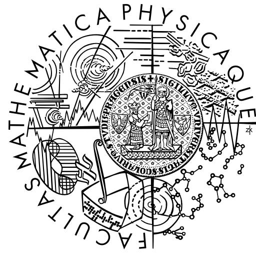


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

BACHELOR THESIS

Gergely Tóth

Sorcerer's Struggle

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Mgr. Jakub Gemrot
Study programme:	Computer Science
Study branch:	IPSS

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

Title: Sorcerer's Struggle

Author: Gergely Tóth

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Department of Software and Computer Science Education

Abstract: This thesis deals with the design and implementation of a multiplayer run and gun game, which can be run on Windows, Linux and MacOS platforms. The thesis contains the selection of a game engine, based on the advantages and disadvantages from the game's viewpoint. Furthermore, the design of a world editor, the problem of serialization and a solution for dynamic image synchronizing is discussed. The result of the work is a two-dimensional game, with a complementing world editor.

Keywords: video game platformer multiplayer multi-platform level editor

I would like to thank Mgr. Jakub Gemrot for the time he spent supervising this thesis. I would also like to thank my family, who have always been there for me.

Contents

Introduction **3**

1 Game inspirations **5**
1.1 Soldat . . . 5
1.2 Worms . . . 6
1.3 Other game mechanics . . . 7
1.4 Points of interest . . . 7

2 Game engine **8**
2.1 Choosing the game engine . . . 8
2.1.1 Unity3D . . . 8
2.1.2 WaveEngine . . . 9
2.1.3 Duality . . . 10
2.1.4 Xenko Game Engine . . . 10
2.1.5 Honorable mentions . . . 10
2.2 A closer look at Unity3D . . . 11
2.2.1 Core Unity functionalities . . . 11
2.2.2 Unity Networking Overview . . . 12

3 Design and implementation of the game **15**
3.1 Lobby room . . . 15
3.1.1 Establishing the connection . . . 15
3.1.2 Lobby room . . . 16
3.2 The game . . . 17
3.2.1 The player object . . . 17
3.2.2 Fireball . . . 19
3.2.3 The tiles shaping the map . . . 19
3.2.4 Neutral enemies . . . 19
3.2.5 Gamemodes . . . 20
3.2.6 Graphics . . . 20
3.2.7 Profiling . . . 21

4 Design of the World Editor **22**
4.1 Implementing the World Editor . . . 22
4.1.1 Creating a visual aid for object placement . . . 22
4.1.2 Creating the custom inspector . . . 23
4.2 TilePickerWindow . . . 24
4.3 Putting it all together . . . 24

5 Map serialization and deserialization **26**
5.1 Determining the goals . . . 26
5.2 Serialization protocols . . . 26

6 **Dynamic image synchronization** **30**
6.1 Requesting and supplying the sprites 31
6.1.1 Image comparison 32
6.2 Preparing the image for the transfer 34
6.2.1 Encoding to JPG 34
6.2.2 Encoding to PNG 34
6.2.3 Zipping 34

Conclusion **35**

Bibliography **36**

List of Figures **39**

User documentation **40**

Introduction

The shoot 'em up video game genre was one of the most popular genres in the history of computer games. This title encapsulates games, where the user is granted control of a spaceship, which tries to make his way through massive waves of enemies by destroying them or dodging their attacks. One of the most popular games of this style - which was also one of the first major arcade game hit - is Asteroids. [1] The run and gun category is one of the subgenres of shoot 'em up. Games belonging to this section have the same base structure as the aforementioned ones, but put more emphasis on evading enemy fire. Typically, the player controls a protagonist fighting on foot and trying to avoid death, while aiming to destroy its enemies. In these type of games the camera's movement is usually defined as continuously moving along the horizontal axis. Although the first representatives of these games were all single player, the advancement of technology made it possible for them to become network aware and therefore multiplayer. These applications allowed friends to play together in a common game environment utilizing a remote server.

In my bachelor thesis I aim to design and implement a multiplayer, two-dimensional run and gun game. The game should also be multi-platform in order for it to be playable by the biggest number of players possible. We will target the three major desktop platforms, these being Windows, Linux and Mac OS X, but omit the deployment for mobile devices, because of the characteristics of the game. Some of these (mainly the precision-based aiming) make it difficult for the player to control the in-game character on a phone.

To look for inspiration for the construction of this application, our first step will be to analyse two games which have gained recognition in their respective fields. After this, I will present eligible game engines which would provide the necessary environment to build our game in and select the one best suited for our purposes. Nowadays, there is a number of great engines out there [2], so we will list their advantages and disadvantages according to the goals we established.

Additionally, in order to prevent the game becoming stale for the player, we will provide a world editor tool for the user to allow him the creation of maps. For this editor, we will prepare tools that are able to shape the map. Moreover, we will permit the end user to choose the spawn positions of the teams and add neutral enemies to the scene.

In the last chapters I will present two technology oriented issues and the process of solving them. The first of these is the serialization of a game map created in the aforementioned world editor. I will present three different methods in order to achieve this, compare their features and performance and pick the one best suited for us. The other problem is the dynamic synchronization of the images used in the game. Because we give the user the option to use any image for the tiles he wishes, we will need a way to synchronize these sprites with the players he chooses to play with as well. I will present the paradigm used to achieve this.

Goals of the thesis

- to gather ideas for a run and gun game
- to choose the most beneficial game engine for the game to be developed in
- the design and implementation the application
- the design and implementation of a world editor for creating custom maps
- solving the problems of serialization and image synchronizing

1. Game inspirations

In this chapter we will present the games that inspired ours. We will highlight their most important characteristics and provide some information about the gameplay mechanics.

1.1 Soldat

Soldat was released in the summer of 2002. Its genre is defined as a basic run and gun, but in reality it is a fast-paced, mass-mulitplayer, precision-based 2D shooter with high customizability (see figure 1.1 for the multiple choices of weapons). Thanks to these attributes and the fact that it is freeware, the game found itself played by a big community. Although nowadays only a few [3] play it mostly only as nostalgia , it was popular a few years back [4] and will serve as a great game idea for us.

Figure 1.1: Screenshot of the game Soldat. [1]

In our game we will try to recreate the atmosphere of this game. For this it is good to note some key aspects of it which we can use later:

1. The movement of the character is fluid. While running, jumping or flying, the player is in constant motion. This is a basic element of a fast-paced game.

[1] Source: `http://www.freegamesutopia.com/public/screenshots/soldat-01.jpg`

2. Adjusting the aim of the weapon is done by just a basic movement of the mouse, but the precision required to be effective with it adds a learning curve to the game.

3. The bullets and projectiles are all affected by gravity adding an extra effect to take into account while trying to hit the opponent.

4. Respawning is instantaneous, meaning the user is alive and playing all the time, keeping him engaged in the game.

1.2 Worms

Worms [5] is a series of games with the first game released in 1995. The gameplay is turn-based, where the goal is to eliminate the enemy team with a wide variety of weapons. Apart from those weapons, there are other utilities available for the player to move him around or to build himself a shelter. Even to this day, the Worms franchise is one of the most popular games of all time (with over 10 million copies sold [6]).

Figure 1.2: Screenshot of the game Worms World Party. [2]

Some great ideas to carry over are:

1. The concept, that the game is playable on almost any map (an example of a procedurally generated map is seen in figure 1.2) is one of the major points of success. Since the map is a main feature in the gameplay - as it determines the positions, allows one but denies another manoeuvre of the player - with it being unique makes every game differ from each other.

2. The number of tools and weapons available. Even in the earlier editions, there are over 40 items to use. [7]

[2] Source: `https://www.team17.com/wp-content/uploads/2015/06/WormsWorldParty_4_11-06-2015.jpg`

1.3 Other game mechanics

A game characteristic that appears in many - mostly role-play - games are abilities based on the class of the character controlled by the player. Although in some games, like *Overwatch* [8], the switch between these classes is allowed during the game, it is mostly common that it remains unchanged.

The user's choice of the class effects the game and his team in a great way. A good team composition would be heterogeneous to divide the different responsibilities amongst the players therefore influence the team performance in a positive way. The cooperation these choices require is often a problem between teammates.

1.4 Points of interest

There are a few ideas we can take away from these examples:

1. The continuous gameplay and flexible movement of Soldat is enticing. It keeps the player completely immersed in the game and the caused atmosphere is unique of Soldat. We will try to reproduce this kind of environment in our game.

2. The aiming technique of the just mentioned game is appealing as well. This kind of functionality adds a certain learning curve to the application and provides a skill, that one can gradually master in order to become better at playing.

3. The map flexibility from Worms adds the necessary variability to the game for it to not become repetitious. By implementing this concept in our application, we open the doors for the player to have different experiences with a single game.

4. The diversity can be further enhanced by giving the player the option to select the class of the character he will be playing (whit each class having different skills).

2. Game engine

In this chapter I will present the game engines I considered for implementing the game and the reasons behind the decision. Then I will briefly present the components and main features of the chosen engine.

2.1 Choosing the game engine

There are multiple options to choose as an engine nowadays. Because of personal preference, I will list here only those that use C# as a scripting language. Note, that because of this we disqualify the likes of UnrealEngine [9] and CryEngine [10], which are otherwise popular and widely-used game engines. [11] For the following pages I will talk about the advantages and disadvantages of the selected engines with respect to the game we want to implement.

2.1.1 Unity3D

Unity is one of the most professional game engines one out there and according to different sources, it is dominating the market. For example, as we can see in figure 2.1, according to a survey conducted in 2014, 62% of video game developers chose Unity to build their games in.

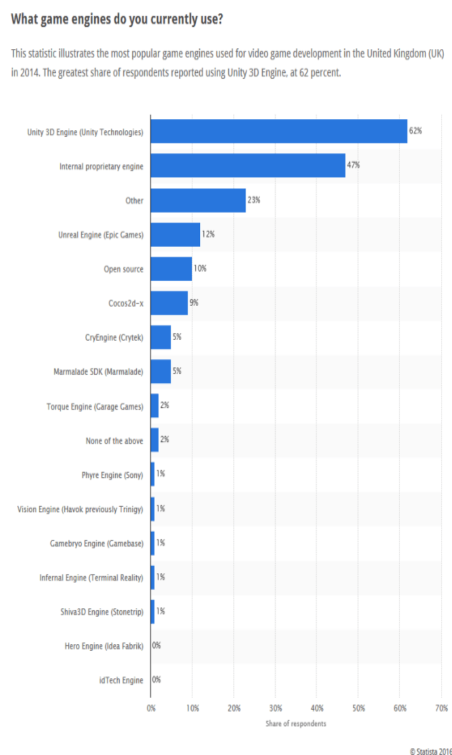


Figure 2.1: Game engine usage in the UK (2014). [1]

[1]Source: https://cdn1.tnwcdn.com/wp-content/blogs.dir/1/files/2016/03/Picture2.png

+ One of the main advantages is its support for over 25 platforms. It allows the developer to deploy the game to almost every big platform, including our three desktop ones: Windows, Mac OS X and Linux. In addition to this, iOS, Android and many others are supported.

+ Although Unity offers three different licenses [12] and the free version is limited in some aspects, it still comes with the core engine features, continuous updates and more, which suffices for most developers who do not want to pay for a more capable version.

+ The editor is powerful and versatile, but equally crowded with a lot of functions which may be overwhelming for beginner game developers.

+ As a direct result of the popularity, the engine has great community support through forums and decent documentation to back that up.

+ Unity provides a system called High Level API for the easier building of multiplayer capabilities, which would help us implement the game more easily.

- Unity is closed source. It is not possible to see the inner workings of it, which could make the debugging process more difficult.

Result: Based on the properties listed, we see that Unity is not considered a popular choice without merit. The engine does have all the functionalities we need and even provides extra tools to make the developing process easier.

2.1.2 WaveEngine

WaveEngine [13] is a cross-platform game engine, with support for both 2D and 3D games. It is relatively young (with the original 1.0 version released in 2011 [14] , but the more significant 2.0 version only in 2015. [15]

+ Wave is capable of targeting the 3 major desktop platforms (being Windows, Linux and Mac) and iOS and Android in addition.

+ The engine is completely free and there are no additional fees or licenses.

+ Several components are open-source and available on GitHub [2], which would allow us to further investigate the mechanics of the engine, if we are faced with a problem.

- There is no support for real-time multiplayer. Even the newest version of the engine provides only turn-based communication over network.

- Bad documentation and as a result of a small community of programmers working in the engine, there is only a small amount of question-answer examples on the forums.

Result: Although the engine looks promising and can be a good fit for some games, the absence of real-time multiplayer makes it purposeless to us.

[2] WaveEngine GitHub page: (`https://github.com/WaveEngine/Components`)

2.1.3 Duality

Duality is an extensible game engine developed specifically for creating 2D games. The project was started in 2011 and it has been updated continuously by the users themselves.

+ Duality is open source. [3] The source code is entirely exposed to the public, which makes it easy for everyone to contribute or debug his own game by seeing the internal works of the engine.

+ It is pointedly made to develop 2D games, consequently the overhead of 3D games present in other, more versatile engines is eliminated.

- No cross-platform support. As stated in the official GitHub page [16], games may run on Linux and Mac, but they are not officially supported.

- No built-in base multiplayer support, which means that we would have to implement the low-level workings of the game's network communication as well.

Result: Duality is a great concept and carries good potential, but factoring in the specifics and goals of our game, the disadvantages make it impractical for us.

2.1.4 Xenko Game Engine

Xenko [17] is a game engine where the emphasis is put mainly on realistic rendering and virtual reality development. With version 2.0 released in 2017 [18] , it is expected to gain more recognition day by day.

+ A free license is available and Xenko offers an educational license for college students. [19]

+ Deployment support for Windows, Linux and all of the major mobile platforms.

- The deployment for Mac is not supported, although the official homepage of Xenko promises more supported platforms to come in the future.

Result: Unfortunately, the lack of Mac support disqualifies Xenko for us, since our goal is to make the game playable on all three major desktop platforms.

2.1.5 Honorable mentions

MonoGame [20] is an open-source game engine with support for the three desktop platforms, iOS, Android and game consoles such as PlayStation4 or Xbox One. It has an active community but minimal documentation. However, MonoGame functions on a lower level than its competitors. This means that it provides more flexibility for the programmer, but also that there are fewer tools available for

[3] Duality source codes: (`https://github.com/AdamsLair/duality`)

the developer to use. An example for this is the lack of the built-in multiplayer support, which - in a similar way as Duality - makes the developer's job harder during the implementation of the network playability (compared to an engine like Unity).
MonoGame is further extended by *Nez* [21], which supplies more useful tools for the aforementioned engine.

Overall result: Based on the listed positive and negative aspects of the presented game engines and the targeted properties and goals of our game taken into consideration, we can eliminate WaveEngine, Duality and Xenko from our pool of choices. The remaining two engines, Unity and MonoGame, both provide the necessary tools and functions for us to implement the game we want.

By directly comparing the two, we must lean towards Unity. The mentioned high level network API, the popularity and the amount of available tutorials make Unity the more eligible option.

2.2 A closer look at Unity3D

To properly explain the implementation of the game, we must first take a closer look at Unity. There are of course far more tools and utilities available in the engine than those listed here, I will try only to present those concepts that are essential to our game. More information can be found in the documentation [22] and in the tutorials provided by Unity [23] or by other services. [24] To develop our game we are using Unity version 5.6.2f1.

2.2.1 Core Unity functionalities

GameObjects [4] are the items that make up a Unity scene. Every GameObject has a number of components (a Transform component is required by all) and scripts (defining the behaviour and the reactions of the GameObject) are considered to be components as well.

Prefabs [5] are GameObjects complete with components and properties. Its purpose is to allow the developer to create an object at runtime with the specified components and properties by simply cloning the prefab and adding the clone to the scene by instantiating it. Prefabs are a key part of Unity and they allow us to create various fundamental entities in the game ranging from the player character through bullets to the tiles in the map without the need to build these objects one by one.

Layers [6] are used to indicate some functionality across different GameObjects. By putting objects on a specific layer we are essentially grouping them together to behave the same way in a given situation. In our game, we will mostly use these layers to check for object collisions. To make this easier, Unity provides a so-called Layer Collision Matrix. This matrix will mark the pairs of layers that

[4] GameObject: (`https://docs.unity3d.com/ScriptReference/GameObject.html`)
[5] Prefab: (`https://docs.unity3d.com/Manual/Prefabs.html`)
[6] Layer: (`https://docs.unity3d.com/Manual/Layers.html`)

are checked for collisions. A good example to demonstrate this is that the layer, which contains the projectiles created to damage opponents, will check for collisions with the default layer (where our player objects are) and react according to their behaviour, but contact with neutral enemies will not be noticed.

Script components define a behaviour for the GameObject they are attached to. Every behaviour script has an *Update()* method, which is called every frame. The majority of the logic directing the game is written in these methods.

Next on the list is to make our GameObjects obey the laws of physics. This can be done by adding a component called RigidBody2D [7] to them and setting the body type property to be Dynamic. There are more settings in this component that can be adjusted to further customize the behaviour of the object.

The framework for creating a user interface is called the UI [8] system. Every UI element has to be a child object of a Canvas object. Once this parent object is created, Unity offers us a collection of interactive items to place in our game: a few of these for example are the Button [9], Text[10] or Slider[11]. We are using this technology to create the lobby room and the heads-up display of our game.

The last core functionality I will mention are Coroutines [12]. A coroutine can be thought of as a function that is executed in intervals. These functions contain yield statements. This statement will temporarily pause the execution of the function, return out of the it and then when the function is called again will resume immediately after the position of the yield command. These *yield return* commands can be accompanied with the instances of WaitForSeconds [13] or WaitUntil[14] objects. These objects will - in the case of the former - suspend the execution of a function for the given amount of time or - in the case of the latter - wait with the continuation until the inserted condition becomes true.

2.2.2 Unity Networking Overview

In order to spare the developer from dealing with the "low-level" technicalities of networking, Unity supplies a so-called "high-level" scripting API [15]. This API will allow us to execute commands on the server called by clients; to instruct a client to carry out a function from the server; or to create complex lobby rooms without the need to implement the details of the networking behaviour. Additionally, we are granted a few more components for our GameObjects that will allow us to make them network aware.

One of these is the NetworkIdentity [16] component. This component will mark the object as unique on the network which will also make it identifiable on every

[7] RigidBody2D: (`https://docs.unity3d.com/Manual/class-Rigidbody2D.html`)
[8] User interface: (`https://docs.unity3d.com/Manual/UISystem.html`)
[9] UI Button: (`https://docs.unity3d.com/ScriptReference/UI.Button.html`)
[10] UI Text: (`https://docs.unity3d.com/ScriptReference/UI.Text.html`)
[11] UI Slider: (`https://docs.unity3d.com/ScriptReference/UI.Slider.html`)
[12] Coroutines: (`https://docs.unity3d.com/ScriptReference/Coroutine.html`)
[13] WaitForSeconds:(`https://docs.unity3d.com/ScriptReference/WaitForSeconds.html`)
[14] WaitUntil: (`https://docs.unity3d.com/ScriptReference/WaitUntil.html`)
[15] HLAPI: (`https://docs.unity3d.com/Manual/UNetUsingHLAPI.html`)
[16] NetworkIdentity: (`https://docs.unity3d.com/Manual/class-NetworkIdentity.html`)

client as well. The two fields - as seen in figure 2.2 - which can be configured are: *Server Only* - marked true, if the object should only exist on the server; and *Local Player Authority* - marked true, if the current (on whose machine this flag is checked) client has the authority over the object. This field will be toggled for the player object whose owner is the current client.

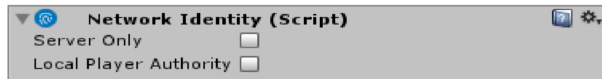


Figure 2.2: NetworkIdentity component. [17]

Every object which has a behaviour that needs to be networked, has to have a behaviour script derived from the NetworkBehaviour [18] base class (instead of MonoBehaviour as discussed in the previous section). It is important to note, that every NetworkBehaviour script will be executed on their attached GameObject on *every* client. In order to only synchronize the movement of the object, it is sufficient to use the NetworkTransform [19] component.

For the general communication between the server and the client, Unity offers a few attributes to mark various methods and give them different network usability. I will present four of these as these often appear in our code:

1. [**Command**] - Methods with this attribute can be invoked on the server from a client. The names of these methods (by Unity standard) must begin with the prefix "Cmd".

2. [**ClientRpc**] - The behaviour of these methods is basically the same as with Command, only working in the opposite direction. These methods can be invoked on the client from the server. Name of the method must begin with the prefix "Rpc".

3. [**TargetRpc**] - The methods marked by this attribute can be invoked from the server on a specific client. The client is specified by the first attribute of the method, which must an object instance of the NetworkConnection [20] type. Name of the method must begin with the prefix "Target".

In order to assure that these methods are called with the correct parameters across the network, the types of the parameters are restricted. This is because every parameter needs to be serialized to be transferred over the network. Valid parameter types among others are basic types, Vector instances and most importantly a NetworkIdentity object. Thanks to the latter can GameObject instances be identified on every client.

[17] Source: `https://docs.unity3d.com/uploads/Main/NetworkIdentityScreenshot.png`

[18] NetworkBehaviour:(`https://docs.unity3d.com/ScriptReference/Networking.NetworkBehaviour.html`)

[19] NetworkTransform:(`https://docs.unity3d.com/ScriptReference/Networking.NetworkBehaviour.html`)

[20] NetworkConnection:(`https://docs.unity3d.com/ScriptReference/Networking.NetworkConnection.html`)

The last element of the Unity network system described here is the [**SyncVar**] attribute. This attribute can be put on variables in NetworkBehaviour classes. The values of these variables will be synchronized from the server to the clients. It is important to note that this synchronization behaves similarly like a ClientRpc call, thus the same rules apply for the type of these variables as for the parameters in the aforementioned methods (with the notable absence of the NetworkIdentity type).

3. Design and implementation of the game

In this chapter I will present the design of the game, as well as the decisions made during the development of it. Firstly, we will have to create a lobby room, where the server-client connections can be established, the players can choose their name and the class of their characters, and the properties of the game can be set. After then, we will implement the game itself and revisit the choices made during the progress.

3.1 Lobby room

The lobby room serves as the preparation stage of the game. After the user has chosen the properties of his character, he can send a signal to the host saying that he is ready and prepared for the game to start. The host player is given the option to make changes to the settings of the game as well, most commonly to select a map or the game mode. However, in order for the player to get to this stage, the connections have to be stabilized and the roles of a server and a client sorted out.

3.1.1 Establishing the connection

As a result of our choice of a game engine we are given a set of useful tools to help us with this process. We will utilize the NetworkLobbyManager class, which comes with the high-level API mentioned in the previous chapter. In our application we will derive a class from this one whose behaviour we will customize in order to fit our needs (figure 3.1).

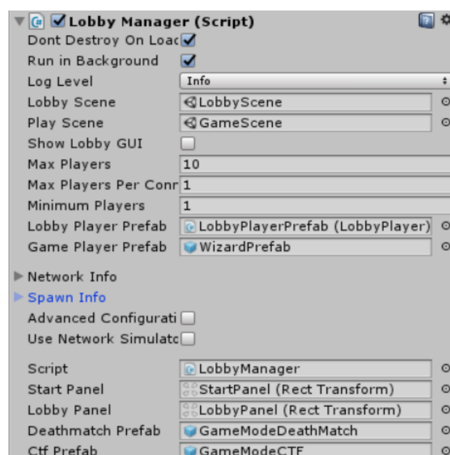


Figure 3.1: Our customized LobbyManager.

There are two kinds of techniques used to establish the connection in our program. The first, more advanced and more convenient way is to use the built-in matchmaking system implemented in the mentioned NetworkLobbyManager.

This system, with the help of another service supplied by the engine called the Unity Multiplayer [25], grants us permission to make use of a server provided by Unity for these purposes. It allows us to create a match on a cloud server, which can be then accessed from anywhere without the need of a public IP address. However, convenience comes with the prize of using a third-party service which will always have its disadvantages. Although a network failure is not expected at a big company like Unity is, the players will still have to rely on another party besides themselves.

The matchmaker works in the following way: at the moment when the player chooses to create a match on the server, an asynchronous method is called which registers the game on the cloud. The command requires a name for the server and allows optional parameters like a password in order to support the establishment of private rooms. When the operation completes, the player is set as the one with host privileges.

On the client side, the user chooses to list the matches currently available on the server. This operation is page-based, meaning that the client can control the amount of games the function returns. This makes it easier to display them in a panel as well. When the player chooses to join a game, an asynchronous request is sent to the server. If the procedure is successful, the user enters the lobby as a client.

The second, more impractical but independent technique is by using a manual connection. In this event the client is needed to be in the possession of the public IP address of the host. This IP address needs to be shared with the client via a third-party messaging service. Nowadays access to one of these services is not a problem, albeit the sharing of this specific address to a lot of people can be proven to be difficult and time-consuming. Because of the reasons listed, a manual connection is more suitable for a small group of players. We have to keep in mind that before attempting to create this type of connection, it is important to adjust the firewall settings of the system.

At the start of the game, the player has to pick one from these options:

1. Create a game on the remote server as part of the matchmaking process and join it immediately as the host.

2. List all the games found on the remote server and join one of them as a client.

3. Choose to function as a host of a manual connection.

4. Enter the IP address of the host and connect to it as a client through manual connection.

Although the technicalities are different, these choices will ultimately lead to a set lobby room, where one of the players functions as the host and the others as the clients.

3.1.2 Lobby room

There are 3 choices every player has to make before the start of the game: decide on the name of the character, select its class and pick the team he will be playing

for. Besides these, the host will have to pick the game mode and opt for a map to play on. Every player has a ready button at his disposal to click when he decides that his choices are final, although it is good practice to allow him to change his mind and readjust something. After all the players have set their state to ready, the host is able to start the game.

The maximum number of players in a game is set to 10, which would ideally mean that 5 players would be playing against 5, but these kind of decisions are left for the users to decide. The lobby manager supports a 1 against 9 composition as well, if all players deem this to be fair (by clicking the ready button).

After entering the lobby, the player is presented with the appropriate GUI elements that allow him to set up the character in the way mentioned before. The selected values are synchronized throughout the network, which gives the option for a player to react to another one's decisions. This kind of system promotes strategizing within or against a team.

3.2 The game

In order to make a functioning game, we need to implement a few key components for it. First of all, we will design the player object, which will be controlled by the user. We will model the mechanics of the movement and the flight and equip the player with a weapon, that has both offensive and defensive capabilities. Additionally, for the mentioned offensive abilities to be available, we will have to construct the projectile. After this object is finished, we will take a look at other elements in the scene. Firstly, we will present the building blocks shaping the map. Secondly, I will introduce the independently functioning neutral enemies, which will provide further variability in the game. As the last point, I will summarize the two supported gamemodes, which determine the goal of the game.

3.2.1 The player object

The most important GameObject in the game is the one controlled by the player. The basic movements most games provide - this being the motion along the horizontal and vertical axis - are implemented in our game as well, with a specification that jumping is only enabled when the player is standing on the ground. Besides these, the player has the ability to fly and in the event of death is respawned instantaneously (as described in the first point of our list of inspirations).

Flight

The variable that will control the mechanics of the flight is the amount of force which is added to the player to push him in the upwards direction. This force is gradually increasing as long as the flight is not interrupted. At the moment the flight is halted, this variable is reset. In the event of flight reactivation the variable is initialized to the starting value and is continuously escalated again.

The time a player can fly for is limited. During a flight the time is regularly decreased and if it runs out, the flight is halted and the player will start to fall. While not amid the process of flying, this time is steadily replenished.

Staff

Although in the first chapter we highlighted the wide variety of weapons of the game *Worms* as an inspiration, in our game the player will only have one firearm and the diversity will stem from the different neutral objects and character classes. As the main weapon for the player, he will be in control of a staff that will have two functions: one is to charge and launch fireballs - presented in the next subsection - and the second is to summon a shield, which will be able to protect the player from the incoming fireballs.

The staff will rotate with respect to the movement of the mouse (the implementation of the second point from our list of inspirations). Thanks to this functionality is the player able to aim the shots at his opponents or direct his shield to deflect a projectile headed towards him. The rotation of the staff is networked, thus every player can see where one's weapon points.

Shield

The shield works in a similar way as the flight. If the player blocks a shot with his shield, the shot is negated, the health of the player remains intact, but the shield loses power and drops its width (the more powerful the shot, the more damage to the shield). If the shield's time runs out, it shatters. Shield time is replenished gradually by not using it. In the event of shattering the shield can not be used and a fixed amount of time has to pass before the shield time will start to refresh.

Class of the character

The player will have control over a special ability, which we will call *ultimate*. The type and behaviour of this skill will be based on the class of the character, although every one of them will share a few common properties (described as the fourth point in the list of inspirations).

Every ultimate will have an active time and a cooldown time. Ultimates are continuously charged and at the moment it is ready, the player can cast a spell on his teammates or opponents based on the type of the spell. The active time will determine how long the effect of the ultimate lasts. After this time has elapsed, the spell will begin to charge itself again and will be ready after the specified cooldown time.

The 4 character classes implemented in the game are:

1. **Tactician:** the player's and his teammates' shield and flight power are replenished

2. **General:** the charging of the fireballs takes twice less time for the player (lasts for a given amount of time)

3. **Puppeteer:** freezes every player of the opposing team in position (lasts for a given amount of time)

4. **Healer:** heals himself and every teammate by setting their health to 100 - additionally, if a teammate is under the effect of the stun by a Puppeteer

and a certain amount of time has already passed, the ultimate will negate the effect of this stun as well

3.2.2 Fireball

The main projectile of the game. Every player and a few of the neutral enemies are able to create and launch fireballs in a given direction.

Players are able to charge a shot. Depending on how long the player holds the shot, its power will be that much stronger, meaning that if the fireball finds its target, the damage dealt will be that much greater. The aim can be adjusted while charging. When the shot is released, a small recoil force will kick the wizard in the direction opposite to the one of the fireball's.

The charging process is not networked on purpose. With this invisible for other players in the game, the user will have to rely on his reflexes, quick decisions and judgements in order to avoid being hit by a projectile that appears suddenly.

There is an important optimization which we can make in the NetworkTransform component of the fireball. After spawning the shot on every client and launching it in the specified direction, we can set the *sendInterval* variable of this component to zero. By doing this we will instruct Unity not to send state updates for the given GameObject, because every client can reliably calculate the next position of the fireball on its own.

The projectiles' collisions will be checked with the player object, the shield objects and the tiles constructing the map. In every case, the fireball gets destroyed upon contact.

3.2.3 The tiles shaping the map

The tiles serve as the building blocks of the game. The main purpose of these elements is to shape the map and to influence the motion of other moving GameObjects. The outlook of these tiles can be easily adjusted in the World Editor which will then be transferred to the game.

3.2.4 Neutral enemies

Besides the ultimates, the other elements contributing to the diversity of the game are the neutrals. The neutral enemies are present to make the user's objective harder to achieve, but they will behave independently of the players. The behaviours of these object are customizable and can be easily done in the World Editor (this will be discussed later).

There are 5 different main types of neutrals implemented in the game, each with a unique ability and purpose.

In order to limit the movement on the floor and to force the player to use his movement capabilities, we design a neutral which will patrol a given section on the ground.

The second type of neutral is the one dynamically changing the layout of the map. This kind of behaviour, if accounted for, can insure a wide range of strategical usability. By thinking a few steps ahead, the player can escape from a chasing enemy or cut one opponent off from reinforcements and corner him for a short amount time.

The other three types are turret based neutrals. These objects will periodically launch a projectile in the specified direction, compelling the player to improve his manoeuvring capabilities and defensive mechanism usage or face the consequences.

1. **Horizontally moving neutral:** a GameObject that will move only horizontally and will kill the player instantly if the two of them collide. In the event of it colliding with a tile from the side it will turn around and continue his movement in the other direction. If it falls down from the ledge of the map, it will obey the laws of the physics, land on the ground and continue his motion.

2. **Builder neutral:** this neutral is basically another block of the map with the addition that it will build a certain amount of tiles in the specified direction after the stated time interval. If it has built all the tiles, it will start removing them until none are left and then start the process again from the beginning. There is no difference between the built and the standard tiles.

3. **Turret neutral:** it will spawn and launch a fireball in the upwards direction relative to its rotation. The fireballs are spawned after the specified time has passed.

4. **Rotating turret neutral:** the previously mentioned turret neutral with the added behaviour of constantly rotating between two specified angles.

5. **Cluster bomb neutral:** a turret neutral that launches cluster fireballs - projectiles that will explode into three fireballs after a specified distance.

3.2.5 Gamemodes

The goal of the game is determined by the selected gamemode. The two classic types of these implemented are Deathmatch [26] and Capture the Flag [27]. The game ends if one the teams reaches the targetted score. Points are earned by killing enemy players in the case of the former or capturing the flag of the enemy team in the case of the latter.

3.2.6 Graphics

In order to display the constantly changing variables of the game for the user, the game contains a heads-up display as well. At the top of this display there is a scoreboard which will illustrate the current state of the fight between the two teams by showing the earned points. At the bottom, there is information about the player's health and the state of his ultimate, shield and flight power respectively.

In order to create an illusion of depth in the game, the application also implements a Parallax Scrolling Background. [28] We divide the elements of the background into different objects and then control their transformation according to the movement of the camera. This effect makes a two-dimensional game feel more immersive.

Every picture or image appearing in the game is created by me (with a few exceptions like the trees or clouds appearing in the background [1]). They are all vector images created in InkScape. [29] All sprites are used in PNG format.

3.2.7 Profiling

Unity's built-in profiler allows us to analyse the performance of the game. It creates a frame timeline, where every frame can be evaluated based on the performance of the CPU, GPU and so on.

By profiling the implementation of our game, we can check a couple of things. First of, the most important is that the game appears continuous and it does not lag. By examining the CPU timeline, we see that the game uses only 30% of the time appointed for the painting of the frame on average. The memory panel shows, that the number of GameObjects in the scene is moving around the same amount throughout the whole game. This means, that even though we are spawning multiple objects during the game, we destroy roughly the same amount, keeping the game performance consistent.

The profiler also has a Network panel, which monitors the multiplayer communication. This shows, that the messages sent are generally made by the need of variable synchronization. Commands and ClientRpc calls are mostly limited to times, when the player requests a spawn of a projectile or uses his ultimate, which is to be expected.

[1] Source: `http://clipartpng.com`

4. Design of the World Editor

In this chapter we will deal with the design and implementation of the game's world editor. By producing a mechanism to vary the game environment, we insure a certain mutability to the game, creating a different experience for the player for every different map he plays (this is the implementation of the third point from the list of inspirations).

Our work will result in a tool, which the user can use to create a game setting he can later use to play in. This editor will let him shape the map by creating tile objects with custom sprite images; decide on the spawn positions of the teams; place the flags for the Capture the Flag gamemode; and add an arbitrary number of neutral enemies with fine-tuned behaviour properties to the scene.

The world editor of our game is created via Unity Editor scripting. The reason behind this is that the engine - as emphasized many times - offers a lot of already implemented tools in its Editor that we can take advantage of during the creation of the game map. However, as a direct result of this decision, the world editor will only be accessible with the help of Unity.

The core functionalities are encapsulated in the form of a custom inspector item. We also create a custom window in order to choose the tile to paint with and utilize a class called *Gizmos* to provide visual help for us during the element placing.

4.1 Implementing the World Editor

Before defining a custom editor for a GameObject, we will create a new type of it and will build our editor on top of this. To make this user friendly, we will add a new item to the GameObject menu. Unity allows us to do this by marking a function with the attribute *[MenuItem]*. In this function we will create an empty object with a custom component which we can use to create and customize the Inspector. As we will reference this component throughout the chapter, let's name it TileMap.

4.1.1 Creating a visual aid for object placement

Before starting to customize the editor, we will draw a grid in the scene to help us position the tiles, neutrals and properties. This is done using the *Gizmos* class. [1]

Every behavioural script in Unity defines a method for drawing a gizmo when selected. By utilizing this, we can draw the grid for our world editor. This grid will not be present anywhere else, it is just a visual assistance for the user to help him better differentiate the elements in the scene. It is important to remember, that the gizmo will only be visible when the component is selected in the Hierarchy.

[1] Gizmos: (`https://docs.unity3d.com/ScriptReference/Gizmos.html`)

4.1.2 Creating the custom inspector

By extending the base Editor [2] class, we are given all the essential functions to use when creating the Inspector. Amongst these functions there are two, that will let us execute commands when the object with the TileMap component is enabled or disabled. Another function will allow us to carry out methods every time the GUI of the Inspector is redrawn and one that will do the same with the GUI of the scene.

Inspector GUI

The GUI elements we create here will be shown in the Inspector of the TileMap component. We will use this to define the settings of the map and add the interactivity for the user to place the properties and neutral enemies on the map.

First and foremost we will need to decide on the size of the map. For this we create a Vector field, where the user can input the width and height of the map, expressed in the number of tiles in a row and in a column. Next we give the user the option to choose the folder, where he collected the artwork for the tiles. Because of the properties of a tile grid, we require that these images had the same size and that there is at least one picture in the specified folder. After this we draw a button, which will carry the functionality of clearing all the tiles in the map. Although this can be done in the hierarchy manually as well, we aim to be as user-friendly as possible and not to force beginner Unity users to use other tools if they do not want to.

Next in line is the property placing panel. Note, that the placement of these properties is essential for the game to function and we can not let the user export the map without setting these positions. The four properties are the spawn positions of the two teams and the spawn positions of the two flags for the Capture The Flag gamemode.

After this comes the neutral enemy placing panel. Here the user can select the type of neutral he wants to place, rotate it if he wishes, set its properties and place it on the map. Although we want to give the user as much freedom as possible, we accept only a range of values he can set. This results in a more constricted world editor, but guarantees the proper and sensible functioning of the neutral.

The last item in the Inspector is the panel that will let us serialize and export the map created in the world editor to the disk to be later used in the game. First, we will force the user to direct us to the game folder where the data of the game are saved. This way we will save the generated map directly into the Maps folder, which will give us the option to use it instantly in our game. It will limit the user interactions with the generated binary file as well. We will use this information to copy the relevant image files to the game folder as well.

If a problem occurs during the creation of the map, we will utilize Unity's bug reporting functions. In the event of an error in the Inspector, we notify the user via Warning and Error messages immediately in the Inspector [3]. If the error

[2] Editor: (`https://docs.unity3d.com/ScriptReference/Editor.html`)

[3] using the functions of EditorGUILayout.HelpBox: (`https://docs.unity3d.com/ScriptReference/EditorGUILayout.HelpBox.html`)

emerges during the serialization process, the user will be notified via the Unity Console window.

Scene GUI

The statements in this method will carry out the actual placement logic. Here we can check the currently occurring events and react to them accordingly. With the current mouse position available to us, we can move the GameObject we are currently placing along with the mouse. By reacting to the presses of the placing buttons, we can actually place the current object in the map.

4.2 TilePickerWindow

Although we almost have all the components necessary to create a map for our game, we still lack the place where we can select the tile we want to place on the scene. For this we will create a custom window.

This window's purpose is to display every tile image in the folder which the user specified in the Inspector. Note, that we have to refresh this window constantly (in the case of the user putting a new image in the folder we have to add that to our list in Unity), but we do not want to import every image all the time because of its ineffectivity. In order to solve this problem we use a HashSet [4] collection, where the *Add()* and *Contains()* functions are $\mathcal{O}(1)$ operations. We draw the sprites of the tiles in the window as textures and generate a highlighter box which will always be drawn around the currently selected tile.

4.3 Putting it all together

The placement of the tiles is now possible, but we still need a way to delete them one by one if necessary. For this, we will construct a two-dimensional array in the size of the grid and keep a reference for the tiles placed in the scene. This way, if the removal button is pressed while hovering over a tile, we can calculate the coordinates of the cell by the position of the mouse, look up the tile in our data structure and destroy it. The other method to do this would be to look up the GameObject by name, but the only way to do this is to iterate through every object in the scene. Because of this, our method is more effective.

Another advantage of this internal structure is that if a tile is placed over another one in the scene, instead of destroying the older, we just replace its sprite property with the sprite of the new tile. Additionally, if the user changes the size of the map after he has placed some tiles already, the structure is programmed to look up any blocks that would end up outside of the boundaries of the new map and is able to destroy them properly (while keeping the still relevant ones stored).

[4] HashSet collection: (`https://msdn.microsoft.com/en-us/library/bb359438(v=vs.110).aspx`)

Because of the advantages of this structure, we use something similar for our property and neutral enemy placer function as well. Consequently, we can look up any cell and alert the user if the placement he is trying to realize is invalid (for example if he tries to place a neutral enemy in a cell where a tile is already present).

5. Map serialization and deserialization

With the world editor working in a different setting than our game does, we need to find a way to export something we created in it. This problem is called serialization, where we need a protocol according to what we convert the elements in the map to a representation which can be condensed into (preferably) a single file. This file can be decoded later on when it is needed in the game.

There are many serialization protocols, every one of them suited better for something than the other. Our job is to find the one best suited for us. For this purpose I will present three methods of serializing, two of which are offered to us by the .NET framework. These two are the XMLSerializer [30] and BinaryFormatter. [31] The third method is implementing a custom serialization protocol tailored for our needs.

5.1 Determining the goals

As the first step, we will look through the objects, components and variables which are necessary to be serialized in order to reconstruct the map in the game.

As a start, we will process the positions of the properties placed. This position field is a third-dimensional vector variable, which is easily serializable by hand and every already existing protocol.

Among the tile objects the things that differ from each other are their position and sprite image. The tile is not rotated or scaled during runtime and its components are not switched out or modified either. It does not have a behaviour script with variables and the only extra component besides the SpriteRenderer is the Collider component, which will always be the same size as the body of the object is. Because we determined that the scale of the tile is not changed at any point, we conclude that the rectangle transform of the Collider will not scale either.

That leaves us with just the position and the Sprite [1]. We process the position the same way as we did at the properties. The image (or texture) field is quite different. The most effective way to compress and transfer a texture through network is still heavily researched and will be discussed more in-depth in the next chapter.

Other objects in the map that are more problematic to process are neutral enemies. Besides the position and the rotation component, they each have a unique behaviour with unique controlling variables, all of which have to be saved in order to recreate the one specific instance of the neutral placed.

5.2 Serialization protocols

As the next step we will compare the two prepared protocols according to the characteristics which matter to us the most.

[1]Sprite: (`https://docs.unity3d.com/ScriptReference/Sprite.html`)

The *XMLSerializer* produces a human readable file, with the structure clearly visible in itself. It remembers the schema of the processed classes and saves the public fields and their values, but not the private private members of the objects. This does not matter in our case, because all the targeted variables are public fields. Moreover, it is easily controllable - by using attributes - how each member is serialized.

The BinaryFormatter produces a binary file, which is not readable by the user intuitively. It serializes the private object members as well, but can not control the way the members are processed. It is also .NET specific and Mono had issues with BinaryFormatter in the past, which is problematic for us because of the portability of our game.

Comparing the serialization protocols is a common topic amongst programmers. One of the more recent tests was performed in April 2017. [32]

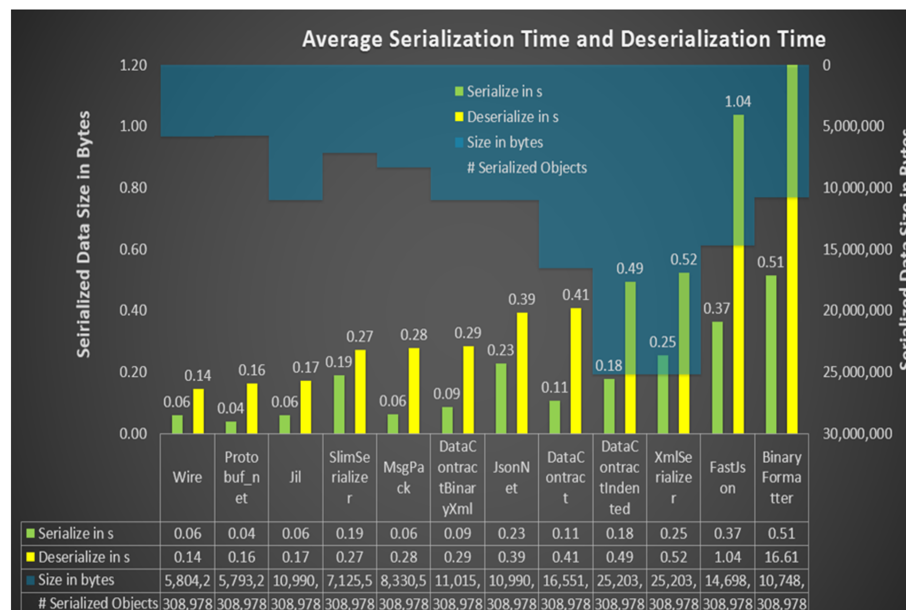


Figure 5.1: The comparison of serializers. [2]

The information we can extract from figure 5.1 is that the XmlSerializer performs twice as better during the serialization and approximately 33 times better at deserialization. In the size department the BinaryFormatter produces a finer result with the generated file size having half the size of the one that XML produced.

The important thing to note here is that this test measures only the time purely spent by serializing and it did not factor in the duration of the initialization before the first item is serialized. For the purpose of evaluating this, the test is conducted with the serialization of just one object. As it can be seen from figure 5.2, the BinaryFormatter produces a much better result in this department than the XmlSerializer.

[2]Source: `https://aloiskraus.files.wordpress.com/2017/04/image11.png`

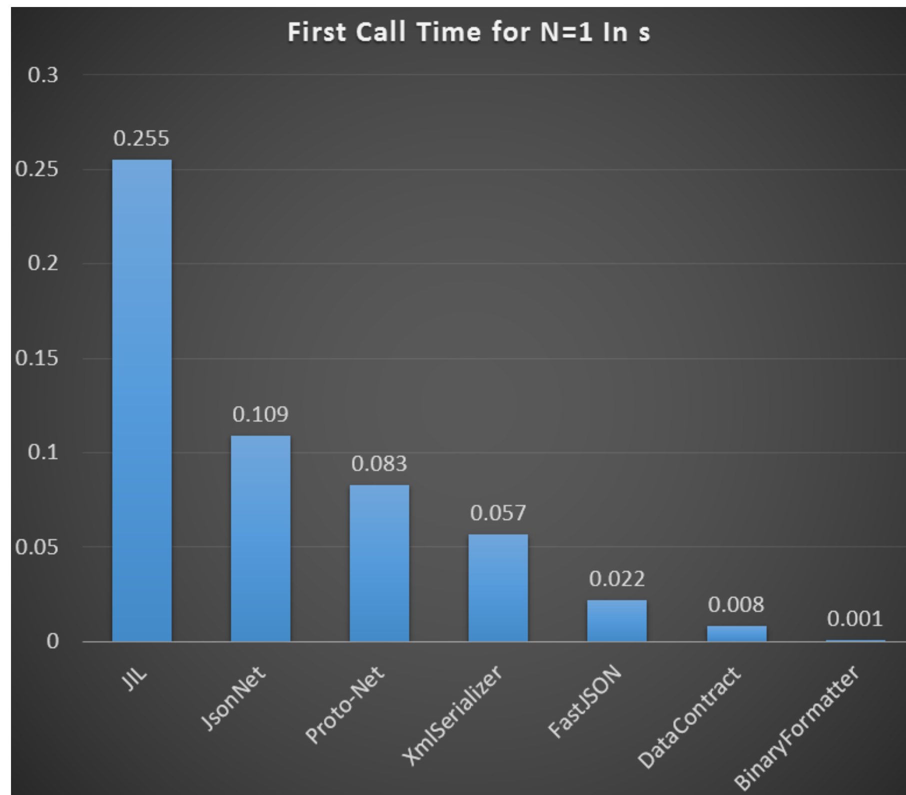


Figure 5.2: The comparison of serializers with serializing only one object. [3]

As the third method for saving the map file to the disk we can discuss writing a custom protocol for serialization. Although it requires more work on the side of the developer, it will result in better size complexity, because we will only process the fields we absolutely require in order to reinitialize the map. Additionally, custom serialization achieves the best performance as well, in part of the aforementioned reasons and in part because we can make certain steps in the direction of enhancing the effectiveness.

We will compare the manual way with the BinaryFormatter, because ultimately it would be better for us to make our file non-readable, as it will provide some basic security. The format won't deter a committed person who wants to modify something in the file, but it will discourage the general user. For this comparison we will use a test provided by CodeProject. [33]

As the tests described on the site are outdated, we will run them on our computer. The specifics of the computer are the following:

- Intel(R) Core(TM) i5-4200H CPU @ 2.8GHz
- DDR3, 8GB RAM
- OS: Windows 8.1

From the results of the test we conclude that the performance ratio stayed approximately the same as it was at the time when the article was written and the manual method still heavily outperforms the .NET supported one.

[3]Source: `https://aloiskraus.files.wordpress.com/2017/04/image4.png`

As a conclusion we determine that the most suitable method for us to use is to implement a custom serialization protocol. Additionally, we conduct a test with randomly generated tiles in order to showcase the performance during the serialization of a big map. For the first test we generate 100.000 (one hundred thousand, figure 5.3) and for the second 1.000.000 (one million, figure 5.4) tiles.

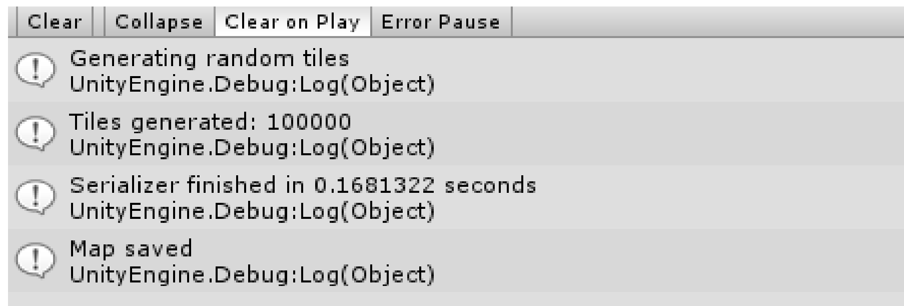


Figure 5.3: Serialization of 100.000 tiles.

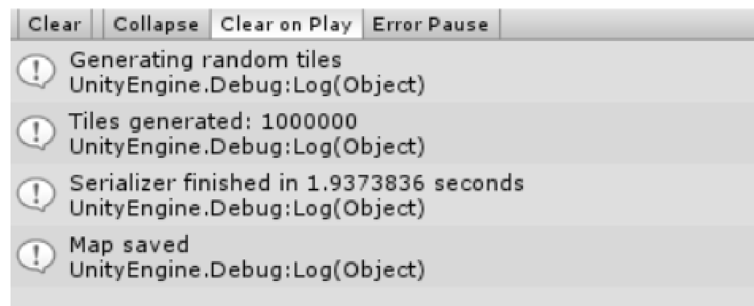


Figure 5.4: Serialization of 1.000.000 tiles.

Further optimization in the future can be realized by using buffers to avoid the continuous access to the disk. Using the hard drive for a limited number of times does not affect the effectiveness in a great way, but in the case of serializing a big number of tiles and neutrals, buffers can improve the performance visibly.

6. Dynamic image synchronization

Now that we have the maps ready and we created a way to use those designed in the World Editor, we need to find a method for the artwork to be dynamically shareable as well. Our goal is to allow the map designer the usage of any images he chooses and then make those textures portable and display them correctly on the side of the client as well. All this without the need to release a new version of the game or update it in some form.

The company that is known to be doing this is Valve [34]. Dynamic texture synchronization is done in both their currently most played games [35] Counter Strike Global Offensive [36] and Dota2 [37]. In the last game of the popular Counter Strike franchise this is manifested in the customization of the weapons the player uses. While requiring a game update for the firearms to be changed, the "skins" of the weapons are interchangable. The same can be said for Dota2, but instead of guns, the customizable units are heroes (figure 6.1).

Figure 6.1: Dota 2 hero customization. [1]

By creating this system, Valve could afford to release the games for a lower price or even free of charge and gather the profit from the customization, if the player wanted to make the game more visually pleasing. With this option still preferential and not required for the player to purchase any items for the game to work properly, many users personalize their items making Valve's step pay off.

Although this was probably a business driven decision for the company to make, the system is functional and we aim to implement something similar by allowing the designer to use custom images besides the ones we provide and instruct the application take care of the rest for the players. For this, we will

[1]Source: `http://i.imgur.com/GtBiMN3.jpg`

need a way to organize the requests received from the clients, prepare the image for the transfer and realize the transfer in the Unity framework.

6.1 Requesting and supplying the sprites

The first step is to define the time when the synchronization would take place. It is important to only share the textures which are used in the chosen map in order to minimize the network traffic and computations. For this we will inevitably have to execute this process after the deserialization of the map. Keeping in mind that we also need to maximize the experience for the player, we have to do this as soon as possible. The timeframe that fits these criteria is the one instantly after the deserialization. In order for the actual painting to be carried out properly, we also need to make sure that the tile is already present on the side of the client. During the downtime while waiting for the tile to become available, we can perform many operations, so it pays off to start the synchronizing process immediately and delay the painting if necessary.

We can extract the information about the used images in the map from the deserialization process. After spawning these tiles on the server, we will iterate through every client connected to the game and signal them, that the host is ready to receive the requests for the transfer. Note, that this done by targetting a client rather than sending a command to all of them, because it is possible that one client has played on this map before and therefore is in possession of the necessary images (figure 6.2), while another have not and would need to receive them in this match (figure 6.3).

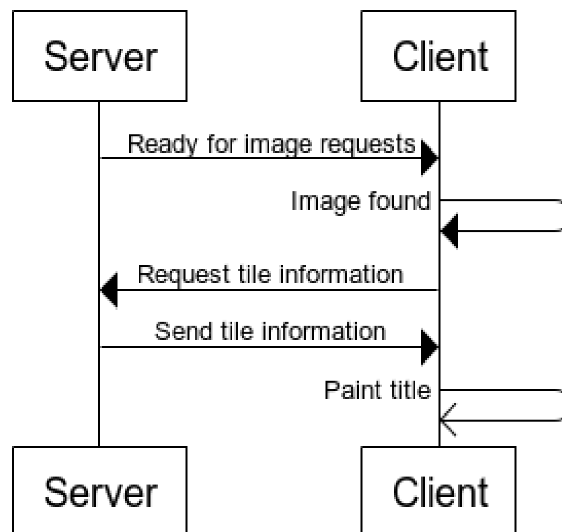


Figure 6.2: The process of sprite synchronization if the client is in possession of it.

The client, upon obtaining the name of the sprite in question, checks the image folder on his side to find out whether he is in the possession of the image or not. If he is, he loads the texture from the disk and sends a query to the server

for the identifications of the tiles he should apply this sprite for. If he is not, he sends a command requesting the specified sprite to be sent from the server.

The server receives the sprite request from the client and prepares the image for the transfer. He sets the control variables necessary for the transmission and starts an asynchronous coroutine with the task to send the texture to the specified client. It is important that the server manages the coroutines in a way that the client will not receive two different sprites concurrently. Upon receiving the sprite, the client saves it to the disk for future use.

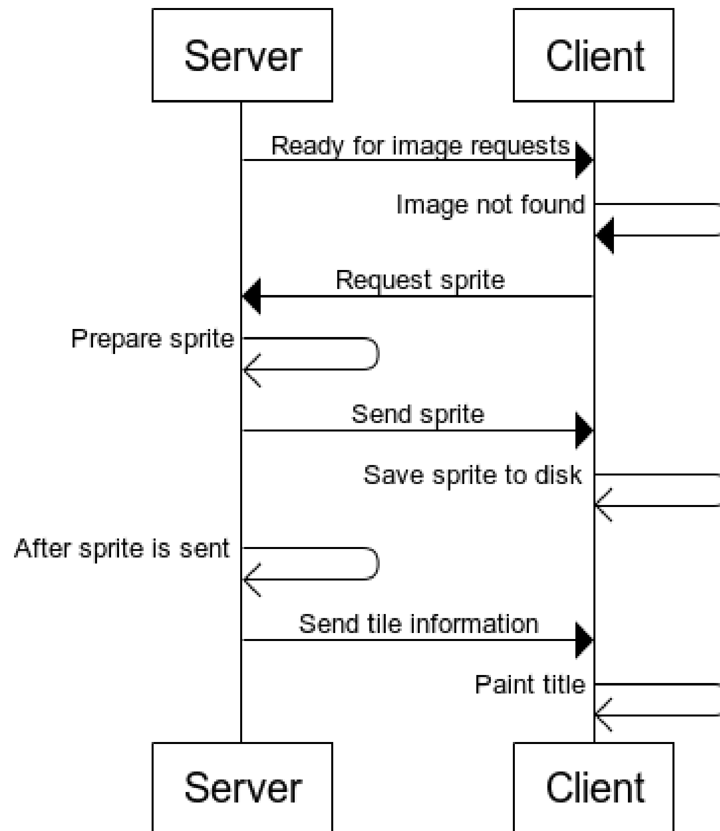


Figure 6.3: The process of sprite synchronization if the client is not in possession of it.

After recognizing that the texture has been sent, the server sends out the network identifications of the tiles which have to be rendered with the specified texture. After accepting the properties of the tile, the client will choose between two actions: if the tile is already present in his scene, he will immediately repaint it with the previously loaded sprite; or if the tile has not yet been spawned on the side of the client, he will save the tile characteristics and start an asynchronous method that will paint the tile when it becomes available.

6.1.1 Image comparison

Note, that we make the assumption that a sprite used in the map is the same as another if they have the same name. This is impractical for the designer and

could result in unintentional behaviour if there is a name collision. However, to verify the equality of two textures through network is not an easy process.

Comparison by pixel colors

One method that could provide an answer for this is to determine the color of a few specified points in the texture and compare only those between the two sprites. The positions of these points could be randomized, but then the instruction sent over network would have to contain these positions as well. Because of this, the colors should be gathered from fixed points, which are specified in the same way at the host and the client side as well (or calculated by the same algorithm).

The problem begins with the definition of the number of these points. We need to choose a sufficiently high amount in order to safely determine the equality of two textures, but a sensibly low amount, in order to not result in elevated and unnecessary network traffic usage. The process of finding a satisfactory balance is not an easy task, but we will give it a try.

Unity defines the Maximum Transmission Unit as 1500 bytes. This is basically the maximal size a packet can take up in a Command or Rpc call counting the internal parameters such as the IP address. This leaves about 1400 bytes for us to use [2], because we are using the Reliable Sequenced [3] channel for communication. We could fill this space with the colors of the chosen points, send it in one message and execute the comparison algorithm. Although this could maybe be an optimal solution if we were trying to transfer bigger images, the default tile textures have a 64*64 size. With the correct compression this file is fit into 3 Rpc calls in average. Consequently, an image comparison which is not guaranteed to work would case a 33% overhead for one sprite, which can add up to be a big number considering the number of textures used in the map and the number of clients connected to the game.

Comparison by hashes

The technique we will be taking a look at is called Perceptual Hashing. [38] The main idea of these hashing algorithms, is to calculate the hash (in this environment called "fingerprint") by identifying distinguishable features in the image and using these features in the computation (instead of the image itself).

Moreover, these functions are applicable not just for images, but for video and audio as well. Provided as an example of real life usage on the pHash website [39], YouTube uses this technique to battle possible copyright violations. Another application is the Google Image Search. [40]

The two algorithms we will mention are the aforementioned pHash and another approach called Average Hash. [41] pHash is a more robust tool, meaning that it is flexible enough to recognize image manipulations like transformations, rotations and so on. On the other side, Average Hash is not this adaptable, although it can be used to find the connection between images, that are almost exactly the same and underwent no modifications.

[2] Unity's maximum message size: (`https://docs.unity3d.com/ScriptReference/Networking.NetworkMessage.MaxMessageSize.html`)

[3] Reliable Sequenced channel: (`https://docs.unity3d.com/ScriptReference/Networking.QosType.html`)

The property, that pHash can be used for a bigger range of purposes, backfires at the examination of performance. We find, that by the characteristics of these algorithms, Average Hash executes the calculations faster than pHash. Although not as adjustable, we find Average Hash to be enough for our purposes of finding out whether a sprite in question matches one in our folder.

As the last step, the similarity of the images are checked using the hamming distance between the two hashes. Based on our choice of encoding (described in the next chapter), we know that the compression of the images will be lossless, resulting in the fact, that two sprites will be equal if the hamming distance is equal to zero.

6.2 Preparing the image for the transfer

I will present three methods for converting the tile sprite into a byte array that we will be able to break into parts and send it to clients. Finding the optimal solution is heavily dependent on the source image used, so we will take our default sprite images as an example.

6.2.1 Encoding to JPG

JPEG loses data at compression. Thanks to this, we can define the targeted size of the compressed image, but once decompressed, the picture will not have all its pixels correctly initialized, which may result in artifacts. An advantage of this method is that it is natively supported in Unity for an instance of the Texture2D class, which helps with the portability of the application to other platforms. This method of compressing is suitable for high definition pictures and photos, as these can withstand high compression. It is not suitable for drawn lines or for pictures containing sharp edges inside the texture, which makes it unusable for us.

6.2.2 Encoding to PNG

PNG is a lossless format and will remain the same after compressing and decompressing. Because of this, we can not control the size of the compact version of the image and the algorithm will decide what it can or can not simplify. PNG is a good fit for our default images, which are all digital images created by vector graphics and it is also able to handle the alpha channel. This type of compression is supported in Unity for the Texture2D class as well.

6.2.3 Zipping

The third method of compression would be to zip the raw texture data of the image using gzip [42]. It is a lossless compression, but the algorithm works in a similar way as the PNG enconding does. Because of this and the fact that the mentioned encoding is supplied by Unity, we will be using that method for the compression of our tile images.

Conclusion

We started this thesis by defining the characteristics of the game we wanted to implement. Subsequently, I presented game engines that would make the implementation possible and listed their advantages and disadvantages from our standpoint based on the goals of our game. By examining these points we selected Unity3D as our engine, which we briefly presented. Afterwards we designed and implemented the application and created a world editor to accompany this game as well. In the last chapters, we analysed the serialization and deserialization process of the world editor constructed map, presented two popular methods for this operation and wrote a custom one. As the last problem, we investigated the synchronization of the sprite images and found an optimal way to compare and compress these pictures.

A function we did not manage to add is the evaluation of the player, which would enable us to create a ranked matchmaking system. With the current methods implemented we make no regulations about who is obliged to play against whom.

The result of this thesis is a two-dimensional, multiplayer run and gun game, playable on the three major desktop platforms. Accompanying this is a world editor operating in the Unity3D setting, which the user can use to create custom maps for the mentioned application.

As the last point, we conclude, that the game engine we selected was suitable for us to develop the targeted application and would recommend it to other programmers.

Bibliography

[1] Asteroids (video game). https://en.wikipedia.org/wiki/Asteroids_(video_game).

[2] Craig Chapple. The top 16 game engines for 2014. 2014. [Accessed on: 2017-07-02] Accessible by: http://www.develop-online.net/tools-and-tech/the-top-16-game-engines-for-2014/0192302.

[3] Soldat current playerbase. https://www.soldat.pl/en/. The current number of players is shown on the homepage below the Download button. Accessed.

[4] Jim Rossignol. Eurogamer's summer of pc plenty - twenty freeware games. *Eurogamer's Summer of PC Plenty*, 2006. [Accessed on: 2017-07-17] Accessible by: http://www.eurogamer.net/articles/a_20bestfreegames. Soldat ranked 11th.

[5] Worms world party. https://www.team17.com/games/worms-world-party/.

[6] Best selling game franchises. [Accessed on: 2017-06-29] Accessible by: http://vgsales.wikia.com/wiki/Best_selling_game_franchises.

[7] Worms 2D weapons. http://worms2d.info/Weapons. Accessed: 2017-07-21.

[8] Overwatch. https://playoverwatch.com/en-gb/.

[9] Unreal engine 4. https://www.unrealengine.com/what-is-unreal-engine-4.

[10] Cryengine. https://www.cryengine.com.

[11] Matan Aspis. 6 top game engines in 2017. 2017. [Accessed on: 2017-07-01] Accessible by: http://www.discoversdk.com/blog/6-top-game-engines-in-2017.

[12] Unity licenses. https://store.unity.com.

[13] Waveengine. https://waveengine.net.

[14] Waveengine release date from gamefromscratch. http://www.gamefromscratch.com/post/2015/10/05/A-Closer-Look-At-Wave-Engine.aspx.

[15] Waveengine 2.0 release. https://geeks.ms/waveengineteam/2015/09/15/whats-new-in-2-0/.

[16] Choosing duality. https://github.com/AdamsLair/duality/wiki/Choosing-Duality.

[17] Homepage of xenko. https://xenko.com.

[18] Xenko release date. http://www.pocketgamer.biz/asia/news/65594/silicon-studio-launches-xenko/.

[19] Xenko licenses. https://store.xenko.com/get-xenko.

[20] Monogame homepage. http://www.monogame.net.

[21] Nez GitHub. https://github.com/prime31/Nez.

[22] Unity documentation. https://docs.unity3d.com/Manual/index.html.

[23] Unity tutorials. https://unity3d.com/learn/tutorials.

[24] Raywenderlich Unity tutorials. https://www.raywenderlich.com/category/unity.

[25] Unity Multiplayer Service. https://docs.unity3d.com/Manual/UnityMultiplayerSettingUp.html.

[26] Deathmatch. https://en.wikipedia.org/wiki/Deathmatch.

[27] Capture The Flag. https://en.wikipedia.org/wiki/Capture_the_flag#Software_and_games.

[28] Parallax scrolling overview by GameDevelopment. https://gamedevelopment.tutsplus.com/tutorials/parallax-scrolling-a-simple-effective-way-to-add-depth-to-a-2d-game--cms- [Accessed on: 2017-07-09].

[29] Inkscape. https://inkscape.org/en/.

[30] Xmlserializer. https://msdn.microsoft.com/en-us/library/system.xml.serialization.xmlserializer(v=vs.110).aspx.

[31] Binaryformatter. https://msdn.microsoft.com/en-us/library/system.runtime.serialization.formatters.binary.binaryformatter(v=vs.110).aspx.

[32] Alois Kraus. The definitive serialization performance guide. 2017. [Accessed on: 2017-07-06] Accessible by: https://aloiskraus.wordpress.com/2017/04/23/the-definitive-serialization-performance-guide/.

[33] Comparing BinaryFormatter and manual serialization. https://www.codeproject.com/Articles/311944/BinaryFormatter-or-Manual-serializing. [Accessed on: 2017-07-09].

[34] Valve software. http://www.valvesoftware.com.

[35] Statistics provided by Valve. http://store.steampowered.com/stats/.

[36] Homepage for the game Counter Strike: Global Offensive. http://blog.counter-strike.net.

[37] Homepage for the game Dota 2. http://blog.dota2.com.

[38] Joe Bertolami. Perceptual hashing. 2014. [Accessed on: 2017-07-15] Accessible by: http://bertolami.com/index.php?engine=blog&content=posts&detail=perceptual-hashing.

[39] phash. http://phash.org.

[40] Google image search. https://www.google.com/intl/es419/insidesearch/features/images/searchbyimage.html.

[41] Chris Pickett. Simple image hashing with python. 2013. [Accessed on: 2017-07-19] Accessible by: https://www.safaribooksonline.com/blog/2013/11/26/image-hashing-with-python/.

[42] gzip. http://www.gzip.org.

List of Figures

1.1 Screenshot of the game Soldat. 5
1.2 Screenshot of the game Worms World Party. 6

2.1 Game engine usage in the UK (2014). 8
2.2 NetworkIdentity component. 13

3.1 Our customized LobbyManager. 15

5.1 The comparison of serializers. 27
5.2 The comparison of serializers with serializing only one object. . . 28
5.3 Serialization of 100.000 tiles. 29
5.4 Serialization of 1.000.000 tiles. 29

6.1 Dota 2 hero customization. 30
6.2 The process of sprite synchronization if the client is in possession of it. 31
6.3 The process of sprite synchronization if the client is not in possession of it. 32

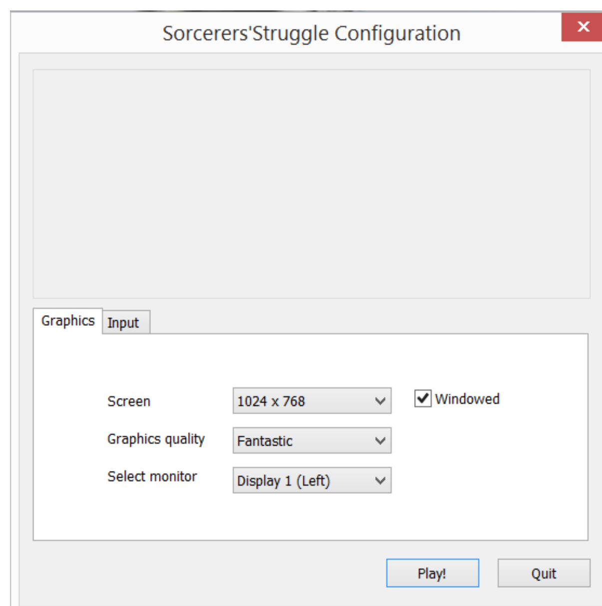
User documentation

The game

In this section, I will present the installation process and familiarize the user with the controls and the course of the game.

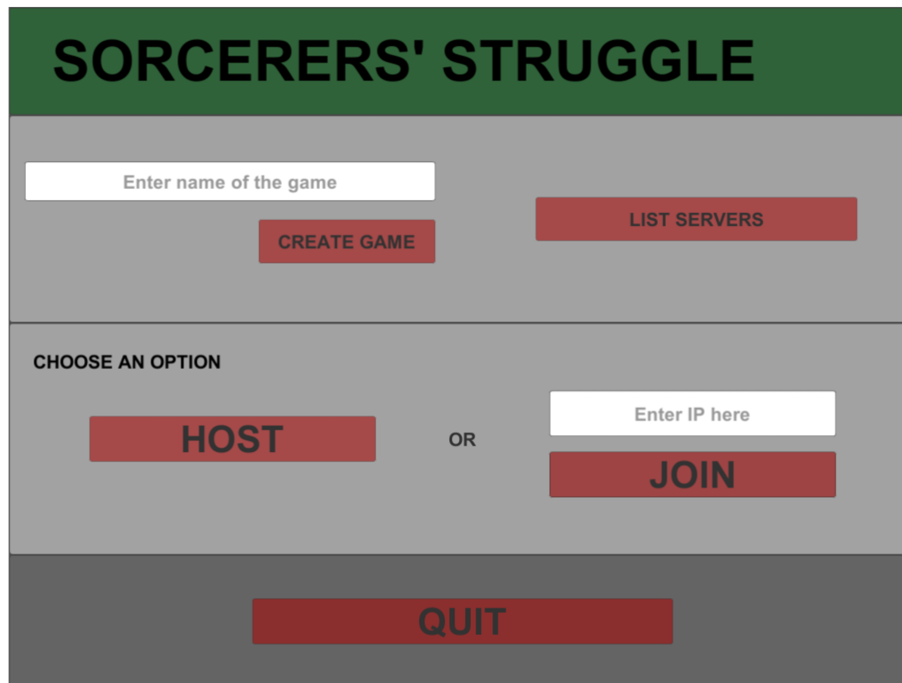
Installation

The game is portable and needs no installation. After the user launches the game (by executing the binary file *SorcerersStruggle*), he is welcomed with the settings screen. In this panel, the player can adjust the resolution of the screen, the quality of the graphics and the targeted monitor as well. Another option, to display the game in windowed context is also present.



Establishing the connection

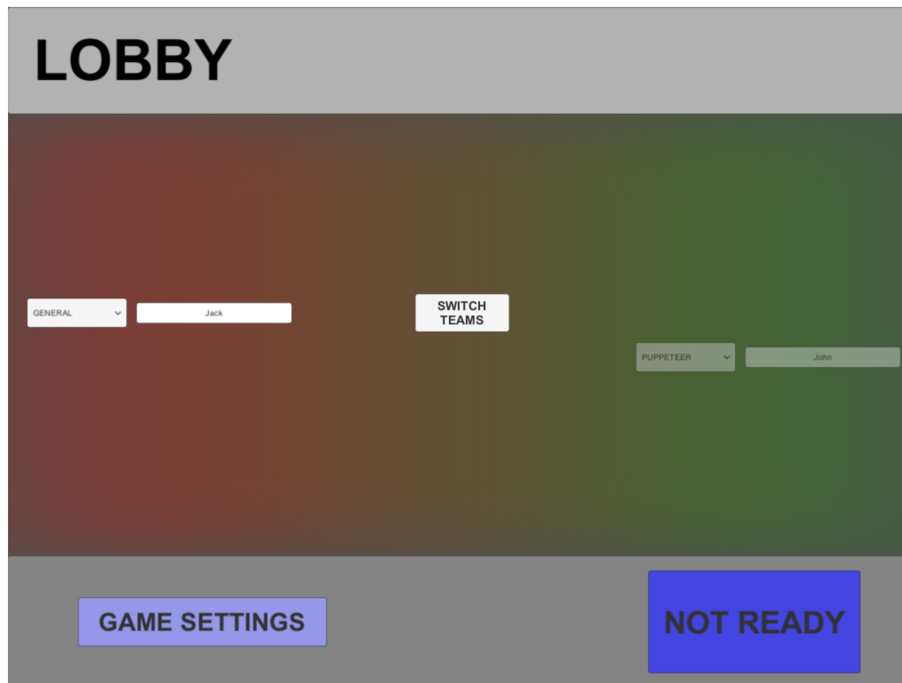
The game provides two options to connect the players: matchmaking and direct connection. If the players agree on using the matchmaking option, the host player inserts the name of the game in the upper panel and clicks the button *Create game*. After the game is created, the other players can access this server by clicking the *List servers* button, where the created match will appear which they can join by clicking the *Join* button.



If the players agree on using the manual connection option, the player taking on the role of the host clicks the *Host* button in the second panel. After this, he shares his public IP address via a third-party messaging service with the other players. Upon receiving this, the other players insert this address into the corresponding field on the second panel and click the *Join* button. Please note, that every player has to adjust the settings of his system's firewall in order to allow the game the creation of the connection.

Lobby room

After joining the lobby, the player can choose the class, name and team of his character. The host player can also set the game settings with the help of the *Game settings* button in the lower left corner. These settings include the gamemode and game map. After a player has finalized his decision he clicks the *Ready* button in the lower right corner. After every player is marked as ready, a *Start game* button appears on the screen of the host and the game can be started.



Game scene

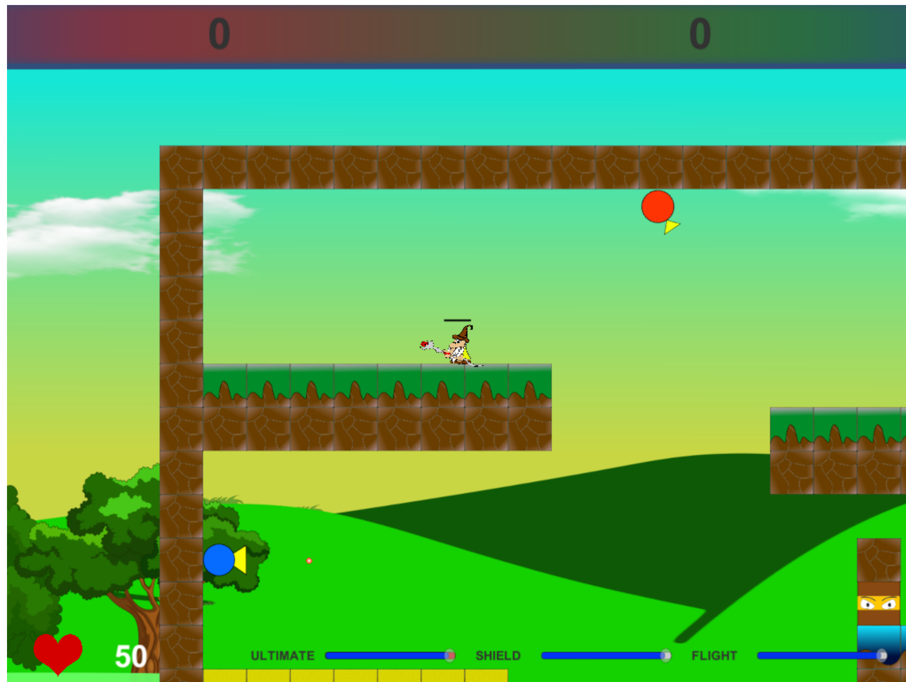
After the game is started, the user is given control over a wizard object. The controls are the following:

Button	Action
W	Jump
A	Move left
D	Move right
Ctrl	Use shield
E	Use ultimate
Mouse 0	Fire
Mouse 1	Flight
Movement of the mouse	Aim

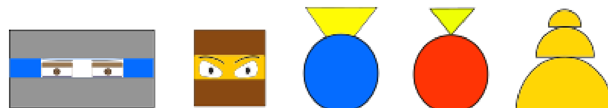
The heads-up display contains various information for the player. At the top of the screen a scoreboard keeps count of the points scored by the two teams. At the bottom, there are four informative panels (from left to right):

- The health panel, which displays the current amount of health points the player has. If it reaches zero, the player dies and is respawned instantly.
- The spell (referenced as ultimate) panel, which displays the current state of the spell determined by the class of the character. If the slider is full, the ultimate is ready to be activated.
- The shield panel, which shows the time still available for the shield to be active. If the shield is shattered, the slider turns red and during this time the defensive mechanism is not available.

- The flight panel, which shows the amount of time the player can fly for.



The five types of neutral enemies described in the thesis are (in respective order): horizontally moving neutral, builder neutral, turret neutral, rotating turret neutral and cluster neutral.



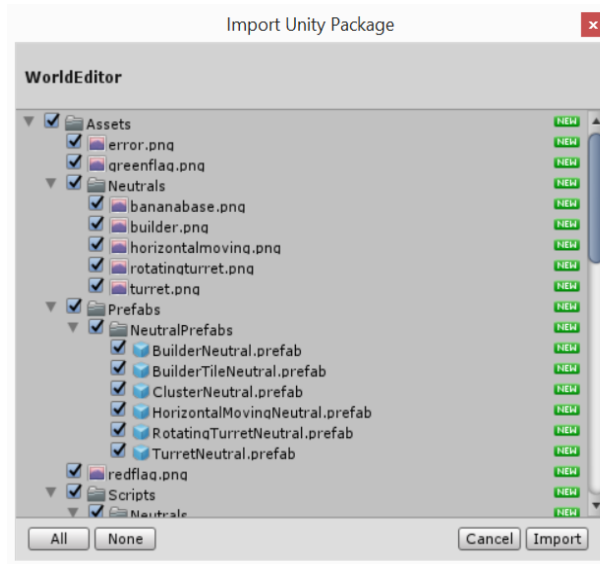
Enemy players are marked with a red bubble. The goal of the game is determined by the gamemode. In deathmatch, the game ends if one of the teams reaches the score of 30. In capture the flag, victory is achieved by capturing the enemy's flag 5 times. After the game is over, the result will appear on the screen and the player can quit the game.

World Editor

Installation

Note that this application requires Unity3D(version 5.6.2f1) to run.

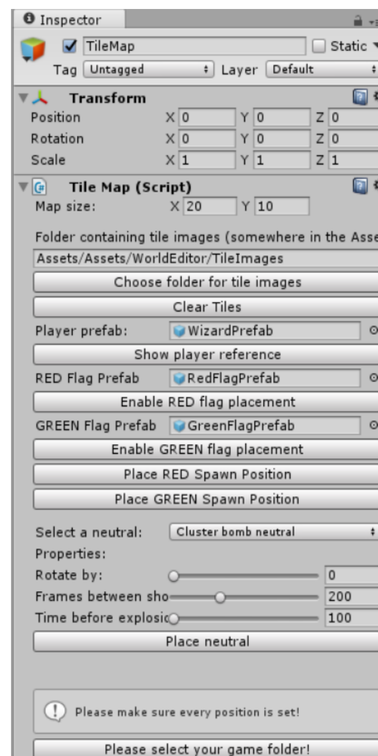
The world editor is supplied as a Unity Package. By double-clicking on the *WorldEditor.unitypackage* file, the Unity3D editor will open and a window will offer to import the files necessary.



The world editor object can be added to the scene by clicking on the *GameObject/WorldEditor* item in the menu. Besides this, in order to select the current tile image, a window will be necessary. This window is defined as the TilePicker window and can be accessed from *Window/TilePicker*.

Building the map

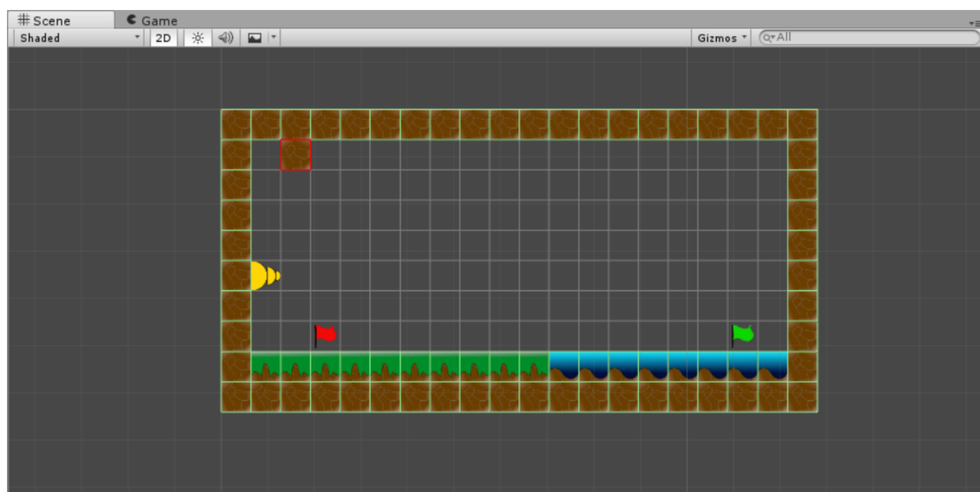
The world editor becomes enabled by clicking on the added TileMap object. The grid of the map is drawn and the custom Inspector gets displayed on the right side.



The settings of the map are shown in the mentioned Inspector. These properties are (from top to bottom):

- The width and height of the map expressed in the number of possible tiles in the given dimension.
- The button that enables to change the folder containing the tile images. The pictures in this folder are displayed in the TilePicker window.
- The Clear Tiles button, which will remove every tile from the scene.
- The button which attaches the wizard object to the mouse. This can be used as a reference to estimate the size of the map.
- The next four buttons place the necessary properties on the map. These properties are the spawn positions of the two teams and the positions of the two flags for the capture the flag gamemode. Note, that the placement of these objects is required in order to export the map.
- The neutral placing panel. Here, the type of the neutral can be selected, which the user wishes to place. This object can then be customized by fine-tuning the controlling values of the neutral's behaviour.
- The export option of the map. This panel will first require the selection of the game data folder (the folder provided together with the game application called *SorcerersStruggle_Data*). Then, the user can input the name of the map and export it. After the Unity Console reports that the map is saved, it is immediately available to be played on and will be given as the option for the player to choose in the game.

When the user is currently not placing a property or a neutral, the tile placing mode is activated. The image of the tile can be selected in the TilePicker window and it can be added into the map by holding down the *Left Shift* button. A tile can removed by hovering over the one the user wishes to delete and pressing the *Left Alt* button. If a tile is put over another object, the object will be removed and the tile will take its place.



The planting of a property or a neutral is realized by pressing the *Left Ctrl* button. A property or a neutral cannot be placed over a tile, but if it gets established on another object of the similar type, the previous object is removed. The removal process is the same as for the tiles (by hovering over and pressing the *Left Alt* button).