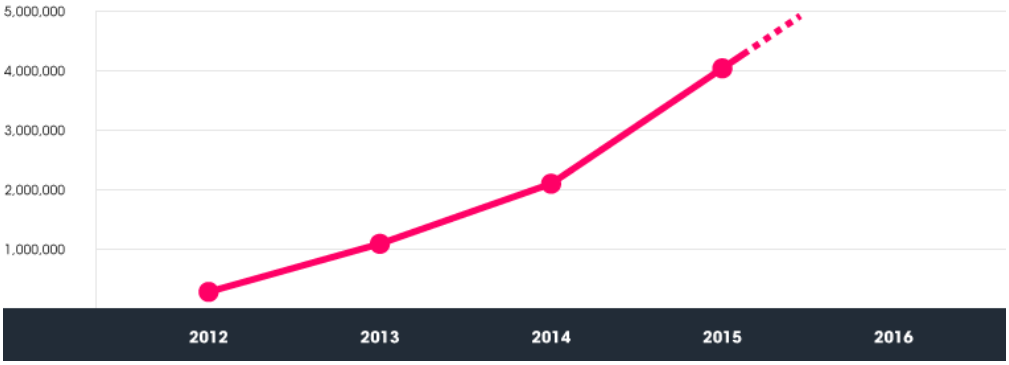


Figure 1. Registered Unity developers 2012-2015

Unity Technologies created said engine to match all the essential needs of game development and to offer the necessary tools for creating and publishing high end software. These built-in components and features include, but are not limited to, a complete physics engine, lighting system, animator and networking.

Originally released in 2005 only for Mac OS X and extended ever since, the Unity Engine reached its fifth version in 2015. With an emphasis on portability, multiplatform build options enable the user to select from over 20 different devices for publishing. This allows game developers more easily to target several different platforms with only minor changes into programming, such as player controls or other device specific optimization. (Unity (game engine) 2016; Unity – Game Engine 2016.)

Unity uses a so called component based model, where every function is its own component. This makes it easy to create reusable components which hand in hand with

the cross-platform options and built-in components, save valuable time and money for developers.

The editor itself can easily be extended by third-party plugins and user generated content can be found in and downloaded from the Asset Store. These user generated contents can range from sound effects and scripts to 3D models and beyond.

A thriving community has evolved around Unity Engine. Community forums are packed with tips or straight out solutions for different kinds of topics, while https://unity3d.com/learn offers comprehensive tutorials for beginners.

Unity personal edition is available for download free of charge, while the professional edition costs 75$/month. Professional edition subscribers gain access to view additional statistical data and perform in-depth analysis of their game, in addition to various customizing options, bug handling and other pro tools. Personal edition may not be licensed or used by an entity with annual gross revenues or budget in excess of $100,000. (Get Unity 2016.)

2.2 Unity Editor

The main interface of Unity Editor is made up of various tabbed windows which can be rearranged, grouped, detached and docked. Each of these windows, or views, have their own purposes and functionality in Unity game development. The most important default views are: Scene, Game, Hierarchy, Project panel and Inspector. Furthermore, Console window may be viewed for errors, warnings and other messages generated by the Editor or user, to aid with debugging.

In this chapter the most essential views of the Editor will be discussed in some detail. The main interface of the Editor can be examined in Figure 2.

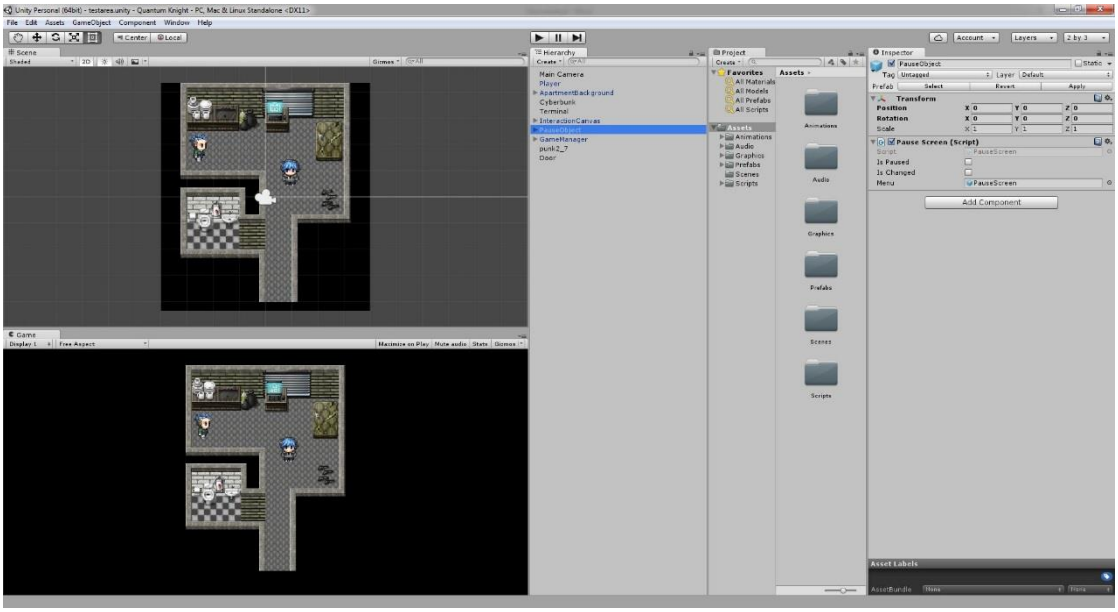


Figure 2. Main interface of Unity Editor

2.2.1 Toolbar

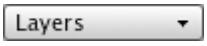
Although the layout of the editor can easily be modified by user and extended by plugins, the toolbar is the only part of Unity interface that cannot be rearranged. Always founded on top (see Figure 3), the toolbar holds a set of important tools each relating to different parts of the Editor.

Figure 3. Editor toolbar

Transform Tools – Move camera, Move object, Rotate, Scale and Rect Tool.

Transform Gizmo Toggles – Affects coordinate system and pivot points.

Play/Pause/Step Buttons – Used to run the current scene.



Layers Drop-down – Handles visibility and locking of layers.

Layout Drop-down – For choosing layout presets.

2.2.2 Scene view

One of the most important views in the Editor is the Scene view – a visual representation of a game world or level (see Figure 4). It enables the maneuvering, manipulation and positioning of all the objects and assets listed in Hierarchy, thus creating a physical space that players may explore and interact with. (Menard 2011, 23.)

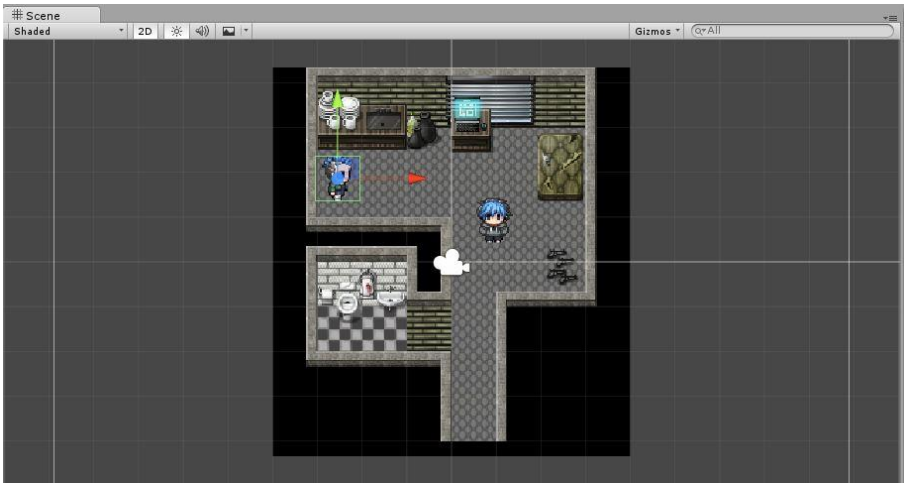


Figure 4. Scene view

When an object is clicked in the Scene view, it becomes selected and the Inspector will be updated with the object's appropriate data. Using the aforementioned tools found in the toolbar, the user may now manually alter the object's position, rotation and scale. These manipulations are called transforms. Corresponding values can be altered through the Inspector as well. (Menard 2011, 29.)

The Translate tool, shown in Figure 5, moves the selected object's position around the scene, either along one of the three axes or freely in space. The Rotate tool rotates the object along selected axis, while the Scale tool allows the object to be scaled either uniformly in all dimensions or by single axis. The desired axis may be selected by clicking on the colored handles. (Menard 2011, 30-32.)

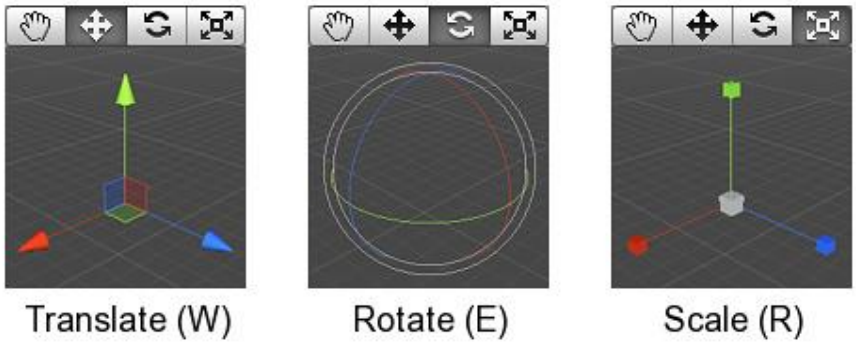


Figure 5. Transform tools and hotkeys

The point of view can be changed by holding down either left or right mouse button and dragging the cursor around. Clicking the 2D button located in the upper left corner of the scene view, changes between 2D and 3D perspective.

2.2.3 Game view

Mainly used for previewing and testing, the Game view (see Figure 6) renders out the game exactly as it would be in the built version. Pressing the Play button on the toolbar makes the editor activate this view and run the game. Any changes made during this time will not be saved, however, changing values when the game is running might help while testing.

Like the Scene view, the Game view also has its own Control Bar. The first area, the Aspect drop-down menu, changes the aspect ratio of the Game view on the fly, even while it is currently playing. Clicking the Maximize on Play toggle button will expand Game view to take up the entire editor view when playing. Pressing the Gizmos button lets the user choose which gizmos will be shown in the Game view. Lastly, the Stats button on the Control Bar will bring up the Render Statistics page which shows

useful optimization information about the game, like FPS, or Frames per Second. (Menard 2011, 38-39.)

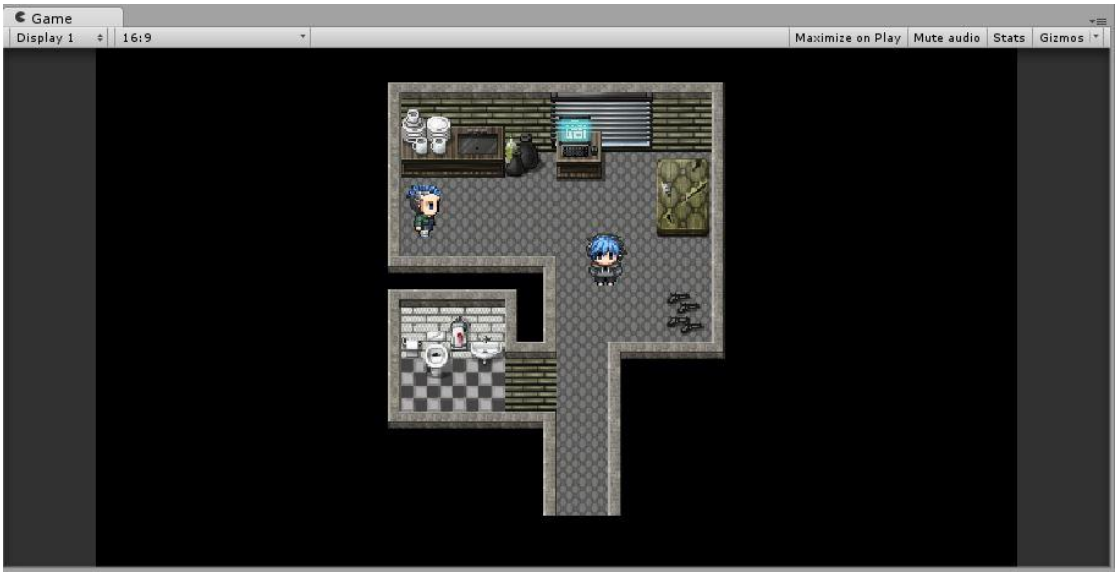


Figure 6. Game view

2.2.4 Hierarchy

The Hierarchy view, demonstrated in Figure 7, contains every GameObject in the current scene, updating with each change as the user adds or removes objects. Some of these could be direct instances of asset files like 3D models, and others might be instances of Prefabs. Each instance of an object will be listed individually, making good naming conventions especially important. (Unity – Manual: Hierarchy 2016.)

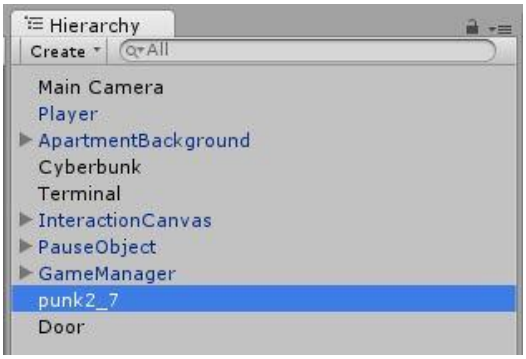


Figure 7. Hierarchy

Parenting objects together in the Hierarchy view might help with organizing and moving a large amount of objects at once. When parenting objects together, unrelated objects are linked together under a single object, the parent. All the other objects under this parent are called its children. All the children will inherit the parent's data, however, they still can be edited independently of each other and the parent object. (Menard 2011, 19-20.)

Selecting an object in the Hierarchy and deleting it will remove the object from the current scene in the game but not from the project's Assets folder.

2.2.5 Project browser

All of the current project's files are organized and arranged under the Project browser. The left panel of the browser, as seen in Figure 8, displays everything in a hierarchical, folder like structure, exactly how the folders and their contents can be found on the hard drive. Using the small triangle expands or collapses the folder, displaying any nested folders it contains. When a folder is selected from the list by clicking, its contents will be shown in the panel to the right as icons. (Unity – Manual: ProjectView 2016.)

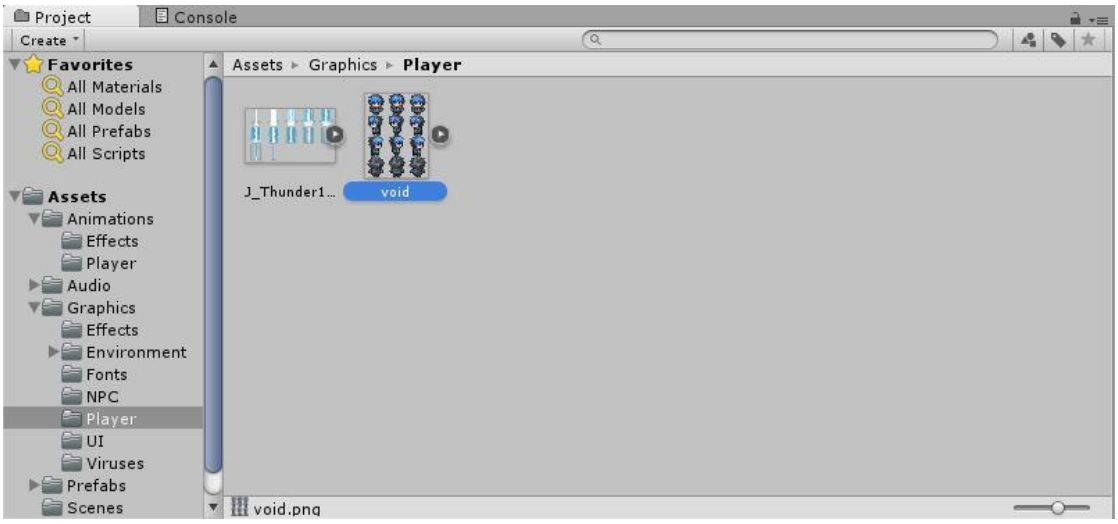


Figure 8. Project browser

Assets can be searched using the browser's search feature that is especially useful for locating files in large or unfamiliar projects. A basic search will filter assets according to the text typed in the search box. Files can also be opened and edited directly from

within the Project browser. Double-clicking will open the file in its default editor, for example a Photoshop file. After completing modifications for the opened file and saving, Unity will reimport the saved file automatically. (Menard 2011, 17.)

The Create menu, located at the left side of the project toolbar allows the creation of new assets and folders. These options for new assets include scripts, materials, and animations and so on. Similar menu can also be opened by right clicking inside the browser panel and choosing create.

2.2.6 Inspector

Through the Inspector view, all of the components and values present in the currently selected GameObject or Asset, can be observed and edited. By adding or removing components and editing their values, the Inspector is used to modify the GameObjects functionality. If several GameObjects with common components are selected at the same time, the corresponding values can be multi-edited simultaneously. The values supplied will be copied to all selected objects. The components attached to a player character can be seen in Appendix 1. (Unity – Manual: Using the Inspector 2016; Unity – Manual: Editing values 2016.)

When GameObjects have custom script components attached to them, the public variables in that script are also shown in the Inspector and can be edited exactly like the properties of Unity's built-in components. Components are added to GameObjects by simply pressing the Add Component button in the Inspector. The GameObject can be deactivated by unticking the checkbox from the left side of the GameObjects name. Underneath the GameObject name, the drop-down menus for Tag and Layer can be found, explained in chapter 2.3. (ibid.)

On the right side of the component present in a GameObject, there is a question mark that leads to the Unity reference page of that particular component. The gear beside the question mark can be used to reset all the values of the component, copying, removing the component completely from the GameObject or to move it up and down in the Inspector view. (ibid.)

2.2.7 Build Settings

When testing outside the editor is needed or a game is ready for publishing, the build settings window can be accessed by choosing “Build Settings” from the File menu. The options in this window (Figure 9) allow the choice of a target platform and various adjustment settings for the building process and the end product.

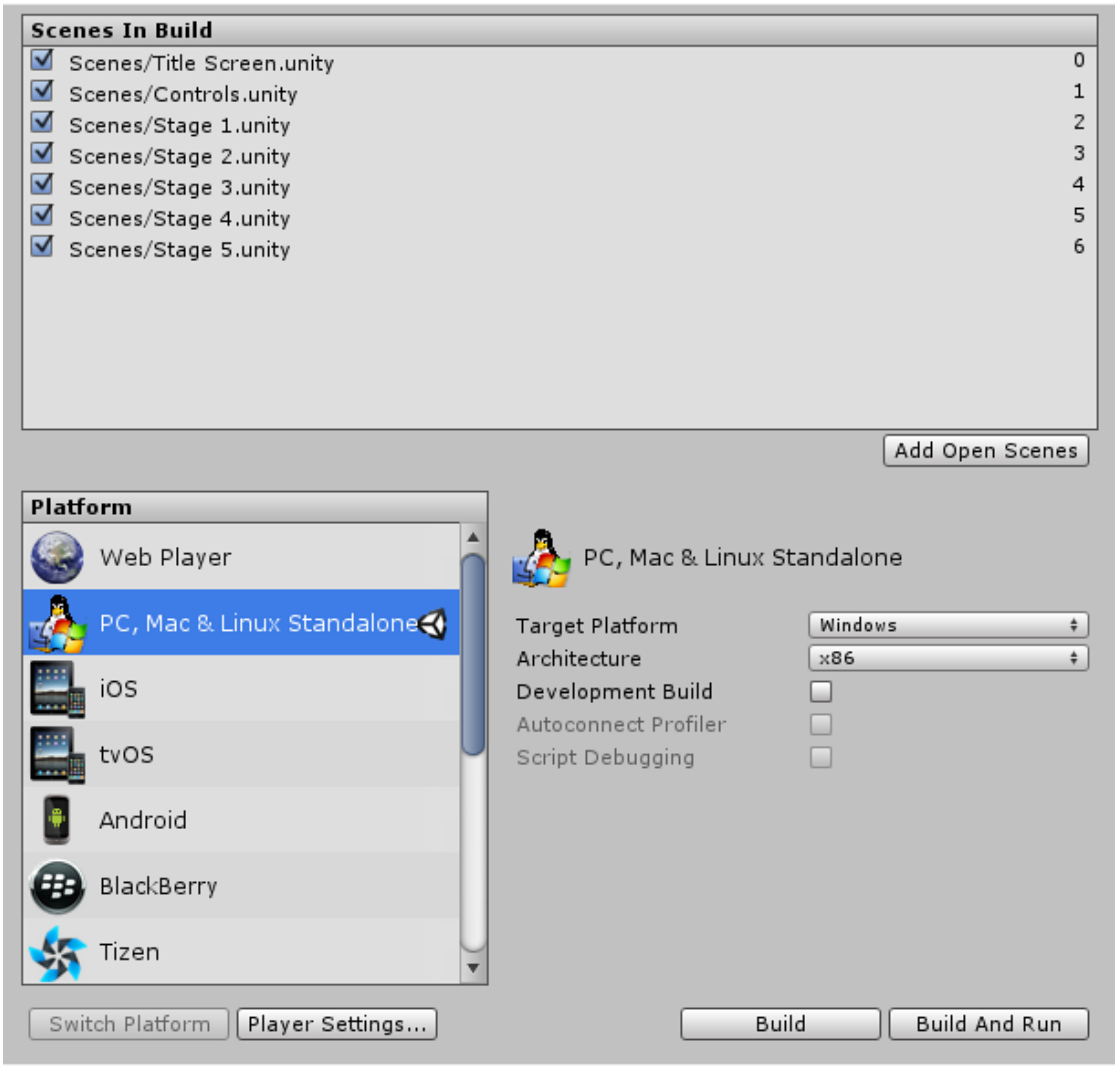


Figure 9. Build settings

“Scenes in Build” part of the window shows the added scenes from the current project that will be included into the build. Scenes can be added by pressing the Add Open Scenes button or by dragging scene assets into this window. Unticking a scene excludes it from the build without removing it from the list. (Unity – Manual: BuildSettings 2016.)

Graphical quality settings for the build can be found in Edit > Project Settings > Quality. The choice between saved quality level presets, designed by the developer, are given to the player when starting the game. Once build settings have been specified, clicking “Build” or “Build and Run” will create a runnable build of the current project on the specified platform.

2.3 Main operating principles of Unity

The Unity Engine uses a component based model in the creation of game worlds and their functionalities. In practice, components are functional pieces of GameObjects, which in turn represent characters, props, scenery and so on, in scenes.

2.3.1 Component

A component can be thought as a smaller piece of a larger machine. They are the functional pieces of every GameObject and in order to execute said functionalities, must always be attached to objects. Components carry out their functionalities through combination of methods and variables. Adding, deleting and editing of a component and its values are handled via the Inspector view. In Figure 10, a Rigidbody component has been attached to a GameObject. (Unity – Manual: UsingComponents 2016.)

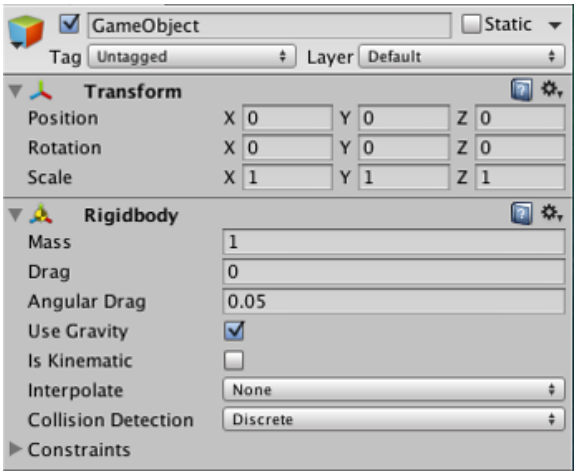


Figure 10. GameObject with a Rigidbody component

In addition to built-in components, such as physics, custom components can be created by writing scripts. When a script is attached to a GameObject it appears in the Inspector just like a component. This is because scripts compile as a type of component and are treated as such by the engine. (Unity – Manual: CreatingComponents 2016.)

The components in a game can be either reusable or made to accomplish a specific task. For example in a vertical space shooter game, a script that handles shooting of a certain enemy, can be used in all instances of the same enemy. Whereas a script used for main character movement might only be used for that particular purpose.

2.3.2 GameObject

GameObjects are fundamental pieces in Unity. Although they do not accomplish much by themselves, they act as containers for components which implement the real functionalities. In Unity Engine, levels and game worlds are composed from multitudes of these objects. (Unity – Manual: GameObject 2016.)

By default, all GameObjects contain only a single Transform component, which defines its position, orientation and scale in space. When different kinds of component combinations are added, they can be made up into characters, lights, trees, sounds, game logic managers or anything the user is willing to build. Components attached to completely different GameObjects are able to communicate between each other. (ibid; Unity – Manual: GameObjects 2016.)

For every GameObject a name, tag and layer can be set according to the user's needs. Stored objects are saved as prefabs.

2.3.3 Asset

Assets are representation of items that can be used by the developer in games or projects. An asset may come from a file created outside of Unity, such as a 3D model, an audio file, an image, or any of the other types of files that Unity supports. There are also some asset types that can be created within Unity, such as an Animator Controller, an Audio Mixer or a Render Texture. Assets created outside of Unity must be imported either by saving or copying them directly into the assets folder of current

project. Unity will automatically detect files as they are added or modified. (Unity – Manual: AssetWorkflow 2016.)

Importing and exporting Unity Asset packages is a simple way to share and re-use projects and asset collections. Packages are collections of files and data from Unity projects, or elements of projects, which are compressed and stored in one file, similar to zip files. The package maintains its original directory structure when it is unpacked, as well as meta-data about assets such as import settings and links to other assets. Items on the Unity Asset Store are also supplied in packages. All imported project assets can be found in the project view. (Unity – Manual: AssetPackages 2016.)

2.3.4 Prefab

Unity has a Prefab asset type that allows storing of GameObjects complete with components and properties, in contrast for having independently editable multiple objects of the same kind in a scene. All modifications made to a prefab asset are immediately reflected in all instances (copies) produced from it, however, overriding components and settings for each instance individually is also possible if needed. Prefabs are particularly useful when instantiating recurring objects such as bullets, enemies and collectibles. (Unity – Manual: Prefabs 2016.)

Prefabs are created by dragging existing GameObjects from hierarchy to project view. The prefab acts as a template from which new object instances can be created in the scene. Objects created as prefab instances will be shown in the hierarchy view in blue text. Like other GameObjects, prefabs are also edited in the inspector view. (ibid.)

Copies of prefabs can also be instantiated through scripting at runtime. In the following code, 10 copies of a prefab are being instantiated each 2 units apart.

```
// Instantiates 10 copies of prefab each 2 units apart from each other

using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour {
    public Transform prefab;
    void Start() {
        for (int i = 0; i < 10; i++) {
            Instantiate(prefab, new Vector3(i * 2.0F, 0, 0), Quaternion.identity);
        }
    }
}
```

2.3.5 Layer

Most commonly used by Cameras to render only certain parts of the Scene (culling mask) or by Lights to illuminate specific areas, Layers can also be used to create collisions or to ignore Raycasting collisions selectively (layerMask). In other words, layers are a way to classify different types of objects from each other in Unity. (Unity – Manual: Layers 2016.)

In 2D games, sorting layers are used in conjunction with sprite graphics. The “sorting” refers to the overlay and drawing order of different sprites. Menu for organizing sorting layers in Figure 11. (Unity – Manual: TagManager 2016.)

Figure 11. Sorting Layers - menu

Layers and sorting layers are applied to GameObjects in the Inspector view. Through the drop-down menu, new layers and sorting layers can be created by the user.

2.3.6 Tag

Tags are words, or Strings that can be issued to GameObjects and used to identify them for scripting purposes. For instance, defining “Player” and “Enemy” Tags for player-controlled characters and non-player characters respectively; a “Collectable”

Tag could be defined for items the player can collect in the Scene. In scripts, they save manual addition of GameObjects to exposed public properties and offer a useful timesaver if the same script code is being used in several GameObjects, for example when using TriggerColliders to detect collisions between player and enemies. Example code of a trigger collision between a bullet and an enemy is presented below. (Unity – Manual: Tags 2016.)

```
void OnTriggerEnter2D(Collider2D other)
{

    if (other.tag == "Enemy")
    {

        other.GetComponent<EnemyShield> ().CalculateDamageTaken (damage, weaponId);

    }

}
```

Tags are applied to GameObjects in the Inspector view. From the same drop-down menu, new tags created by the user are added.

2.3.7 Scene

Each scene file, with all the GameObjects it contains, can be thought as a unique level or a title screen, for example. Scenes encase all the information of what will happen when a game is being played. In each Scene, environments, decorations, characters and gameplay mechanics define a game level by level.

Scenes may be edited one at a time in the editor or by opening additive scenes in the Hierarchy view, to allow multi scene editing. By default, all GameObjects are destroyed when switching between scenes, however, they can be set not to, by using DontDestroyOnLoad () – method in scripts. (Unity – Manual: MultiSceneEditing 2016.)

2.4 Scripting

While classic Object Oriented Programming (OOP) can be used, the Unity workflow highly builds around the structure of components and custom component scripting. The fine line between programming and scripting has become even more blurred

over the years. Essentially, scripting or writing scripts, is programming within a program whereas programming is writing software that runs independent of an exterior program. (Porter 2013.)

In line with the Unity Engine's emphasis on usability and flexibility, it natively supports three different scripting languages: C#, Boo (a dialect of Python) and JavaScript, also referred to as UnityScript. A single project can contain scripts written in whichever of the three languages. While possible, but not recommended, scripts written in different languages can access each other's functions and GameObjects can have them attached and running, all at the same time. As mentioned before, attached scripts behave like components in GameObjects. Unity also supports various scripting environments like: Visual Studio, MonoDevelop, UnityDevelop and SubEthaEdit to name a few. Below, the initial contents of a C# script file are displayed. (Menard 2011, 156 – 158.)

```
using UnityEngine;
using System.Collections;

public class Collision2D : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

One of the greatest differences between traditional OOP and Unity scripting is, that constructors are not used in a similar fashion, when a class inherits Unity's MonoBehaviour. Instead, Awake () and Start () functions of MonoBehaviour are used for initialization. MonoBehaviour is the base class every script derives from; except when a script contains network functionality, NetworkBehaviour is used. NetworkBehaviour is discussed in detail in chapters 3 and 5. (Menard 2011, 157.)

JavaScript is automatically derived from MonoBehaviour, while C# and Boo have to be explicitly derived from it. When MonoBehaviour is inherited in a class, it is able to use all basic features and functions of Unity Engine. Most commonly used and the most important functions are: Awake (), Start (), Update (), FixedUpdate (), OnGUI ()

and functions concerning collisions. Appendix 2 introduces several important MonoBehaviour functions and their uses. The execution order of functions during a script's lifetime is as follows: (Unity – MonoBehaviour 2016; Unity – Manual: ExecutionOrder 2016.)

- Editor
  - Reset
- Initialization
  - Awake
  - OnEnable
  - Start
- Physics
  - FixedUpdate
  - yield waitForFixedUpdate
  - OnTrigger
  - OnCollision
- Input Events
- Game logic
  - Update
  - yield waitForSeconds
  - StartCoroutine
  - Animation update
  - LateUpdate
- Scene rendering
- Gizmo rendering
- GUI rendering
  - OnGUI
- End of frame
  - yield waitForEndOfFrame
- Pausing
  - OnApplicationPause
- Disable
  - OnDisable
- Decommissioning
  - OnApplicationQuit
  - OnDisable
  - OnDestroy

Appendix 3 demonstrates simple 2D collision and Coroutine scripts, while Appendix 4 shows an example of a player movement script written in C# using MonoDevelop. It handles the movement of a main character according to user input as well as animation control and raycasting to detect if player object collides with interactable objects.

2.5 Asset Store

Whether you're a programmer, game designer, texture artist or 3D modeler, you're welcome to share your creations with everybody in the Unity developer community! (Sell Assets 2016.)

Originally launched in November 2010, the Unity Asset Store acts as an online marketplace for Unity users and other developers or artists alike to sell project assets to each other. With over 40,000 asset packages available in various categories and prices ranging from free to over $1000, Asset Store offers a potential chance for saving resources during game development. Publisher of an asset receives 70% cut of their set price in the store, while Unity takes 30% off of each sale. Free tutorials, sample projects and standard assets are also available in courtesy of Unity Technologies. (Sell Assets 2016; Stats Monitor – Asset Store 2016.)

The store may be accessed through a simple interface built into the Unity Editor (Figure 12) or a web browser. Purchased assets are downloaded and imported directly into your project. Three most popular asset packages in mid-2015 were NGUI: Next-Gen UI, Playmaker and "Unity-Chan!"-model, excluding Unity Technologies assets. (Stats Monitor – Asset Store 2015; Unity – Manual: Importing from the Asset Store 2016.)

Figure 12. Main view of Asset Store

3 Multiplayer solutions for Unity game development

3.1 Unity Networking

Released with update 5.1 and recently coming out of beta, the UNet, or simply Unity Networking, is designed to replace the old networking system. It offers a new set of highly customizable components and tools for creating real time, networked games with Unity. It can be divided into two layers.

The Lower Level API, called the Transport Layer, is a thin layer working on top of the operating system's sockets-based networking. The LLAPI targets experienced network programmers and users, who are building network infrastructures or advanced multiplayer games. This chapter mainly concentrates on the High Level API, which gives access to commands that cover most of the common requirements for multi-user games without the need of worrying about the "lower level" implementation details. (Unity – Manual: Networking Overview 2016.)

3.1.1 The High Level API

Built on top of the lower level transport real-time communication layer, the High Level Application Programming Interface (HLAPI) is a system for building multiplayer capabilities for Unity games. The HLAPI contains a new set of networking commands built into Unity within a new namespace: UnityEngine.Networking and it handles many of the common tasks that are required for multiplayer games. While the transport layer supports any kind of network topology, the HLAPI is a server authoritative system; although, it allows one of the participants to be a client and the server at the same time, so no dedicated server process is necessarily required. Working in conjunction with the internet services, this allows multiplayer games to be played over the internet without major effort from developers. The layers which add functionality to the HLAPI are displayed in Figure 13. (Unity – Manual: The High Level API 2016.)

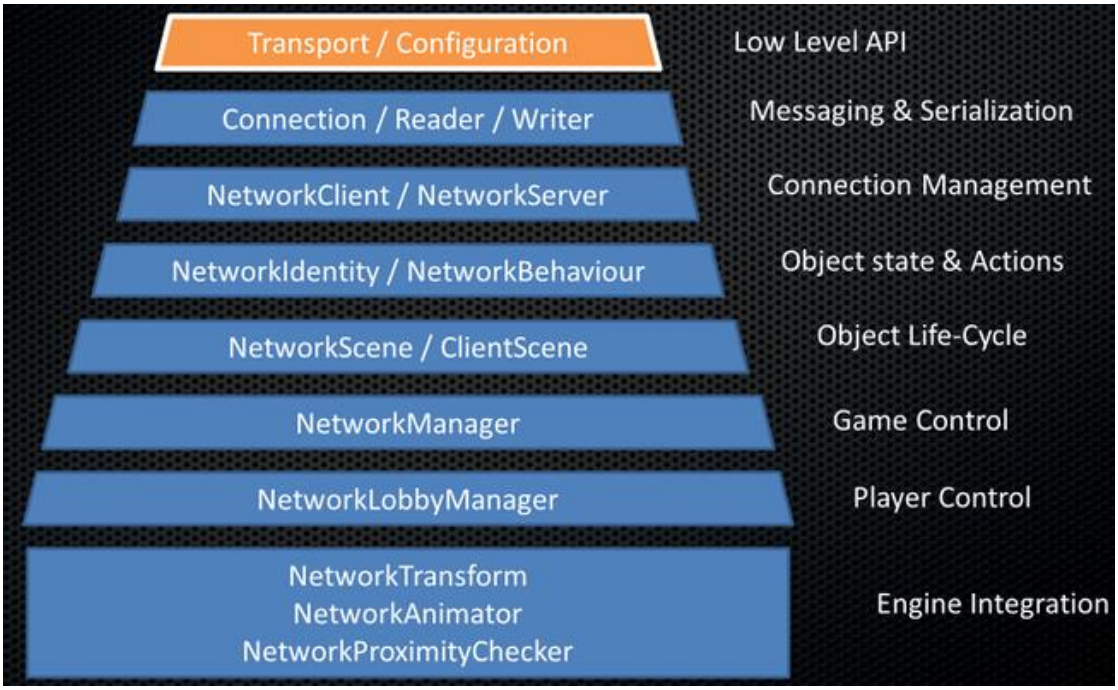


Figure 13. HLAPI layers

In the Unity Networking system, games have a Server and multiple Clients. When there is no dedicated server to be found, one of the clients is able to play the role of the server, or “host”. The host is a server and a client in the same process. The host uses a special kind of client called the LocalClient, while other clients are RemoteClients. The LocalClient communicates with the (local) server through direct function calls and message queues, since it is in the same process, sharing the scene with the server. RemoteClients communicate with the server over a regular network connection. (Unity – Manual: Network System Concepts 2016.)

3.1.2 NetworkBehaviour

The NetworkBehaviour base class is, as a matter of fact, an “extension” of the MonoBehaviour class and scripts that inherit it have also access to MonoBehaviour functions. Designed to work with objects containing the NetworkIdentity component, these special scripts are able to perform HLAPI functions such as Commands, ClientRPCs, SyncEvents and SyncVars, and they should be inherited by scripts which contain Networking functionality. (Unity – Manual: NetworkBehaviour 2016.)

Using the networking features it provides, the user is able to synchronize member variables from server to clients or perform virtual, overridable callback functions for various network events, for example. Since the HLAPI is a server authoritative system, usage of NetworkBehaviour Commands is the definitive way for clients to request to do something on the server while Client RPC calls are used by server objects to cause things to happen on client objects. Simplified interaction between client and server is demonstrated in Figure 14. Appendix 5 introduces the most notable NetworkBehaviour functions and variables, as well as their uses. (ibid.)

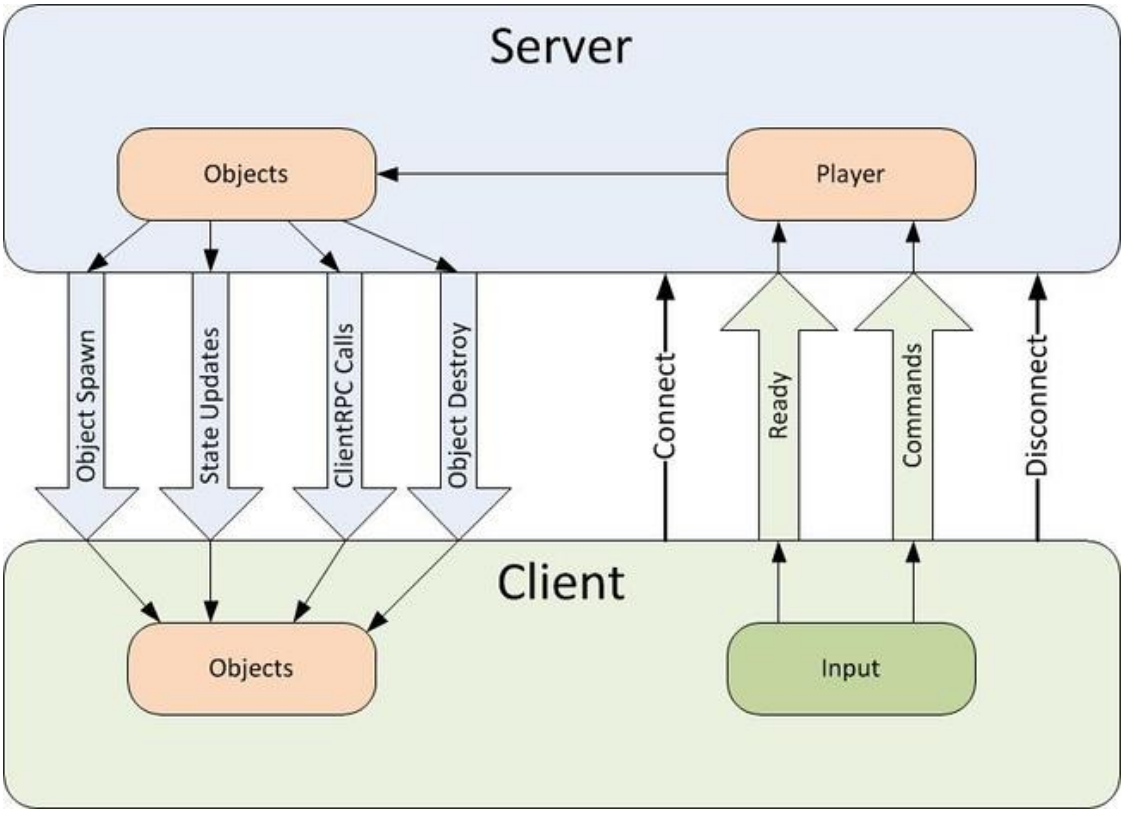


Figure 14. UNet function directions

3.1.3 NetworkManager

The NetworkManager can be thought as the core controlling component of a multiplayer game. Usable even completely without scripting, it manages the network states of a game, as its name suggests. In advanced games scripting is naturally a must, in order to access network functions. When added into a GameObject of the users liking, the inspector for the NetworkManager in the editor allows configuration and controlling of several features related to networking. These features include:

- Game state management
- Spawning management
- Scene management
- Debugging information
- Matchmaking
- Network customization

Another useful component to be used alongside with the NetworkManager is the NetworkManagerHUD which supplies a simple, default user interface at runtime that allows the network state to be controlled by the user. It is not a substitute for a proper interface found in real games, however, it might prove useful when getting started with Unity Networking. In Figure 15, both components have been attached into a GameObject.

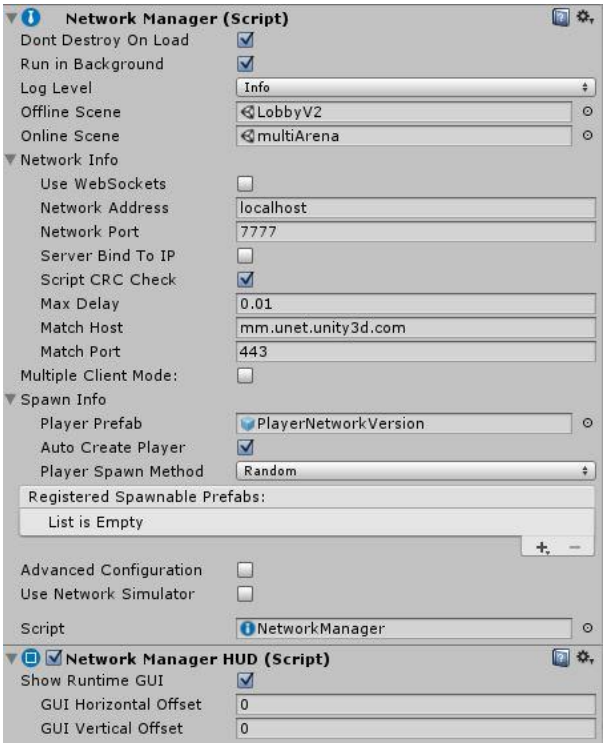


Figure 15. NetworkManager and HUD

A UNet multiplayer game can run in three modes - as a client, as a dedicated server, or as a Host, which is both a client and a server at the same time. The NetworkManager holds methods for entering each of these modes. StartClient (), StartServer (), StartHost () and their counterparts for stopping are all available to be invoked from input handlers or from custom user interfaces, in addition to numerous other functions. (Unity – Manual: Using the NetworkManager 2016.)

By default, there are two slots for scenes on the NetworkManager inspector, the offlineScene and the onlineScene. Dragging scene objects into these slots activates networked scene management. When a server or host is started, the online scene will be loaded and it will then become the current network scene. Any clients connecting to that particular server will be instructed to also load that scene. When the network is stopped, by stopping the server or host, or by a client disconnecting, the offline scene will be loaded. This allows the game to automatically return to an offline menu scene when disconnected. (ibid.)

For managing the spawning of networked prefab objects, the NetworkManager has slots for both the main player, as well as for any other objects to be spawned on the server. When a player prefab is set, it will automatically be spawned from that prefab for each user in the game. This applies to the local player on a hosted server, and remote players on remote clients. The other dynamically spawnable prefab objects must be added into the Registered Spawnable Prefabs – list in order to register them with the ClientScene or by using the ClientScene.RegisterPrefab () functions. All spawnable prefabs that hold networking functions must have a NetworkIdentity component attached to themselves. (ibid.)

Starting positions for players can be set by adding a NetworkStartPosition component to an object in the play scene. The NetworkManager automatically registers the position and orientation of the object as a start position. When a client joins the game and a player is added, the player object will be created at one of the starting positions with the same transform values. (ibid.)

The NetworkManager runtime UI and NetworkManager inspector allow interactions with the UNet matchmaker service. The function StartMatchmaker () enables matchmaking, and populates the NetworkManager.matchmaker property with a NetworkMatch object. The default match host for matchmaking is the Unity Technology “mm.unet.unity3d.com” – server. (ibid.)

3.1.4 NetworkLobbyManager

The NetworkLobbyManager is a specialized NetworkManager component that provides a staging area for players to join before playing the actual game. In this area,

often called the lobby, the players may be able to change settings and set themselves ready before the game starts. As it is derived from the NetworkManager, it implements many of the virtual functions provided by the NetworkManager class, in addition to its own. (Unity – Manual: Multiplayer Lobby 2016.)

Some of the features provided by the NetworkLobbyManager are listed below:

- Limit on number of players that can join
- Support for multiple players per client with a limit on number of players per client
- Prevent players from joining game in-progress
- Pre-player ready state, so that game starts when all players are ready
- Per-player configuration data
- Re-joining the lobby when the game is finished
- Virtual functions that allow custom logic for lobby events
- A simple user interface for interacting with the lobby

In similar fashion to the NetworkManager, the lobby component has two slots for scenes by default. One for the lobby scene and another for the play scene. The key differences reveal themselves when player objects come into question. There are two kinds of player objects - each which has a prefab slot in the NetworkLobbyManager. (ibid.)

A LobbyPlayer is created when a client connects and joins the lobby or when a new player is added. One prefab instance of LobbyPlayer is created for every player in the lobby and it persists to exist until that client disconnects from the server. It handles the commands given while in the lobby scene and holds the ready flags for all players. The LobbyPlayer prefab must have a NetworkLobbyPlayer component attached to itself; the NetworkIdentity is added automatically after attaching. (ibid.)

In turn, the GamePlayer is created from a prefab when the game scene has loaded. It handles commands given by the players during the game and gets destroyed when re-entering the lobby scene or getting disconnected. This prefab must have a NetworkIdentity component attached. (ibid.)

Below are some virtual callback functions of the LobbyPlayer, which are used for creating custom lobby behaviour.

```
public virtual void OnClientEnterLobby();

public virtual void OnClientExitLobby();

public virtual void OnClientReady(bool readyState);
```

The function OnClientEnterLobby gets called on the client when the game enters the lobby. This happens when the lobby scene starts for the first time, and also when returning to the lobby from the game-play scene. The OnClientExitLobby is called on the client when the game exists the lobby. This happens when switching to the play scene. OnClientReady is called on the client when the ready state of that player changes. (ibid.)

When the minimum amount of ready flags given by players (represented by the "minimum players" field in the manager) have been set, the manager is able to transit from the lobby scene to the game scene. NetworkLobbyPlayer.SendReadyToBeginMessage () function can utilized to tell the server that this player is ready for the game to begin. The NetworkLobbyManager component and the aforementioned fields are displayed in Figure 16.

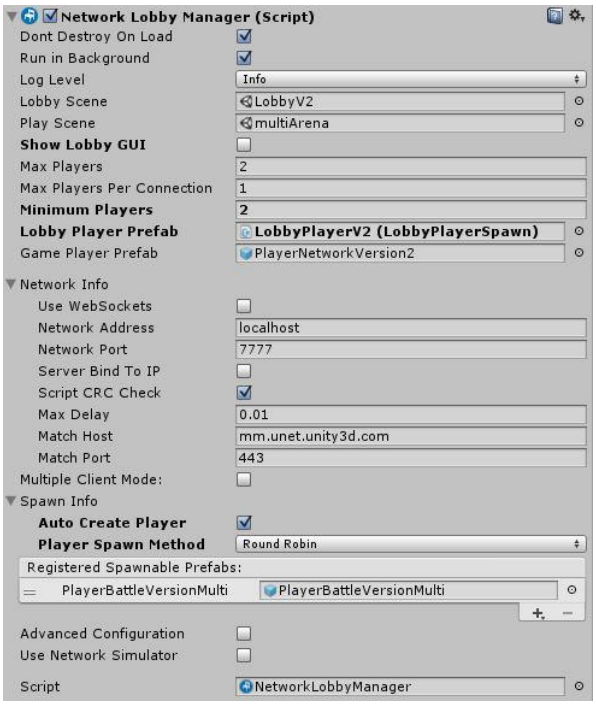


Figure 16. NetworkLobbyManager component

3.1.5 Essential Networking Components

Along with the two managers and HUD, Unity Networking offers several other components to be used in multiplayer applications. In this chapter the most crucial components and their uses and functionalities are explained shortly to give a better understanding about the workings of UNet.

NetworkIdentity is at the heart of the new Networking system. When attached to a GameObject, it makes the networking system aware of the object's presence and assigns unique NetworkInstanceId for it when spawned. There might be multiple objects instantiated of a particular object type, and the network ID is used to identify which object, for example, a network update should be applied to. The NetworkIdentity component has two visible checkboxes in the inspector (Figure 17). The "Server Only" checkbox sets off a flag that will ensure this particular object will not be spawned or enabled on clients. The "Local Player Authority" checkbox allows the object to be controlled only by the client that owns it and have authority over it. Local player authority is used mainly to handle player commands, so that only the appropriate object has authority over controls. (Unity – Manual: NetworkIdentity 2016.)

Figure 17. NetworkIdentity

NetworkTransform component synchronizes movement across the network for all spawned GameObjects. It takes authority into account, so local player objects synchronize their position from the client to server, then out to other clients, whereas other objects with server authority will be synchronized from the server to clients. In order to function properly, this component requires NetworkIdentity to be attached into the same object. (Unity – Manual: NetworkTransform 2016.)

NetworkStartPosition, as mentioned in chapter 3.1.3, is used by the Network(Lobby)Manager when spawning player objects. It is registered automatically as a starting position by the NetworkManager when fixed into a GameObject. When positioned in a scene like the user wishes, player objects are spawned after a game

level has been loaded, at one of the starting positions with the same transform values by default. To quickly alter how starting positions are issued, the Player Spawn Method can be set to Random or Round Robin from the manager components. (Unity – Manual: Using the NetworkManager 2016.)

NetworkProximityChecker is a component which relies on physics to calculate and control the visibility of objects for network clients based on proximity. It has some configurable parameters (Figure 18), such as Vis Range and Vis Update Interval. Objects further away than Vis Range will not be visible to a player, and each player's set of visible objects will be recalculated every Vis Update Interval seconds. In order to check proximities, objects must have colliders. (Unity – Manual: Object Visibility 2016.)

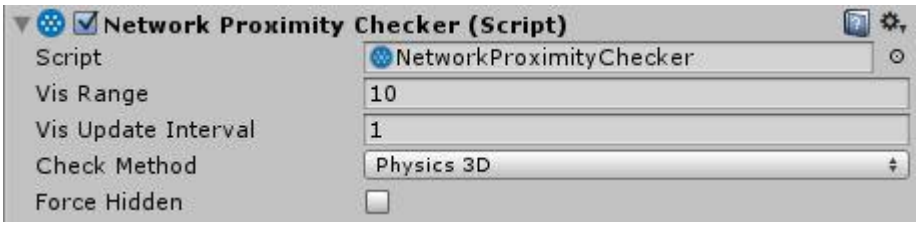


Figure 18. NetworkProximityChecker

NetworkAnimator is a component which synchronizes animation states for networked objects. Although it behaves very similarly as the regular Animator, it has some properties of its own. To name a few, the local authority controls whether a certain object is animated on the local client, and SetParameterAutoSend sets whether an animation parameter should be auto sent. (Unity – Manual: NetworkAnimator 2016.)

NetworkDiscovery allows Unity applications to find each other in a local area network by broadcasting their presence and by listening for other broadcasts. NetworkDiscovery is able to run in a server mode by calling StartAsServer, where it broadcasts to other computers on the local network, or in a client mode by calling StartAsClient where it listens for broadcasts from a server. (Unity – Scripting API: NetworkDiscovery 2016.)

3.1.6 Objects and authority in UNet

In addition to the isLocalPlayer flag and the local authority property of player objects, which are provided by NetworkIdentity to prevent command invoking on another player, starting with Unity release 5.2, it is possible to have client authority over non-player objects. Non-player objects with client authority can send commands just like the players, although these commands are run on the server instance of the object. Setting authority over these objects can be achieved by spawning the object using NetworkServer.SpawnWithClientAuthority or by using NetworkIdentity.AssignClientAuthority with the network connection of the client to take ownership. Nevertheless, objects that should not be controlled by players, like enemies, are recommended to be spawned on the server with server authority, as shown in Figure 19. (Unity – Manual: Network System Concepts 2016.)

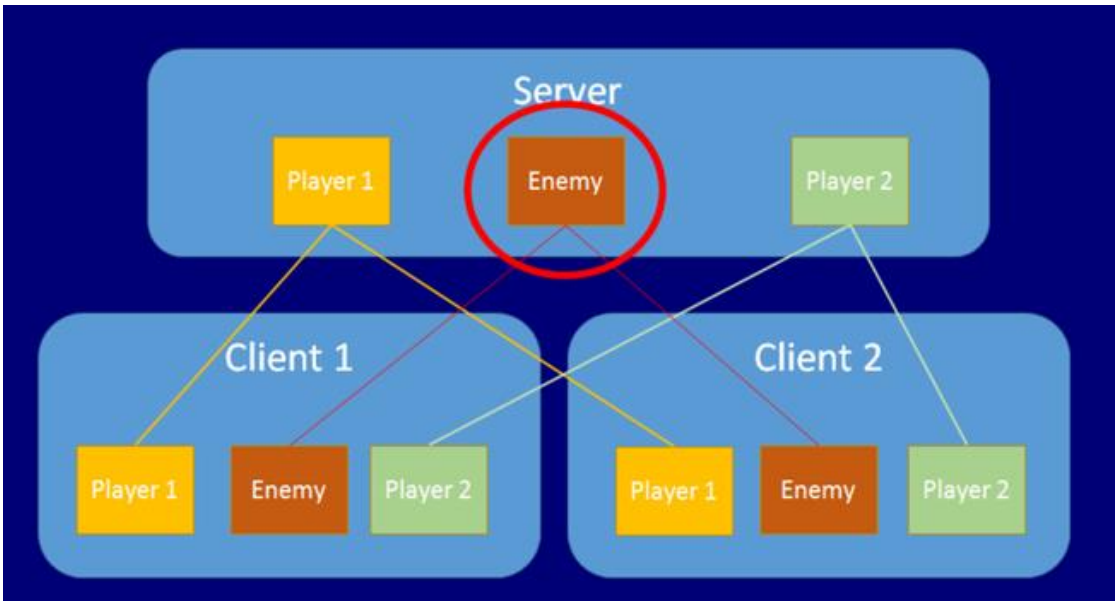


Figure 19. Network authority

In the server authoritative model of the UNet, registered spawnable network prefabs are instantiated on the server. The two formerly mentioned managers offer an easy drag-and-drop solution for registration. Spawning an object on the server means that it is created by the server onto currently connected clients, and state updates are sent to clients when the object changes on the server. The following code snippet

shows an enemy prefab to be instantiated on a server. (Unity – Manual: Object Spawning 2016.)

```
public GameObject enemyPrefab;

public void Spawn()
{
    GameObject enemy = (GameObject)Instantiate(enemyPrefab, transform.position, transform.rotation);
    NetworkServer.Spawn(enemy);
}
```

GameObjects already existing in a scene are handled differently, when compared to dynamically instantiated objects. They are loaded with the scene on both the client and server and exist at runtime before any spawn messages are sent. If the NetworkIdentity component is present in these objects and when a scene is fully loaded, NetworkServer.SpawnObjects () is called automatically by the managers to activate these networked scene objects. Due to this special way of instancing, they are hooked up to the network, which means that if an object of this kind is destroyed before a client joins the game, it will never be spawned on any new clients that join; however, these objects come with the benefit of having special modifications that differ from prefabs. (ibid.)

3.1.7 State Synchronization

State synchronization is set up from the server to remote clients and since local clients do not have data serialized to them, SyncVar hooks are needed. SyncVars are variables of NetworkBehaviour scripts that are synchronized from the server to clients. SyncVars can be of any basic type such as integers, floats or strings; or Unity types like Vector3. SyncVar updates are sent automatically by the server when objects are spawned or when new players connect to a game already in progress. The state of SyncVars is applied to objects on clients before OnStartClient () is called, which guarantees the state of the objects to be up-to-date when a client connects. Variables are made into SyncVars by tagging them with the [SyncVar] custom attribute, like such: (Unity – Manual: State Synchronization 2016.)

```
[SyncVar]
int health;

public void TakeDamage(int amount)
{
    if (!isServer)
        return;

    health -= amount;
}
```

In order to synchronize a list of values, instead of individual values the usage of SyncLists is recommended. SyncLists are specific classes and do not require the SyncVar tag. There are built-in SyncList types for basic variables:

- SyncListString
- SyncListFloat
- SyncListint
- SyncListUInt
- SyncListBool

There is also a type called SyncListStruct, which can be used for lists of user-defined structs. The struct used in SyncListStruct derived class, like in the following code, can contain members of basic types, arrays, and common Unity types. (ibid.)

```
public struct test
{
    public string name;
    public int health;
    public float speed;
};

public class TestList : SyncListStruct<test> {}
TestList m_tests = new TestList();
```

Sometimes SyncVars are not enough for scripts to serialize their state to clients, so the virtual functions on NetworkBehaviour can be implemented by developers to perform custom serialization. These functions are:

```
public virtual bool OnSerialize(NetworkWriter writer, bool initialState);

public virtual void OnDeSerialize(NetworkReader reader, bool initialState);
```

Appendix 6, made by Unity Technologies, demonstrates the difference between custom serialization and the simple usage of SyncVars. (Unity – Manual: State Synchronization 2016.)

3.1.8 Remote Actions

There are two types of Remote Procedure Calls, or RPCs, in the UNet system, to perform actions across the network. Commands, which are called from the client and run on the server and ClientRPCs, which are called on the server and run on clients. (Unity – Manual: Remote Actions 2016.)

To make functions into Commands, a custom [Command] attribute tag must be set before the actual function. A “Cmd” prefix must also be added in front of the name of the function. These Commands are sent from player objects on the client, to player objects on the server and for security reasons they can only be sent by this particular client, to prevent taking over the controls of another player. If any arguments are present, they are passed automatically to the server with Commands. The following code shows a Command function for player firing a bullet, with isLocalPlayer check, that prevents the execution of said Command if isLocalPlayer returns false. (ibid.)

```
[Command]
void CmdFire()
{
    GameObject bullet = (GameObject)Instantiate(bulletPrefab, transform.position, Quaternion.identity);

    bullet.GetComponent<Rigidbody>().velocity = -transform.forward*4;

    NetworkServer.Spawn(bullet);

    Destroy(bullet, 2.0f);
}

void Update()
{
    if (!isLocalPlayer)
        return;

    if (Input.GetKeyDown(KeyCode.Space))
    {
        CmdFire();
    }
}
```

In similar fashion as Commands, the ClientRPC functions are created by setting a custom [ClientRPC] tag and ”Rpc” prefix. The ClientRPC calls are sent from objects on the

server to objects on clients, by any spawned server objects which have NetworkIdentity component attached. In other words, ClientRPC calls are a way for server objects to cause things to happen on client objects. Since the server has authority, there are no security issues with server objects being able to send these calls. The behaviour of a ClientRpc call is the same for LocalClients and RemoteClients; even though LocalClient is in the same process as the server. The following code shows an example of a ClientRpc call. (ibid.)

```
[SyncVar]
int health;

[ClientRpc]
void RpcDamage(int amount)
{
    Debug.Log("Took damage:" + amount);
}

public void TakeDamage(int amount)
{
    if (!isServer)
        return;

    health -= amount;
    RpcDamage(amount);
}
```

3.1.9 Internet services

The Unity Technologies offer a multiuser server service for games to communicate across the internet, which provides the users an ability to create, join, list and advertise online matches. In order to enable the internet services for a project, it needs to be registered by clicking the cloud icon in the upper right corner of the editor, which leads to the cloud multiplayer website from where a project ID can be acquired. There is also a 20 CCU (concurrent users) limit for multiplayer development in the Unity Personal Edition. (Unity – Manual: Internet Services 2016.)

When using the UNet internet services, network traffic goes through a relay server hosted by Unity in the cloud instead of going directly between the clients, which avoids problems with firewalls and NATs. The internet service matchmaking functionality can be utilized with a script in the UnityEngine.Networking.Match namespace. The most notable matchmaking features are as follows. (ibid.)

```
//Starts and stops the Matchmaker for the NetworkManager
NetworkManager.singleton.StartMatchMaker ();
NetworkManager.singleton.StopMatchMaker ();

//Creates a new match with default Networkmanager values
CreateMatchRequest create = new CreateMatchRequest ();
networkMatch.CreateMatch (create, OnMatchCreate);

//Lists found matches
networkMatch.ListMatches (0, 20, "", OnMatchList);

//Joins an internet match
networkMatch.JoinMatch (match.networkId, "", OnMatchJoined);
```

3.2 Photon Unity Networking

3.2.1 Feature overview

Photon Unity Networking, or PUN, is a Unity plugin package for creating multiplayer games. It provides authentication options, matchmaking and in-game communication through the Photon backend. The PUN multiplayer features are based around room creation and games are hosted in globally distributed Photon Clouds, in order to guarantee low latencies for players worldwide. PUN exports to almost all platforms supported by Unity and there are two different packages to choose from: PUN Free and PUN Plus. Figure 20 displays the comparison between these two packages. (Photon – Photon Unity Networking Intro N.d.)

	PUN	PUN+
Price	FREE	$ 95
Free CCU	20	100
World wide hosting	✔	✔
Unity Networking compatible	✔	✔
Unity FREE: Web, Standalone	✔	✔
Unity 4 FREE: iOS, Android	✖	✔
Unity 5 FREE: iOS, Android	✔	✔
Host migration	✔	✔
Rooms, Lobby support	✔	✔
Player Quality of Service (QoS)	✔	✔

Figure 20. PUN versus PUN+

As is the case with the UNet, PUN also has its own networking components, callback functions, remote procedure calls and classes. The most important classes are: (Photon Unity Networking: Class List N.d.)

1. PhotonNetwork, which is the main class to use the PhotonNetwork plugin.
2. Photon.Monobehaviour, which inherits MonoBehaviour and is inherited by Photon.PunBehaviour. This class adds the PhotonView property.
3. Photon.PunBehaviour provides the PhotonView and all callbacks/events PUN is able to call. PunBehaviour uses the namespace “using.Photon”.

The chapter 3.2 introduces the most essential features, functions and components of the PUN system.

3.2.2 Initial setup

After the PUN has been imported into a Unity project, the PUN Setup Wizard (Figure 21) will pop up. Registering a new Photon Cloud account provides a personal application id for the user, which is needed for Photon Cloud hosting. (Photon – Initial Setup N.d.)

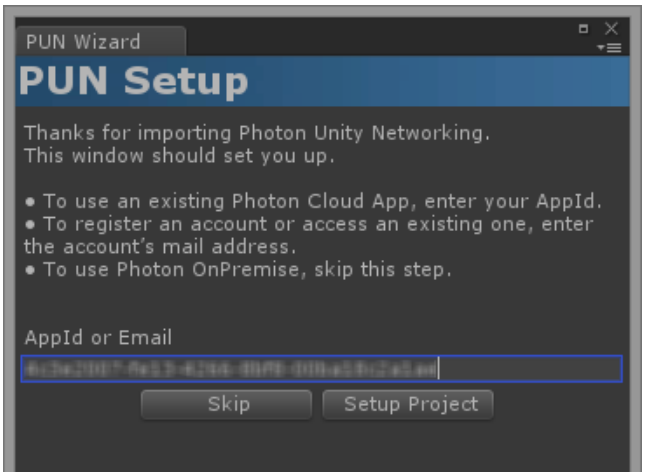


Figure 21. PUN Wizard

The Wizard also adds a PhotonServerSettings file into the project’s asset folder, which can be used to alter the server configurations easily. Through the configuration file the user is able to edit the hosting types, hosting regions, protocols, client settings and remote procedure calls. Figure 22 portrays the PhotonServerSettings file and the options it contains. (ibid.)

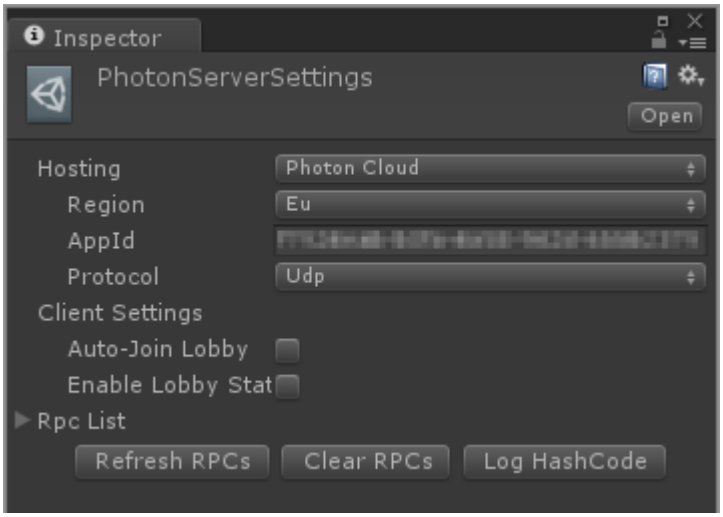


Figure 22. PhotonServerSettings

From the Hosting drop-down menu the user is able to select which server will handle the networking of the game. Both Photon Cloud and Best Region options relate to the services managed by Photon. The Best Region mode will ping all specified regions when the application starts for the first time and lets the clients to select the region with best ping, if more than one region has been specified. The Self Hosted mode offers a choice of running a Photon Server on one's own. By default the connection protocol is set as UDP, however, Photon also supports TCP. (Photon – Initial Setup N.d.)

The Client Settings section contains the options for "Auto-Join Lobby" and "Enable Lobby Stats". If the "Auto-Join Lobby" is checked, the PUN will then automatically join players into the default lobby when connection has been established or when leaving rooms. Lobby statistics enable the receiving of statistical data from servers, which might prove useful when dealing with a game that uses multiple lobbies or when server activity, such as player count, needs to be displayed to the clients. (ibid.)

PUN keeps a list of the game's remote procedure calls, which are used to call functions on all clients in a room. For further discussion about these functions, see Chapter 3.2.7. (ibid.)

3.2.3 Essential components

The PhotonView is the equivalent of UNet's NetworkIdentity component and is used to send messages across the network. PUN requires one PhotonView per instantiated prefab in order to track networking references, object ownership and observed component references. PUN keeps track of the PhotonViews it has instantiated locally and the referenced observed components are able to send updates to the other clients at runtime. For example, if a Transform component has been set as "observed", its position, rotation and scale values are now synchronized across the network to other players. Figure 23 displays a PhotonView component. (Photon – Feature Overview N.d.)

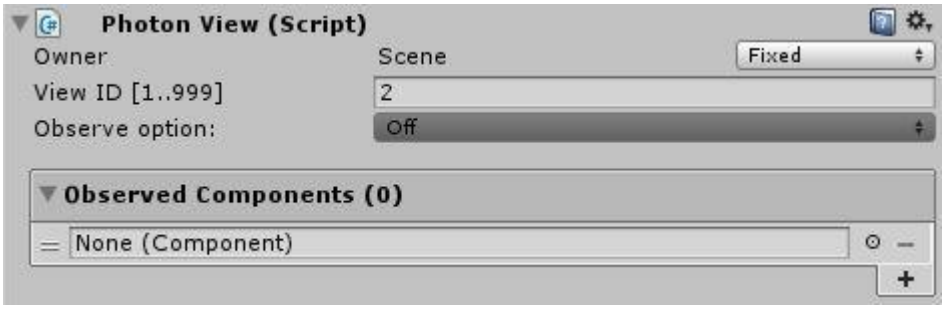


Figure 23. PhotonView component

The PhotonAnimatorView (Figure 24) allows the developer to define which animation layer weights and parameters have to be synchronized. The layer weights only need to be synchronized if they change during the game and the same goes for parameters. Each parameter can be synchronized either discretely or continuously. In practice, discrete synchronization sends values 10 times per second (in OnPhotonSerializeView) and the receiving clients pass the value on to their local Animator. Continuous synchronization means that the PhotonAnimatorView records additional values. When the OnPhotonSerializeView is called (10 times per second), the values recorded since the last call are sent together. The receiving client then applies the values in sequence to retain smooth transitions. While continuous synchronization is smoother, it also needs to send more data. (Photon – Tutorials: Mecanim Demo N.d.)

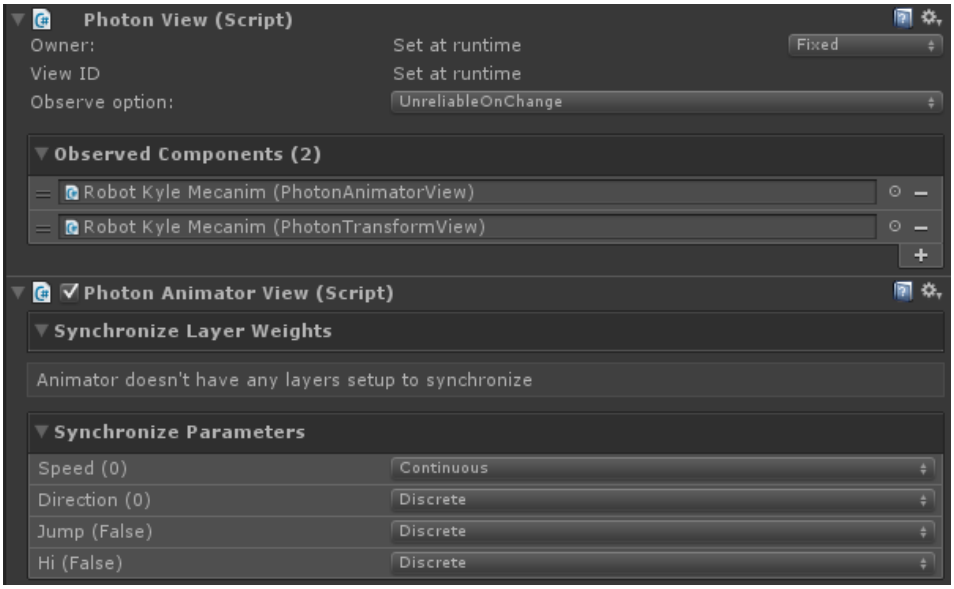


Figure 24. PhotonAnimatorView

The PhotonTransformView, **PhotonRigidbodyView** and **PhotonRigidbodyView2D** components offer a selection of customizable options for advanced synchronization. The RigidbodyView components can be used to synchronize velocities according to Rigidbody physics, while the PhotonTransformView opens up more options on how the Transform data is put in sync. All three components should be added into the "observed" field of the PhotonView, if they are attached into a GameObject. Figure 25 presents the options found in the PhotonTransformView component.

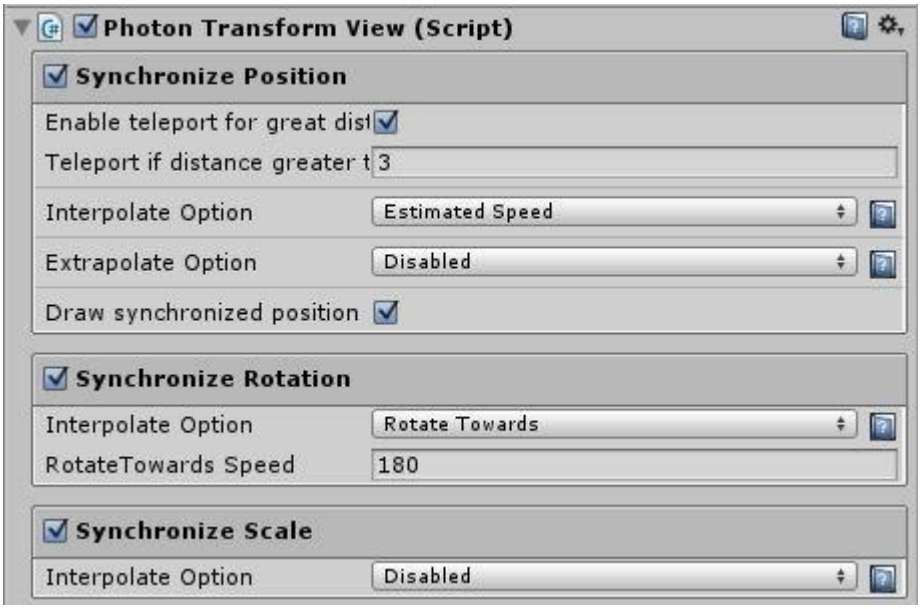


Figure 25. PhotonTransformView

3.2.4 Matchmaking

The PhotonNetwork class always uses a master server and one or more game servers. The master server manages the currently available games and does matchmaking, while the game servers handle actual gameplay once a room has been found or created. PhotonNetwork.ConnectUsingSettings ("v1.0") is all the user needs in order to make use of Photon's features. It sets the client's game version and uses the PhotonServerSettings to connect. Alternatively, Connect () may be used to ignore the server settings file altogether. There are no hosts in the same sense as in Unity Networking, however PUN has a replacement; The "Master Client" which is always the player with the lowest ID in a room. All clients are capable to check if they are currently the master with PhotonNetwork.isMasterClient. (Photon – Matchmaking & Room Properties N.d.)

The following code showcases the basic functions for creating, joining and listing rooms:

```
//Join a room
PhotonNetwork.JoinRoom(roomName);

//Create a room
//Fails if it already exists and calls: OnPhotonCreateGameFailed
PhotonNetwork.CreateRoom(roomName);

//Tries to join any random game
//Fails if there are no matching games: OnPhotonRandomJoinFailed
PhotonNetwork.JoinRandomRoom();

//Returns a list of all available games
PhotonNetwork.GetRoomList();

//Creates or joins a room with custom RoomOptions
//Room is created only if a room with the same name does not already exist
RoomOptions roomOptions = new RoomOptions() { isVisible = true, maxPlayers = 4 };
PhotonNetwork.JoinOrCreateRoom(roomName, roomOptions, TypedLobby.Default);
```

3.2.5 Instantiation

In Photon, prefabs are instantiated during runtime with the PhotonNetwork.Instantiate function. PUN automatically registers and takes care of the spawning of these networked objects by passing starting position, rotation and prefab name to the instantiate function. All networked prefabs must contain a PhotonView component

and should be located directly under a resources/ folder for runtime accessing. (Photon – Instantiation N.d.)

```
GameObject ex = PhotonNetwork.Instantiate("example", Vector3.zero, Quaternion.identity,0);
```

If the developer wishes not to rely on the resources folder for object instantiation, "manual" instantiation is an option as well. Spawning objects through remote procedure calls will accomplish this task. The PhotonView.viewID is the key for routing network messages to the correct GameObjects and scripts, while PhotonNetwork.AllocateViewID () allocates new viedIDs, so everyone in the room has the same ID on the new object. Manual instantiation is carried out as follows: (ibid.)

```
void SpawnPlayerEverywhere()
{
    // Manually allocate PhotonViewID
    int id1 = PhotonNetwork.AllocateViewID();
    PhotonView photonView = this.GetComponent<PhotonView>();
    photonView.RPC("SpawnOnNetwork", PhotonTargets.AllBuffered, transform.position,
                                    transform.rotation, id1, PhotonNetwork.player);
}

[RPC]
void SpawnOnNetwork(Vector3 pos, Quaternion rot, int id1, PhotonPlayer np)
{
    Transform newPlayer = Instantiate(playerPrefab, pos, rot) as Transform;

    // Set player's PhotonView
    PhotonView[] nViews = newPlayer.GetComponentsInChildren<PhotonView>();
    nViews[0].viewID = id1;
}
```

GameObjects spawned with the PhotonNetwork.Instantiate function will exist on other clients as long as the client who owns or creates them stays in the same room. If this kind of behaviour is not desirable, the "PhotonNetwork.autoCleanUpPlayerObjects" can be set as false. Alternatively, the Master Client can create GameObjects which have the same lifetime as the room by using PhotonNetwork.InstantiateSceneObject (). Objects created this way are associated with the room, not the Master Client. By default, the Master Client controls the created objects, however control may be passed on to other clients with PhotonView.TransferOwnership (). (ibid.)

If the GameObjects need to be set up as they get instantiated, it is possible to call the OnPhotonInstantiate (PhotonMessageInfo info) on them, with the info who triggered the instantiation. For example, setting an object as a player's Tag object is as follows: (ibid.)

```
void OnPhotonInstantiate(PhotonMessageInfo info)
{
    info.sender.TagObject = this.gameObject;
}
```

3.2.6 Player Authority

There are a few ways to have authority over player controls in PUN. As the PhotonNetwork.Instantiate returns the GameObject it created, a simple bool variable is suitable for this task. This is demonstrated in the code below, where a player object gets instantiated and its isControllable value is set as true for input registering. (Photon – Tutorials: Marco Polo N.d.)

```
void OnJoinedRoom()
{
    GameObject player = PhotonNetwork.Instantiate("playerprefab",
                                Vector3.zero, Quaternion.identity, 0);

    player.GetComponent<ThirdPersonController>().isControllable = true;
}
```

```
if(isControllable)
{
    Input.GetAxisRaw("Vertical");
    Input.GetAxisRaw("Horizontal");
}
```

Another way to achieve similar functionality is to use the isMine property of the PhotonView component, which returns true if this particular client owns the PhotonView in question. (ibid.)

```
if (photonView.isMine)
{
    Input.GetAxisRaw("Vertical");
    Input.GetAxisRaw("Horizontal");
}
```

3.2.7 State synchronization

As mentioned in chapter 3.2.3, GameObjects can easily be made network aware by assigning a PhotonView component on them. The PhotonView must be setup to observe other components such as Transform, or more commonly other custom script components. While a script is observed, the PhotonView component regularly calls the method OnPhotonSerializeView. The duty of this function is to create the info the

clients want to pass on to others and handle such incoming info, depending on who created the PhotonView. (Photon – Synchronization and State N.d.)

A PhotonStream is passed on to the OnPhotonSerializeView and the value of isWriting tells the client if it needs to write or read remote data from it. The subsequent code snippet showcases this functionality as it sends and receives positional and rotational data. (Photon – Tutorials: Marco Polo N.d.)

```
public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
    if (stream.isWriting)
    {
        // We own this player: send the others our data
        stream.SendNext(transform.position);
        stream.SendNext(transform.rotation);
    }
    else
    {
        // Network player, receive data
        this.transform.position = (Vector3) stream.ReceiveNext();
        this.transform.rotation = (Quaternion) stream.ReceiveNext();
    }
}
```

It is also possible to set custom synchronizable properties through code for player objects and rooms with the SetCustomProperties (). Photon's custom properties consist of key-value Hashtables, which are synched and cached on clients. For example, to set custom properties for a player, PhotonPlayer.SetCustomProperties (Hashtable propsToSet) must be used. Similarly, PhotonNetwork.room.SetCustomProperties (Hashtable propsToSet) function is used for rooms. (Photon – Synchronization and State N.d.)

3.2.8 RPCs and RaiseEvent

In Photon, the PhotonView components are like "targets" for remote procedure calls. When a function has been tagged with [PunRPC], it is executed on the clients (defined by PhotonTarget values) only on the networked GameObject with a specific PhotonView. For example, if a client damages another object in a game and calls for an "ApplyDamage" RPC function, all the receiving clients will apply the damage to the same object on their end. In order to call the functions marked as RPC, a PhotonView

is needed on a GameObject, as well as a Photon.MonoBehaviour or Photon.PunBehaviour in scripts. An example code of a message being sent through an RPC function is as follows: (Photon – RPCs and RaiseEvent N.d.)

```
    PhotonView photonView = PhotonView.Get(this);
    photonView.RPC("ChatMessage", PhotonTargets.All, playerName, message);
}

[PunRPC]
void ChatMessage(string a, string b)
{
    Debug.Log("ChatMessage " + a + " " + b);
}
```

The PhotonTargets might have some parameters ending on “Buffered”. The server remembers these RPC functions and when a new player joins in, it receives the RPC, even though it happened prior to joining. Alternatively, parameters ending on ViaServer disables the “All” parameter of PhotonTargets. The ViaServer sends RPCs through the server, executing them in the same order as they arrive on the server. Typically in Photon, when the sending client has to execute an RPC, it does so immediately without sending the RPC through the server. The PhotonServerSettings file lists and stores all currently present RPCs. (Photon – RPCs and RaiseEvent N.d.)

In some cases the RPCs are not exactly what is needed. With the PhotonNetwork.RaiseEvent, the developers are able to create custom events and send them without any relation to networked objects or PhotonViews. To receive events, a script must implement with an EventCallback. The RaiseEventOptions parameter of the PhotonNetwork.RaiseEvent can be set as null or to define which clients receive this event, is it buffered, etc. (ibid.)

```
// Setup OnEvent as callback:
void Awake()
{
    byte evCode = 0;
    byte[] content = new byte[] { 1, 2, 5, 10 };
    bool reliable = true;
    PhotonNetwork.RaiseEvent(evCode, content, reliable, null);
    PhotonNetwork.OnEventCall += this.OnEvent;
}

// Handle events:
private void OnEvent(byte eventcode, object content, int senderid)
{
    if (eventcode == 0)
    {
        PhotonPlayer sender = PhotonPlayer.Find(senderid);  // who sent this?
        byte[] selected = (byte[])content;
        foreach (byte unitId in selected)
        {
            // do something
        }
    }
}
```

3.3 Other plugins

There are several other third-party plugins to be found in the Asset Store and around the internet, either for a fee or completely free of charge, although many of the available plugins offer trial- or lite-versions for free and the unlocking of extended features requires a payment. In addition to PUN, some of the most popular plugins are: DarkRift, Forge, uLink and PlayFab. Each of these plugins have their own operating principles and networking libraries to be integrated into a Unity project.

4 Project Quantum Knight

4.1 Game story

In the not so distant future the megacorporations rule the world. Human rights have been diminished in the city of Kyoto, as the malicious Yú Corporation enforces peace and obedience with an iron grip. The Cybernatural Affairs Department (C.A.D), a section of Yú Corporation's paramilitary security forces, responds swiftly and ruthlessly to all cyber security transgressions.

The main character, only known by the name of Void, is a synthetic lifeform created by Yú Corporation's Quantum Knight Program, in order to simulate how a machine is

able to coexist with organic lifeforms. Quantum Knights have the ability to physically move in and out of the Kyoto city's labyrinthine data network (cyber world) through terminals and into different places in the real world. Void is completely oblivious to his/her true nature, while several other versions of Quantum Knights with the same ability have been produced.

At the beginning of the game, the main character's personal domain is attacked by a tracker virus. The virus causes a lockdown in the apartment block and Void must enter the cyber world in order to defeat it. The cyber world bears a close resemblance to the real world, as perceived by organics, however, with different threats and challenges. Security protocols dispose infiltrators with mechanical precision, all the while some parts of the network have been infested by viruses. Although Void manages to conquer the virus in his/hers personal domain, the C.A.D agents will soon arrive. Before that happens, Void must flee the scene and escape from the district…

Quantum Knight is a JRPG-styled game (Japanese Role Playing Game), with a battle system similar to Final Fantasy Tactics and Fire Emblem series, including online gameplay. The game is set in a cyberpunkish city of Kyoto for the player to explore. QK features retro style graphics and music to emulate 80s visions of dystopian future.

4.2 Project realization

The Quantum Knight – prototype project was carried out by a two-man development team, handling everything from design and programming to music and sound effects, except graphics. The work load was distributed evenly for menial tasks, such as music, effects, menus and level layouts, while tasks concerning topics of our separate theses, were divided accordingly; the main focal points being in the studying of multiplayer solutions and in Object Oriented Programming versus Entity Component System in Unity.

Unity Engine was a natural choice over other game development platforms, as both team members had over a year of previous experience working with it. Another reason were the built-in features of Unity, which greatly sped up the development pro-

cess and workflow in a tight schedule. At the beginning, at the specification and designing phase of the project, the development timeframe was set to four months and priorities for important features as follows:

1. Basic gameplay
2. Battle mechanics
3. Viruses
4. Multiplayer
5. Environment building
6. UI
7. Music
8. Dialog
9. Sound effects
10. Testing

As with every proper software project, the development process went through a set of different phases and in this particular case, iteratively. The first step was to write out a requirement specification and Game Design Document (GDD), which includes all the assets needed, gameplay mechanics, overall game idea, game flow and how the development process is carried out.

After the designing of key mechanics, functions and environments, the project advanced to implementation phase, where all achievable features were created in Unity and tested constantly as production moved forward. A small-scale end testing session was held for the prototype, to ensure no critical errors were to be found.

To assure keeping on schedule, as well as task visualization and listing of working hours, team members chose Trello and Scrum for project handling tools. Trello is a web-based project management application originally made by Fog Creek Software and Scrum an iterative and incremental agile software development framework for managing product development. Other tools utilized during the development included Renoise, a digital audio workstation tracker software and Paint.NET, a freeware raster graphics editor program for Microsoft Windows.

4.3 Audiovisual design

In order to create the feel of a future, where humanity has more or less failed, cyberpunk and retro futurism heritage was used as an inspiration. Through common interests, an East Asian city, set in a dystopian future, was chosen as the setting for the game prototype, to reflect genuine 80s futuristic visions to the player.

4.3.1 Graphics

To save limited production time, retro styled graphics from RPG Maker were implemented into the project. This enabled the quick creation of characters and environments as well as obtaining the desired retro style visually. In Figure 26, the different 2D sprites needed for animating the main character's movements can be inspected.

Figure 26. Void, the main character

The RPG Maker graphics are delivered in Sprite sheets. A sprite sheet is generally a large image, containing a collection of 2D sprites. All the characters and environments seen in the prototype were created by using various suitable sheets. Figure 27 presents one of them.

Figure 27. Sprite sheet

Graphical design started with the choosing of fitting sprites for characters and environments, followed by sketches drawn on grid paper. This made it easier to perceive the place and scale of an each object. After several discussions what the environments should include, which objects are interactable and the appearances of characters, the levels were assembled in the Unity editor. One of the resulted levels is displayed in Figure 28. It shows a portion of the Streets of Kyoto level which acts as a hub for smaller locations.

Figure 28. Streets of Kyoto

4.3.2 Sounds

Due to the common previous experience of the Renoise tracker software, it was chosen as the platform to create sound effects and background music for the game. Effects and songs were composed by using a diverse selection of audio samples and VSTs, Virtual Studio Technology plugins. VST is a software interface that integrates software audio synthesizer and effect plugins with audio editors and recording systems. Thousands of VST plugins exist, both commercial and freeware. (Virtual Studio Technology 2016.)

Virtual instruments, including synthetisators and drums, for example, were chosen appropriately, to further the 80s feel aurally. Team members picked the sound effect samples together and held listening sessions and discussions over background music tracks. The prototype does not feature spoken dialogue for any character. Figure 29 presents the user interface of Renoise.

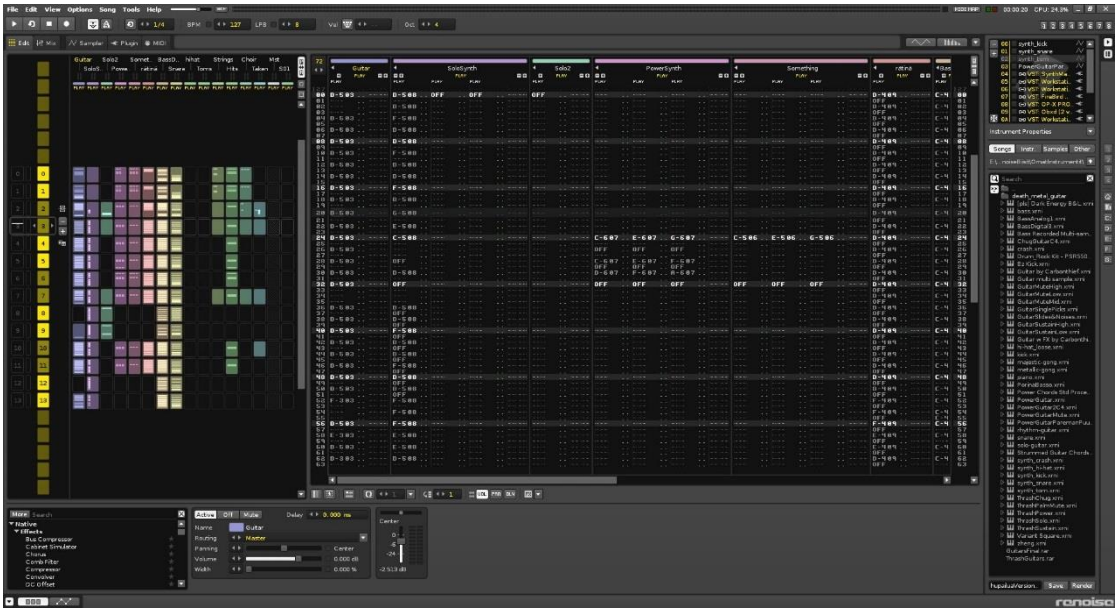


Figure 29. Renoise user interface

4.4 Gameplay and key mechanics

In practice, the fundamental gameplay mechanics of Quantum Knight lean heavily on the standards set by JRPGs. Influenced by such classic games as Final Fantasy Tactics, Fire Emblem and Pokémon, the key mechanics include:

- Exploration of the game world and dialogue exchange with NPCs (non-playable characters)
- Interacting with predestined objects
- Switching between the physical world and cyber world
- Puzzle solving that affect both of the worlds
- Battling viruses in the cyber world and capturing them
- Completing quests given by the NPCs
- A multiplayer tournament mode

4.4.1 Controls

From using the menus and selecting combat actions during battle, to interacting with objects, a preset control scheme allows the prototype to be played with either keyboard or Xbox 360 controller. Table 1 presents the actions and corresponding buttons for both controlling options. Appendix 4, previously mentioned in chapter 2.4, shows a C# script for handling the player movement inputs and raycasting, to detect interactable GameObjects.

Table 1. Control Scheme

Action	Keyboard	Gamepad
Move left/selection left	Left key / A	Left directional / stick
Move right/selection right	Right key / D	Right directional / stick
Move up/selection up	Up key / W	Up directional / stick
Move down/selection down	Down key / S	Down directional / stick
Interact/Submit/ Advance dialogue	Enter / E	A
Cancel	Backspace / Q	B
Pause menu	Escape	Start

4.4.2 Dialogue system

When interacting with NPCs or predestined objects, the game enters into a dialogue mode. When this mode is active, other player controls and events occurring in the game, are disabled. By using a simple array of public strings, dialogue lines are displayed in a Unity.UI text component, inside a panel object on the top of the screen (Figure 30). If the player stands next to an interactable object the same panel shows an indicator that this particular object can be interacted with. Pressing interact buttons initiates and advances the dialogue between the main character and engaged GameObjects.

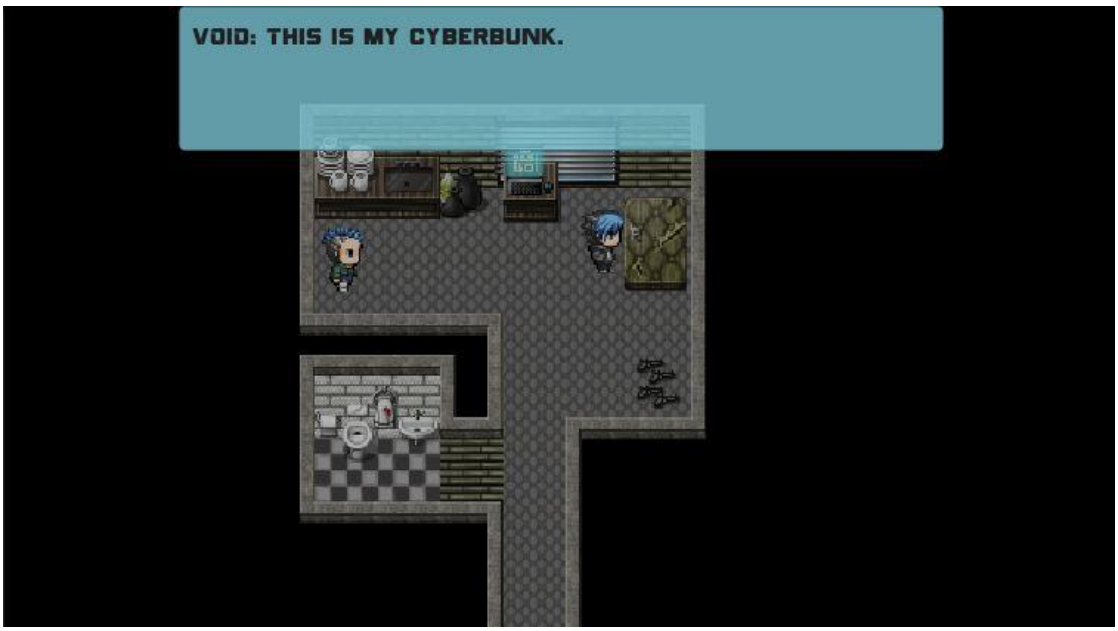


Figure 30. Dialogue system

4.4.3 Battle system

One of the defining characteristics of the JRPG genre is turn-based combat. In Quantum Knight, battles are fought in virtual arenas that have various layouts and obstructions. The arenas are composed from square tiles, where each character has the ability to move according to their movement points and/or take one action per turn. At the start of the battle, the currently present characters are arranged in order by their speed attribute. The character with the highest value takes action first. If two characters have equal values, the order is randomized between the two. Each combatant has their own set of skills and combat statistics. Figure 31 displays the initiation of the shrine demon battle.

Figure 31. Shrine demon battle

Skills and statistics are dependent on the role of each fighter. The stats are divided into five different main categories: Health points, attack strength, cyber magic, defense and speed. In turn, the stats are set as low, medium or high for each role, or class. For example, a character class with high damage dealing abilities has a low defense and a character with high cyber magic, a low attack strength. In the single player campaign, the main character takes place as one of the fighters. If the main character falls in battle, the game is over. Void also has the ability to capture viruses, not including bosses if they are weakened enough.

During the battle, commands are issued through a simple menu, which gives the player a choice between “move”, “attack”, “skills” or “end turn”. After selecting an action, besides turn ending, a “selector-icon” becomes visible on the battlefield. Moving this selector on top of enemies or allies, displays information about them in a panel at the top of the screen. With the selector, the player chooses where he/she wants to move or which enemy to attack. If player both moves and acts on same turn, end turn is initiated automatically. Figure 32 demonstrates the functionality of the selector and menu.

Figure 32. Selector, stats and menu

Arena layouts, skills and roles bring tactical aspects to play when battling against enemies. For instance, clever positioning of melee and ranged characters in order to block lines of sight by standing behind obstructions or smart co-operative usage of skills between different classes is needed for victory. In the single player campaign, battle scenarios are predetermined and played against AI, but in multiplayer tournaments, two players go head-to-head; each representing their own team. The prototype has two distinct battle modes, which are:

1. Node Conquest, both or only one of the players have a node, which must be protected from enemy damage.
2. Deathmatch, one must defeat the opponent’s main character.

4.4.4 Programs

The “viruses”, from now on referred to as programs, are named according to real cybersecurity terminology. They can be harmful damage dealers, like Malware or beneficial “healers” such as Antivirus. Tables 2 and 3 illustrate the stats and class types of each program.

Table 2. Program classes

Class	Tank	Healer	Damage	DoT	AoE	Buffs	Debuffs	Melee	Ranged
Adware				X			X		X
Antivirus		X				X			X
Bot					X		X		X
Firewall	X					X		X	
Malware			X		X			X	
Spyware		X	X				X		X
Trojan	X						X	X	
Virus			X	X				X	
Worm			X				X	X	

Table 3. Program stats

Class	HP	Attack	Magic	Defense	Speed
Adware	low	medium	high	low	high
Antivirus	medium	low	high	low	high
Bot	medium	medium	medium	medium	medium
Firewall	high	low	medium	high	low
Malware	low	high	high	low	medium
Spyware	medium	medium	medium	low	high
Trojan	high	medium	low	high	low
Virus	low	high	medium	low	high
Worm	medium	high	low	low	high

4.4.5 Single player

The short single player campaign of the prototype involves the main character Void exploring the two separate worlds. Conversing with NPCs, completion of quests and interaction with key objects trigger events which advance the plot. The progression is managed via an EventHandler script and a state machine it contains. When certain requirements are met, like completing a quest, a trigger value is sent to the state machine and it activates the designed subsequent plot objective. Appendix 7 shows the C# code of the EventHandler script.

Game flow and progression key points of the single player campaign as described in the project's game design document are listed as follows:

1. Player wakes up in his/hers rundown apartment
2. There is a light flickering on the screen of a local cyberterminal
3. Tracker program attacks the players cyber domain
4. Player cannot leave the apartment before interacting with the terminal
5. Player uses the terminal and enters the cyberworld
6. Tutorials in the cyberworld
7. Player fights with the program and wins (battle scene)
8. Cyberworld outside own domain is now open for access
9. Player can now remove the lockdown from the apartment door
10. Player leaves the apartment
11. Player enters the rundown streets of Kyoto
12. Player monologue
13. Player movements are restricted by a murder scene, road construction site and virus infested security barrier
14. Player can walk around the streets or visit a shrine, cybershop and an adult shop
15. There are various NPC:s the player can talk and interact with
16. There are non-significant events occurring, like protester getting shot by security droid
17. Owner of the cybershop cannot access his/hers domain and player must enter the cyberworld to open access
18. Reward
19. Player must fix a glitch with the cyber shrine (demon virus)
20. Reward
21. When player has powerful enough programs the security barrier becomes hackable
22. If player doesn't have powerful enough programs, he/she can enter the cyberworld in order to try and capture some
23. When player has hacked the gate, a plot twist for teasing purposes appears
24. Game Ends!

Figure 33 displays the event of a protester getting shot.

Figure 33. An event

4.4.6 Multiplayer

In addition to the single player campaign, the prototype has a separate multiplayer mode, where two players can battle against each other locally or over an internet connection. Battles are fought in a similar fashion as in the single player mode, but the length of each turn is now limited by a timer, to keep things more hectic.

Multiplayer lobby menu, as seen in Figure 34, offers choices between hosting and joining games over the internet, hosting and joining locally or running a dedicated server. Once two players have joined the same lobby and both have set their status to ready, multiplayer version of a battle arena is loaded. After the combat is over, winners and losers are declared and players have the option to choose a rematch or return to the lobby. Chapter 5 describes in detail how the implementation of multi-player gameplay was carried out.

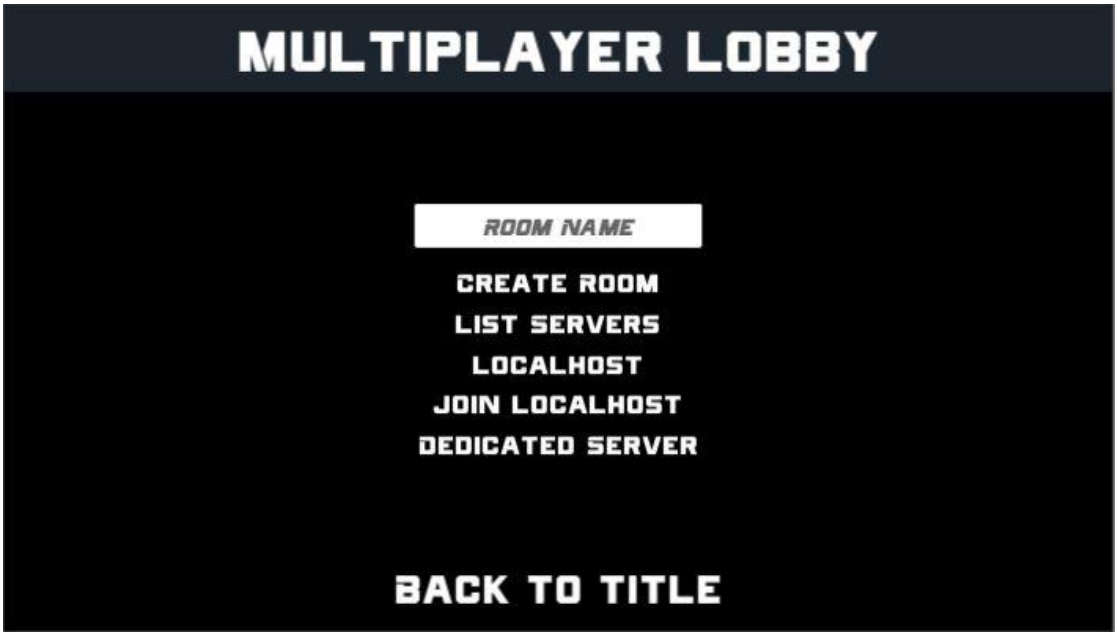


Figure 34. Multiplayer lobby main menu

5 Implementation of multiplayer features

Chapter 5 presents two different hands-on approaches on how the multiplayer features of the prototype were actually implemented. The main focus is on the networking functionality of the lobby, player prefabs and how remote actions, synchronization, animations and gameplay mechanics are handled in the game scene. The feature implementation is examined through UNet and one of the most popular free Asset Store third-party plugins.

5.1 Unity Networking

5.1.1 Lobby scene

The main menu of the lobby, as previously seen in Figure 28, is built up by using the Unity UI components. It holds an input field and five different buttons to handle user requests on how to create and join games. When a button is clicked, it executes a function via an OnClick () event. These OnClick () events handle all the transitions in the lobby system by activating and deactivating predestined UI panels, as well as carrying out the network functionality. Figure 35 displays a UI button component.

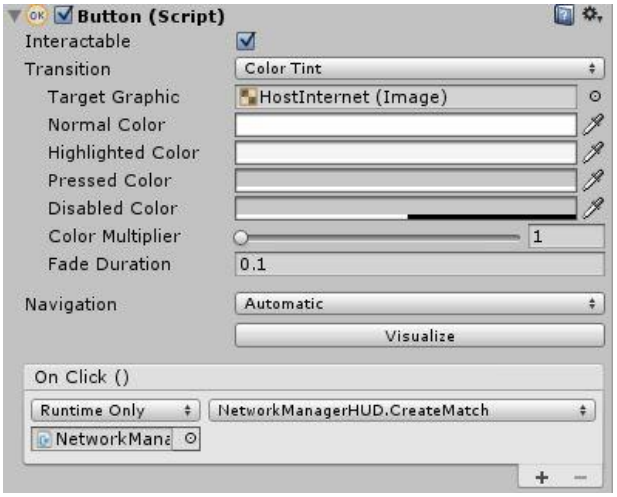


Figure 35. Button and OnClick ()

When the **"Create Room" button** is clicked, it checks at first if the NetworkServer and NetworkClient return false. If they do, it instructs the NetworkLobbyManager to start the matchmaker service with appropriate parameters and creates a room with the name specified in the input field. By disabling and activating panels, the player is taken to the actual lobby which holds the ready status checks; now awaiting for another player to join the same room. In the following code, the functionality behind this button can be inspected.

```
public void CreateInternetMatch()
{
    if (!NetworkServer.active && !NetworkClient.active)
    {
        if (manager.matchMaker == null)
        {
            manager.StartMatchMaker ();
            manager.matchMaker.CreateMatch (manager.matchName, manager.matchSize, true, "", manager.OnMatchCreate);
        }
    }
}

public void CreateMatch ()
{
    mainPanel.SetActive (false);
    lobbyPanel.SetActive (true);
    CreateMatchRequest create = new CreateMatchRequest();
    create.name = inputF.text ;
    create.size = 2 ;
    create.advertise = true ;
    create.password = "" ;
    networkMatch.CreateMatch(create, MatchCreated);
}

public void MatchCreated (CreateMatchResponse matchResponse)
{

    if (matchResponse.success)
    {
        matchCreated = true ;
        manager.SetMatchHost ("mm.unet.unity3d.com", 443, true);
        manager.matchInfo  = new MatchInfo(matchResponse);
        Utility.SetAccessTokenForNetwork(matchResponse.networkId, new NetworkAccessToken(matchResponse.accessTokenString));
        manager.StartHost(manager.matchInfo) ;
    }
}
```

The "List Servers" button requests a list of the currently active matches from the UNet matchmaker. After instructing the manager to start matchmaking, it lists the matches by using the ListMatches function. Then, a Coroutine starts with a waiting time of three seconds, in order to ensure all matches are surely found. After three seconds have passed, a foreach-loop iterates through all the found matches and instantiates a GameObject with room name and join button, to visually represent each room found (Figure 36).

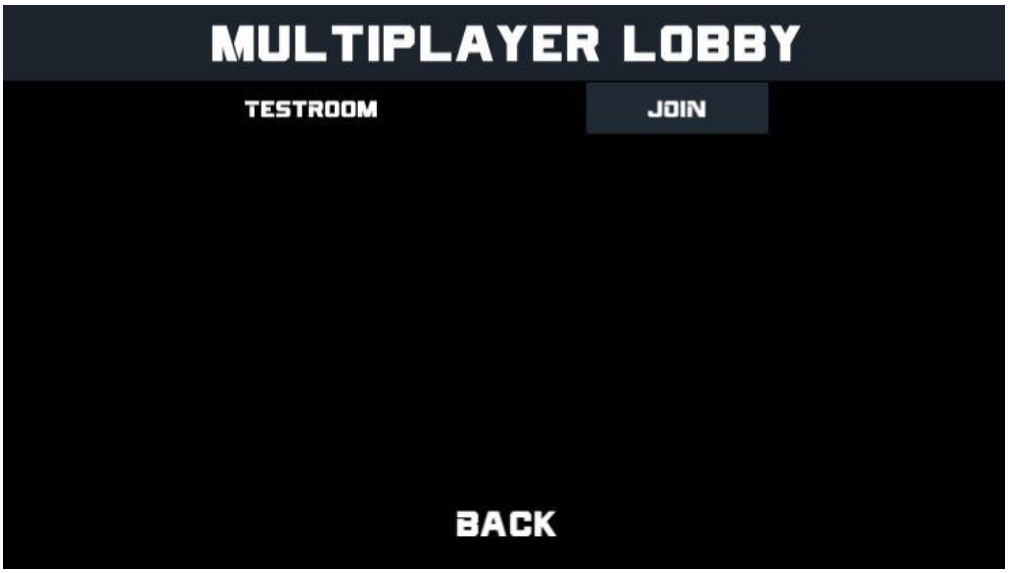


Figure 36. Server list

When the corresponding **"Join" button** is clicked, it connects the client to that specific room. The "Join" – button identifies the correct room to connect the client to, by using the room's NetworkID. The "**Back" buttons** found in each panel, depending on where the user is in the lobby system, hold the functionality for StopHost (), StopClient (), StopServer () and StopMatchmaker (). When the game scene is running, an OnGUI button with similar functionality is displayed. The code for the whole server listing process is presented as follows:

```
public void GetServerList()
{
    if (!NetworkServer.active && !NetworkClient.active)
    {
        manager.StartMatchMaker ();
        mainPanel.SetActive (false);
        serverListPanel.SetActive (true);

        if (manager.matchInfo == null)
        {
            if (manager.matches == null)
            {
                manager.matchMaker.ListMatches (0, 10, "", manager.OnMatchList);
                StartCoroutine (WaitForServers());
            }
        }
    }
}

IEnumerator WaitForServers()
{
    yield return new WaitForSeconds (3);
    int yPos = 0;

    foreach (var match in manager.matches)
    {
        GameObject serverO = (GameObject)Instantiate (serverObject,
                                serverListPanel.transform.position, serverListPanel.transform.rotation);
        serverO.transform.SetParent (GameObject.FindWithTag ("ServerList").transform, false);
        serverO.transform.localPosition = new Vector3 (0, yPos, 0);
        serverO.GetComponentInChildren<Text> ().text = match.name;
        serverO.GetComponentInChildren<Button> ().onClick.AddListener(() => JoinMatch(match.networkId));;

        yPos -= 100;
    }

    if (manager.matches.Count == 0)
    {
        noServersPanel.SetActive (true);
    }
    else
    {
        noServersPanel.SetActive (false);
    }
}

void JoinMatch(Networking.Types.NetworkID id)
{
    serverListPanel.SetActive (false);
    lobbyPanel.SetActive (true);
    manager.matchMaker.JoinMatch (id, "", manager.OnMatchJoined);
}
```

The “Localhost” button creates a game in the local network and in similar fashion as the “Create Room” button, it takes the player to the actual lobby. The StartHost () function creates a local server and client at the same time.

```
public void LanHost()
{
    if (!NetworkClient.active && !NetworkServer.active && manager.matchMaker == null)
    {
        manager.StartHost();
        mainPanel.SetActive (false);
        lobbyPanel.SetActive (true);
    }
}
```

Like the “Join” button in the server list, the **“Join Localhost“ button** connects a client to a game, however, in this case, into a local network.

```
public void LanClient()
{
    if (!NetworkClient.active && !NetworkServer.active && manager.matchMaker == null)
    {
        manager.StartClient ();
        mainPanel.SetActive (false);
        lobbyPanel.SetActive (true);
    }
}
```

If the user wants to start up a dedicated server for a game, the **"Dedicated Server" button** accomplishes this particular task. When the minimum required amount of players have connected into the server, it initiates the loading of the game scene. The next code snippet starts up a server and displays informative text via UI panels to the user, to visualize if the server is running or not. Figure 37 shows a server running on localhost at network port 7777 and awaiting for players to join.

```
public void LanServerOnly()
{
    if (!NetworkClient.active && !NetworkServer.active && manager.matchMaker == null) {
        manager.StartServer ();
        mainPanel.SetActive (false);
        dedicatedPanel.SetActive (true);
        dedicatedPanel.GetComponentInChildren<Text> ().text = "Server running" +
        "\n" + "Address: " + manager.networkAddress + "\n" + " Port: " + manager.networkPort +
        "\n" + "Waiting for players...";
    }
    else
    {
        dedicatedPanel.GetComponentInChildren<Text> ().text = "Server startup error";
    }
}
```

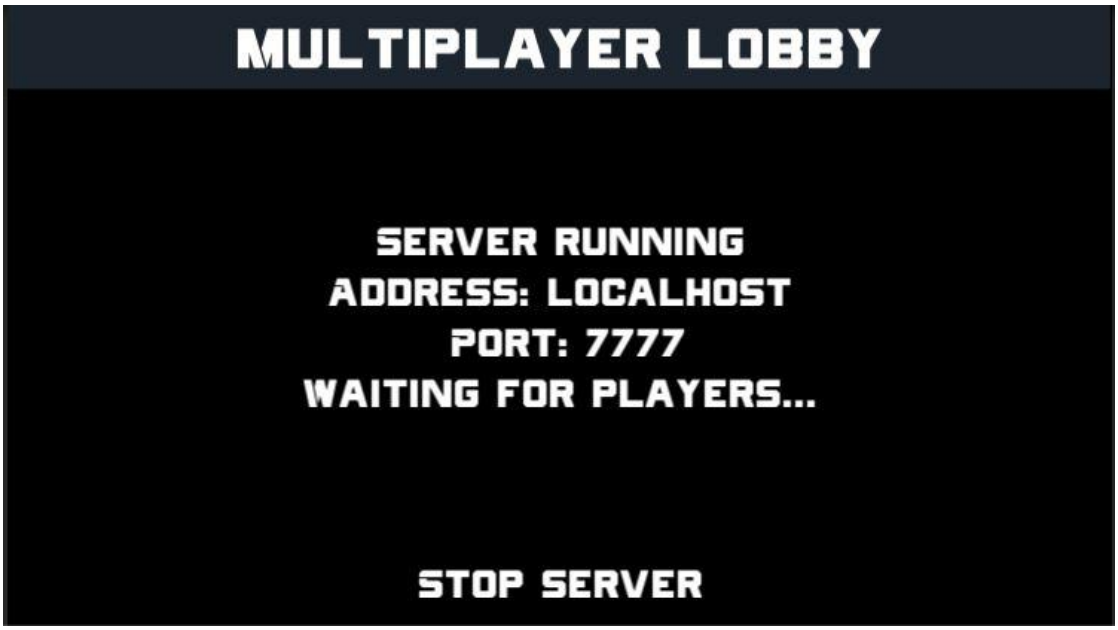


Figure 37. Dedicated server

When two players have joined the same lobby (Figure 38), their LobbyPlayer instances are represented by two panels; Player 1 and Player 2. Each of these panels holds a button for sending a ready to begin message to the server. When both players have checked their status as ready, the game scene loading initiates.

Figure 38. Two players in lobby

As advised in Chapter 3, the NetworkLobbyManager has the lobby and game scenes assigned into their appropriate slots, as well as the player and registered spawnable prefabs (Figure 16). Because a custom interface for the lobby was needed, a modified NetworkManagerHUD script instructs the manager through the formerly mentioned button functions. Appendix 8 presents this script in its entirety.

To mention an alternative option to custom lobby UI creation, the Unity Technologies offer a sample lobby asset package for beginners in the Asset Store, which provides a simple drag-and-drop solution.

5.1.2 LobbyPlayer

The LobbyPlayer GameObject is instantiated from a prefab, with local player authority set in the NetworkIdentity component, when a client connects to a server. It visually represents this specific connected player and holds the ready to begin flag for it.

The ready status can only be controlled by the local player via an OnClick () event, which executes a Command function:

```
[Command]
public void CmdPlayerReady()
{
    if(!isLocalPlayer)
    {
        return;
    }

    if (readyToBegin == false)
    {
        SendReadyToBeginMessage ();

    }
    else if (readyToBegin == true)
    {
        SendNotReadyToBeginMessage ();

    }
}
```

If the status is set as “not ready” and the player sends an OnClick () event by clicking the button attached to the player object, it sends the ready to begin message as well as changes the button’s text to “Ready”; and vice versa. The LobbyPlayers persists to exist until the client disconnects from the server, although the panels and buttons are deactivated from code when both players are ready. Appendix 9 shows the C# code for LobbyPlayer functionality in a custom NetworkLobbyPlayer script.

5.1.3 Game scene

The game scene itself behaves very similarly to its single player counterpart. The only differences between these two are the absence of AI controlled objects; the BattleManager object now organizes the turn sequence for two online players and the presence of two NetworkStartPositions. Only the starting positions contain real networking behaviour for the actual scene and are registered automatically by the NetworkLobbyManager, as it is carried over from the lobby scene. The relevant online functions in the game are performed by the GamePlayer objects.

5.1.4 GamePlayer

During runtime there are only two instances of the GamePlayer prefab existing in the game scene; one instance for each player. These objects perform their tasks through

an **action selection** script which is functionally tied to a canvas. By activating and de-activating script components present in these objects and depending on which action has been chosen from the canvas, the action selection handles moving, attacking, passing and skipping turns as well as spawning of networked animation objects. Figure 39 displays all the attached components of the player object.

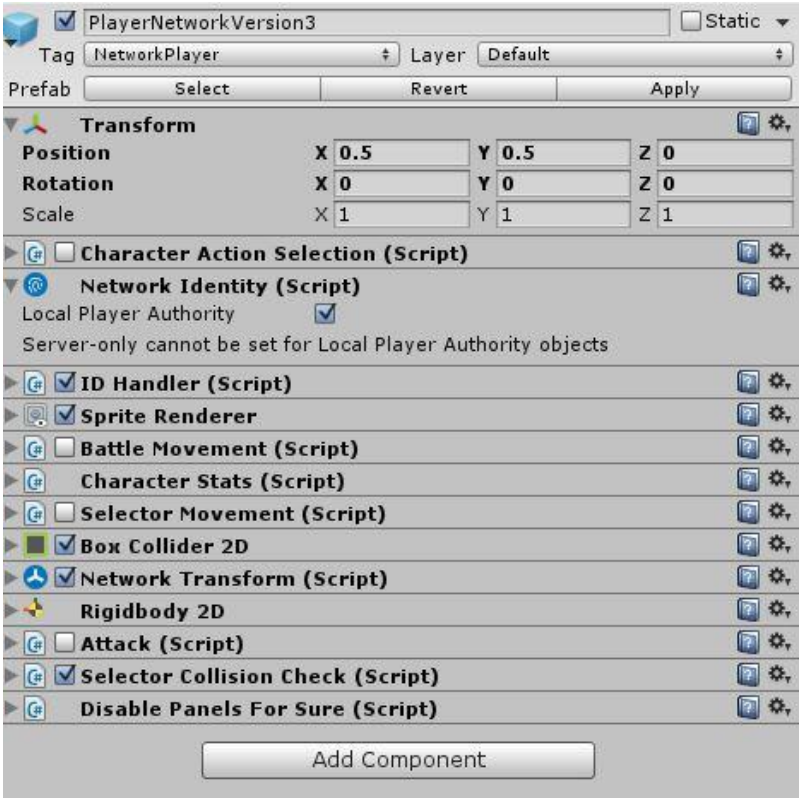


Figure 39. Network GamePlayer

The action selecting respects local player authority and exits if someone else is trying to access it. The functionality itself is carried out through a series of Commands and ClientRPCs. A perfect example of this is the choice of turn skipping. When player input for selecting it has been received, it sends a Command to the server that the player wants to skip a turn. If the server receives this Command, it executes a ClientRPC function which tells both players that the turn has been skipped. The RPC function then compares the two NetworkIDs found in both players and transfers the turn to the player with an id that differs from the current. While it is not the local player's turn, all canvas actions are locally frozen. The turn skipping functionality written in C# is as follows:

```
[Command]
public void CmdSkipTurn()
{
    Debug.Log ("Skipping turn");

    RpcEnableOther ();
}

[ClientRpc]
void RpcEnableOther()
{
    selectorScript.isMoving = false;
    selectorScript.isAttacking = false;

    hasMoved = false;
    hasActed = false;
    selection = 0;

    BattleCanvas.SetActive (false);


    GameObject[] players2 = GameObject.FindGameObjectsWithTag ("NetworkPlayer");

    for (int i = 0; i < players2.Length; i++)
    {
        if (players2 [i].GetComponent<NetworkIdentity> ().netId.Value !=
            gameObject.GetComponent<NetworkIdentity>().netId.Value)
        {
            players2[i].GetComponent<CharacterActionSelection>().enabled = true;

            this.enabled = false;
        }
    }
}
```

The **movement** of the GamePlayer object should be automatically synchronized by the NetworkTransform component. In this turn-based case, however, old sprites may still linger on the clients and cause seemingly endless instances of players littering the arena when they move around. This problem can be solved through some “remote procedure call” magic. At first, a networked BattleMovement script orders the player to move into a new position on the arena, while a Command spawns corresponding teleport effect on to the server. Inside this Command the server sends a ClientRPC order to hide the player sprite across the network. Similarly, when the teleporting effect is over, the player is revealed in its new position to both players, by using a Command and ClientRPC. The beginning of character movement is depicted in the following code.

```
public void MoveCharacterToPlaceTheBeginning(GameObject Character, bool isEnemy)
{

    CmdStealthBegin (Character);

    currentPosition = selectorScript.currentPosition;

    Vector2 oldPosition = new Vector2 (Character.transform.position.x, Character.transform.position.y);

    Character.transform.position = currentPosition + new Vector2(0, 0.25f);


    if (isEnemy == false)
    {
        Destroy (MovementTiles);
    }
}

[Command]
public void CmdStealthBegin(GameObject Character)
{
    GameObject teleportEffect2 = (GameObject)Instantiate
        (teleportEffect, Character.transform.position + new Vector3(0,0.75f,0), Quaternion.identity);

    NetworkServer.Spawn (teleportEffect2);

    Destroy (teleportEffect2, 2.0f);

    RpcHide1 (Character);
}

[ClientRpc]
public void RpcHide1(GameObject Character)
{
    Character.GetComponent<SpriteRenderer> ().color = col1;
}
```

From the networking point of view, **attacking** works very similarly as the movement does. For example, when a character manages to hit another in the arena, two Commands are sent in order to spawn an effect and deal damage.

```
int hitchange = 90 - (CharacterTakingTheHit.GetComponent<CharacterStats> ().currentSpeed
    - CharacterAttacking.GetComponent<CharacterStats> ().currentSpeed);

if (Random.Range(0, 100) <= hitchange)
{
    // It's a hit!
    pos = CharacterTakingTheHit.transform.position;

    CmdHit (pos);

    CmdDamage (CharacterTakingTheHit, CharacterAttacking);
}
```

Another good example of Commands and RPCs is when the character who deals damage utilizes melee weapons. A corresponding effect for it will be spawned by sending a Command, however, a ClientRPC is needed to flip the effect in relation to the characters. The next code snippet demonstrates the implementation of this functionality.

```
[Command]
public void CmdMelee(Vector3 attackDirection, Vector3 pos)
{
    GameObject Melee = (GameObject)Instantiate (MeleeEffect, pos, Quaternion.identity);
    NetworkServer.Spawn (Melee);

    RpcFlip (Melee, attackDirection);

    Destroy(Melee, 1f);
}

[ClientRpc]
public void RpcFlip(GameObject Melee, Vector3 attackDirection)
{
    if (attackDirection.x >= 0)
    {
        Melee.GetComponent<SpriteRenderer> ().flipX = true;
    }

    else
    {
        Melee.GetComponent<SpriteRenderer> ().flipX = false;
    }
}
```

When a character gets hit by an attack or a skill is used in order to heal it, the **synchronized character stat values** are needed. The variables holding these values are set for synchronization by simply tagging them as SyncVars. The values are then updated across the network into the statistic panel; as seen before in Chapter 4, Figure 26. An example of synchronized character stat variables can be inspected below, in addition to a damage dealing Cmd function which accesses some of these values.

```
public class CharacterStats : NetworkBehaviour
{
    [SyncVar]
    public int currentHealth, maxHealth;
    [SyncVar]
    public int currentAttack, maxAttack;
    [SyncVar]
    public int currentDefense, maxDefense;
```

```
[Command]
public void CmdDamage(GameObject CharacterTakingTheHit, GameObject CharacterAttacking)
{
    int damage = CharacterAttacking.GetComponent<CharacterStats> ().currentAttack
        - CharacterTakingTheHit.GetComponent<CharacterStats> ().currentDefense;

    if (damage > 0)
    {
        CharacterTakingTheHit.GetComponent<CharacterStats> ().currentHealth -= damage;
    }

    Debug.Log (CharacterAttacking.name + " attacks " + CharacterTakingTheHit.name +
                " for " + damage + " damage!");
}
```

The Networked animations were created from their single player counterparts. The conversion was a simple process of adding a NetworkIdentity and NetworkAnimator components into the effect objects. The default Animator needs to be given onto the NetworkAnimator by dragging it into the corresponding slot. Neither local authority nor server only is checked on the NetworkIdentity component as local authority is mainly for player Commands and server only for objects that exist only on the server, such as enemy spawners. A GameObject which contains an animation for a melee hit is displayed in Figure 40.

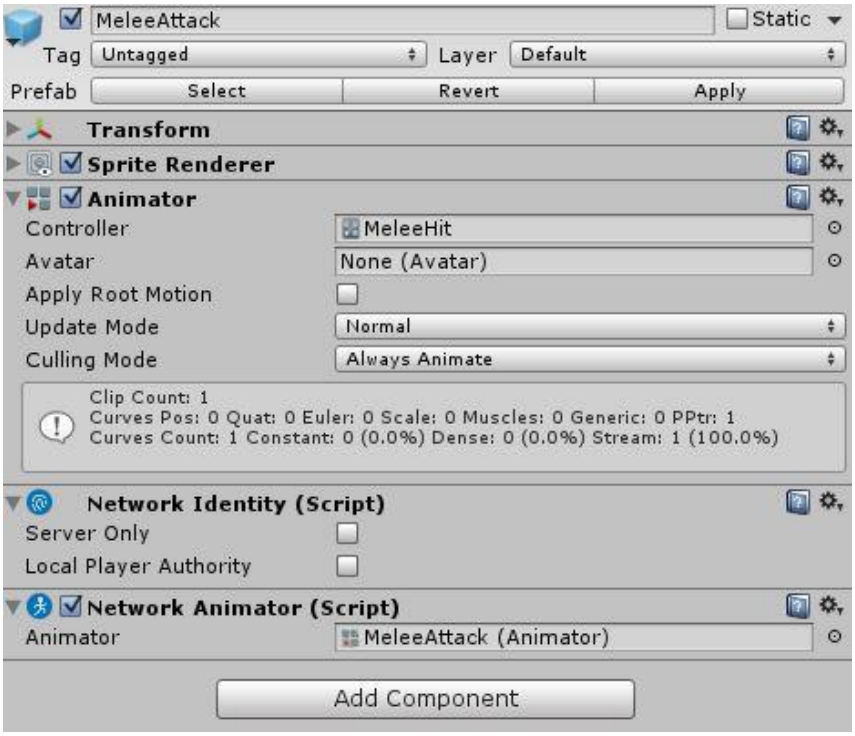


Figure 40. Networked animation object

The three most relevant GamePlayer NetworkBehaviour scripts are displayed in Appendices 10 – 12. Figure 41 shows two players in two separate windows, while connected into the same game. This gives also a sense of the programming process of the network functionality, as with every minor change the game has to be built again and executed in two separate instances.

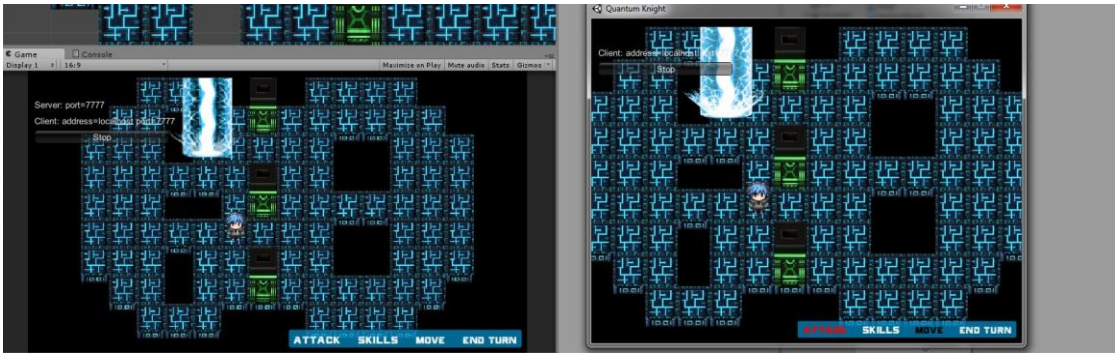


Figure 41. Networking in action

5.2 Photon Unity Networking

5.2.1 Lobby Scene

The actual implementation of Photon networking functionality for the lobby scene is at the same time very similar and also completely different when compared with UNet. The main difference between these two is the absence of a dedicated server and local hosting options in the lobby menu for Photon. Photon is mainly focused on the cloud server services it provides and the creation of rooms. While it is possible for players to create their own servers and host locally, it is somewhat impractical and redundant in this context. Figure 42 demonstrates the Photon lobby main menu.

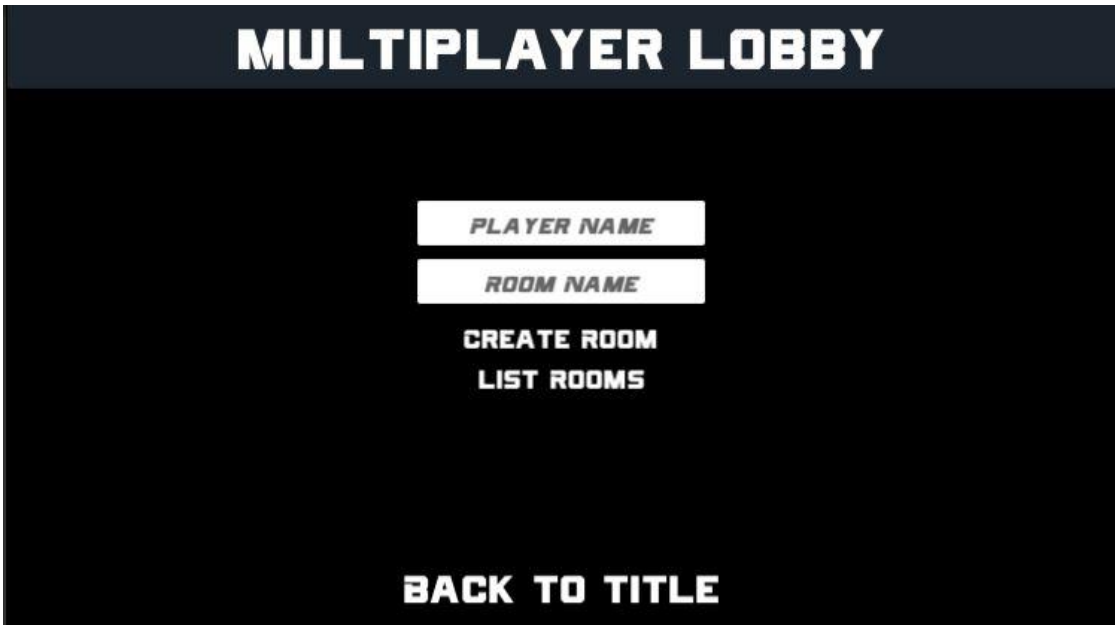


Figure 42. Photon lobby main menu

To begin with, the users are automatically connected to the lobby system when the main menu has been opened. This is achieved by checking the “Auto-Join Lobby” from the PhotonServerSettings file and using the following code in Awake (), in a script which handles the network functionalities of the lobby.

```
if (PhotonNetwork.connectionStateDetailed == PeerState.PeerCreated)
{

    PhotonNetwork.ConnectUsingSettings("1.0");
}
```

After a connection has been established, the users are now able to create rooms with input specified names, list rooms and also set player name aliases.

The **player name** for the local user is randomized when the lobby is accessed for the first time, after which the user may change it at will. In Photon, the usernames of players can easily be stored locally with PhotonNetwork.playerName. When either the room creation or server listing buttons have been clicked, the player name is saved in order to ensure that any changes in the name are properly stored. The first section in the following code takes care of the initial name randomizing, and the second stores the current name when a button has been clicked.

```
if (String.IsNullOrEmpty(PhotonNetwork.playerName))
{
    PhotonNetwork.playerName = "Guest" + Random.Range(1, 9999);

}

inputFName.text = PhotonNetwork.playerName;


//Store name
PhotonNetwork.playerName = inputFName.text;
PlayerPrefs.SetString("playerName", PhotonNetwork.playerName);
```

Reminiscent of UNet, the “**Create Room” button** creates a room with the specified name and connects the user into it. When the room has been created, the appropriate UI panels are set active and a visual representation of the lobby player gets instantiated with PhotonNetwork.Instantiate () inside the OnCreatedRoom callback function. All the buttons in the lobby system execute functions through OnClick () events. The following code showcases the room creation:

```
public void CreateRoomButton ()
{
    roomName = inputFRoom.text;
    PhotonNetwork.playerName = inputFName.text;
    PlayerPrefs.SetString("playerName", PhotonNetwork.playerName);
    PhotonNetwork.CreateRoom(roomName, new RoomOptions() {maxPlayers = 2}, null);
}

public void OnCreatedRoom()
{
    mainPanel.SetActive (false);
    lobbyPanel.SetActive (true);
    GameObject player = PhotonNetwork.Instantiate("LobbyPlayerV2", Vector3.zero, Quaternion.identity, 0);
    player.GetComponentInChildren<Text> ().text = PhotonNetwork.playerName;
    isCreator = true;
}
```

When the **“List Rooms” button** is clicked, it requests a list of currently active games inside a Coroutine and instantiates a visual representation for each of them via the PhotonNetwork.GetRoomList () function. In Photon, it is extremely straightforward to access the roomInfo properties of created rooms; and unlike in the UNet realization, this implementation displays the current and maximum number of players in each room inside the instantiated server objects (Figure 43). If no active rooms are to be found, a “No servers found” text is set as active.

```
serverO.GetComponentInChildren<Text> ().text = roomInfo.name +
    "    " + roomInfo.playerCount + "/" + roomInfo.maxPlayers;
```

Figure 43. Server Object

When the corresponding **“Join” button** is clicked, it connects the client to that specific room. The “Join” button then identifies the correct room to connect the client to by using the room’s name; in UNet the identification was handled using the NetworkIDs. Similarly as with the room creation, the lobby player prefab is instantiated when joining a room after interacting with said button. The code behind room joining is as follows:

```
void JoinMatch(String roomInfoName)
{
    PhotonNetwork.JoinRoom(roomInfoName);
}

public void OnJoinedRoom()
{
    Debug.Log ("Joined room");

    if (isCreator == false)
    {
        mainPanel.SetActive (false);
        serverListPanel.SetActive (false);
        lobbyPanel.SetActive (true);
        GameObject player = PhotonNetwork.Instantiate ("LobbyPlayerV2", Vector3.zero, Quaternion.identity, 0);
        player.GetComponentInChildren<Text> ().text = PhotonNetwork.playerName;
    }
}
```

If the client leaves a room while still in the lobby scene, the visual representation of this particular client must be destroyed across the network in order to guarantee that no unwanted player objects remain on other clients. The OnLeftRoom callback function is perfect for this task. In this case, it surveys the PhotonViews of player objects and checks the isMine attributes of both players. According to the returning values of isMine, it destroys the correct lobby player GameObjects.

```
public void OnLeftRoom()
{
    if (GameObject.FindGameObjectsWithTag ("LobbyPlayer").Length == 1)
    {
        if (GameObject.FindGameObjectWithTag ("LobbyPlayer").GetPhotonView ().isMine)
        {
            Destroy (GameObject.FindGameObjectWithTag ("LobbyPlayer"));
        }
    }

    else if (GameObject.FindGameObjectsWithTag ("LobbyPlayer").Length == 2)
    {
        if (GameObject.FindGameObjectsWithTag ("LobbyPlayer") [0].GetPhotonView ().isMine)
        {
            Destroy (GameObject.FindGameObjectsWithTag ("LobbyPlayer") [0]);
        }
        else
        {
            Destroy (GameObject.FindGameObjectsWithTag ("LobbyPlayer") [1]);
        }
    }
}
```

This, and everything else regarding the lobby scene menu functionality, such as the code for the whole server listing process, are to be found in the Appendix 13.

5.2.2 LobbyPlayer

PUN does not have LobbyPlayer objects in the similar sense as in UNet; hence, very little functionality is being handled automatically behind the curtains. In this case, as the client joins a room, the representing player object is instantiated and registered with the PhotonNetwork.Instantiate function, as previously mentioned. The custom lobby player GameObject (see Figure 44) contains a PhotonView component and a Photon.MonoBehaviour script to carry out the required functionalities. The PhotonView component is set to observe the Transform and “Photon Lobby Player“ script in order to update their statuses across the network.

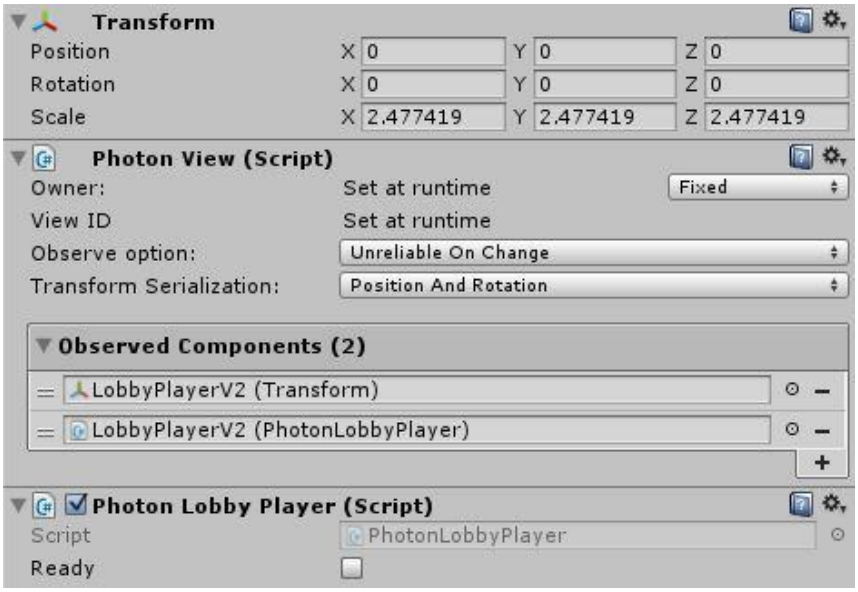


Figure 44. Photon lobby player object

Furthermore, it is not enough to just add components into the observed field of the PhotonView. The necessary values and parameters of these observed components must also be sent and received inside the OnPhotonSerializeView function. When two players have entered the same room, their transformational values and ready statuses are being written and read through the PhotonStream. Even though the positions of lobby player objects do not change after being instantiated into the lobby scene, as the players have no movement controls over them, this ensures they are not accidentally spawned on top of each other. The player objects are instantiated on preset positions for both clients, according to the isMine ownership parameters

of the PhotonView. The next code snippet showcases the reading and writing from the PhotonStream.

```
void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
{
    if (stream.isWriting)
    {
        stream.SendNext(transform.position);
        stream.SendNext(transform.rotation);
        stream.SendNext (ready);


    }
    else
    {
        this.correctPlayerPos = (Vector3)stream.ReceiveNext();
        this.correctPlayerRot = (Quaternion)stream.ReceiveNext();
        this.ready = (bool)stream.ReceiveNext ();
    }
}
```

The toggling of the ready status, however, is a different story; since it can be constantly updated and there is no suitable equivalent to the UNet's SendReadyToBeginMessage () in Photon. In order to achieve similar functionality, a local bool value is being sent and received via the PhotonStream. When the local player toggles the ready status, it is received on the other client's end and vice versa. In order to prevent access into other player's controls, the custom SetReady () function first verifies that this particular photonView.isMine returns true.

```
public void SetReady()
{
    if (photonView.isMine && ready == false)
    {
        ready = true;

    }
    else if (photonView.isMine && ready == true)
    {
        ready = false;
    }
}
```

The name of the remote client is updated locally with photonView.owner.name if the isMine parameter returns false, while the corresponding ready status texts are set normally inside the update function.

```
if (!photonView.isMine)
{
    gameObject.GetComponentInChildren<Text> ().text = photonView.owner.name;
```

When both players have set their status as ready, PhotonNetwork.LoadScene () is called on both players in order to proceed into the actual game scene. Figure 45 demonstrates two players in the same room before the game begins; with separate names and the remote client status set as ready. See Appendix 14 for the whole photon lobby player script.

Figure 45. Two players in Photon lobby

5.2.3 Game Scene

The Photon game scene implementation builds upon its UNet counterpart by introducing a new custom, self-made GameObject called “PhotonBattleFeatures”, which handles the instantiation of player objects, along with an entirely new feature altogether: an in-game chat. The battle features object contains a PhotonView component and is owned by the scene itself, while the BattleManager object still organizes the turn sequence for two online players. Figure 46 displays the PhotonBattleFeatures GameObject and all the components attached into it.

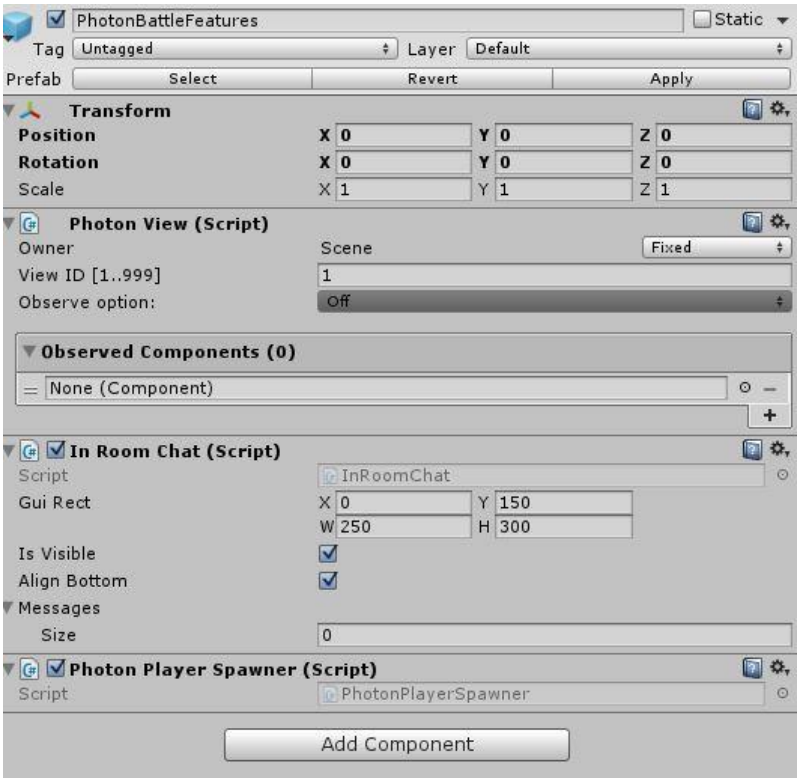


Figure 46. PhotonBattleFeatures GameObject

Currently there are no automatically registered starting position components in Photon, and therefore the positions are set manually through code in the Photon Player Spawner script. When the game scene has been loaded, two player objects are instantiated on the server and then positioned consistently according to the Master Client parameter of the PhotonView. As mentioned before, the Master Client always is the player with the lowest ID in the room, and by means of thorough testing, it is safe to say the player who created the currently present room is always set as the Master Client by default. The following code snippet illustrates the instantiation of player objects, where the Master Client is positioned to the left side of the arena and the other one onto the right side.

```
if (PhotonNetwork.isMasterClient)
{
    GameObject player = PhotonNetwork.Instantiate ("PlayerNetworkPhoton",
                    new Vector3 (-5.5f, 0.5f, 0f), Quaternion.identity, 0);
}
else
{
    GameObject player = PhotonNetwork.Instantiate ("PlayerNetworkPhoton",
                    new Vector3 (5.5f, 0.5f, 0f), Quaternion.identity, 0);
}
```

After examining and researching the Photon documentation and demo projects, it was extremely easy to implement the in-game chat as an “extra” feature into the prototype. A chat window is now displayed on the lower left side of the screen and through it, the players are able to converse in real time while they are in the same room. The chat messages are sent as strings across the network to both players via PunRPC function calls with the name of the sender and shown in a GUILayout field (Figure 47). When the messages that have been sent take up a certain amount of space in the screen, a scrollbar appears next to the field while keeping focus on the most recent message. This prevents the messages from taking up too much screen space.

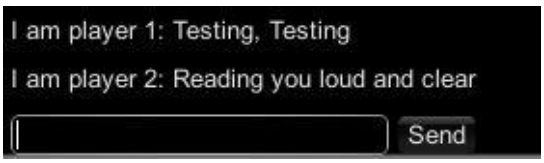


Figure 47. In-game chat

The PunRPC function currently in question demonstrates an excellent example of the chat’s functionality. When the “Send” – button is clicked, it calls the PunRPC on the PhotonView of the PhotonBattleFeatures object; targeting all players and passing the message string as a parameter.

```
this.photonView.RPC("Chat", PhotonTargets.All, this.inputLine);
```

The consecutive C# code showcases the actual remote procedure call function:

```
[PunRPC]
public void Chat(string newLine, PhotonMessageInfo mi)
{
    string senderName = "anonymous";

    if (mi != null && mi.sender != null)
    {
        if (!string.IsNullOrEmpty(mi.sender.name))
        {
            senderName = mi.sender.name;
        }
        else
        {
            senderName = "player " + mi.sender.ID;
        }
    }

    this.messages.Add(senderName +": " + newLine);
}
```

If no connection has been properly established, the in-game chat script exits without executing any chat functionality.

```
if (!this.IsVisible || PhotonNetwork.connectionStateDetailed != PeerState.Joined)
{
    return;
}
```

Furthermore, a “Return to lobby” – button, located on the upper left side of the screen, orders the client to leave the room and initiates the loading of the lobby scene. The in-game chat script can be seen in its entirety in Appendix 15.

5.2.4 GamePlayer

To begin with, the game player prefab and all the animation effect prefabs it instantiates were relocated into the resources folder along with one PhotonView component attached into each of them. The PhotonView of the player object is set to observe its Transform, as well as the Character Stats script in order to synchronize the values it contains. Figure 48 displays all the attached components of the Photon game player object.

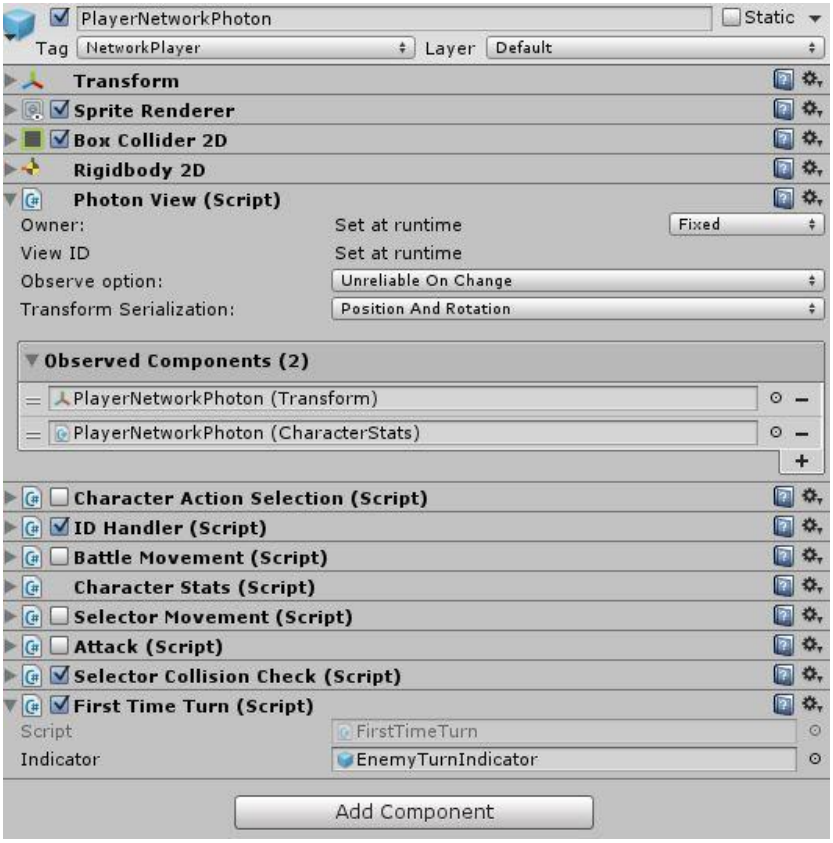


Figure 48. Photon game player object

In this case, the greatest difference between Photon and UNet game player object scripting was how the variables are synchronized across the network. As aforementioned, the PhotonView observes the **Character Stats** script; hence, the values it contains must be sent and received through the PhotonStream. In addition, a remote procedure call function was added into the script in order to properly handle the subtraction of player health.

```
public class CharacterStats : Photon.MonoBehaviour
{

    public int currentHealth, maxHealth;
    public int currentAttack, maxAttack;
    public int currentDefense, maxDefense;
    public int currentMagic, maxMagic;
    public int currentSpeed, maxSpeed;
    public bool isEnemy;
    public bool hasRangedAttack;

    void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
    {
        if (stream.isWriting)
        {
            stream.SendNext (currentHealth);
            stream.SendNext (currentAttack);
            stream.SendNext (currentDefense);
            stream.SendNext (currentMagic);
            stream.SendNext (currentSpeed);
        }
        else
        {
            this.currentHealth = (int)stream.ReceiveNext ();
            this.currentAttack = (int)stream.ReceiveNext ();
            this.currentDefense = (int)stream.ReceiveNext ();
            this.currentMagic = (int)stream.ReceiveNext ();
            this.currentSpeed = (int)stream.ReceiveNext ();
        }
    }

    [PunRPC]
    public void Damage(int dama)
    {
        currentHealth -= dama;
    }
}
```

This time, the **character action selection** script is set to respect local player authority via the photonView.isMine property, and it exits if someone else is trying to access the local controls.

```
void Update ()
{
    if (!photonView.isMine)
    {
        return;
    }

    Action ();
}
```

As previously stated, the Photon only has one type of remote procedure calls, the PunRPC, as opposed to Commands and ClientRPCs of UNet. For example, while the **battle movement** positioning is now automatically synchronized through the PhotonView observer field, the server only needs to be told to hide the player object on the remote clients end. When the StealthBegin () function is executed, it sends an RPC targeting all players and ordering them to hide this object with this PhotonView.

```
public void StealthBegin()
{
    GameObject teleportEffect2 = PhotonNetwork.Instantiate
        ("TeleObject", gameObject.transform.position + new Vector3(0,0.75f,0) , Quaternion.identity, 0);

    StartCoroutine (waitForDestroy (teleportEffect2));

    this.photonView.RPC ("Hide1", PhotonTargets.All);

}

[PunRPC]
public void Hide1()
{
    gameObject.GetComponent<SpriteRenderer> ().color = col1;
}
```

Another good example of PunRPCs is when the players attack and damage each other. If the currently attacking player manages to hit the other player, the **attack** script retrieves the PhotonView from the character which is taking the hit and calls an RPC on its PhotonView. This RPC targets all players to synchronize the damage dealing process across the network and passes the damage attribute into the Character Stats script for health subtraction. On a footnote, it is not possible to pass GameObjects inside PunRPC functions as it is unable to serialize them. The following code showcases the aforementioned damage dealing process.

```
int hitchange = 90 - (CharacterTakingTheHit.GetComponent<CharacterStats> ().currentSpeed
    - CharacterAttacking.GetComponent<CharacterStats> ().currentSpeed);

if (Random.Range(0, 100) <= hitchange)
{
    // It's a hit!
    pos = CharacterTakingTheHit.transform.position;

    Hit (pos);

    int damage = CharacterAttacking.GetComponent<CharacterStats> ().currentAttack
        - CharacterTakingTheHit.GetComponent<CharacterStats> ().currentDefense;

    if (damage > 0)
    {

        PhotonView pv = CharacterTakingTheHit.GetPhotonView ();

        pv.RPC("Damage", PhotonTargets.All, damage);
    }

}
```

A perfect case of the **networked animations** is the “Melee” effect animation. When a character which deals melee damage lands a hit on another character, the melee effect gets instantiated from a prefab through the PhotonNetwork.Instantiate function, to all players in the current room. At the same time, an RPC is called on its PhotonView in order to flip the animation sprites in the right direction. The melee animation object itself contains a script with the said remote procedure call. The instantiated object is then destroyed on all players after one second has passed.

```
public void Melee(Vector3 attackDirection, Vector3 pos)
{
    GameObject Melee = PhotonNetwork.Instantiate ("MeleeAttack", pos , Quaternion.identity, 0);

    PhotonView MpV = Melee.GetPhotonView ();

    MpV.RPC ("Flip", PhotonTargets.All, attackDirection);

    StartCoroutine (waitForDestroy (Melee));
}

IEnumerator waitForDestroy(GameObject obj)
{
    yield return new WaitForSeconds (1.0f);

    PhotonNetwork.Destroy (obj);
}
```

Additionally, the melee animation object has its PhotonView set to observe the Photon Animator View component as can be seen in Figure 49.

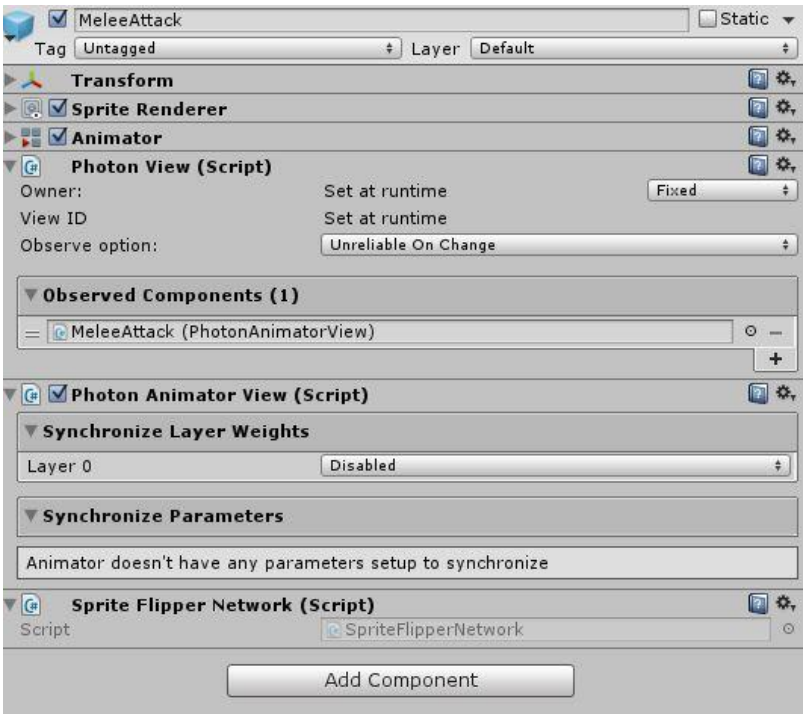


Figure 49. Melee animation object

Apart from the player authority, RPC function calls and networked animations; the movement, attack and other scripts are essentially almost the same as the corresponding scripts of the UNet implementation. To mention few minor improvements over the UNet implementation:

The stats panel now displays the player names correctly by accessing their PhotonViews and retrieving the owner names. Also, when it is the opposing player's turn, the local battle canvas is completely hidden and a flashing "Enemy Turn" GameObject is displayed instead. Figure 50 showcases these functionalities, while Appendices 16-18 display the action selection, movement and attack scripts for comparison between Photon and UNet.

Figure 50. Photon in action

6 Results and Comparison

This chapter compares the two implementation methods against each other according to the evaluation criteria chosen by the author, after which the end results and the acquired networking data are investigated.

6.1 Evaluation

In Table 4, both UNet and Photon are evaluated by means of ten distinct criteria, and given points from zero to two, based on the author's experience; 0 means poor, 1

stands for satisfactory and 2 is exemplary. After the table, each criterion along with its point distribution is shortly explained and justified in text.

Table 4. Comparison

Criterion of Evaluation	UNet	Photon
Components	2	1
Scripting	2	1
Usability	2	1
Performance	1	2
Stability	1	2
Flexibility	1	1
Exportability	2	1
Testability	1	2
Documentation	2	1
Sample projects	0	2
Total points	**14**	**14**

Components

In terms of the number of built-in components and their automatically performed functions, UNet wins hands down. Even though the PhotonView component of Photon is universally usable along with its observer field, the two NetworkManagers of UNet, automatically registered player prefabs and easy to understand NetworkIdentity component allow the development workflow to be more intuitive.

Scripting

In both implementation methods, it is relatively easy and straightforward to utilize the networking classes as well as access the components and their functions through scripting. Naturally, some features are more effortless to achieve with the other, such as sending ready messages in UNet or the usage of player object properties in Photon. However, Photon loses this round as well, since it is not able to serialize GameObjects inside remote procedure calls and the synchronization of variables is considerably more complex.

Usability

As stated earlier, the components of UNet enable the workflow to be more intuitive along with the fact that it has been developed by the same company as the engine itself. Therefore, the UNet offers a more user friendly experience for beginners without the need for additional trickery, such as relocating the networked prefabs under the resources folder, if the developer does not wish to handle instantiating “manually”. Besides, the UNet is guaranteed to stay up to date with the engine, and it is easier for users to request support from the developers.

Performance

While gathering statistical data, it was taken into notice that the UNet version seems to have slightly higher round trip times (RTT). The only visible outcome of this was that the networked animations appeared with a minor delay for the remote player. No such delay was ever observed in the Photon implementation. This topic is further discussed in the Network statistics chapter.

Stability

During the testing process, the version utilizing Photon did not crash even once, and the cloud servers were always in up status, while the UNet project kept crashing randomly. Furthermore, working hours got wasted while the UNet servers were down and no error notifications concerning this were ever received. This led the author to assume there were some faults in the programming, while the truth was quite different. This may originate from the fact that UNet just recently came out of its beta status.

Flexibility

Both methods seem to hold all the necessary tools and functionalities to create various kinds of multiplayer games. However, neither receives two points due to the fact how tedious it is to create turn based controls. The foundations have clearly been built around the development of games with “real time” controls, such as platformers and 3D action games.

Exportability

In short, all UNet and PUN Plus projects can currently be exported to all platforms supported by Unity, while PUN Free exports to *"almost all platforms"*. Furthermore, as Photon is developed by a third party company, some future platforms might be supported with a slight delay or not at all, hence, UNet receives two points and Photon a single point. (Photon – Photon Unity Networking Intro N.d.)

Testability

The testing of a multiplayer game is a tedious process as with every change the project must be rebuilt and run on two separate instances. As this was the reality with both methods the differences come down to the error messages. The messages in UNet are at the moment somewhat insufficient and lack vital information. Photon outperforms UNet in this aspect and it is possible to set the error message logging to even higher levels with PhotonNetwork.loglevel = PhotonLogLevel.Full.

Documentation

Photon has been around longer than UNet and for this reason, it is extraordinary how poor its documentation is. The scripting references are unreadable when compared with the UNet documentation. In addition, some Photon documentation example codes are outright incorrect and defunct.

Sample projects

On the other hand, Photon has more than enough well made, illustrative sample projects which demonstrate the basic networking functionalities satisfyingly. The only official example project released for UNet is the Network Game Lobby (beta) Asset Store package, which does little in the way of coherently introducing the networking features of UNet.

Summary

The comparison between these two different implementation methods results in a tie, which indicates that both practices are viable options, even for beginners, when creating the functionality of multiplayer games. Both approaches excel in different areas: UNet being more intuitive to work with and functionally more advanced, while

Photon is more stable and efficient. As UNet is freshly out of beta, it will be interesting to see how it might evolve in the future and what Photon's response to it will be.

6.2 Network statistics

Unfortunately, it is not possible to use exactly the same tools for both implementation methods in order to gather statistical network data. The Profiler tool of Unity Engine displays data concerning the network traffic of UNet, while the "Photon Stats Gui" component showcases only the information related to Photon. Therefore, the majority of the information these two tools display is not directly comparable. For example, while the Profiler logs all incoming and outgoing, buffered and unbuffered messages frame by frame, it is unclear if the Stats Gui component does the same. The Stats Gui clumps all outgoing and incoming messages together as follows:

Out |In |Sum: 88 | 81 | 169

This makes it impossible to tell the difference between buffered and unbuffered messages; or perhaps the Stats Gui counts merely the other ones, since this issue is not properly documented. Fortunately, there was a single clearly distinguishable property, the round trip time (RTT, also known as ping time), which is the average time in milliseconds until a message is acknowledged by the server and echoed back to its source. Through this value, a simple comparison can be achieved between the UNet matchmaking service and Photon cloud servers, when networked actions are being performed.

The measurements were taken while using the movement, attack and skip turn functions in runtime, with two players in the same game. Each measurement was repeated five times and an average RTT value was calculated for each function. The UNet measurements were obtained by observing the RTT field of the Profiler tool (see Figure 51). The Profiler is most commonly used for game optimization, as it records and displays runtime data regarding, for example, CPU and memory usage in addition to network messages. Clicking inside one of the fields it contains makes it possible to furtherly inspect the recorded data by dragging the vertical bar back and forth.

Figure 51. The Profiler

Consecutively, as the functions were executed in the Photon implementation, the "To Log" button of the Photon Stats Gui component was pressed simultaneously in order to print out the exact RTT values at the right moment. In Figure 52 the RTT, or ping, has been outlined with red.

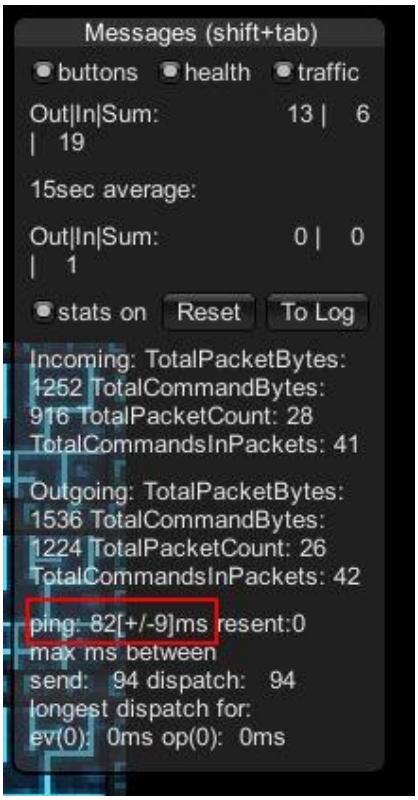


Figure 52. Photon Stats Gui

The following chart presents the average round trip times for all three functions in milliseconds:

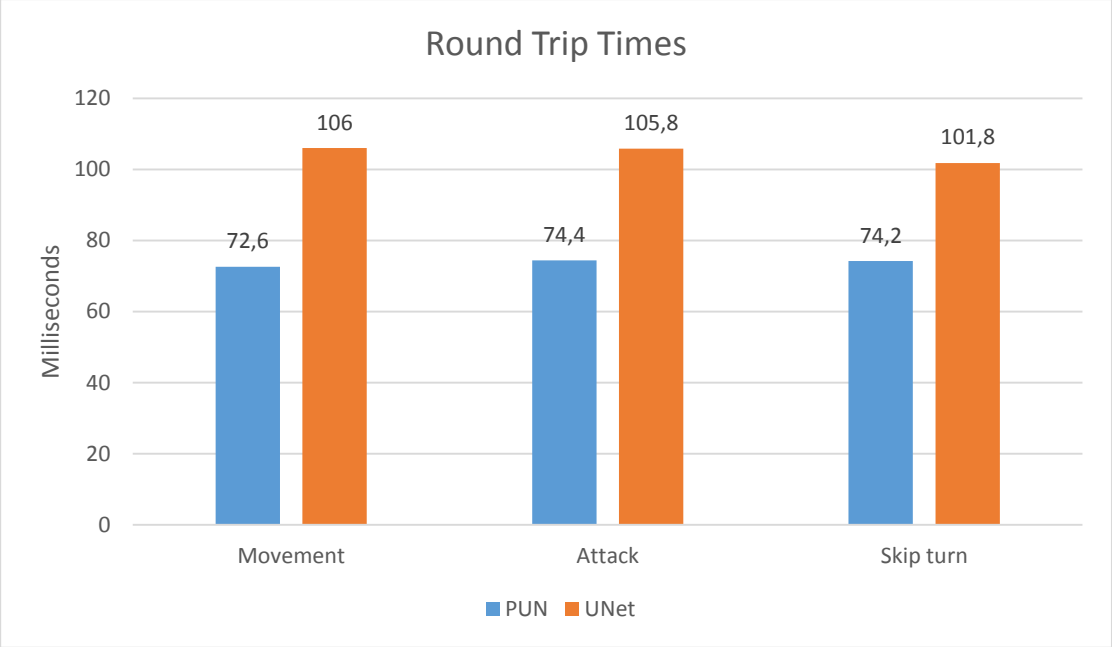


Figure 53. Round Trip Times

As the chart depicts, Photon has lower round trip times for every function. This presumably originates from the fact how the Unity matchmaker servers are globally situated or that the UNet packages being sent are considerably larger in size.

6.3 End results

There were some more or less severe problems along the way of development. The most puzzling UNet issue was encountered while scripting the lobby system. Once the CCU limit had been reached, the matchmaker service seemed to redirect the connection to localhost without any proper notifications from the editor. As it turns out, similar behaviour was discovered even when using the Network Game Lobby (beta) Asset Store package by Unity Technologies. This was the only severe identified UNet problem that remained unsolved after testing.

Similarly, the most substantial issues occurred with the Photon lobby implementation. The entire system broke down systematically when the join button in the room list was clicked and the same client had previously created a room. The only way to

solve this issue in runtime was to back all the way down to the game's title screen. This lead the author to believe there were some faults in the actual network functionality programming. As it turns out, the problem was rather embarrassing to some extent. When the user creates a room, the isCreator bool value for that user is set as true, in order to properly instantiate the joining player. The same value was never set as false when backing out from a room. This lead to the very reason of the whole lobby systematically breaking down along with the loss of several precious working hours.

Nevertheless, in the end, the results are satisfactory since both project variants operate mainly as originally intended. The minor discrepancies between the end results stem from the actuality how various tasks are being carried out differently in UNet and Photon, and also from time constraints. The Photon implementation is a bit more polished due to the fact it was built upon the foundations laid by the UNet project and for having outstanding sample projects to be examined. The circumstances might have probably been completely different if UNet had come out of the beta status sooner.

7 Conclusion

Ultimately, it was the Photon implementation which was chosen by the team to be in the final build of the prototype. The choice resulted directly from the reality of Photon's servers being currently more reliable, along with the in-game chat functionality and overall refinement of the final outcome. However, neither the execution nor outcome is perfect for either version, since both are relatively complex and advanced networking systems, offering functionalities far beyond this thesis. Therefore, both implementation variants hold room for future improvement and investigation, as some issues remained unsolved, such as the aforementioned CCU problem with UNet.

Personally, I found the whole research and development process intriguing and educational. Even though the Unity Engine was a familiar working environment for me from my previous experience with it; both UNet and Photon were not. Initially, before acquiring the basic knowledge concerning these two implementation

methods, I did not have the slightest clue where to even begin. Furthermore, I found myself facing various problems over and over again, for which there was no example code to be found. While this can be considered as a beneficial aspect regarding the learning process, it also led to programming by trial and error. On my part, the whole project realization took approximately 600 hours, when all completed tasks are taken into account along with network programming and writing.

At the end of the day, as chapters 5 and 6 demonstrate, I was able to create two separate, satisfyingly functional lobby systems and gameplay mechanic implementations while using unfamiliar technologies. In my opinion, this fulfills the initially set research goals as well as the betterment of my own professional skills.

References

Fast Facts. 2016. Unity homepage. Accessed on 2016, February 24. Retrieved from https://unity3d.com/public-relations

Get Unity. 2016. Unity homepage. Accessed on 2016, February 24. Retrieved from https://unity3d.com/get-unity

Menard, M. 2011. Game Development with Unity. Course Technology.

Multiplayer video game. 2016. Wikipedia page about multiplayer games. Accessed on 2016, May 2. Retrieved from https://en.wikipedia.org/wiki/Multiplayer_video_game

Online game. 2016. Wikipedia page about online games. Accessed on 2016, May 2. Retrieved from https://en.wikipedia.org/wiki/Online_game

Photon – Feature Overview. N.d. Photon documentation about crucial features. Accessed on 2016, April 14. Retrieved from http://doc.photonengine.com/en/pun/current/getting-started/feature-overview

Photon – Initial Setup. N.d. Photon setup instructions. Accessed on 2016, April 13. Retrieved from http://doc.photonengine.com/en/pun/current/getting-started/initial-setup

Photon – Instantiation. N.d. Photon documentation about network instantiation. Accessed on 2016, April 15. Retrieved from http://doc.photonengine.com/en/pun/current/tutorials/instantiation

Photon – Matchmaking & Room Properties. N.d. Photon documentation about matchmaking. Accessed on 2016, April 14. Retrieved from http://doc.photonengine.com/en/pun/current/tutorials/matchmaking-and-lobby

Photon – Photon Unity Networking Intro. N.d. Documentation about Photon features. Accessed on 2016, April 13. Retrieved from http://doc.photonengine.com/en/pun/current/getting-started/pun-intro

Photon – RPCs and RaiseEvent. N.d. Photon documentaiton about Remote Procedure Calls. Accessed on 2016, April 16. Retrieved from http://doc.photonengine.com/en/pun/current/tutorials/rpcsandraiseevent

Photon – Synchronization and State. N.d. Photon documentation about state synchronization. Accessed on 2016, April 15. Retrieved from http://doc.photonengine.com/en/pun/current/tutorials/synchronization-and-state

Photon – Tutorials: Marco Polo. N.d. Photon tutorial about basic features. Accessed on 2016, April 14. Retrieved from http://doc.photonengine.com/en/pun/current/tutorials/tutorial-marco-polo

Photon – Tutorials: Mecanim Demo. N.d. Photon tutorial about animations. Accessed on 2016, April 14. Retrieved from http://doc.photonengine.com/en/pun/current/tutorials/mecanim-demo

Photon Unity Networking: Class List. N.d. Photon class descriptions. Accessed on 2016, April 13. Retrieved from http://docapi.exitgames.com/en/pun/current/annotated.html

Porter. 2013. Unity: Now you are thinking with components. Article about Unity components at Gamedevelopment.tutsplus. Accessed on 2016, March 12. Retrieved from http://gamedevelopment.tutsplus.com/articles/unity-now-youre-thinking-with-components--gamedev-12492

Sell Assets. 2016. Unity homepage. Accessed on 2016, February 26. Retrieved from https://unity3d.com/asset-store/sell-assets

Stats Monitor – Asset Store. 2015. Statistics about Asset Store assets. Accessed on 2016, February 26. Retrieved from http://assetsreporter.com/statistics-assets-072015/

Unity – Game Engine. 2016. Unity homepage. Accessed on 2016, February 24. Retrieved from https://unity3d.com/unity

Unity (game engine). 2016. Wikipedia. Accessed on 2016, February 24. Retrieved from https://en.wikipedia.org/wiki/Unity_(game_engine)

Unity – Manual: AssetPackages. 2016. Unity documentation on asset packages. Accessed on 2016, March 4. Retrieved from http://docs.unity3d.com/Manual/AssetPackages.html

Unity – Manual: AssetWorkflow. 2016. Unity documentation on assets. Accessed on 2016, March 4. Retrieved from http://docs.unity3d.com/Manual/AssetWorkflow.html

Unity – Manual: BuildSettings. 2016. Unity documentation on the build setting. Accessed on 2016, March 3. Retrieved from http://docs.unity3d.com/Manual/BuildSettings.html

Unity – Manual: CreatingComponents. 2016. Unity documentation on components. Accessed on 2016, March 4. Retrieved from http://docs.unity3d.com/Manual/CreatingComponents.html

Unity – Manual: EditingProperties. 2016. Unity documentation on value editing. Accessed on 2016, March 3. Retrieved from http://docs.unity3d.com/Manual/EditingValueProperties.html

Unity – Manual: ExecutionOrder. 2016. Unity documentation on script lifecycle. Accessed on 2016, March 12. Retrieved from http://docs.unity3d.com/Manual/ExecutionOrder.html

Unity – Manual: GameObject. 2016. Unity documentation on GameObjects. Accessed on 2016, March 4. Retrieved from http://docs.unity3d.com/Manual/class-GameObject.html

Unity – Manual: GameObjects. 2016. Unity documentation on GameObjects. Accessed on 2016, March 4. Retrieved from http://docs.unity3d.com/Manual/GameObjects.html

Unity – Manual: Hierarchy. 2016. Unity documentation on Hierarchy. Accessed on 2016, March 3. Retrieved from http://docs.unity3d.com/Manual/Hierarchy.html

Unity – Manual: Importing from the Asset Store. 2016. Unity documentation on asset workflow. Accessed on 2016, February 26. Retrieved from http://docs.unity3d.com/Manual/AssetStore.html

Unity – Manual: Internet Services. Unity documentation about the matchmaking service. Accessed on 2016, April 6. Retrieved from http://docs.unity3d.com/Manual/UNetInternetServicesOverview.html

Unity – Manual: Layers. 2016. Unity documentation on layers. Accessed on 2016, March 9. Retrieved from http://docs.unity3d.com/Manual/Layers.html

Unity – Manual: MultiSceneEditing. 2016. Unity documentation on scene editing. Accessed on 2016, March 9. Retrieved from http://docs.unity3d.com/Manual/MultiSceneEditing.html

Unity – Manual: Multiplayer Lobby. 2016. Unity documentation on MultiplayerLobbyManager. Accessed on 2016, April 1. Retrieved from http://docs.unity3d.com/Manual/UNetLobby.html

Unity – Manual: NetworkAnimator. 2016. Unity documentation on the NetworkAnimator component. Accessed on 2016, April 3. Retrieved from http://docs.unity3d.com/Manual/class-NetworkAnimator.html

Unity – Manual: NetworkBehaviour. 2016. Unity documentation on NetworkBehaviour class. Accessed on 2016, March 30. Retrieved from http://docs.unity3d.com/Manual/class-NetworkBehaviour.html

Unity – Scripting API: NetworkDiscovery. 2016. Unity scripting references about the NetworkDiscovery component . Accessed on 2016. April 3. Retrieved from http://docs.unity3d.com/ScriptReference/Networking.NetworkDiscovery.html

Unity – Manual: NetworkIdentity. 2016. Unity documentation on NetworkIdentity component. Accessed on 2016, April 3. Retrieved from http://docs.unity3d.com/Manual/class-NetworkIdentity.html

Unity – Manual: NetworkTransform. 2016. Unity documentation on NetworkTransform component. Accessed on 2016, April 3. Retrieved from http://docs.unity3d.com/Manual/class-NetworkTransform.html

Unity – Manual: Networking Overview. 2016. Unity documentation on Networking. Accessed on 2016, March 30. Retrieved from http://docs.unity3d.com/Manual/UNetOverview.html

Unity – Manual: Network System Concepts. 2016. Unity documentation on Networking. Accessed on 2016, March 30. Retrieved from http://docs.unity3d.com/Manual/UNetConcepts.html

Unity – Manual: Object Spawning. 2016. Unity documentation about object spawning in network. Accessed on 2016, April 3. Retrieved from http://docs.unity3d.com/Manual/UNetSpawning.html

Unity – Manual: Object Visibility. 2016. Unity documentation on the NetworkProximityChecker component. Accessed on 2016, April 3. Retrieved from http://docs.unity3d.com/Manual/UNetVisibility.html

Unity – Manual: Prefabs. 2016. Unity documentation on prefabs. Accessed on 2016, March 9. Retrieved from http://docs.unity3d.com/Manual/Prefabs.html

Unity – Manual: ProjectView. 2016. Unity documentation on Project browser. Accessed on 2016, March 3. Retrieced from http://docs.unity3d.com/Manual/ProjectView.html

Unity – Manual: Remote Actions. 2016. Unity documentation about the remote procedure calls in UNet. Accessed on 2016, April 3. Retrieved from http://docs.unity3d.com/Manual/UNetActions.html

Unity – Manual: State Synchronization. 2016. Unity documentation about state synchronization in network. Accessed on 2016, April 3. Retrieved from http://docs.unity3d.com/Manual/UNetStateSync.html

Unity – Manual: Tags. 2016. Unity documentation on tags. Accessed on 2016, March 9. Retrieved from http://docs.unity3d.com/Manual/Tags.html

Unity – Manual: TagManager. 2016. Unity documentation on tags and layers. Accessed on 2016, March 9. Retrieved from http://docs.unity3d.com/Manual/class-TagManager.html

Unity – Manual: The High Level API. 2016. Unity documentation on HLAPI. Accessed on 2016, March 30. Retrieved from http://docs.unity3d.com/Manual/UNetUsingHLAPI.html

Unity – Manual: UsingComponents. 2016. Unity documentation on components. Accessed on 2016, March 4. Retrieved from http://docs.unity3d.com/Manual/UsingComponents.html

Unity – Manual: UsingTheInspector.2016. Unity documentation on the Inspector. Accessed on 2016, March 3. Retrieved from http://docs.unity3d.com/Manual/UsingTheInspector.html

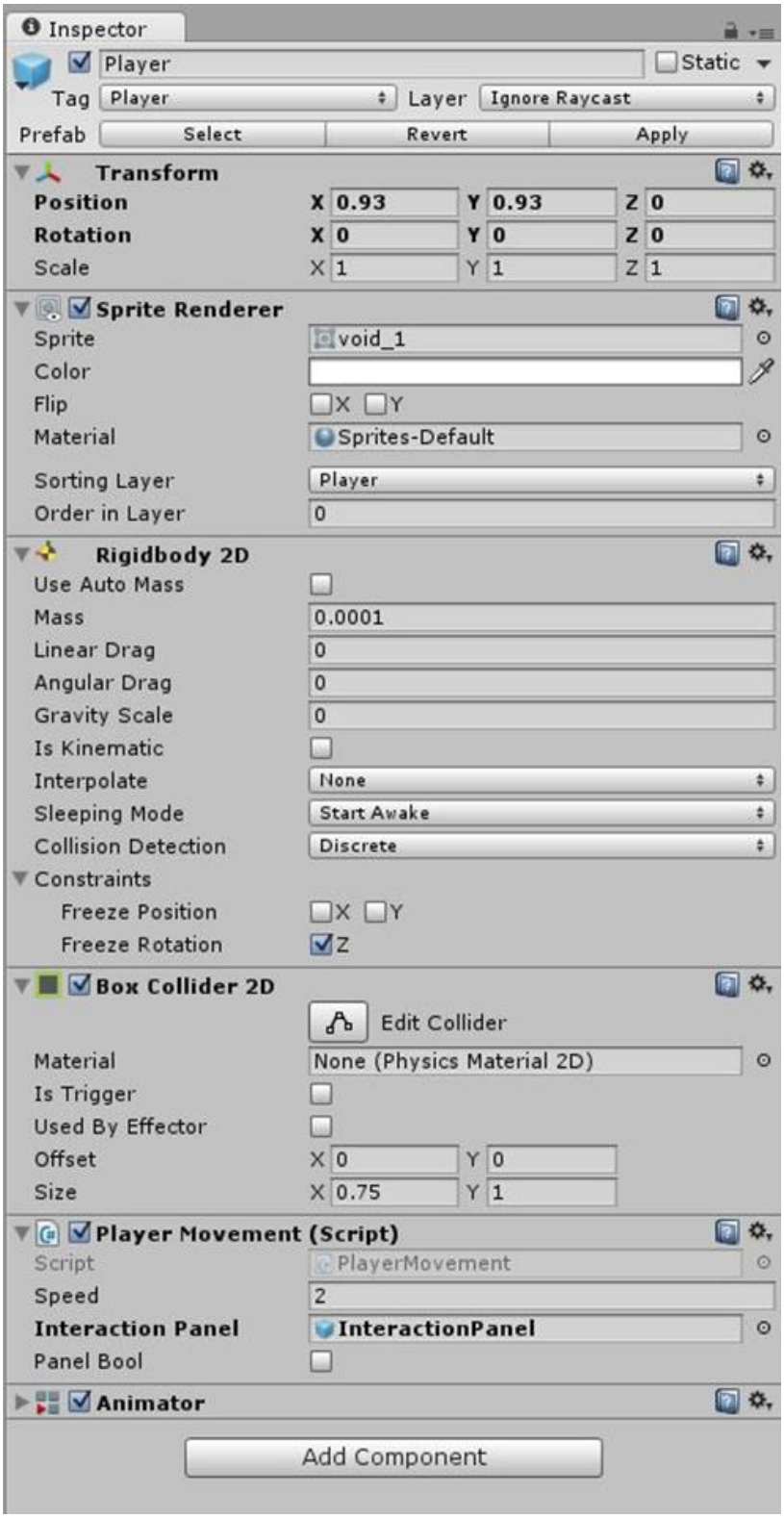
Unity – MonoBehaviour. 2016. Unity documentation on MonoBehaviour. Accessed on 2016, March 12. Retrieved from http://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Virtual Studio Technology. 2016. Wikipedia page about VST plugins. Accessed on 2016, March 16. Retrieved from https://en.wikipedia.org/wiki/Virtual_Studio_Technology

Unity – Manual: Using the NetworkManager. 2016. Unity documentation on NetworkManager. Accessed on 2016, March 31. Retrieved from http://docs.unity3d.com/Manual/UNetManager.html

Appendices

Appendix 1. Inspector view



Appendix 2. MonoBehaviour functions

Function	Usage
Awake	Awake is called when the script instance is being loaded. Awake is used to initialize any variables or game state before the game starts. Awake is called only once during the lifetime of the script instance.
Start	Start is called on the frame when a script is enabled just before any of the Update methods is called the first time. Start is called exactly once in the lifetime of the script.
Update	Update is called once every frame if MonoBehaviour is enabled. Update is the most commonly used function to implement any kind of game behavior.
FixedUpdate	This function is called every fixed framerate frame, if the MonoBehaviour is enabled. FixedUpdate should be used instead of Update when dealing with physics.
LateUpdate	LateUpdate is called after all Update functions have been called.
OnApplicationQuit	Sent to all game objects before the application is quit.
OnApplicationPause	Sent to all game objects when the player pauses.
OnCollisionEnter **OnCollisionEnter2D**	OnCollisionEnter is called when a collider/rigidbody has begun touching another rigidbody/collider.
OnCollisionExit **OnCollisionExit2D**	OnCollisionExit is called when this collider/rigidbody has stopped touching another rigidbody/collider.
OnCollisionStay **OnCollisionStay2D**	OnCollisionStay is called once per frame for every collider/rigidbody that is touching rigidbody/collider.
OnDestroy	Called when MonoBehaviour will be destroyed. OnDestroy will only be called on game objects that have previously been active.
OnEnable	This function is called when a script becomes enabled and active.

OnDisable	Called when a script becomes disabled or inactive. Also called when destroyed.
OnGUI	OnGUI is called for rendering and handling Graphical user interface events. OnGUI implementation might be called several times per frame.
OnTriggerEnter **OnTriggerEnter2D**	OnTriggerEnter is called when a Collider enters a trigger. This message is sent to the trigger collider and the rigid-body (or the collider if there is no rigidbody) that touches the trigger. Trigger events are only sent if one of the colliders also has a rigidbody attached.
OnTriggerExit **OnTriggerExit2D**	OnTriggerExit is called when a Collider has stopped touching a trigger. This message is sent to the trigger and the collider that touches the trigger.
OnTriggerStay **OnTriggerStay2D**	OnTriggerStay is called once per frame for every Col-lider that is touching a trigger. This message is sent to the trigger and the collider that touches the trigger.
GetComponent	Returns specific requested component if the game ob-ject has one attached, null if it doesn't.
Instantiate	This function makes a copy of an object. Instantiate is most commonly used to instantiate projectiles, AI Ene-mies, particle explosions or wrecked object replace-ments from prefabs.
Destroy	Removes a GameObject, component or asset.
StartCoroutine	A function that has the ability to pause execution and return control to Unity but then continue where it left off on the following frame. Declared with a return type of IEnumerator and with yield return statement.

Appendix 3. 2D Collision and Coroutine

```
using UnityEngine;
using System.Collections;

public class Collision2D : MonoBehaviour {

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.tag == "Enemy")
        {
            Destroy (other.gameObject, 0.5f);
        }
    }
}
```

```
using UnityEngine;
using System.Collections;

public class CoroutineExample : MonoBehaviour {

    void Start()
    {
        print("Starting " + Time.time);
        StartCoroutine(WaitAndPrint(2.0F));
        print("Before WaitAndPrint Finishes " + Time.time);
    }

    IEnumerator WaitAndPrint(float waitTime)
    {
        yield return new WaitForSeconds(waitTime);
        print("WaitAndPrint " + Time.time);
    }
}
```

Appendix 4. PlayerMovement script

```
using UnityEngine;
using System.Collections;

public class PlayerMovement : MonoBehaviour {

    float inputX = 0.0f;
    float inputY = 0.0f;
    public float speed;
    private Vector2 movement;
    Animator anim;
    Vector3 previous = Vector3.zero;
    public GameObject InteractionPanel;
    public bool panelBool = false;

    // Use this for initialization
    void Start ()
    {
        anim = GetComponent<Animator> ();
    }

    // Update is called once per frame
    void Update ()
    {

        inputX = Input.GetAxis ("Horizontal");
        inputY = Input.GetAxis ("Vertical");

        movement = new Vector2 (speed * inputX, speed * inputY);

        //Normalizing speed for diagonal movement
        if (movement.magnitude > speed)
        {
            movement = movement.normalized * speed ;
        }

        AnimationControl (inputX, inputY);

    }

    void FixedUpdate()
    {
        transform.Translate (movement * speed * Time.deltaTime);

        InteractRaycast (inputX, inputY);
    }

    void AnimationControl(float inputX , float inputY)
    {
        if (inputY > 0.02f && anim.GetBool("WalkRight") == false && anim.GetBool("WalkLeft") == false)
        {
            anim.SetBool ("WalkUp", true);

            if (anim.GetBool ("WalkUp") == true)
            {
                anim.SetBool ("WalkRight", false);
                anim.SetBool ("WalkLeft", false);
                anim.SetBool ("WalkDown", false);
            }
        }

        else if (inputY == 0.0f)
        {
            anim.SetBool ("WalkUp", false);
        }

        if (inputY < -0.02f && anim.GetBool("WalkRight") == false && anim.GetBool("WalkLeft") == false)
        {
            anim.SetBool ("WalkDown", true);

            if (anim.GetBool ("WalkDown") == true )
```

```
        {
            anim.SetBool ("WalkRight", false);
            anim.SetBool ("WalkLeft", false);
            anim.SetBool ("WalkUp", false);
        }
    }
    else if (inputY == 0.0f)
    {
        anim.SetBool ("WalkDown", false);
    }

    if (inputX > 0.02f && anim.GetBool("WalkUp") == false && anim.GetBool("WalkDown") == false)
    {
        anim.SetBool ("WalkRight", true);
        anim.SetBool ("WalkLeft", false);

        if (anim.GetBool("WalkRight") == true)
        {
            anim.SetBool ("WalkUp", false);
            anim.SetBool ("WalkDown", false);
        }
    }
    else if (inputX == 0.0f)
    {
        anim.SetBool ("WalkRight", false);
    }

    if (inputX < -0.02f && anim.GetBool("WalkUp") == false && anim.GetBool("WalkDown") == false)
    {
        anim.SetBool ("WalkLeft", true);
        anim.SetBool ("WalkRight", false);

        if (anim.GetBool("WalkLeft") == true)
        {
            anim.SetBool ("WalkUp", false);
            anim.SetBool ("WalkDown", false);
        }
    }
    else if (inputX == 0.0f)
    {
        anim.SetBool ("WalkLeft", false);
    }

}

void InteractRaycast(float x, float y)
{
    RaycastHit2D hit = Physics2D.Raycast(transform.position, -Vector2.up);

    Vector3 dir = new Vector2(x,y);

    if (dir == Vector3.zero)
    {
        dir = previous;
    }
    else
    {
        previous = dir;
    }

    hit = Physics2D.Raycast(transform.position, dir, 1);
    Debug.DrawRay(transform.position,dir, Color.green);

    if (hit.collider != null && hit.collider.tag == "Interactable")
    {
        if (panelBool == false && hit.collider.GetComponent<InteractionTest>().enabled == true)
        {
            InteractionPanel.SetActive (true);
            panelBool = true;
        }

        if (Input.GetButtonDown ("Submit") && hit.collider.GetComponent<InteractionTest>().enabled == true)
        {
```

```
            panelBool = false;
            hit.collider.GetComponent<InteractionTest> ().interacted = true;

            InteractionPanel.SetActive (false);
        }
    }

    if (hit.collider == null && panelBool == true)
    {

        InteractionPanel.SetActive (false);
        panelBool = false;
    }

  }
}
```

Appendix 5. NetworkBehaviour functions

Function/Variable	Usage
connectionToClient/ connectionToServer	The NetworkConnection associated with this NetworkIdentity. Only valid for player objects on the server.
hasAuthority	Returns true if this object is the authoritative version of the object in the distributed network application.
isClient	Returns true if running as a client and spawned by a server.
isLocalPlayer	Returns true if this object is the one that represents the player on the local machine. Used to filter out input for non-local players.
isServer	Returns true if this object is active on an active server. Only true if the object has been spawned.
localPlayerAuthority	Value set on the NetworkIdentity. Returns true if this player has authority.
netId	The unique network Id of this object. Assigned at runtime by the network server and will be unique for all objects for that network session.
OnNetworkDestroy	Invoked on clients when the server has caused this object to be destroyed.
OnStartAuthority	Invoked on behaviours that have authority, based on context and localPlayerAuthority. Called after OnStartServer and OnStartClient.
OnStopAuthority	Invoked on behaviours when authority is removed. Called on the client that owns the object.
OnStartClient	Called on every NetworkBehaviour when it is activated on a client. Objects on the host have this function called, as there is a local client on the host. SyncVars on object are guaranteed to be initialized correctly with the latest state from the server.

PreStartClient	A method called on client objects to resolve GameObject references.
OnStartLocalPlayer	Called when the local player object has been set up. Happens after OnStartClient (), as it is triggered by an ownership message from the server. Appropriate place to activate components or functionality that should only be active for the local player.
OnStartServer	Called when the server starts listening.
CmdDelegate	Delegate for Command functions.
EventDelegate	Delegate for Event functions.
OnConnectedToServer	Called on the client when it successfully connects to a server.
OnDisconnectedFrom-Server	Called on the client when the connection was lost or it disconnects from the server.
OnFailedToConnect	Called on the client when a connection attempt fails for some reason.
OnPlayerConnected	Called on the server whenever a new player has successfully connected.
OnPlayerDisconnected	Called on the server whenever a player disconnected from the server.
InvokeCommand	Manually invokes a Command. Returns true if successful.
InvokeRPC	Manually invokes an RPC function. Returns true if successful.
InvokeSyncEvent	Manually invoke a SyncEvent. Returns true if successful.
OnNetworkInstantiate	Called on objects which have been network instantiated. Useful for disabling or enabling components for objects which have been instantiated and their behavior depends on if they are locally or remotely owned.
SetDirtyBit	Used to set the behaviour as dirty, so that a network update will be sent for the object.
ClearAllDirtyBits	Clears all the dirty bits that were set on this script by SetDirtyBits()

Appendix 6. State serialization

```
public class data : NetworkBehaviour
{

   [SyncVar]
   public int int1 = 666;

   [SyncVar]
   public int int2 = 23487;

   [SyncVar]
   public string MyString = "exampleText";
}

public class data : NetworkBehaviour
{

   public int int1 = 666;
   public int int2 = 23487;
   public string MyString = "exampleText";

   public override bool OnSerialize(NetworkWriter writer, bool forceAll)
   {
      if (forceAll)
      {
         // the first time an object is sent to a client, send all the data (and no dirty bits)
         writer.WritePackedUInt32((uint)this.int1);
         writer.WritePackedUInt32((uint)this.int2);
         writer.Write(this.MyString);
         return true;
      }
      bool wroteSyncVar = false;
      if ((base.get_syncVarDirtyBits() & 1u) != 0u)
      {
         if (!wroteSyncVar)
         {
            // write dirty bits if this is the first SyncVar written
            writer.WritePackedUInt32(base.get_syncVarDirtyBits());
            wroteSyncVar = true;
         }
         writer.WritePackedUInt32((uint)this.int1);
      }
      if ((base.get_syncVarDirtyBits() & 2u) != 0u)
      {
         if (!wroteSyncVar)
         {
            // write dirty bits if this is the first SyncVar written
            writer.WritePackedUInt32(base.get_syncVarDirtyBits());
            wroteSyncVar = true;
         }
         writer.WritePackedUInt32((uint)this.int2);
      }
      if ((base.get_syncVarDirtyBits() & 4u) != 0u)
      {
         if (!wroteSyncVar)
         {
            // write dirty bits if this is the first SyncVar written
            writer.WritePackedUInt32(base.get_syncVarDirtyBits());
            wroteSyncVar = true;
         }
         writer.Write(this.MyString);
      }

      if (!wroteSyncVar)
      {
         // write zero dirty bits if no SyncVars were written
         writer.WritePackedUInt32(0);
      }
      return wroteSyncVar;
   }
}
```

Appendix 7. EventHandler script

```
using UnityEngine;
using System.Collections;

public class EventManager : MonoBehaviour
{
   public GameObject ToCyberWorld;
   public bool playerIsInCyberWorld;
   public bool droneEventHasHappened;

   public void EventStater(int index, GameObject sender)
   {
      switch (index)
      {
      default:
         break;

      case 0: // prologue part 1

         GameObject.Find ("PROLOGUE").GetComponent<PrologueEvent> ().pleaseContinue = true;

         break;

      case 1: // prologue part 2

         GameObject.FindWithTag ("Player").GetComponent<PlayerMovement> ().enabled = true;

         GameObject.Find ("PROLOGUE").GetComponent<PrologueEvent> ().Bunk.SetActive (true);

         Destroy (GameObject.Find ("PROLOGUE"));

         break;

      case 2: // To cyber world and back

         if (playerIsInCyberWorld == true)
         {
            playerIsInCyberWorld = false;
         }

         else
         {
            playerIsInCyberWorld = true;
         }

         Destroy (Instantiate(ToCyberWorld, Camera.main.ScreenToWorldPoint(new Vec-
tor3(Screen.width/2, Screen.height/2, Camera.main.nearClipPlane)), Quaternion.Euler(0, 0, -90)), 1f);

         GameObject.Find (sender.name).GetComponent<SceneChanger> ().ActivateSceneChange ();

         break;

      case 3: // drone event

         GameObject.Find("Drone").GetComponent<DroneEvent> ().enabled = true;

         droneEventHasHappened = true;

         break;

      case 4: // battle start

         gameObject.GetComponentInParent<PlayerPositionHandler> ().playerStartPosition = GameOb-
ject.FindWithTag ("Player").transform.position;

         GameObject.Find (sender.name).GetComponent<SceneChanger> ().ActivateSceneChange ();

         break;
      }

   }
}
```

Appendix 8. Custom NetworkManagerHUD

```
#if ENABLE_UNET
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
using UnityEngine.SceneManagement;
using UnityEngine.Networking;
using UnityEngine.Networking.Types;
using UnityEngine.Networking.Match;
namespace UnityEngine.Networking
{
  [AddComponentMenu("Network/NetworkManagerHUD")]
  [System.ComponentModel.EditorBrowsable(System.ComponentModel.EditorBrowsableState.Never)]

  public class NetworkManagerHUD : MonoBehaviour
  {
    public NetworkLobbyManager manager;
    public InputField inputF;
    public GameObject mainPanel;
    public GameObject serverListPanel;
    public GameObject dedicatedPanel;
    public GameObject lobbyPanel;
    public GameObject panels;
    public GameObject noServersPanel;
    public GameObject serverObject;
    string roomName;
    public bool ready = false;
    NetworkMatch networkMatch;
    bool matchCreated;
    bool extraOfExtras = false;

    [SerializeField] public bool showGUI = true;
    [SerializeField] public int offsetX;
    [SerializeField] public int offsetY;


    bool showServer = false;

    void Awake()
    {
      manager = GetComponent<NetworkLobbyManager>();
      panels.SetActive (true);
      networkMatch = gameObject.AddComponent<NetworkMatch>();
    }

    void Update()
    {
```

```
    if(GameObject.FindGameObjectsWithTag("NetworkPlayer").Length == 0)
    {
        panels.SetActive (true);
    }
}

public void LanHost()
{
    if (!NetworkClient.active && !NetworkServer.active && manager.matchMaker == null)
    {
        manager.StartHost();
        mainPanel.SetActive (false);
        lobbyPanel.SetActive (true);
    }
}

public void LanClient()
{
    if (!NetworkClient.active && !NetworkServer.active && manager.matchMaker == null)
    {
        manager.StartClient ();
        mainPanel.SetActive (false);
        lobbyPanel.SetActive (true);
    }
}

public void LanServerOnly()
{
    if (!NetworkClient.active && !NetworkServer.active && manager.matchMaker == null) {
        manager.StartServer ();
        mainPanel.SetActive (false);
        dedicatedPanel.SetActive (true);
        dedicatedPanel.GetComponentInChildren<Text> ().text = "Server running" +
        "\n" + "Address: " + manager.networkAddress + "\n" + " Port: " + manager.networkPort +
        "\n" + "Waiting for players...";
    }
    else
    {
        dedicatedPanel.GetComponentInChildren<Text> ().text = "Server startup error";
    }
}

public void CreateInternetMatch()
{
    if (!NetworkServer.active && !NetworkClient.active)
    {
        if (manager.matchMaker == null)
        {
            manager.StartMatchMaker ();
```

```
            manager.matchMaker.CreateMatch (manager.matchName, manager.matchSize, true, "", manager.OnMatchCreate);
        }
      }
    }

    public void CreateMatch ()
    {
      mainPanel.SetActive (false);
      lobbyPanel.SetActive (true);
      CreateMatchRequest create = new CreateMatchRequest();
      create.name = inputF.text ;
      create.size = 2 ;
      create.advertise = true ;
      create.password = "" ;
      networkMatch.CreateMatch(create, MatchCreated);
    }

    public void MatchCreated (CreateMatchResponse matchResponse)
    {

      if (matchResponse.success)
      {
        matchCreated = true ;
        manager.SetMatchHost ("mm.unet.unity3d.com", 443, true);
        manager.matchInfo  = new MatchInfo(matchResponse);
        Utility.SetAccessTokenForNetwork(matchResponse.networkId, new NetworkAccessToken(matchResponse.ac-
cessTokenString));
        manager.StartHost(manager.matchInfo) ;
      }
    }


    public void GetServerList()
    {
      if (!NetworkServer.active && !NetworkClient.active)
      {
        manager.StartMatchMaker ();
        mainPanel.SetActive (false);
        serverListPanel.SetActive (true);

        if (manager.matchInfo == null)
        {
          if (manager.matches == null)
          {
            manager.matchMaker.ListMatches (0, 10, "", manager.OnMatchList);
            StartCoroutine (WaitForServers());
          }
        }
      }
```

```
}

IEnumerator WaitForServers()
{
   yield return new WaitForSeconds (3);
   int yPos = 0;

   foreach (var match in manager.matches)
   {
      GameObject serverO = (GameObject)Instantiate (serverObject,
                        serverListPanel.transform.position, serverListPanel.transform.rotation);
      serverO.transform.SetParent (GameObject.FindWithTag ("ServerList").transform, false);
      serverO.transform.localPosition = new Vector3 (0, yPos, 0);
      serverO.GetComponentInChildren<Text> ().text = match.name;
      serverO.GetComponentInChildren<Button> ().onClick.AddListener(() => JoinMatch(match.networkId));;

      yPos -= 100;
   }

   if (manager.matches.Count == 0)
   {
      noServersPanel.SetActive (true);
   }
   else
   {
      noServersPanel.SetActive (false);
   }
}

void JoinMatch(Networking.Types.NetworkID id)
{
   serverListPanel.SetActive (false);
   lobbyPanel.SetActive (true);
   manager.matchMaker.JoinMatch (id, "", manager.OnMatchJoined);
}

public void StopServer()
{
   if (NetworkServer.active)
   {
      manager.StopServer();
      dedicatedPanel.SetActive (false);
      mainPanel.SetActive (true);
   }
}

public void StopLanHost()
{
   if (NetworkServer.active || NetworkClient.active)
```

```
        {
            manager.StopHost ();
            lobbyPanel.SetActive (false);
            mainPanel.SetActive(true);
        }
    }

    public void StopMatchMaker()
    {
        manager.StopMatchMaker ();
        lobbyPanel.SetActive (false);
        noServersPanel.SetActive (false);
        serverListPanel.SetActive (false);
        mainPanel.SetActive (true);
    }

    public void BackToTitle()
    {
        Destroy (GameObject.FindWithTag ("EventSystem"));
        GameObject.FindWithTag ("LobbyManager").GetComponent<NetworkLobbyManager> ().dontDestroyOnLoad = false;
        Destroy (GameObject.FindWithTag ("LobbyManager"));

        SceneManager.LoadScene ("Title");
    }

    void OnGUI()
    {
        if (!showGUI)
            return;

        int xpos = 10 + offsetX;
        int ypos = 40 + offsetY;
        int spacing = 24;


        if (NetworkServer.active)
        {
            GUI.Label(new Rect(xpos, ypos, 300, 20), "Server: port=" + manager.networkPort);
            ypos += spacing;
        }
        if (NetworkClient.active)
        {
            GUI.Label(new Rect(xpos, ypos, 300, 20), "Client: address=" + manager.networkAddress + " port=" + manager.network
Port);
            ypos += spacing;
        }


        if (NetworkServer.active || NetworkClient.active)
```

```
        {
            if (GUI.Button(new Rect(xpos, ypos, 200, 20), "Stop"))
            {
                manager.StopHost();
                manager.StopClient ();
                manager.StopServer ();
                panels.SetActive (true);
                dedicatedPanel.SetActive (false);
                lobbyPanel.SetActive (false);
                mainPanel.SetActive (true);
            }
            ypos += spacing;
        }
    }
  }
};
#endif //ENABLE_UNET
```

Appendix 9. NetworkLobbyPlayer script

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Networking;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

public class LobbyPlayerSpawn : NetworkLobbyPlayer
{
    public int id = 0;
    bool boolOfExtras = false;
    public GameObject playerPanel;
    string readyText = "Ready";
    string notReadyText = "Not Ready";

    void Awake()
    {

        if (GameObject.FindGameObjectsWithTag ("LobbyPlayer").Length == 2)
        {
            id = 1;
        }

    }

    void Start()
    {
        if (id == 0)
        {
            gameObject.transform.SetParent (GameObject.FindWithTag ("LobbyManager").transform, false);

            gameObject.GetComponentInChildren<Text> ().text = "Player 1 (HOST)";

            transform.localPosition = new Vector3 (-250, 0, 0);
        }
        else
        {
            gameObject.transform.SetParent (GameObject.FindWithTag ("LobbyManager").transform, false);
            gameObject.GetComponentInChildren<Text> ().text = "Player 2 (CLIENT)";

            transform.localPosition = new Vector3 (250, 0, 0);
        }
    }


    [Command]
    public void CmdPlayerReady()
    {
        if(!isLocalPlayer)
        {
            return;
        }

        if (readyToBegin == false)
        {
            SendReadyToBeginMessage ();

        }
        else if (readyToBegin == true)
        {
            SendNotReadyToBeginMessage ();

        }
    }

    public void OnClientReady()
    {
        if (readyToBegin)
        {
            gameObject.GetComponentInChildren<Button> ().GetComponentInChildren<Text>().text = readyText;
```

```
        }
        else
        {
            gameObject.GetComponentInChildren<Button> ().GetComponentInChildren<Text>().text = notReadyText;
        }
    }

    void Update()
    {
        if (playerPanel.activeSelf == true)
        {
            OnClientReady ();
        }
    }

    void LateUpdate()
    {

        if (GameObject.FindGameObjectsWithTag ("LobbyPlayer").Length == 2 && boolOfExtras == false)
        {
            GameObject player1 = GameObject.FindGameObjectsWithTag ("LobbyPlayer") [0];
            GameObject player2 = GameObject.FindGameObjectsWithTag ("LobbyPlayer") [1];

            if (player1.GetComponent<NetworkLobbyPlayer>().readyToBegin == true && player2.GetComponent<NetworkLobby-
Player>().readyToBegin == true)
            {
                boolOfExtras = true;
                playerPanel.SetActive (false);
            }
        }
    }
}
```

Appendix 10. Character Action Selection script

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Networking;
using System.Collections;
using System.Collections.Generic;

public class CharacterActionSelection : NetworkBehaviour
{
    BattleMovement movementScript;
    public bool hasMoved = false;
    public bool hasActed = false;
    Attack attackScript;
    public int selection;
    float inputXLast;
    public Text[] selectionObjects;
    public List<GameObject> ObstacleColliders;
    public SelectorMovement selectorScript;
    public bool forEnemyPurposes;
    public GameObject BattleCanvas;
    public uint playerID;
    public GameObject[] Allies;
    public bool extra = false;


    void Awake()
    {
        movementScript = gameObject.GetComponent<BattleMovement> ();

        attackScript = gameObject.GetComponent<Attack> ();

        selectorScript = gameObject.GetComponent<SelectorMovement> ();

        selection = 0;
    }


    IEnumerator waitForEnabling()
    {
        Debug.Log ("waiting");

        yield return new WaitForSeconds (0.1f);

        this.enabled = true;
    }



    void OnEnable ()
    {
        playerID = gameObject.GetComponent<NetworkIdentity> ().netId.Value;

        foreach (GameObject collider in ObstacleColliders)
        {
            Destroy (collider);

        }

        BattleCanvas.SetActive (true);

        if (hasMoved && hasActed)
        {
            CmdSkipTurn ();
        }
        else if (hasMoved)
        {
            selection = 0;
        }
        else if (hasActed)
```

```
    {
        selection = 1;
    }
}

[Command]
public void CmdSkipTurn()
{
    Debug.Log ("Skipping turn");

    RpcEnableOther ();
}

[ClientRpc]
void RpcEnableOther()
{
    selectorScript.isMoving = false;
    selectorScript.isAttacking = false;

    hasMoved = false;
    hasActed = false;
    selection = 0;

    BattleCanvas.SetActive (false);


    GameObject[] players2 = GameObject.FindGameObjectsWithTag ("NetworkPlayer");

    for (int i = 0; i < players2.Length; i++)
    {
        if (players2 [i].GetComponent<NetworkIdentity> ().netId.Value !=
            gameObject.GetComponent<NetworkIdentity>().netId.Value)
        {
            players2[i].GetComponent<CharacterActionSelection>().enabled = true;

            this.enabled = false;
        }
    }
}

void Update ()
{
    if (!isLocalPlayer)
    {

        return;
    }

    Action ();
}


public void Action()
{
    float inputX = Input.GetAxis ("Horizontal");

    if (inputX >= 0.5f && inputXLast < 0.5f)
    {
        selection += 1;

        if (hasActed == true && selection == 0)
        {
            selection += 1;
        }

        if (hasMoved == true && selection == 1)
        {
            selection += 1;
        }

        if (selection > 2)
        {
            selection = 0;
```

```
        if (hasActed == true && selection == 0)
        {
            selection += 1;
        }

        if (hasMoved == true && selection == 1)
        {
            selection += 1;
        }
    }

}

else if (inputX <= -0.5f && inputXLast > -0.5f)
{
    selection -= 1;

    if (hasActed == true && selection == 0)
    {
        selection -= 1;
    }

    if (hasMoved == true && selection == 1)
    {
        selection -= 1;
    }

    if (selection < 0)
    {
        selection = 2;

        if (hasActed == true && selection == 0)
        {
            selection -= 1;
        }

        if (hasMoved == true && selection == 1)
        {
            selection -= 1;
        }
    }


}

ShowSelected ();

if (Input.GetButtonDown("Submit"))
{
    // Actions when activated

    switch (selection)
    {
    default:
        break;

    case 0:

        if (hasActed == false)
        {
            selectorScript.isMoving = false;
            selectorScript.isAttacking = true;

            attackScript.enabled = true;
            this.enabled = false;
        }

        break;

    case 1:
```

```
            if (hasMoved == false)
            {
               selectorScript.isMoving = true;
               selectorScript.isAttacking = false;

               movementScript.enabled = true;
               this.enabled = false;
            }

            break;

         case 2:

            CmdSkipTurn ();

            break;
         }
      }

      inputXLast = Input.GetAxis("Horizontal");
   }

   public void ShowSelected() // Shows the selected item graphically, currently changes its color to red
   {
      switch (selection)
      {
      default:
         break;

      case 0:

         selectionObjects [0].GetComponent<Text> ().color = new Color (1f, 0f, 0f, 1f); // Currently se-
lected text color is changed to red


         if (hasMoved == false)
         {
            selectionObjects [1].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f); // Others are returned normal

         }
         else
         {
            selectionObjects [1].GetComponent<Text> ().color = new Color (0f, 0f, 0f, 1f);
         }

         selectionObjects [2].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f);

         break;

      case 1:


         selectionObjects [1].GetComponent<Text> ().color = new Color (1f, 0f, 0f, 1f);


         if (hasActed == false)
         {
            selectionObjects [0].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f);

         }
         else
         {
            selectionObjects [0].GetComponent<Text> ().color = new Color (0f, 0f, 0f, 1f);
         }

         selectionObjects [2].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f);

         break;

      case 2:

         selectionObjects [2].GetComponent<Text> ().color = new Color (1f, 0f, 0f, 1f);
```

```
            if (hasMoved == false)
            {
                selectionObjects [1].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f); // Others are returned normal

            }
            else
            {
                selectionObjects [1].GetComponent<Text> ().color = new Color (0f, 0f, 0f, 1f);
            }

            if (hasActed == false)
            {
                selectionObjects [0].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f);

            }
            else
            {
                selectionObjects [0].GetComponent<Text> ().color = new Color (0f, 0f, 0f, 1f);
            }

            break;
        }
    }
}
```

Appendix 11. Battle Movement script

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;

public class BattleMovement : NetworkBehaviour
{
    public GameObject CharacterToMove;
    public GameObject ColoredTiles;

    [SyncVar]
    public bool readyToMove = false;

    public GameObject teleportEffect;
    public Vector2 currentPosition;

    [SyncVar]
    float timeStart, timeNow;

    [SyncVar]
    private Color col1 = new Color (1, 1, 1, 0);

    [SyncVar]
    private Color col2 = new Color (1, 1, 1, 1);

    public SelectorMovement selectorScript;

    [SyncVar]
    bool timerIsOn = false;

    GameObject MovementTiles;
    CharacterActionSelection actionSelectScript;


    void Awake()
    {
        selectorScript = gameObject.GetComponent<SelectorMovement> ();

        actionSelectScript = gameObject.GetComponent<CharacterActionSelection> ();
    }

    void OnEnable ()
    {

        CharacterToMove = gameObject;

        MovementTiles = (GameObject) Instantiate (ColoredTiles, CharacterToMove.transform.position, Quaternion.identity);

        selectorScript.enabled = true;

        selectorScript.isMoving = true;

        currentPosition = selectorScript.currentPosition;

        selectorScript.spriteRenderer.color = new Color (1, 1, 1, 1);
    }

    void Update ()
    {
        if (Input.GetButtonDown("Cancel"))
        {
            Destroy (MovementTiles);

            selectorScript.isMoving = false;

            actionSelectScript.hasMoved = false;

            actionSelectScript.enabled = true;
```

```
        selectorScript.enabled = false;

        this.enabled = false;
    }

    if (readyToMove)
    {
        MoveCharacterToPlaceTheBeginning (CharacterToMove, false);

        timeStart = Time.time;

        timerIsOn = true;

        readyToMove = false;
    }

    else if(timerIsOn)
    {
        timeNow = Time.time;

        if (timeNow - timeStart >= 1.0f)
        {
            MoveCharacterToPlaceTheEnd (CharacterToMove, false);

            timerIsOn = false;
        }
    }
}


public void MoveCharacterToPlaceTheBeginning(GameObject Character, bool isEnemy)
{

    CmdStealthBegin (Character);

    currentPosition = selectorScript.currentPosition;

    Vector2 oldPosition = new Vector2 (Character.transform.position.x, Character.transform.position.y);

    Character.transform.position = currentPosition + new Vector2(0, 0.25f);


    if (isEnemy == false)
    {
        Destroy (MovementTiles);
    }
}

[Command]
public void CmdStealthBegin(GameObject Character)
{
    GameObject teleportEffect2 = (GameObject)Instantiate
        (teleportEffect, Character.transform.position + new Vector3(0,0.75f,0), Quaternion.identity);

    NetworkServer.Spawn (teleportEffect2);

    Destroy (teleportEffect2, 2.0f);

    RpcHide1 (Character);
}

[ClientRpc]
public void RpcHide1(GameObject Character)
{
    Character.GetComponent<SpriteRenderer> ().color = col1;
}

public void MoveCharacterToPlaceTheEnd(GameObject Character, bool isEnemy)
{

    CmdStealtEnd (Character);

    selectorScript.isMoving = false;
```

```
        if (isEnemy == false)
        {
            actionSelectScript.hasMoved = true;

            actionSelectScript.enabled = true;

            selectorScript.enabled = false;

            this.enabled = false;
        }
    }

    [Command]
    public void CmdStealtEnd(GameObject Character)
    {
        GameObject teleportEffect3 = (GameObject)Instantiate (teleportEffect, Character.transform.position + new Vec-
tor3(0,0.75f,0), Quaternion.identity);

        NetworkServer.Spawn (teleportEffect3);

        Destroy (teleportEffect3, 2.0f);

        RpcHide2 (Character);
    }


    [ClientRpc]
    public void RpcHide2(GameObject Character)
    {
        Character.GetComponent<SpriteRenderer> ().color = col2;
    }
}
```

Appendix 12. Attack script

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;

public class Attack : NetworkBehaviour
{
    public bool isRanged = false;
    public GameObject CharacterAttacking;
    public GameObject CharacterTakingTheHit;
    public SelectorMovement selectorScript;
    public GameObject MeleeAttackTiles, RangedAttackTiles;

    [SyncVar]
    public bool attackNow = false;

    CharacterActionSelection actionSelectScript;
    GameObject AttackTiles;
    public GameObject RangedEffect, MeleeEffect, HitEffect, MissEffect;

    [SyncVar]
    public Vector3 pos;

    void Awake()
    {
        selectorScript = gameObject.GetComponent<SelectorMovement> ();
        actionSelectScript = gameObject.GetComponent<CharacterActionSelection> ();
    }

    void OnEnable ()
    {
        CharacterAttacking = gameObject;

        isRanged = CharacterAttacking.GetComponent<CharacterStats> ().hasRangedAttack;

        if (isRanged == false)
        {
            AttackTiles = (GameObject) Instantiate (MeleeAttackTiles, CharacterAttacking.transform.position, Quaternion.identity);
        }

        else
        {
            AttackTiles = (GameObject) Instantiate (RangedAttackTiles, CharacterAttacking.transform.position, Quaternion.identity);
        }

        selectorScript.isAttacking = true;

        selectorScript.enabled = true;

        selectorScript.spriteRenderer.color = new Color (1, 1, 1, 1);
    }

    void Update ()
    {
        if (Input.GetButtonDown("Cancel"))
        {
            Destroy (AttackTiles);

            selectorScript.isAttacking = false;

            actionSelectScript.hasActed = false;

            actionSelectScript.enabled = true;

            selectorScript.enabled = false;

            this.enabled = false;
        }
```

```
        if (attackNow)
        {
            Destroy (AttackTiles);



            if (isRanged)
            {
                // calculate angle from enemy to player

                float angle = Mathf.Atan2(CharacterTakingTheHit.transform.position.y - CharacterAttacking.transform.position.y,
                    CharacterTakingTheHit.transform.position.x - CharacterAttacking.transform.position.x) * 180 / Mathf.PI;

                pos = CharacterTakingTheHit.transform.position;
                CmdRanged (angle, pos);
            }

            else
            {
                // check from which direction the attack is coming

                Vector3 attackDirection = (CharacterTakingTheHit.transform.position - CharacterAttacking.transform.position).normal-
ized;
                pos = CharacterTakingTheHit.transform.position;

                CmdMelee (attackDirection, pos );

            }

            // hit change calculation, if target's speed is higher -> the change is lower, if attacker's speed is higher -> change is higher

            int hitchange = 90 - (CharacterTakingTheHit.GetComponent<CharacterStats> ().currentSpeed
                - CharacterAttacking.GetComponent<CharacterStats> ().currentSpeed);

            if (Random.Range(0, 100) <= hitchange)
            {
                // It's a hit!
                pos = CharacterTakingTheHit.transform.position;

                CmdHit (pos);

                CmdDamage (CharacterTakingTheHit, CharacterAttacking);
            }

            else
            {
                // miss

                pos = CharacterTakingTheHit.transform.position;
                CmdMiss (pos);
            }

            selectorScript.isAttacking = false;

            actionSelectScript.hasActed = true;

            actionSelectScript.enabled = true;

            selectorScript.enabled = false;

            attackNow = false;

            this.enabled = false;
        }
    }

    [Command]
    public void CmdDamage(GameObject CharacterTakingTheHit, GameObject CharacterAttacking)
    {
        int damage = CharacterAttacking.GetComponent<CharacterStats> ().currentAttack
            - CharacterTakingTheHit.GetComponent<CharacterStats> ().currentDefense;

        if (damage > 0)
```

```
        {
            CharacterTakingTheHit.GetComponent<CharacterStats> ().currentHealth -= damage;
        }

        Debug.Log (CharacterAttacking.name + " attacks " + CharacterTakingTheHit.name +
                " for " + damage + " damage!");
    }

    [Command]
    public void CmdMiss(Vector3 pos)
    {
        GameObject Miss = (GameObject)Instantiate (MissEffect, pos, Quaternion.identity);
        NetworkServer.Spawn (Miss);
        Destroy(Miss, 1f);
    }

    [Command]
    public void CmdHit(Vector3 pos)
    {
        GameObject Hit = (GameObject)Instantiate (HitEffect, pos, Quaternion.identity);
        NetworkServer.Spawn (Hit);
        Destroy(Hit, 1f);
    }



    [Command]
    public void CmdRanged(float angle, Vector3 pos)
    {
        GameObject Ranged = (GameObject)Instantiate (RangedEffect, pos, Quaternion.Euler(0, 0, angle));
        NetworkServer.Spawn (Ranged);
        Destroy(Ranged, 1f);
    }

    [Command]
    public void CmdMelee(Vector3 attackDirection, Vector3 pos)
    {
        GameObject Melee = (GameObject)Instantiate (MeleeEffect, pos, Quaternion.identity);
        NetworkServer.Spawn (Melee);

        RpcFlip (Melee, attackDirection);

        Destroy(Melee, 1f);
    }

    [ClientRpc]
    public void RpcFlip(GameObject Melee, Vector3 attackDirection)
    {
        if (attackDirection.x >= 0)
        {
            Melee.GetComponent<SpriteRenderer> ().flipX = true;
        }

        else
        {
            Melee.GetComponent<SpriteRenderer> ().flipX = false;
        }
    }

}
```

Appendix 13. Photon LobbyManager script

```
using UnityEngine;
using System;
using System.Collections;
using UnityEngine.UI;
using UnityEngine.SceneManagement;
using Random = UnityEngine.Random;

public class LobbyPhoton : Photon.MonoBehaviour {

    public InputField inputFName;
    public InputField inputFRoom;
    public GameObject mainPanel;
    public GameObject serverListPanel;
    public GameObject lobbyPanel;
    public GameObject panels;
    public GameObject noServersPanel;
    public GameObject serverObject;
    string roomName;
    bool isCreator = false;

    public static readonly string SceneNameMenu = "LobbyV2";
    public static readonly string SceneNameGame = "multiArena";

    void Awake()
    {
        panels.SetActive (true);

        PhotonNetwork.automaticallySyncScene = true;

        if (PhotonNetwork.connectionStateDetailed == PeerState.PeerCreated)
        {

            PhotonNetwork.ConnectUsingSettings("1.0");
        }

        if (String.IsNullOrEmpty(PhotonNetwork.playerName))
        {
            PhotonNetwork.playerName = "Guest" + Random.Range(1, 9999);

        }

        inputFName.text = PhotonNetwork.playerName;

    }

    public void CreateRoomButton ()
    {
        roomName = inputFRoom.text;
        PhotonNetwork.playerName = inputFName.text;
        PlayerPrefs.SetString("playerName", PhotonNetwork.playerName);
        PhotonNetwork.CreateRoom(roomName, new RoomOptions() {maxPlayers = 2}, null);
    }

    public void OnCreatedRoom()
    {
        mainPanel.SetActive (false);
        lobbyPanel.SetActive (true);
        GameObject player = PhotonNetwork.Instantiate("LobbyPlayerV2", Vector3.zero, Quaternion.identity, 0);
        player.GetComponentInChildren<Text> ().text = PhotonNetwork.playerName;
        isCreator = true;
    }

    public void GetRoomListButton()
    {

        mainPanel.SetActive (false);
        serverListPanel.SetActive (true);
        PhotonNetwork.playerName = inputFName.text;
```

```
    PlayerPrefs.SetString("playerName", PhotonNetwork.playerName);
    StartCoroutine (WaitForRooms ());
}

IEnumerator WaitForRooms()
{
    yield return new WaitForSeconds (3);
    int yPos = 0;

    foreach (RoomInfo roomInfo in PhotonNetwork.GetRoomList())
    {

        GameObject serverO = (GameObject)Instantiate (serverObject,
            serverListPanel.transform.position, serverListPanel.transform.rotation);

        serverO.transform.SetParent (GameObject.FindWithTag ("ServerList").transform, false);
        serverO.transform.localPosition = new Vector3 (0, yPos, 0);

        serverO.GetComponentInChildren<Text> ().text = roomInfo.name +
            "    " + roomInfo.playerCount + "/" + roomInfo.maxPlayers;

        serverO.GetComponentInChildren<Button> ().onClick.AddListener(() => JoinMatch(roomInfo.name));;

        yPos -= 100;
    }

    if (PhotonNetwork.GetRoomList().Length == 0)
    {
        noServersPanel.SetActive (true);
    }
    else
    {
        noServersPanel.SetActive (false);
    }
}

void JoinMatch(String roomInfoName)
{
    PhotonNetwork.JoinRoom(roomInfoName);
}

public void OnJoinedRoom()
{

    if (isCreator == false)
    {
        mainPanel.SetActive (false);
        serverListPanel.SetActive (false);
        lobbyPanel.SetActive (true);
        GameObject player = PhotonNetwork.Instantiate ("LobbyPlayerV2", Vector3.zero, Quaternion.identity, 0);
        player.GetComponentInChildren<Text> ().text = PhotonNetwork.playerName;
    }
}

public void BackToMainFromRoom()
{
    PhotonNetwork.LeaveRoom ();
    isCreator = false;
    lobbyPanel.SetActive (false);
    noServersPanel.SetActive (false);
    serverListPanel.SetActive (false);
    mainPanel.SetActive (true);

}

public void BacktoMainFromList()
{
    lobbyPanel.SetActive (false);
    noServersPanel.SetActive (false);
    serverListPanel.SetActive (false);
    mainPanel.SetActive (true);
}
```

```
    public void BackToTitle()
    {
        Destroy (GameObject.FindWithTag ("EventSystem"));

        SceneManager.LoadScene ("Title");
    }

    public void OnLeftRoom()
    {
        isCreator = false;

        if (GameObject.FindGameObjectsWithTag ("LobbyPlayer").Length == 1)
        {
            if (GameObject.FindGameObjectWithTag ("LobbyPlayer").GetPhotonView ().isMine)
            {
                Destroy (GameObject.FindGameObjectWithTag ("LobbyPlayer"));
            }
        }

        else if (GameObject.FindGameObjectsWithTag ("LobbyPlayer").Length == 2)
        {
            if (GameObject.FindGameObjectsWithTag ("LobbyPlayer") [0].GetPhotonView ().isMine)
            {
                Destroy (GameObject.FindGameObjectsWithTag ("LobbyPlayer") [0]);
            }
            else
            {
                Destroy (GameObject.FindGameObjectsWithTag ("LobbyPlayer") [1]);
            }
        }
    }
}
```

Appendix 14. Photon Lobby Player script

```
using UnityEngine;
using UnityEngine.UI;
using Photon;
using System.Collections;

public class PhotonLobbyPlayer : Photon.MonoBehaviour {

    private Vector3 correctPlayerPos;
    private Quaternion correctPlayerRot;
    public bool ready = false;
    string readyText = "Ready";
    string notReadyText = "Not Ready";

    public static readonly string SceneNameGame = "multiArena";

    void Update()
    {
        if (!photonView.isMine)
        {
            gameObject.GetComponentInChildren<Text> ().text = photonView.owner.name;

            transform.SetParent (GameObject.FindWithTag ("LobbyManager").transform, false);

            transform.localPosition = new Vector3 (250, 0, 0);
        }
        else
        {
            transform.SetParent (GameObject.FindWithTag ("LobbyManager").transform, false);
            transform.localPosition = new Vector3 (-250, 0, 0);
        }

        OnReady ();

        if (GameObject.FindGameObjectsWithTag ("LobbyPlayer").Length == 2)
        {
            if (GameObject.FindGameObjectsWithTag ("LobbyPlayer") [0].GetComponent<PhotonLobbyPlayer> ().ready == true &&
                GameObject.FindGameObjectsWithTag ("LobbyPlayer") [1].GetComponent<PhotonLobbyPlayer> ().ready == true)
            {
                StartCoroutine (waitForLoading ());
            }
        }
    }

    IEnumerator waitForLoading()
    {
        yield return new WaitForSeconds (0.5f);
        PhotonNetwork.LoadLevel (SceneNameGame);


    }

    void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)
    {
        if (stream.isWriting)
        {
            stream.SendNext(transform.position);
            stream.SendNext(transform.rotation);
            stream.SendNext (ready);


        }
        else
        {
            this.correctPlayerPos = (Vector3)stream.ReceiveNext();
            this.correctPlayerRot = (Quaternion)stream.ReceiveNext();
            this.ready = (bool)stream.ReceiveNext ();
        }
    }
```

```
    public void SetReady()
    {
        if (photonView.isMine && ready == false)
        {
            ready = true;

        }
        else if (photonView.isMine && ready == true)
        {
            ready = false;
        }
    }

    public void OnReady()
    {
        if (ready)
        {
            gameObject.GetComponentInChildren<Button> ().GetComponentInChildren<Text>().text = readyText;

        }
        else
        {
            gameObject.GetComponentInChildren<Button> ().GetComponentInChildren<Text>().text = notReadyText;
        }
    }
}
```

Appendix 15. In-game chat script

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;

[RequireComponent(typeof(PhotonView))]
public class InRoomChat : Photon.MonoBehaviour
{
  public Rect GuiRect = new Rect(0,0, 250,300);
  public bool IsVisible = true;
  public bool AlignBottom = false;
  public List<string> messages = new List<string>();
  private string inputLine = "";
  private Vector2 scrollPos = Vector2.zero;

  public static readonly string ChatRPC = "Chat";

  public void Start()
  {
    if (this.AlignBottom)
    {
      this.GuiRect.y = Screen.height - this.GuiRect.height;
    }
  }

  public void OnGUI()
  {
    if (!this.IsVisible || PhotonNetwork.connectionStateDetailed != PeerState.Joined)
    {
      return;
    }

    if (GUILayout.Button("Return to Lobby"))
    {
      PhotonNetwork.LeaveRoom();
    }

    if (Event.current.type == EventType.MouseDown)
    {
      if (!string.IsNullOrEmpty(this.inputLine))
      {
        this.photonView.RPC("Chat", PhotonTargets.All, this.inputLine);

        this.inputLine = "";
        GUI.FocusControl("");
        return;
      }
      else
      {
        GUI.FocusControl("ChatInput");
      }
    }

    GUI.SetNextControlName("");
    GUILayout.BeginArea(this.GuiRect);

    scrollPos = GUILayout.BeginScrollView(new Vector2(0,100000));
    GUILayout.FlexibleSpace();
    for (int i = 0; i <= messages.Count-1; i++)
    {
      GUILayout.Label(messages[i]);
    }
    GUILayout.EndScrollView();

    GUILayout.BeginHorizontal();
    GUI.SetNextControlName("ChatInput");
    inputLine = GUILayout.TextField(inputLine);
    if (GUILayout.Button("Send", GUILayout.ExpandWidth(false)))
```

```
        {
            this.photonView.RPC("Chat", PhotonTargets.All, this.inputLine);
            this.inputLine = "";
            GUI.FocusControl("");
        }
        GUILayout.EndHorizontal();
        GUILayout.EndArea();
    }

    [PunRPC]
    public void Chat(string newLine, PhotonMessageInfo mi)
    {
        string senderName = "anonymous";

        if (mi != null && mi.sender != null)
        {
            if (!string.IsNullOrEmpty(mi.sender.name))
            {
                senderName = mi.sender.name;
            }
            else
            {
                senderName = "player " + mi.sender.ID;
            }
        }

        this.messages.Add(senderName +": " + newLine);
    }

    public void AddLine(string newLine)
    {
        this.messages.Add(newLine);
    }

    public void OnLeftRoom()
    {
        SceneManager.LoadScene(LobbyPhoton.SceneNameMenu);
    }
}
```

Appendix 16. Character Action Selection script Photon

```
using UnityEngine;
using UnityEngine.UI;
using Photon;
using System.Collections;
using System.Collections.Generic;

public class CharacterActionSelection : Photon.MonoBehaviour
{
  BattleMovement movementScript;
  public bool hasMoved = false;
  public bool hasActed = false;
  Attack attackScript;
  public int selection;
  float inputXLast;
  public Text[] selectionObjects;
  public List<GameObject> ObstacleColliders;
  public SelectorMovement selectorScript;
  public bool forEnemyPurposes;
  public GameObject BattleCanvas;
  public int playerID;
  public GameObject[] Allies;
  public bool extra = false;
  public bool isControllable = false;
  public GameObject Indicator;


  void Awake()
  {
    movementScript = gameObject.GetComponent<BattleMovement> ();

    attackScript = gameObject.GetComponent<Attack> ();

    selectorScript = gameObject.GetComponent<SelectorMovement> ();

    selection = 0;
  }


  IEnumerator waitForEnabling()
  {
    Debug.Log ("waiting");

    yield return new WaitForSeconds (0.1f);

    this.enabled = true;
  }



  void OnEnable ()
  {
    playerID = photonView.ownerId;

    foreach (GameObject collider in ObstacleColliders)
    {
      Destroy (collider);

    }

    if (isControllable)
    {
      BattleCanvas.SetActive (true);

    }

    if (photonView.isMine)
    {
      Indicator.SetActive (false);
```

```
        Destroy (gameObject.GetComponent<FirstTimeTurn> ());
    }

    if (hasMoved && hasActed)
    {
        this.photonView.RPC("SkipTurn", PhotonTargets.All);
    }
    else if (hasMoved)
    {
        selection = 0;
    }
    else if (hasActed)
    {
        selection = 1;
    }
}

[PunRPC]
public void SkipTurn()
{
    Debug.Log ("Skipping turn");

    this.photonView.RPC("EnableOther",PhotonTargets.All);
}

[PunRPC]
void EnableOther()
{
    selectorScript.isMoving = false;
    selectorScript.isAttacking = false;

    hasMoved = false;
    hasActed = false;
    selection = 0;

    BattleCanvas.SetActive (false);


    GameObject[] players2 = GameObject.FindGameObjectsWithTag ("NetworkPlayer");

    for (int i = 0; i < players2.Length; i++)
    {
        if (players2 [i].GetPhotonView().ownerId !=
            gameObject.GetPhotonView().ownerId)
        {
            players2[i].GetComponent<CharacterActionSelection>().enabled = true;

            if (photonView.isMine)
            {
                Indicator.SetActive (true);
                Indicator.transform.localPosition = new Vector3 (0, 0, 0);
            }
            this.enabled = false;
        }
    }
}

void Update ()
{
    if (!photonView.isMine)
    {
        return;

    }

    Action ();
}


public void Action()
{
    float inputX = Input.GetAxis ("Horizontal");
```

```
if (inputX >= 0.5f && inputXLast < 0.5f)
{
   selection += 1;

   if (hasActed == true && selection == 0)
   {
      selection += 1;
   }

   if (hasMoved == true && selection == 1)
   {
      selection += 1;
   }

   if (selection > 2)
   {
      selection = 0;

      if (hasActed == true && selection == 0)
      {
         selection += 1;
      }

      if (hasMoved == true && selection == 1)
      {
         selection += 1;
      }
   }

}

else if (inputX <= -0.5f && inputXLast > -0.5f)
{
   selection -= 1;

   if (hasActed == true && selection == 0)
   {
      selection -= 1;
   }

   if (hasMoved == true && selection == 1)
   {
      selection -= 1;
   }

   if (selection < 0)
   {
      selection = 2;

      if (hasActed == true && selection == 0)
      {
         selection -= 1;
      }

      if (hasMoved == true && selection == 1)
      {
         selection -= 1;
      }
   }


}


ShowSelected ();

if (Input.GetButtonDown("Submit"))
{
   // Actions when activated

   switch (selection)
   {
   default:
```

```
            break;

        case 0:

            if (hasActed == false)
            {
                selectorScript.isMoving = false;
                selectorScript.isAttacking = true;

                attackScript.enabled = true;
                this.enabled = false;
            }

            break;

        case 1:

            if (hasMoved == false)
            {
                selectorScript.isMoving = true;
                selectorScript.isAttacking = false;

                movementScript.enabled = true;
                this.enabled = false;
            }

            break;

        case 2:

            this.photonView.RPC("SkipTurn", PhotonTargets.All);

            break;
        }
    }

    inputXLast = Input.GetAxis("Horizontal");
}

public void ShowSelected() // Shows the selected item graphically, currently changes its color to red
{
    switch (selection)
    {
    default:
        break;

    case 0:

        selectionObjects [0].GetComponent<Text> ().color = new Color (1f, 0f, 0f, 1f); // Currently se-
lected text color is changed to red


        if (hasMoved == false)
        {
            selectionObjects [1].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f); // Others are returned normal

        }
        else
        {
            selectionObjects [1].GetComponent<Text> ().color = new Color (0f, 0f, 0f, 1f);
        }

        selectionObjects [2].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f);

        break;

    case 1:


        selectionObjects [1].GetComponent<Text> ().color = new Color (1f, 0f, 0f, 1f);


        if (hasActed == false)
```

```
            {
                selectionObjects [0].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f);

            }
            else
            {
                selectionObjects [0].GetComponent<Text> ().color = new Color (0f, 0f, 0f, 1f);
            }

            selectionObjects [2].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f);

            break;

        case 2:

            selectionObjects [2].GetComponent<Text> ().color = new Color (1f, 0f, 0f, 1f);

            if (hasMoved == false)
            {
                selectionObjects [1].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f); // Others are returned normal

            }
            else
            {
                selectionObjects [1].GetComponent<Text> ().color = new Color (0f, 0f, 0f, 1f);
            }

            if (hasActed == false)
            {
                selectionObjects [0].GetComponent<Text> ().color = new Color (1f, 1f, 1f, 1f);

            }
            else
            {
                selectionObjects [0].GetComponent<Text> ().color = new Color (0f, 0f, 0f, 1f);
            }

            break;
        }
    }
}
```

Appendix 17. Battle Movement script Photon

```
using UnityEngine;
using System.Collections;
using Photon;
using UnityEngine.Networking;

public class BattleMovement : Photon.MonoBehaviour
{
  public GameObject CharacterToMove;
  public GameObject ColoredTiles;


  public bool readyToMove = false;

  public GameObject teleportEffect;
  public Vector2 currentPosition;


  float timeStart, timeNow;


  private Color col1 = new Color (1, 1, 1, 0);


  private Color col2 = new Color (1, 1, 1, 1);

  public SelectorMovement selectorScript;

  bool timerIsOn = false;

  GameObject MovementTiles;
  CharacterActionSelection actionSelectScript;


  void Awake()
  {
    selectorScript = gameObject.GetComponent<SelectorMovement> ();

    actionSelectScript = gameObject.GetComponent<CharacterActionSelection> ();
  }

  void OnEnable ()
  {

    CharacterToMove = gameObject;

    MovementTiles = (GameObject) Instantiate (ColoredTiles, CharacterToMove.transform.position, Quaternion.identity);

    selectorScript.enabled = true;

    selectorScript.isMoving = true;

    currentPosition = selectorScript.currentPosition;

    selectorScript.spriteRenderer.color = new Color (1, 1, 1, 1);
  }

  void Update ()
  {
    if (!photonView.isMine)
      return;

    if (Input.GetButtonDown("Cancel"))
    {
      Destroy (MovementTiles);

      selectorScript.isMoving = false;

      actionSelectScript.hasMoved = false;
```

```
            actionSelectScript.enabled = true;

            selectorScript.enabled = false;

            this.enabled = false;
        }

        if (readyToMove)
        {
            MoveCharacterToPlaceTheBeginning (false);

            timeStart = Time.time;

            timerIsOn = true;

            readyToMove = false;
        }

        else if(timerIsOn)
        {
            timeNow = Time.time;

            if (timeNow - timeStart >= 1.0f)
            {
                MoveCharacterToPlaceTheEnd (false);

                timerIsOn = false;
            }
        }
    }


    public void MoveCharacterToPlaceTheBeginning(bool isEnemy)
    {

        StealthBegin ();

        currentPosition = selectorScript.currentPosition;

        Vector2 oldPosition = new Vector2 (gameObject.transform.position.x, gameObject.transform.position.y);

        gameObject.transform.position = currentPosition + new Vector2(0, 0.25f);


        if (isEnemy == false)
        {
            Destroy (MovementTiles);
        }
    }


    public void StealthBegin()
    {
        GameObject teleportEffect2 = PhotonNetwork.Instantiate ("TeleObject", gameObject.transform.position + new Vec-
tor3(0,0.75f,0) , Quaternion.identity, 0);

        StartCoroutine (waitForDestroy (teleportEffect2));

        this.photonView.RPC ("Hide1", PhotonTargets.All);

    }

    [PunRPC]
    public void Hide1()
    {
        gameObject.GetComponent<SpriteRenderer> ().color = col1;
    }

    public void MoveCharacterToPlaceTheEnd(bool isEnemy)
    {

        StealthEnd ();
```

```
        selectorScript.isMoving = false;

        if (isEnemy == false)
        {
            actionSelectScript.hasMoved = true;

            actionSelectScript.enabled = true;

            selectorScript.enabled = false;

            this.enabled = false;
        }
    }


    public void StealthEnd()
    {
        GameObject teleportEffect3 = PhotonNetwork.Instantiate ("TeleObject", gameObject.transform.position + new Vec-
tor3(0,0.75f,0) , Quaternion.identity, 0);

        StartCoroutine (waitForDestroy (teleportEffect3));

        this.photonView.RPC ("Hide2", PhotonTargets.All);

    }


    [PunRPC]
    public void Hide2()
    {
        gameObject.GetComponent<SpriteRenderer> ().color = col2;
    }

    IEnumerator waitForDestroy(GameObject obj)
    {
        yield return new WaitForSeconds (2.0f);

        PhotonNetwork.Destroy (obj);
    }
}
```

Appendix 18. Attack script Photon

```
using UnityEngine;
using Photon;
using System.Collections;

public class Attack : Photon.MonoBehaviour
{
    public bool isRanged = false;
    public GameObject CharacterAttacking;
    public GameObject CharacterTakingTheHit;
    public SelectorMovement selectorScript;
    public GameObject MeleeAttackTiles, RangedAttackTiles;


    public bool attackNow = false;

    CharacterActionSelection actionSelectScript;
    GameObject AttackTiles;
    public GameObject RangedEffect, MeleeEffect, HitEffect, MissEffect;


    public Vector3 pos;

    void Awake()
    {
        selectorScript = gameObject.GetComponent<SelectorMovement> ();
        actionSelectScript = gameObject.GetComponent<CharacterActionSelection> ();
    }

    void OnEnable ()
    {

        CharacterAttacking = gameObject;

        isRanged = CharacterAttacking.GetComponent<CharacterStats> ().hasRangedAttack;

        if (isRanged == false)
        {
            AttackTiles = (GameObject) Instantiate (MeleeAttackTiles, CharacterAttacking.transform.position, Quaternion.identity);
        }

        else
        {
            AttackTiles = (GameObject) Instantiate (RangedAttackTiles, CharacterAttacking.transform.position, Quaternion.identity);
        }

        selectorScript.isAttacking = true;

        selectorScript.enabled = true;

        selectorScript.spriteRenderer.color = new Color (1, 1, 1, 1);
    }

    void Update ()
    {
        if (!photonView.isMine)
            return;

        if (Input.GetButtonDown("Cancel"))
        {
            Destroy (AttackTiles);

            selectorScript.isAttacking = false;

            actionSelectScript.hasActed = false;

            actionSelectScript.enabled = true;

            selectorScript.enabled = false;
```

```
            this.enabled = false;
        }

        if (attackNow)
        {
            Destroy (AttackTiles);


            if (isRanged)
            {
                // calculate angle from enemy to player

                float angle = Mathf.Atan2(CharacterTakingTheHit.transform.position.y - CharacterAttacking.transform.position.y,
                    CharacterTakingTheHit.transform.position.x - CharacterAttacking.transform.position.x) * 180 / Mathf.PI;

                pos = CharacterTakingTheHit.transform.position;
                Ranged (angle, pos);
            }

            else
            {
                // check from which direction the attack is coming

                Vector3 attackDirection = (CharacterTakingTheHit.transform.position - CharacterAttacking.transform.position).normal-
ized;
                pos = CharacterTakingTheHit.transform.position;

                Melee (attackDirection, pos );

            }

            // hit change calculation, if target's speed is higher -> the change is lower, if attacker's speed is higher -> change is higher

            int hitchange = 90 - (CharacterTakingTheHit.GetComponent<CharacterStats> ().currentSpeed
                - CharacterAttacking.GetComponent<CharacterStats> ().currentSpeed);

            if (Random.Range(0, 100) <= hitchange)
            {
                // It's a hit!
                pos = CharacterTakingTheHit.transform.position;

                Hit (pos);

                int damage = CharacterAttacking.GetComponent<CharacterStats> ().currentAttack
                    - CharacterTakingTheHit.GetComponent<CharacterStats> ().currentDefense;

                if (damage > 0)
                {

                    PhotonView pv = CharacterTakingTheHit.GetPhotonView ();

                    pv.RPC("Damage", PhotonTargets.All, damage);
                }

            }

            else
            {
                // miss

                pos = CharacterTakingTheHit.transform.position;
                Miss (pos);
            }

            selectorScript.isAttacking = false;

            actionSelectScript.hasActed = true;

            actionSelectScript.enabled = true;

            selectorScript.enabled = false;
```

```
            attackNow = false;

            this.enabled = false;
        }
    }

    public void Miss(Vector3 pos)
    {

        GameObject Miss = PhotonNetwork.Instantiate ("Miss", pos , Quaternion.identity, 0);
        StartCoroutine (waitForDestroy (Miss));
    }


    public void Hit(Vector3 pos)
    {
        GameObject Hit = PhotonNetwork.Instantiate ("GettingHit", pos , Quaternion.identity, 0);
        StartCoroutine (waitForDestroy (Hit));
    }


    public void Ranged(float angle, Vector3 pos)
    {
        GameObject Ranged =  PhotonNetwork.Instantiate ("RangedAttack", pos , Quaternion.Euler(0,0,angle), 0);
        StartCoroutine (waitForDestroy (Ranged));
    }


    public void Melee(Vector3 attackDirection, Vector3 pos)
    {
        GameObject Melee = PhotonNetwork.Instantiate ("MeleeAttack", pos , Quaternion.identity, 0);

        PhotonView MpV = Melee.GetPhotonView ();

        MpV.RPC ("Flip", PhotonTargets.All, attackDirection);

        StartCoroutine (waitForDestroy (Melee));
    }

    IEnumerator waitForDestroy(GameObject obj)
    {
        yield return new WaitForSeconds (1.0f);

        PhotonNetwork.Destroy (obj);
    }

}
```