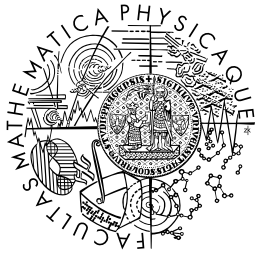

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Gergely Tóth

Sorcerer's Struggle

Department of Software and Computer Science Education

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

Prague 2019

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, Ph.D., 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 discussions about the gaming platforms and the most important components of the application and of a suitable world editor. Furthermore, the problems of map smoothing and dynamic image synchronization are explored and the last part addresses the design and functionalities of a matchmaking server. The result of the thesis is a two-dimensional game, with a complementing world editor and server.

Keywords: video game, platformer, multiplayer, multi-platform, level editor

I would like to thank Mgr. Jakub Gemrot, Ph.D. 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

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 will 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 kind of application, our first step after choosing the target platforms will be to analyse two games, which have gained recognition in their respective fields. After this, we will establish the game design, perform the analysis and talk about the implementation of the game and the world editor.

This latter tool makes the creation of maps possible for the user. For this editor, we will design and implement tools, which will allow the user to create tilemaps and polygon maps as well. Moreover, we will also permit the end user to choose the spawn positions of the teams and add neutral enemies to the scene.

In the last chapters, we will first present the problem of dynamic image synchronization 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. In the last chapter, we also design a server capable of hosting lobbies for unranked matches and creating balanced games according to the players' skill.

Goals of the thesis

- select the gaming platforms for our application (run and gun game)
- establish the game design and game flow
- select the best suited libraries for the implementation
- implement tools for creating tile-based and polygon maps
- solve the problem of custom map synchronization
- create a matchmaking server with custom lobby options

1. Platform selection

In this chapter, we elaborate on the thesis assignment and then select the video game platforms, which we will develop our game for.

1.1 Assignment breakdown

The goal of this thesis is to develop a multiplayer multi-platform run and gun game. If we break this definition down, we can specify the game we aim to create (here we describe only the high-level construction of the game, the more detailed design is discussed in the following chapters):

- for the game to be multi-platform, we design it to run on more than one system, which helps us reach a wider audience
- the game being multiplayer means that the user plays via the Internet with other people instead of computer controlled entities
- a run and gun game implies a two-dimensional side-scroller, where the user can control only his own character, which he uses to launch projectiles at opponents trying to eliminate them while attempting to avoid being hit by them

Additionally, as a supplement of the game, we also include a game specific world editor module. In the interest of variability, the editor will make it possible to create tilemaps and polygon maps too. Finally, because visual elements play a big role in run and gun games, we will also offer the map creator the chance to use his own tile designs for the games, even if the tile images have not yet been seen by other players.

In the following section, we will select the platforms most suitable for our application.

1.2 Choosing the platforms

Nowadays game companies aim to target as many platforms with the same game as possible. Extended options attract more players and therefore the profit increases.

According to a survey done in 2017 by BusinessInsider (figure 1.1), desktop computer gaming (PC, Mac and Linux combined) still leads the charts despite the incredible growth mobile devices attracted in the last decade.

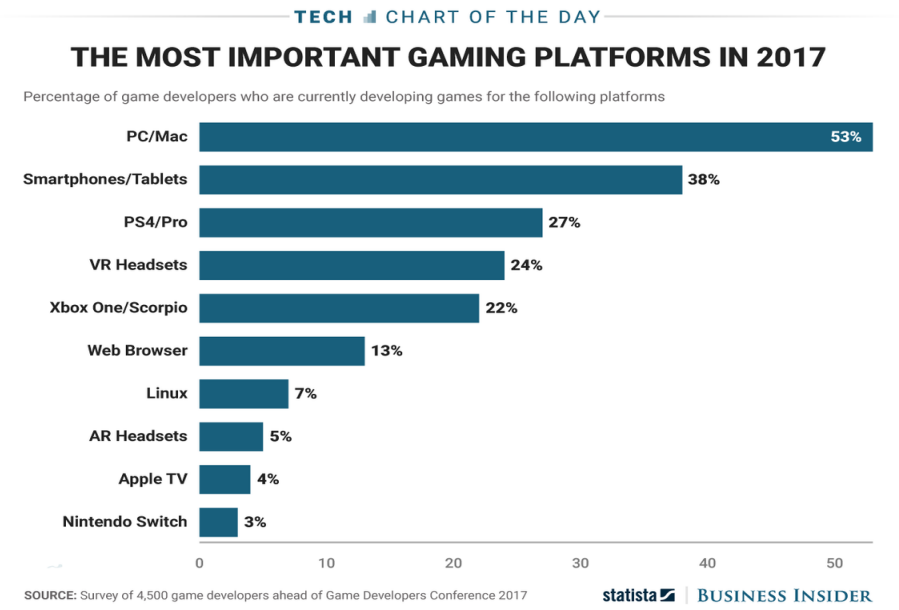

Figure 1.1: The most important gaming platforms in 2017. [1]

Let us now look at 2 points characterizing the differences between the platforms:

1. When considering smaller devices, the both-hand coordination and the complex control of the character does not allow a sensible design for the game. Mobile phones struggle with a lot of technical restrictions, some of which can be solved by upgrades and the constant improvements rolled out by manufacturers, but the one that is always apparent and most important for us is the size of the device. Since handheld devices share the same surface for both the display and the controls, when a game has a large number of buttons for navigation, the screen inevitably gets covered.

2. There are significant differences between how the platforms are controlled that can not be overlooked. Let us use a first person shooting game as an example and note how much of a disadvantage the mobile user has against the player on his desktop computer playing with a keyboard and mouse. We can see that the same multiplayer game can not be necessarily implemented well for both types of gaming platforms (the same argument is also true for gaming consoles, such as the PlayStation or Xbox) because the difference between the controls makes the game unfair towards a group of the players.

The user playing a run and gun game must control his character's movement plus the offensive and defensive actions all at once, the latter two of which require precise targeting. Because of this and according to **point 1**, we can conclusively decide that we cannot include mobile devices in our target platforms.

By considering **point 2** in the list above, we also see that we cannot design the game for both gaming consoles and desktop computers. According to the research above, the widest audience can be reached by developing a game for PC

[1] Source: `https://www.businessinsider.com/most-popular-game-platforms-developers-chart-2017-3`

and Mac. In addition, there is no reason why we cannot include Linux based computers as well, as they differ from other desktop platforms only in technical aspects, but not control-wise.

As a result we have determined, in order to attract the most users possible, we will design our game for all 3 desktop platforms - Windows, Linux and Mac.

2. Design of the game

In this chapter we list the games that inspired ours, describe the points which we carry over and which we change. Finally, we describe our game in detail.

2.1 Inspirations

Firstly we will present the games that inspired ours and highlight their most important characteristics while providing some information about the gameplay mechanics.

2.1.1 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-multiplayer, precision-based 2D shooter with high customizability (see figure 2.1 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 [2] play it mostly only as nostalgia, it was moderately popular a few years back [3].

The most interesting characteristics of the game are:

- The player is controlling a soldier with jet boots, which allow him to fly for a limited amount of time and which help the user navigate the entirety of the playing field. The soldier also possesses a weapon, which can be selected only upon the player's death and not during the active time. If the player is killed, his soldier character is returned to the spawn point and after a short amount of time, the user is active again.

- The game is played on 2D maps comprising of several polygon blocks, which are tailored for fast-paced gameplay. The map elements are static and the player is not able to interact with the layout, but there are various neutral elements (health packs, ammunition, etc.) scattered around, which the user can use or pick up.

- The goal of the game is based on the selected gamemode (of which there are several). This goal can be object based (where the teams fight for each other's flag or areas of interest) or free-for-all, where the objective is simply to eliminate the opposing players.

Figure 2.1: Screenshot of the game Soldat. [1]

2.1.2 Worms

Worms [4] 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 [5]).

There are many variations of the game, but we will be interested in the earlier editions, which are 2D side-scrolling video games.

- The goal of the game is the elimination of other (possible multiple) enemy teams.
- In order to achieve this, there is a number of tools and weapons available for the player to use and this choice is made during the active time of the player and can be changed freely. In the earlier editions, there are over 40 items to select from. [6]
- The concept, that the game is playable on almost any map (an example of a procedurally generated map is seen in figure 2.2) 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.

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

- Additionally, the environment is fully destructible. This gives the option to employ different tactics, e.g. destroying the map below an enemy worm can make the worm drown, which results in instant death, independent on its current remaining health.

Figure 2.2: Screenshot of the game Worms World Party. [2]

2.1.3 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* [7], 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.

2.2 Main ideas

In this section we briefly present the main ideas which we can take away from the games listed above. We describe that characteristics we choose to implement, ignore or improve upon (or possibly expand on).

Our goal is to create a two-dimensional side-scroller run and gun game with continuous gameplay, which will keep the player immersed. In order to achieve this, the downtime of the user - caused only by the death the of the character object - will be short, i.e. the respawn will be almost instantaneous (a few seconds is necessary so the player can shift focus).

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

The game is also mass-multiplayer, which means that it must be ready to handle hundreds of players.

2.2.1 Game objects

The user will be playing against other players via the Internet, distributed to teams, where each player will control only one game object - *the character object*. This character will have the ability to traverse the entire map with the help of basic movements and the ability to fly for a limited (replenishable) amount of time.

In addition to movement, the player will be able to attack and defend himself against enemies. For the offense to be successful, the user will have several weapons at his disposal, which can be selected and changed at any given point in time. 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. Additionally, we introduce character classes, which equip the players with *ultimates* - various spells with longer cooldowns to limit their usage. Other game objects include neutral enemies, which are uncontrollable. These objects give way for various tactics and therefore can make the player's life easier or harder.

2.2.2 Playing field

It is important to guarantee a level of variability to the game for it to not become repetitious. In order to achieve this, we will give the user the possibility to create custom maps and therefore we open the doors for the player to have different experiences with a single game.

One thing not applicable for our game is the concept of the destructible environment. With 10 players constantly active on the map, each of them firing different projectiles at each other every few seconds, this mechanic would quickly reduce the map to nothing.

2.3 Game design and flow

In this section we will look at the models of the key game components. 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 projectiles.

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.

2.3.1 The player object

The most important game object in the game is the one controlled by the player (illustrated in figure 2.3). 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. Beside these, in the event of his death, the player is respawned instantaneously (to mirror the continuous gameplay of Soldat).

Figure 2.3: The in-game player object.

Projectiles

After the movement is specified, the second prevalent characteristic of a run and gun game is that players can shoot, hurl or throw projectiles at each other, which makes these objects the main subjects of the users' interactions.

Inspired by the gameplay of Soldat, every player is able to create and launch projectiles in a given direction. This direction can be adjusted freely up until the shot is fired. When the shot is released, a small recoil force will kick the wizard in the direction opposite to the one of the projectile's.

The projectiles' collisions will be checked with the player object, the shield objects and the tiles constructing the map. In every case, the projectile gets destroyed upon contact, which causes keeps the number of objects constant in the scene during the game (as new projectiles are spawned, the older ones are destroyed).

Flight

Flight is another mechanism motivated by Soldat, which helps the player traverse the map faster, while respecting basic physics laws making the game feel more natural. Because of this, we will design the functionality the following way.

The variable that will control the process 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 increased 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.

In order to force the user to exploit the other types of movement, we limit the time a player can fly for. 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.

Shield

As described above in the previous sections, players in Worms are allowed to select their approach to the situation by choosing their tools during the gameplay. On this basis and because in the course of the game, the player is consistently targeted with projectiles, we balance this by introducing a shield unit capable of blocking missiles. Additionally, so the usage of this component becomes a calculated choice, we limit the time the shield can be used.

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.

Weapon system

As discussed regarding the shield mechanics, we aim to allow the player to select their tools on the fly. We also aim to provide the user with different types of these devices, so he can always choose the best fit for the presented situation (this idea is one of the core principles of the game Worms).

In our game the player will possess 3 different weapons, namely the staff, the rod and the dagger (figure 2.4). Different weapons have different properties, ranging from reload time to maximum possible damage dealt by the different projectiles. Each weapon is suited best for distinct situations: there is a chargeable weapon (where the damage dealt by the projectile is based upon the time it spent being charged by the player), an automatic weapon (where there are multiple lower-damage projectiles fired in rapid succession) and a one-off weapon (which deals substantial amount of damage to other players, but has a longer reload time than other firearms).

Figure 2.4: Available in-game weapons.

The targeting system mirrors the one from Soldat. The weapons rotate with respect to the movement of the mouse, thanks to which the player is able to aim the shots at his opponents or direct his shield to deflect a projectile headed towards him. The rotation and selection of the weapons are networked, thus every player can see the other one's choice and its position.

Class of the character

In order to put more focus on correct teamplay (the team with players choosing complementing abilities will usually perform better than the one which does not care about synergy), games often introduce character classes with unique strengths (as referenced for the Overwatch).

We add this functionality to our game by granting the player 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.

Every ultimate will have an active time and a cooldown time (so they are used strategically and not constantly). Ultimates are continuously charged and at the moment it is ready, the player can cast the spell. After release, an expanding circle emits from the given player and the effects are applied on players whom collide with this circle. 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 reload time of all weapons are halved (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 teammates by setting their health to 100 - additionally, if the player or 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

2.3.2 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 game objects. Our game supports both tile based maps as well as polygon maps. Additionally, since graphical elements play a big role in run and gun games, we enable the creators of maps to add their own style to it as well. The designers are able to use their possible unique tile designs in maps they create. This function will be described in a later chapter in detail.

2.3.3 Neutral enemies

The last type of objects not present in the referenced games are neutral enemies. Beside the previously discussed ultimates, they can contribute to the diversity of the game by unlocking new strategies a player can use. 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 done in the World Editor (this will be discussed later).

There are 2 different types of neutrals (illustrated in figure A.6) implemented in the game, both with a unique ability and purpose:

1. **Knight:** a game object that will move only horizontally and will damage 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. **Archer:** the neutral will spawn and launch a projectile in the upwards direction relative to its rotation. The projectiles are spawned after the specified time has passed.

Figure 2.5: Neutral enemies in-game.

2.3.4 Gamemodes

The goal of the game is determined by the selected gamemode. The two classic types of these implemented are Deathmatch [8] and Capture the Flag [9].

DeathMatch is possibly the most popular gamemode in shooting games. The objective is to eliminate other players while trying to die as few times as possible. In our game, we will play the variant Team Deathmatch, where the players are separated into 2 teams. A team earns a point if a player from the opposing side dies.

The Capture The Flag gamemode is focused on the principles of obtaining and defending a specific game object. Each team has his own flag and the objective of the game is to bring the enemy team's flag to your own base (or more specifically, the place where one's flag's base point is), while not losing your own flag to the enemy team (called "capturing the flag"). In this style of play, kills are not as important and various strategies can be implemented to try to find the optimal approach.

The game ends if one the teams reaches the target score.

2.3.5 WorldEditor

In this section we will deal with the design of the game's world editor. By permitting the user to create the layout, we insure a certain mutability to the game, creating a different experience for the player for every different map he plays.

The WorldEditor is a tool, which the user can use to create a game setting he can later use to play in. This editor will allow the user to:

- shape the map by creating tile objects with custom sprite images
- create custom polygon maps using the tiles drawn (from here on in this thesis we will refer to this functionality as "smoothing" the map - because it transforms the right-angled outline of the map to a curved-surfaced one - as shown in figure 2.6)
- decide on the spawn positions of the teams (and other gamemode specific objects - e.g. the flags for the gamemode Capture The Flag)
- add an arbitrary number of neutral enemies with fine-tuned behaviour properties to the scene

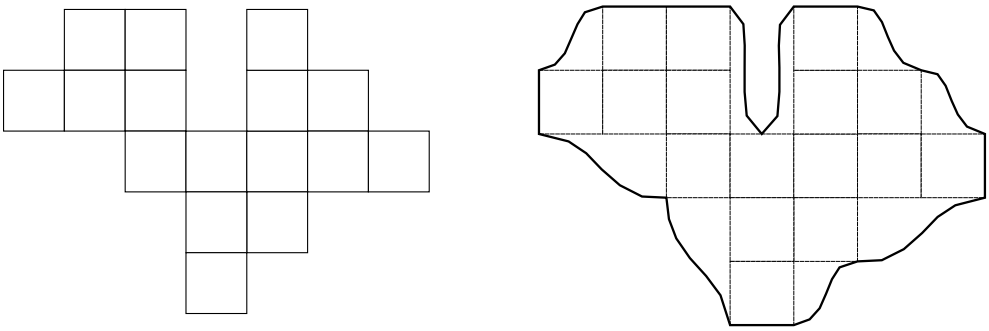

Figure 2.6: Converting ("smoothing") the tilemap to a polygon map.

3. Analysis and implementation

In the previous chapter we designed the game, including the objects, game flow and other things. In this chapter we go through the necessities, which we will need to implement the objects and concepts described previously. We will discuss the lobby system, evaluation of players and the map smoothing algorithm. These things will contribute to the decisions made about the game and network engine and the world editor in the next chapter.

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, the game mode and set the length of the match by modifying target objective number. 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.

At the start of the game, the player picks one from these options:

1. Create a custom game on the remote server (public or private) 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. Enter the matchmaking pool and play a balanced and ranked match.

This process is described in detail later in the thesis.

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.

3.1.1 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 (in case of a ranked balanced match, this option is disabled) - illustrated in figure 3.1. Besides these, the host will have to pick the game mode, set the game length 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 a 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.

Figure 3.1: Representation of the player in the lobby.

The maximum number of players in a game is set to 10, so in the event of a custom lobby this 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 Player evaluation

The goal of every game is to make the user enjoy themselves while playing it, which can only be achieved when the players involved in a match are similarly skilled. Imbalanced games tend to cause bitterness in people - especially in those playing for the disadvantageous side - which make them less likely to play the game the next time. Estimating a player's skill however is not a trivial task, but fortunately there were a few models researched and put together for exactly this purpose.

In this chapter, we will explore our options for a matchmaking model and find that Microsoft's TrueSkill model is suited to our needs.

3.2.1 Matchmaking models

There are 3 popular matchmaking models used nowadays - ELO, Glicko and TrueSkill.

The *ELO* system [10] was designed and developed by Arpad Elo and is now widely used in competitive games - the most prominent of them being chess, for which it was officially adopted by the World Chess Federation. Unfortunately, ELO works only for two-player games and cannot handle team-based ones. One attempt to bridge this problem was the so-called *duelling heuristic* as referred to in the already mentioned TrueSkill paper [11]. This approach treats teams as individual players who play against the players on the enemy team and then calculates the average of these duels.

The next model is called the *Glicko* system, which was developed by Dr. Mark E. Glickman in 1999 [12] as an extension of the ELO system. This model introduces a skill trust factor (standard deviation in statistical terminology) referred to as *ratings deviation*, which describes how much confidence does the system have in the player's rating (this confidence gradually declines when the player has not played in a while for example). While this system addressed key defects of ELO, it still remained a two-player rating model.

The last model (which will be our pick as a model by process of elimination) is the already cited *TrueSkill* system developed by Microsoft. It is viewed as a generalisation of the ELO system, which can deal with teams containing more than only one player and calculate the updated rank of the individual from the team's performance. This model provides similar results as the above mentioned

ELO system with the duelling heuristic, but performs twice as fast and tends to converge better (according to tests conducted by Jeff Moser [13]).

3.2.2 More about TrueSkill

The TrueSkill system uses Bayesian inference for ranking players. Each player is assumed to have a prior skill, which is described by two variables: the mean (the assumed rank of the player) and the standard deviation (the confidence in the established rank).

In short, the algorithm uses a factor graph (figure 3.2) with message passing to calculate the new ranks based on the match outcome. If we follow the path down the graph, the process predicts the player's performance based on his prior skill. According to the predicted performance we can anticipate the performances of the teams (which TrueSkill determines simply as the sum of the player performances), which can then be compared. We iterate this process until the values at the bottom of the graph converge. After this, we reverse the directions each factor and propagate messages back up the graph. The new skill rating of the player is established once the messages reach the top level of the graph.

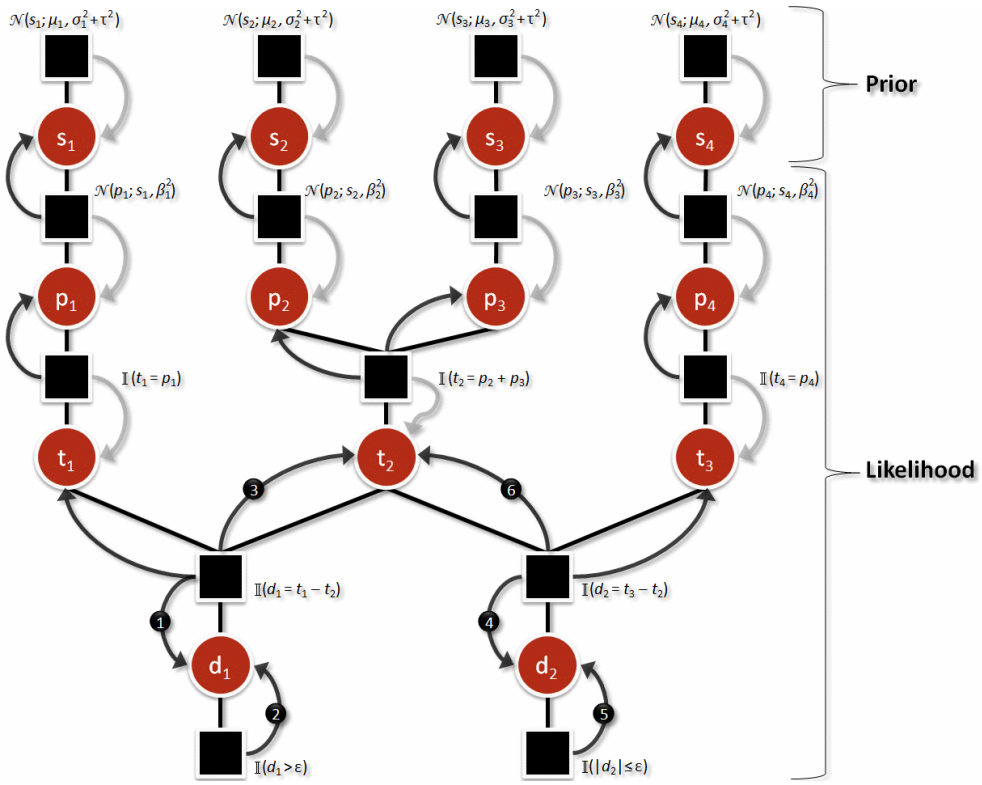

Figure 3.2: TrueSkill factor graph. [1]

[0]Note: for the sake of completeness we mention one more system called Rankade and their skill score called Ree [14]. In the time of writing this thesis the model counts as new so there is little information available about it and almost all of them are originated from the developer team and is therefore considered subjective. There is also no information about the mechanics, only some key points highlighted in which it is supposedly superior to TrueSkill.

[1]Source: `http://www.moserware.com/2010/03/computing-your-skill.html`

Another plus characteristic of the method of using a factor graph to calculate new ranks is the built-in extensibility of the model. In the March of 2018 Microsoft released an improved version of their system labelled TrueSkill2 [15], which introduces the concept of metric-driven modelling and extends the basic factor graph by adding in variables, which model the in-game behaviour of the player better than the original version. Unfortunately, these metrics are game-specific and can only be determined from previous real game data, so the analysis and implementation of this model is left as a future improvement (for after the game is released and sufficient amount of data is collected).

3.3 WorldEditor

In this section, we go through the required functionalities for the world editor.

3.3.1 Creating a visual aid for object placement

First of all, we will require a grid in order to help us position the tiles, neutral enemies, properties, etc. We need a grid which will not be present anywhere else, which would only serve as a visual assistance for the user to help him better differentiate and place the elements in the scene.

3.3.2 Required elements

After the grid is set, we need to define the settings of the map and add the interactivity for the user to place the tiles, properties and neutral enemies on the map.

First and foremost we will need to decide on the size of the map. We will define map size as 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. These artworks are then synchronized amongst the players, which is discussed further in a later chapter. In addition, we need a way to clear all tiles in the map for the user's convenience as we aim to be as user-friendly as possible.

After the main building blocks of the map are set, we compel the user to set the map properties. Note, that the placement of these properties is essential for the game to function and we can not let the user create 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 these necessary properties are set, we must offer the user to optionally place the neutral enemies. The creator must be able to 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 (e.g. the time interval of two shots of the Archer object). This results in a more constricted world editor, but guarantees the proper and sensible functioning of the neutral.

In order to allow the user the creation of polygon maps, we must also offer a mechanism to convert the designed tilemap to a polygon one (referred to as smoothing). We choose to create polygon maps this way because it is generally easier and more intuitive for the creator than with a free hand drawing tool (which can also create possible unwanted artifacts, e.g. a random dot somewhere in the air, which would restrict movement, block shots and cause undesirable effects). Naturally, the tradeoff for this optimization is that the editor becomes more constricted, but this way we can guarantee smooth movement through the map.

Lastly, we need a way to save the created map and artwork so the player can use it in the game. First, we will need 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 and it will also 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.

As an extra element, we will require a logger system, where we can notify the user if a problem occurs during the creation or serialization of the map.

3.3.3 Custom tile picking

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 specific tile we want to place on the scene.

This component's purpose is to display every tile image in the given folder (specified by the user). Note that we would like the user to see the images of the tiles and not just the names of the files. Furthermore, we would also like the component to show the tiles as soon as they are moved into the folder, so the component will require a refresh utility.

3.3.4 Easing the editing

Beside the placement mechanisms, we must also give place to human error and so provide additional ways of deletion or of replacement. Additionally, if the user changes the size of the map after he has placed some tiles already, the structure of the map still relevant between new boundaries must remain and any blocks outside of the new map must be destroyed properly.

Because of the advantages of this structure, we use something similar for our property and neutral enemy placer functions 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).

4. Libraries and game engine

In this chapter, we explore the libraries used to implement the game and world editor, which we selected according to the characteristics and specifics described in the previous chapters. We present the libraries, their properties and also compare them to alternative ones.

4.1 Game engine selection

In order to select the game engine best suited for our needs, we can establish a few basic criteria according to which we can make our decision:

- Because there are many engines out there using different scripting language, we can settle on the usage of C# (mostly due to the author's familiarity with it).
- Due to our project being non-commercial, we limit our search for libraries and engines which are free.
- The engine being ready to support the targeted platforms: Windows, Linux and Mac.
- The support for real-time multiplayer games.

Now let us analyse the different game engines according to our conditions above.

According to the first two conditions, we narrow down our search for 5 game engines: Unity3D [16], WaveEngine2.0 [17], Duality [18], Xenko Game Engine [19] and MonoGame [20]. It is also the reason why we do not discuss the likes of UnrealEngine [21] and CryEngine [22], which are otherwise popular and widely-used game engines [23] (but both of which use C++ as a scripting language).

Let us now move on to the third filter in our list. Unity3D, WaveEngine and MonoGame all satisfy this condition as specifically stated on the homepages of the engine. However, Duality and Xenko fall short.

Duality is Windows-only and has no cross-platform support. As stated in the official GitHub page [24], games may run on Linux and Mac, but they are not officially supported.

As for Xenko Game Engine, the deployment for Mac is not supported, although the official homepage of Xenko promises more supported platforms to come in the future. [25]

We can now examine the fourth and last condition in our list. Unfortunately, for WaveEngine2.0 there is no support for real-time multiplayer games and only turn-based communication is offered through the network. *On the 18^{th} of June, 2019 the team behind Wave released a brand new version WaveEngine3.0 [26]. In the release notes there is no information about added support and this release was published only after the analysis for this thesis was completed.*

This leaves us with Unity3D and MonoGame. For the latter game engine, the principle of multiplayer support is exhausted in a very simple networking API,

which has not been maintained for the at least 2 years (and the last update for game engine as a whole was released on March 1st, 2017 [27].

However, Unity3D is not without its problems. The engine's legacy multiplayer solution - referred to as UNet - is being deprecated as of 2018 [28], which would interfere with our project in the future (regarding future development). The replacement for UNet will also not be ready for at least until 2020 according to the source cited above. Fortunately, thanks to Unity's popularity, there are other options to consider.

Probably the most popular alternative is Photon Bolt, which provides seamless integration with Unity and is a correct replacement for UNet. Unfortunately, Photon is a commercial project and only allows 20 concurrent users for the free package [29].

Another alternative is called Forge Networking, which has been rebranded as Forge Networking Remastered following the open-source announcement in January 2019 [30]. This networking system is flexible and most importantly it does not have a concurrent user limit, making it a good choice for us (we can handle hundreds and even thousands of players cost-free). This solution is also released under MIT license, which means that we can also customize it to our liking.

To sum it up, we arrive at the conclusion that the most preferable choice for our game is to use Unity supplemented by Forge. Additionally, Unity is also well documented and contains a lot of advanced features which prove useful to us, such as the built-in Animation system, the ParticleSystem or the Unity Asset Store, which hosts many audio-visual elements, which we can use in our game. In order to develop our game, we will be using the version 2018.3.7f1 of Unity.

4.2 World Editor

Our next task is to choose an appropriate tool to serve as the world editor, but the selection process here is much more difficult. As specified in the previous chapters, our editor must support tilemaps as well as polygon maps while maintaining ease of use and giving the creators options to paint each tile separately and with custom designs. Additionally, we must be able to place neutral enemies and world properties as well without them colliding with the map and as the final thing we must be able to serialize what we created.

There are several packages in the Unity Asset Store to build tilemaps, but there are no free options, which offer conversions to polygon maps. One interesting option would be Terrainify 2D [31], which could handle tiles and polygon based maps as well. However, this solution generates maps using Perlin noise, which would interfere with custom tile designs.

Since the 2018 editions, Unity also has a built-in Tilemap [1] system aimed to fill the need to use 3rd party software (such as Tiled [32]). The system is quite robust, unfortunately the painted tilemap is handled as one unit (one game object) and we have no information about the individual tiles, which is indispensable for map smoothing and the placing of other map elements.

Consequently, the world editor of our game must be implemented by us. This can be achieved via Unity Editor scripting. The reason behind this is that the

[1] Tilemap: (`https://docs.unity3d.com/Manual/class-Tilemap.html`)

engine 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.

4.3 Player evaluation

As we previously established, we use the TrueSkill in our game to update the skill representation of a player after a game is finished. There are a few freely accessible implementations. We will use Jeff Moser's library [33], which has a few advantages from our point of view, mainly that it is written in C# and therefore is directly accessible from our project. In addition to this, the implementation is also understandable and documented by the accompanying paper.

5. Map smoothing

Now that we have established that the our World Editor will be custom written, we need an algorithm for the map smoothing in order to offer polygon based maps besides tilemaps as well. We present this functionality in this section.

The algorithm receives a list of tiles, which the user can select manually. For the user's convenience it is also possible to pass multiple "blocks" of tiles on one call (i.e. more shapes can be passed) in which case the algorithm generates a polygon map for each. The following algorithm is executed for every block of tiles.

1. Our first task is to find the contour of the block, i.e. the list of outer edges. We realize that every edge belongs to at most 2 tiles. If an edge does belong to exactly 2 tiles, it means that the edge is positioned on the inside of the block (represented as a red edge in figure 5.1), whereas if it belongs to only tile it is an outer one and therefore a contour edge (green edges in figure 5.1).

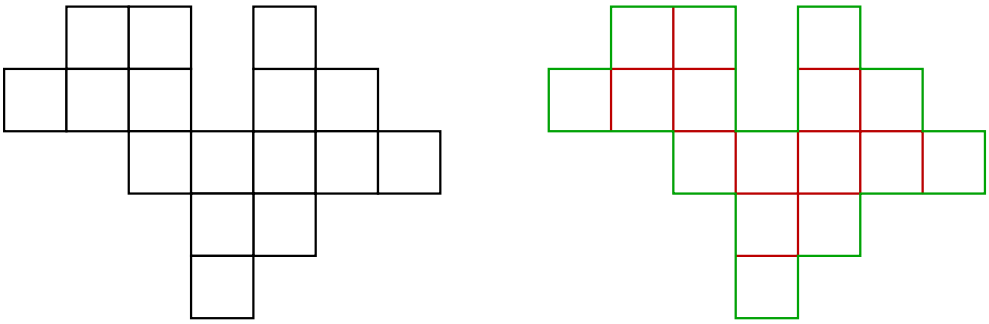

Figure 5.1: Tilemap contour.

```
Input: list of tiles
Result: dictionary of <coordinate, list of edges>
contourEdges = [];
for every tile do
    for every edge in tile do
        if edge ∈ contourEdges then
            remove edge from contourEdges;
        else
            add edge to contourEdges;
        end
    end
end
edgesByCoordinates = {};
for every edge in contourEdges do
    add edge to edgesByCoordinates[first coordinate of edge];
    add edge to edgesByCoordinates[second coordinate of edge];
end
return edgesByCoordinates
```

2. The second step is to gather the points on the contour in a sequential order. For our ordering we chose clockwise orientation, which means that if we arrive at a contour point for example from the left (as highlighted in figure 5.2), we will consider the edges in the order of up, right, down and left and choose the first unused edge. This is important especially if we have an outlier tile, which is connected to the block by only one vertex (as depicted in figure 5.2) so we do not leave out a tile.

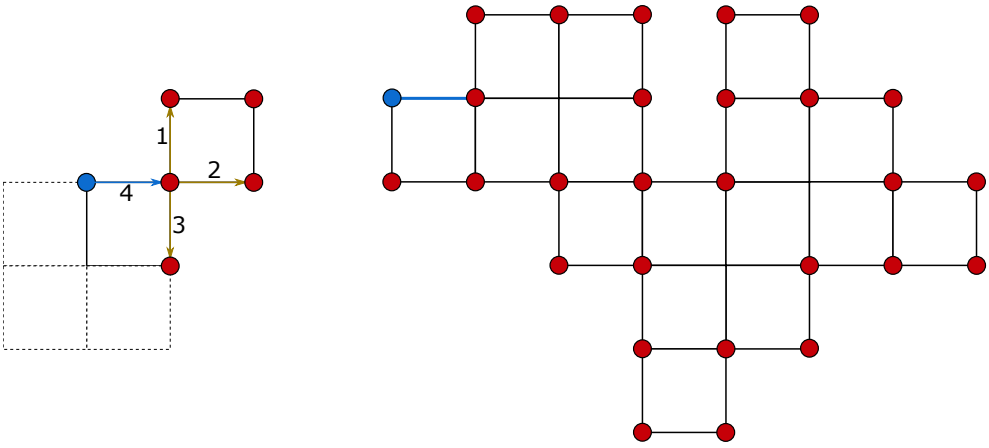

Figure 5.2: Tilemap contour points.

```
Input: edgesByCoordinates = dictionary of <coordinate, list of edges>
Result: list of contour points
contourPoints = [];
usedEdges = [];
edge = select first clockwise edge (x2 > x1 or y2 > y1);
add edge to usedEdges;
add first point of edge to contourPoints;
currentPoint = second point of edge;
while true do
    sort the edges in edgesByCoordinates[currentPoint] according to
     clockwise orientation;
    edge = first unused edge in edgesByCoordinates[currentPoint];
    if edge is null then
        break;
    end
    add edge to usedEdges;
    add currentPoint to contourPoints;
    currentPoint = other point of edge;
end
return contourPoints
```

3. Our third step is to find the corner points in the tilemap. We take into consideration all contour points available and check whether the previous and next contour point share one coordinate (i.e. their x or y coordinates are equal) or not. If the latter is true, we denote the contour point as corner point (as seen in figure 5.3).

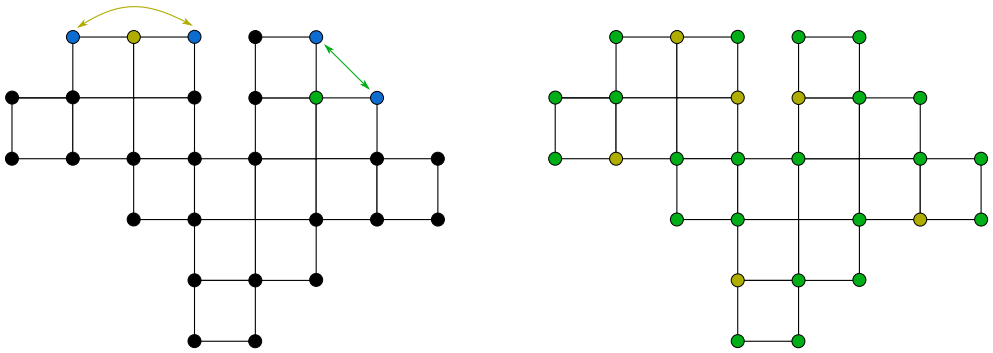

Figure 5.3: Tilemap corner points.

```
Input: list of contour points
Result: list of corner points
cornerPoints = [];
for every contour point do
    previousPoint = previous contour point;
    nextPoint = next contour point;
    if PointsShareCoordinate(previousPoint, nextPoint) then
        continue;
    end
    add contour point to cornerPoints
end
return cornerPoints
```

4. In this step we are starting to gather the points of the polygon. First we start with those that are situated on the tilemap and are necessarily corner points. It is important to note that not all corner points are polygon points, only those from which the outgoing edge is in a clockwise orientation relative to the incoming edge (e.g. if we arrive to the point from the left, the edge pointing down is considered to be in clockwise orientation, while the edge pointing up is not). These points are denoted as green points (including the blue point, which we choose as a starting point in the next step) in figure 5.4.

```
Input: list of contour points, list of corner points
Result: list of polygon-corner points
polygonPoints = [];
for cornerPoint ∈ list of corner points do
    previousContourPoint = the contour point before cornerPoint;
    incomingEdge = Edge(previousContourPoint, cornerPoint);
    nextContourPoint = the contour point after cornerPoint;
    outgoingEdge = Edge(cornerPoint, nextContourPoint);
    if EdgeTurnsRight(incomingEdge, outgoingEdge) then
        add cornerPoint to polygonPoints;
    end
end
return polygonPoints
```

5. This is the step where we calculate the polygon points which smooth the map. We begin by choosing a starting polygon point (marked as blue in figure 5.4) and we proceed to find the next one as well. According to these two points we have three options. The first option is that the points share a coordinate and the second polygon point is the first corner point immediately after the first polygon point (denoted as option a, in figure 5.4), in which case no extra action is necessary. The second option is that the two points share a coordinate and there are at least two corner points between them (situation denoted as option b, in the figure). In this case we find the "midpoint" of the valley and calculate various number of points (according to the distance between the 2 polygon points) from the first polygon point to the midpoint and additional points from the midpoint to the second polygon point according to a smoothing function. The function we use for this purpose is called the ease-in-ease-out function, which guarantees a nicer slope than for example a linear function would. The third option is that the two polygon points do not share a coordinate (marked as option c, in the figure), in which case we create a slope between the points the same way as we did for a polygon point and the valley midpoint.

```
Input: list of corner points, list of polygon-corner points
Result: list of polygon points
polygonPoints = [];
polygonPoint = select first polygonPoint;
/* the first polygon point is purposefully there twice for
   the MeshCreator to work                                    */
add polygonPoint to polygonPoints;
for nextPolygonPoint ∈ tail(list of polygon-corner points) do
    if PointsShareCoordinate(polygonPoint, nextPolygonPoint) then
        if IsNotTheFollowingCornerPoint(polygonPoint,
         nextPolygonPoint) then
            /* option b, in the figure                        */
            midPoint = valley midpoint;
            downSlopePoints = EaseInEaseOut(polygonPoint, midPoint);
            add downSlopePoints to polygonPoints;
            add midPoint to polygoingPoints;
            upSlopePoints = EaseInEaseOut(midPoint,
             nextPolygonPoint);
            add upSlopePoints to polygonPoints;
        end
    else
        /* option c, in the figure                            */
        slopePoints = EaseInEaseOut(polygonPoint, nextPolygonPoint);
        add slopePoints to polygonPoints;
    end
    add nextPolygonPoint to polygonPoints;
    polygonPoint = nextPolygonPoint;
end
return polygonPoints
```

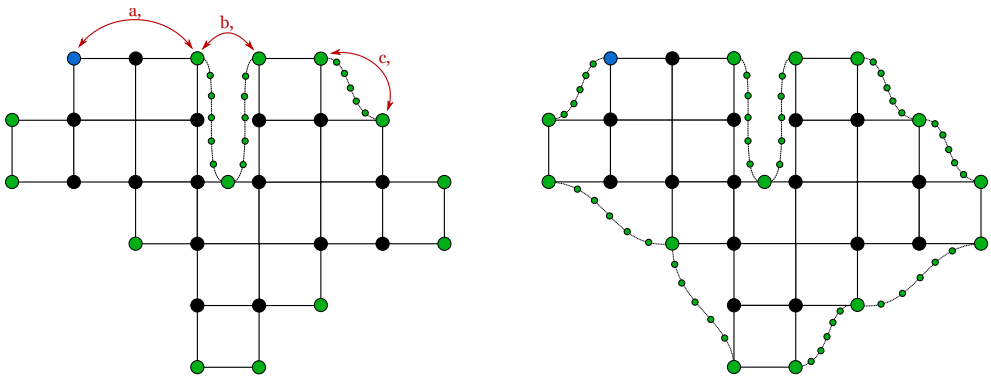

Figure 5.4: Tilemap polygon points.

6. The last step is now only to create the mesh from the polygon points (figure 5.5).

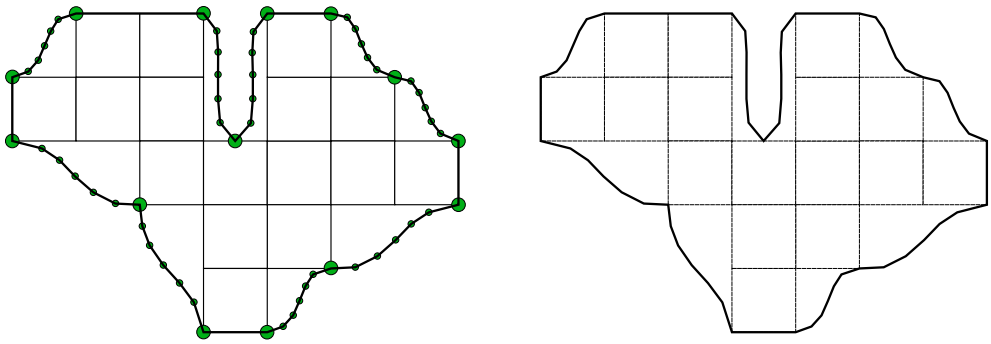

Figure 5.5: Finished polygon.

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 business model, where the company releases the game for free and collects the revenue from character customization is growing more popular nowadays and one of the companies 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).

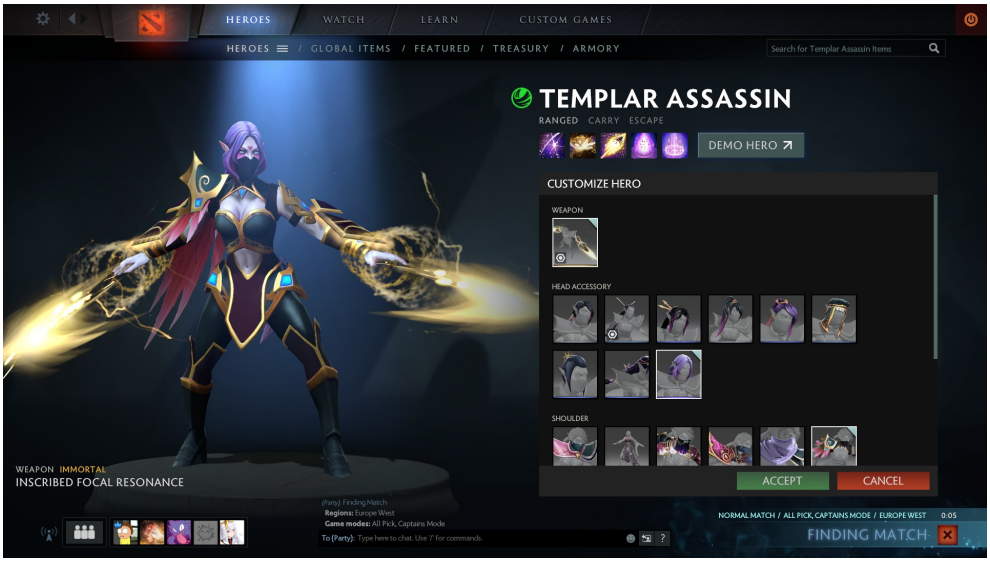

Figure 6.1: Dota 2 hero customization. [1]

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, it resulted in granting the designers to use custom images besides the ones provided. This is where we draw our inspiration from to permit unique styles to manifest in our games. For this, we will need a way to organize the requests received from

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

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).

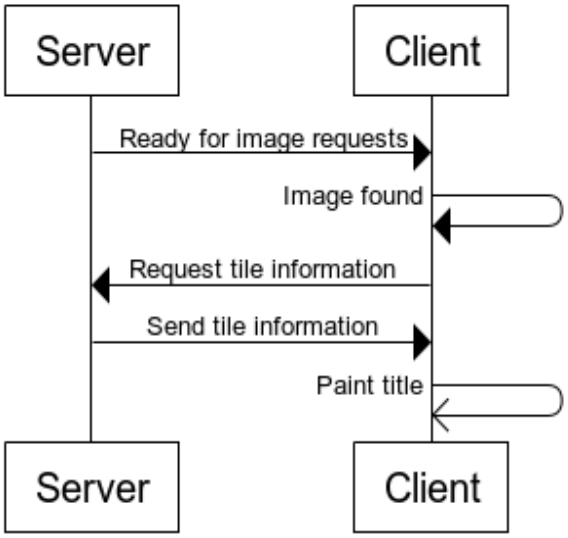

Figure 6.2: The process of sprite synchronization if the client is in possession of it.

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.

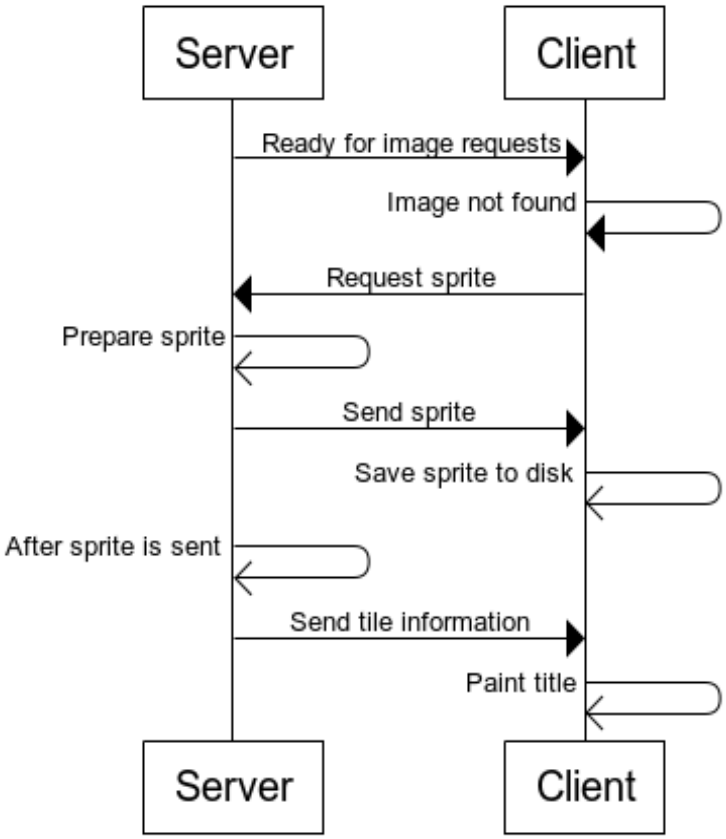

Figure 6.3: The process of sprite synchronization if the client is not in possession of it.

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.

7. MasterServer

Now that we have our game designed and implemented, the only thing left to do is to design a service, which will allow the players to find and play with other players.

In our thesis, as is typical for most multiplayer games, we design this server for two purposes (figure 7.1). The players will have the ability to create custom lobby matches, which can either be private (password protected) or public. This mode allows the players to have an unranked, possibly imbalanced leisurely match with their friends or others from the world. The other service is a matchmaking service, which will automatically group players according to their ranks and form matches where the emphasis is on the balance between the two teams. In this thesis we use TrueSkill to rank players and discuss the matchmaking algorithm in detail.

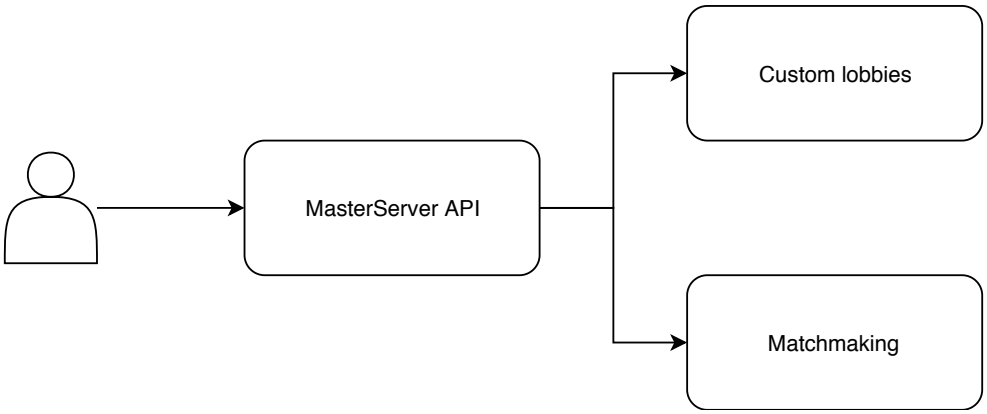

Figure 7.1: High level overview of the server.

7.1 Custom lobbies

The purpose of the custom lobbies module is to store up-to-date information about the registered servers and to supply them to players on demand. For this task, we define 3 operations (as illustrated in figure 7.2):

- **register** - creates the match on the server with the supplied information. This is where the player can decide to create a private match with specifying a corresponding password for the lobby room, which is necessary for others to join.
- **update** - the method is used to update match information on the server. In our case we use it to keep the player count in the lobby up-to-date and to signal if the host player changes the gamemode of the match.
- **get** - this is a simple query command, which triggers the server to send the player the list of matches currently registered. The player can then enter the lobby with the help of the supplied connection information.

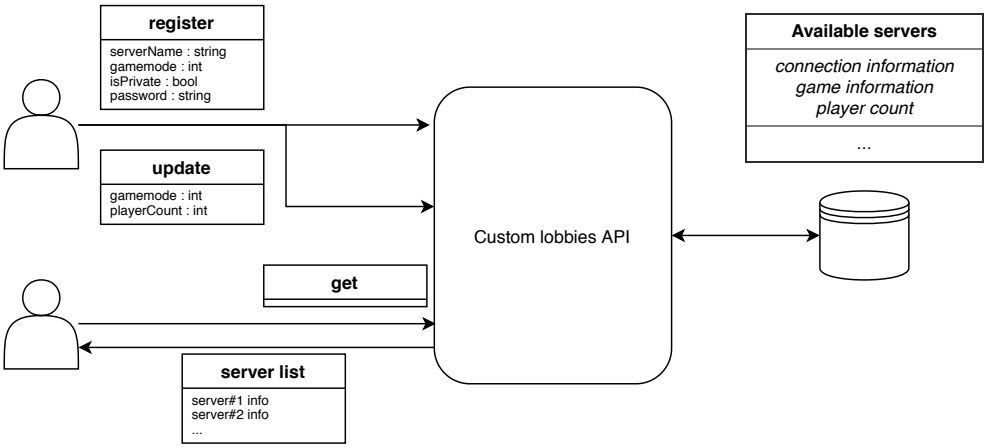

Figure 7.2: Custom lobby creation.

An important aspect of these games is that they are considered unranked. This could be problematic if we consider a player who plays only custom lobby matches and picks up sufficient experience with the game before playing any ranked matchmaking games. This is solved by the very idea of TrueSkill ranking, which does not represent level of expertise by only one values, but two (so there is a rank uncertainty variable as well). This is explored more in detail in the next section.

7.2 Ranked matches

We already discussed the different player evaluation algorithms in detail in a previous chapter. We already established that TrueSkill is the appropriate choice for our application and now we will develop the matchmaking process further.

An important equation, which will be useful to us for automatic match creation is the one for determining match quality between two teams. We can also view this as the probability of a draw between the two teams, so naturally this value is in range $[0, 1]$. The higher the chance for a draw, the higher chance that the teams are balanced and therefore the match quality is better. The formula for two players in two teams is supplied in the original TrueSkill paper as equation (7):

$$q_{draw}(\beta^2, \mu_i, \mu_j, \sigma_i, \sigma_j) = \sqrt{\frac{2\beta^2}{2\beta^2 + \sigma_i^2 + \sigma_j^2}} exp(-\frac{(\mu_i - \mu_j)^2}{2(2\beta^2 + \sigma_i^2 + \sigma_j^2)})$$

Since we established that in TrueSkill the team performance equals the sum of the players' performance in the team, we can introduce the team ratings as following:

$$\mu_t = \sum_{player \in team} \mu_{player}$$

$$\sigma_t = \sum_{player \in team} \sigma^2_{player}$$

With this we can formulate the equation for the match quality with two teams:

$$q_{draw}(\beta^2, \mu_{t1}, \mu_{t2}, \sigma_{t1}, \sigma_{t2}) = \sqrt{\frac{n\beta^2}{n\beta^2 + \sigma_{t1} + \sigma_{t2}}} exp(-\frac{(\mu_{t1} - \mu_{t2})^2}{2(n\beta^2 + \sigma_{t1} + \sigma_{t2})})$$

where n is the total number of players

$$n = |t1| + |t2|$$

7.3 Matchmaking

Now that we have established how the ranks of the users are updated, in the second part of this chapter we will examine how the matchmaking algorithm should work (i.e. grouping together the players to create a balanced match).

There are several factors influencing the creation of the match. Our top priority is to create a team composition, where each team has the same chance to win, however, we must also monitor for example the players' times spent in queue. We will look these and other factors' impact on the matchmaking process in the following section and construct a formula, which will determine whether we create the pondered match or cancel it.

7.3.1 Matchmaking aspects

In the paper Theoretical Foundations of Team Matchmaking [43] by Josh Alman and Dylan McKay, the factors listed are *fairness*, *uniformity* and *time-sensitive queueing* (plus another one called *role-restricted queueing*, however, this does not pertain to our game as players choose their classes after the match is established), which we will detail in the following paragraphs.

In this context, *fairness* means that the two teams have a roughly equal chance of winning. We note however, that we already deduced a function calculating the fairness of a match in a previous section - the q_{draw} function of the TrueSkill algorithm. For our function to fit into the equations presented in the paper, we simply denote

$$d = 1 - q_{draw}$$

as the fairness factor, because we try to maximize q_{draw} and the paper identifies $d = 0$ as the fairest game possible.

Uniformity is a measure of how equally the players are skilled. Note that this is not the same as *fairness*: we can imagine a game where there is one extremely skilled player in each team, while the others have relatively low rating. It is not hard to imagine that not all players would enjoy this game. This is why this factor is introduced and represents how spread out the players's ratings are. The paper characterizes this generally as *q-uniformity*, but it will suffice for us to use the standard deviation:

$$v_{match} = \sqrt{\frac{1}{n} \sum_{player \in match} (\mu_{player} - \mu_{match})^2}$$

where n is total number of players

$$n = |t1| + |t2|$$

and μ_{match} is the mean skill of all players

$$\mu_{match} = \sum_{player \in match} \frac{\mu_{player}}{n}$$

Then the *imbalance function* looks the following:

$$f = \alpha d + v_{match}$$

In our thesis we use $\alpha = \frac{1}{2}$, so we deem fairness and uniformity equally important.

Additionally, we also introduce a *time-sensitive priority* variable t_{player} to address the queue length problem. Then we can modify our imbalance function to look like

$$g = f - \beta \max_{player \in match} t_{player}$$

β is a hyperparameter, which characterizes how much emphasis should be put on low queue times (in exchange of the fairness and uniformity of the game). This parameter is set empirically, which we will discuss later during the testing phase.

One last factor not presented in the paper but which can be of use for us is what we will call the *game-completeness factor*. This is another technique to reduce the necessary time spent in queues. Although we prefer games in which the number of participating players is the maximum possible, it is reasonable to allow less players to play as long as the teams are balanced. We can control the emphasis put on this factor with the hyperparameter γ, which we can find empirically as well:

$$h = g - \gamma(10 - n)$$

To put it together, the function we will try to minimize is

$$h = \frac{1}{2}(1 - q_{draw}) + v_{match} - \beta \max_{player \in match} t_{player} - \gamma(10 - n)$$

7.3.2 MatchmakingWorker

We first present a single instance of a *MatchmakingWorker* object (illustrated in figure 7.3). This class has the responsibility to find the best match with a given set of players, for which we will utilize the function described in the previous section.

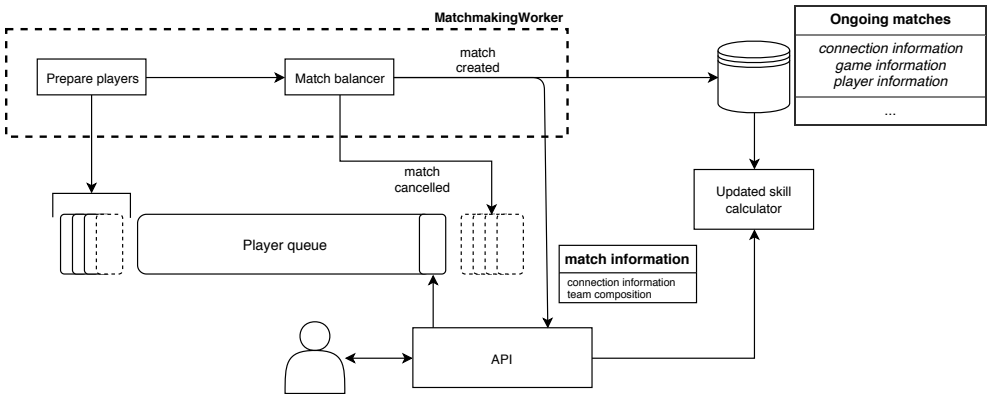

Figure 7.3: The workings of the prototype matchmaker illustrated.

The module works in a loop:

- The *Worker* dequeues at most 10 players from the front of the queue and passes them to the MatchBalancer function. If the number of players in the queue is less than 10, the *Worker* selects all the players in the queue.
- The MatchBalancer function tries to create the best match possible (by examining every viable team composition, where the number of players in the teams are equal - of which there are $\binom{10}{5}$ options) with the given players (finding the minimum value of h possible). Since the only component in the h function that is different with different team compositions is the fairness factor, we are looking for

$$\underset{team\ composition}{\operatorname{argmin}} \; d = \underset{team\ composition}{\operatorname{argmin}} \; 1 - q_{draw}$$

  from which we can calculate the value of function h. If this value is not less than some parameter λ, the match is cancelled and the players are put back into the queue and the execution jumps to the first point. If the match quality is good enough, we proceed to the next point.
- The game is added to the list of ongoing matches and the connection information and team composition are sent to the player.
- When the game is concluded, the server player reports back the result, according to which the new player ratings are calculated (and the match is naturally removed from the list of ongoing matches).

Now we can describe the method with which we can incorporate this unit into the server and create the games for the players.

7.3.3 Creating a multi-threaded matchmaker

In this section we introduce bins to our matchmaking model. The main idea here is that when a player connects to the matchmaking service, he is assigned to one of the bins according to his skill rating. In reality, the bins are just queues, upon which the *MatchmakingWorker* instances can operate similarly as described above

(illustrated in figure 7.4). Additionally, there is no reason why these instances could not operate simultaneously, therefore we can also introduce multithreading, which increases the performance of the server and guarantees lower queue times for the players.

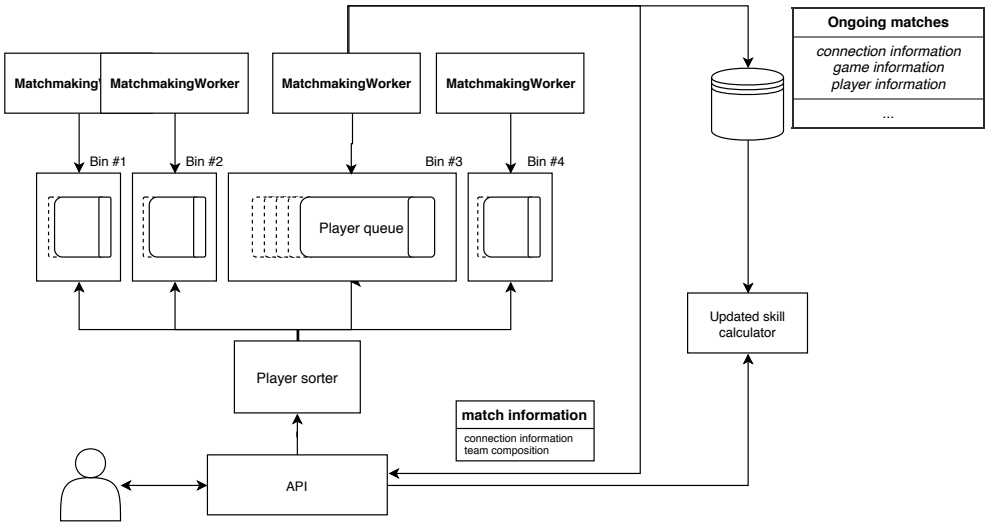

Figure 7.4: The layout of the multithreaded server.

It is easy to see the advantages the matchmaker bins bring us. If a match is cancelled and players are put back into the queue as last ones in line, the queue is inherently a lot shorter (than the one big data structure we would have), which improves the queue time. It also addresses the uniformity problem, because a bin contains approximately equally skilled users only so the standard deviation is perceived as negligible.

However, a problem this model introduces is the choosing of the appropriate number and represented intervals of the bins. A higher than optimal bin count would mean that the players are segmented too much, which would make the game creation a harder process and would result in longer queue times. A lower bin count would however defeat the purpose of the model with increased queue time and worse uniformity. We would also need more bins for more concurrent players and fewer bins for fewer.

Additionally, since we established that the players' skill ratings follow the normal distribution, uniformly distributed bins are not the best solution. Ideally we would establish more bins around the mean of the ratings and less with the outliers, making the bins contain approximately the same amount of players (illustrated in figure 7.5).

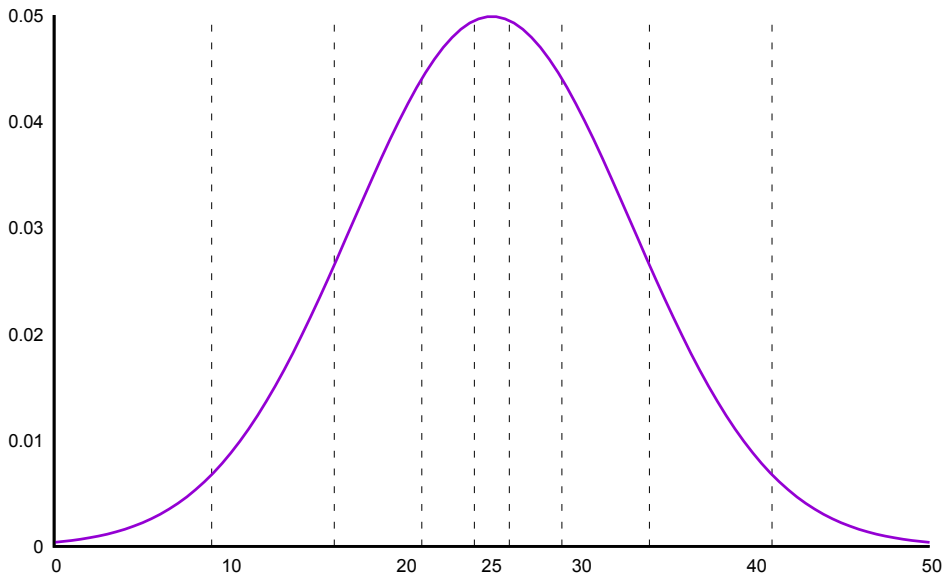

Figure 7.5: The distribution of player skills grouped into bins.

Because of this reasoning we will implement *adaptive bin counts*, which will mean that if the number of players in a queue exceeds some pre-set amount, we will cut the interval represented by the bin in half. The already existing bin will take the first half of the interval with the players belonging to it and we create a second bin, which will take the second half. Similarly, if the number of players in a bin is lower than a pre-set constant, we will merge the bin with one adjacent (the one which has fewer players in it).

7.3.4 Testing the algorithm

Playerbase generation

In order to test the algorithm presented, we will try to generate a realistic playerbase, which means that we have to generate the ratings (i.e. sample the μ and σ values). According to the central limit theorem, the player skill distribution will follow the normal distribution.

Before we go further, we must specify the default values we use for a new player. There are many options of course, but a general recommendation comes from the original TrueSkill paper, which uses mean $\mu = 25$ and standard deviation $\sigma = \frac{25}{3}$. In our thesis, we will be using these values as well but we leave the option to change them later if necessary.

According to Microsoft [44] since the ranking system uses a Gaussian belief distribution all mean skills will theoretically lie within ± 4 times the initial σ. In addition, empirical study showed a stronger constraint, where 99.99% of the rankings happen to be even within ± 3 times the initial standard deviation. Therefore we can sample player ranks from the truncated normal distribution $\mu \sim \mathcal{N}(25, \frac{25}{3})$ using rejection sampling (because the rejection rate is only 0.01%).

Additionally, we must sample the standard deviation. These values can be sampled from a uniform distribution and because the "uncertainty" of the skill cannot increase (because as time progresses we gather additional information

about the player regarding his skill and more information can only help us and not confuse us) we sample from $\sigma \sim \mathcal{U}(0, \frac{25}{3})$.

As a last step, we try to determine the average amount of players who are in queue. We must consider a lower count of players as well, recognizing that (without any major marketing strategies) at release our game will have a low number of concurrent users. However, keeping in mind that matchmaking algorithms are created for the purpose of handling a big player pool, we can set the lower bound to 1000.

To determine the higher bound, we can take a look at some of the more popular multiplayer games. According to SteamCharts [45] Dota2 had an all-time peak of concurrent users of $\sim 1.3 million$. If we consider the average game length to be 30 minutes, we estimate that at this peak there were about 40000 players in queue (this is only a very rough estimate of course as we do not consider logged in but passive players, players in lobbies or custom matches, etc.).

Matchmaking aspect weights

Now that we can sample our playerbase, we must also make a decision about the weights, which control the matchmaking aspects. In our tests, we will consider the match to be a decent game if the following conditions are met (note, that these values are subjective and can be modified to adjust to the size of player base, maximum allowed queue length, etc.):

- the fairness of the match is at most 0.5 (the middle point of the d function)
- the uniformity is at most 0.1 (we push for the players to be equally rated)
- we tolerate up to 120 seconds of queue time, during which we will gradually allow the values above to be higher - so we will set β to be $\frac{1}{8}$
- if the player count is not at the maximum allowed (10), we also tolerate a more imbalanced match in order for these players to get a game - we will set $\gamma = \frac{1}{8}$

We must also normalize the seconds and the player count to be in the same range as the fairness and uniformity and so we set $\beta = \frac{1}{60}\frac{1}{8}$ and $\gamma = \frac{1}{10}\frac{1}{8}$.

Taking the above points into consideration, we can now calculate the boundary below which we accept and create the match:

$$\lambda = \frac{1}{2}0.5 + 0.1 - \frac{1}{480}30 - \frac{1}{80}0 = 0.2875$$

Test

The experiments were run on a computer with the following specifics:

- Intel(R) Core(TM) i5-4200H CPU @ 2.8GHz (2 cores, 4 threads)
- DDR3, 8GB RAM
- OS: Windows 8.1

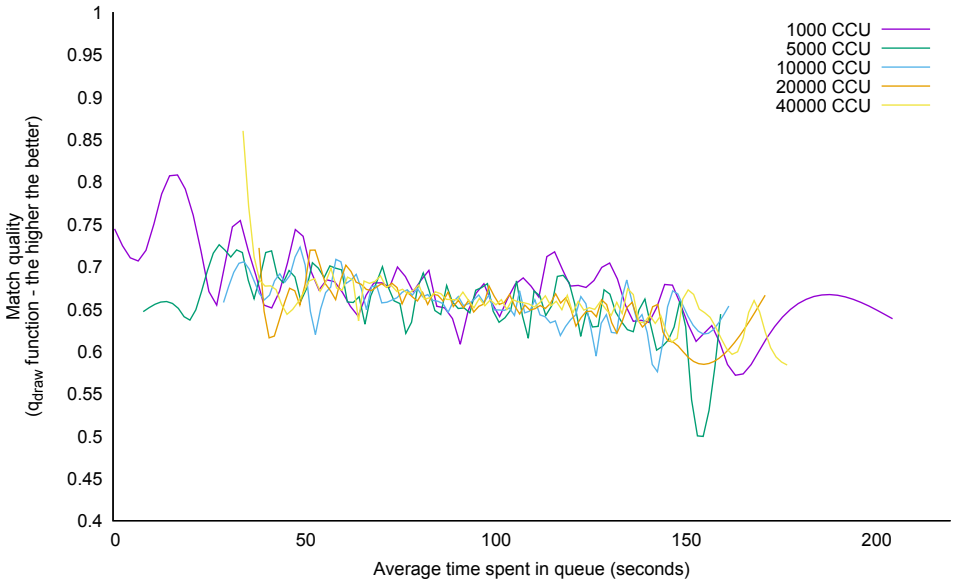

Figure 7.6: Multithreaded server results (trendlines) for different number of concurrent users (CCU)

Figure 7.6 shows us the results of the experiments and we can see that the our server can handle 40000 concurrent players as well as 1000. Additionally, the match quality decreases gradually with time, but no so drastically that it would have a significantly negative impact on the users' game.

Conclusion

We started this thesis by selecting the video gaming platforms, defining the design and characteristics of the game and of the world editor we wanted to implement. Then we discussed some details of the implementation. Subsequently, we selected the libraries and the game engine best suited for our needs, in which we implemented our application. We chose Unity3D as our game engine, which we briefly presented. We composed the map smoothing algorithm and we investigated the synchronization of the sprite images and found an optimal way to compare and compress these pictures. In the last chapter we discussed the responsibilities and operations of the MasterServer, we analysed the matchmaking algorithm and proposed a prototype and an improved multi-threaded model.

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 (the user and programming documentations are included as attachments).

Bibliography

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

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

[3] Jim Rossignol. Eurogamer's summer of pc plenty - twenty freeware games. *Eurogamer's Summer of PC Plenty*, 2006. [Accessed on: 2019-07-07] Accessible by: http://www.eurogamer.net/articles/a_20bestfreegames. Soldat ranked 11th.

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

[5] Best selling game franchises. [Accessed on: 2019-06-29] Accessible by: http://vgsales.wikia.com/wiki/Best_selling_game_franchises.

[6] Worms 2d weapons. http://worms2d.info/Weapons. Accessed: 2019-05-21.

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

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

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

[10] Arpad Elo. *The rating of chess players: Past and present*. Arco Publishing, 1978.

[11] Thore Graepel Ralf Herbrich, Tom Minka. Trueskilltm: A bayesian skill rating system. 2006. [Accessed on: 2019-07-17] Accessible by: https://www.microsoft.com/en-us/research/wp-content/uploads/2007/01/NIPS2006_0688.pdf.

[12] Dr. Mark E. Glickman. The glicko system. 1999. [Accessed on: 2019-06-02] Accessible by: http://www.glicko.net/glicko/glicko.pdf.

[13] Jeff Moser. Computing your skill. 2010. [Accessed on: 2019-07-10] Accessible by: https://github.com/moserware/Skills.

[14] Rankade, 2018. [Accessed on: 2019-05-23] Accessible by: https://rankade.com/ree.

[15] Yordan Zaykov Tom Minka, Ryan Cleven. Trueskill 2: An improved bayesian skill rating system. 2018. [Accessed on: 2019-07-16] Accessible by: https://www.microsoft.com/en-us/research/uploads/prod/2018/03/trueskill2.pdf.

[16] Unity homepage. https://unity.com.

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

[18] Duality homepage. https://www.duality2d.net.

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

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

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

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

[23] Matan Aspis. 6 top game engines in 2017. 2017. [Accessed on: 2019-07-01] Accessible by: http://www.discoversdk.com/blog/6-top-game-engines-in-2017.

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

[25] Xenko - supported platforms. https://doc.xenko.com/latest/en/manual/platforms/index.html.

[26] Waveengine 3.0 release announcement. https://geeks.ms/waveengineteam/2019/06/18/waveengine-3-0-preview/.

[27] Last update release of monogame engine. http://www.monogame.net/2017/03/01/monogame-3-6/.

[28] Don Glover. Unet deprecation faq. [Accessed on: 2019-07-05] Accessible by: https://support.unity3d.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ.

[29] Photon bolt pricing. https://www.photonengine.com/en-US/BOLT/pricing.

[30] Forge networking remastered. https://github.com/BeardedManStudios/ForgeNetworkingRemastered.

[31] Terrainify 2d in unity asset store. https://assetstore.unity.com/packages/tools/terrain/terrainify-2d-139257.

[32] Tiled map editor. https://www.mapeditor.org.

[33] Detailed implementation of the trueskill algorithm by jeff moser. https://github.com/moserware/Skills.

[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: 2019-07-15] 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: 2019-07-07] Accessible by: https://www.safaribooksonline.com/blog/2013/11/26/image-hashing-with-python/.

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

[43] Josh Alman and Dylan McKay. Theoretical foundations of team matchmaking. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, pages 1073–1081, Richland, SC, 2017. International Foundation for Autonomous Agents and Multiagent Systems.

[44] Microsoft Research. Trueskill ranking system description. [Accessed on 2019-07-04] Accessible by: https://www.microsoft.com/en-us/research/project/trueskill-ranking-system/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fprojects%2Ftrueskill%2Fcalculators.aspx.

[45] Concurrent player analysis on steam - dota2. https://steamcharts.com/app/570.

List of Figures

A. User documentation

This attachment serves as a general instruction manual for the player. We go through the course of the game starting with launching the game, gameplay and ending.

A.1 Launching the server

In order to allow the players to find matches, the game server must be launched (only one instance of the server is required for all games). To achieve this, navigate to the *MasterServer* folder and launch the application with the *MasterServer.exe* file. After the program starts up, the user can configure the network information (pressing *Enter* sets the default value) and the server is launched.

> Please make sure that your firewall does not interfere with the server. Either specify the application as an exception in your firewall settings, or turn the firewall off entirely (not recommended).

A.2 Installation

The application 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 (displayed in figure A.3).

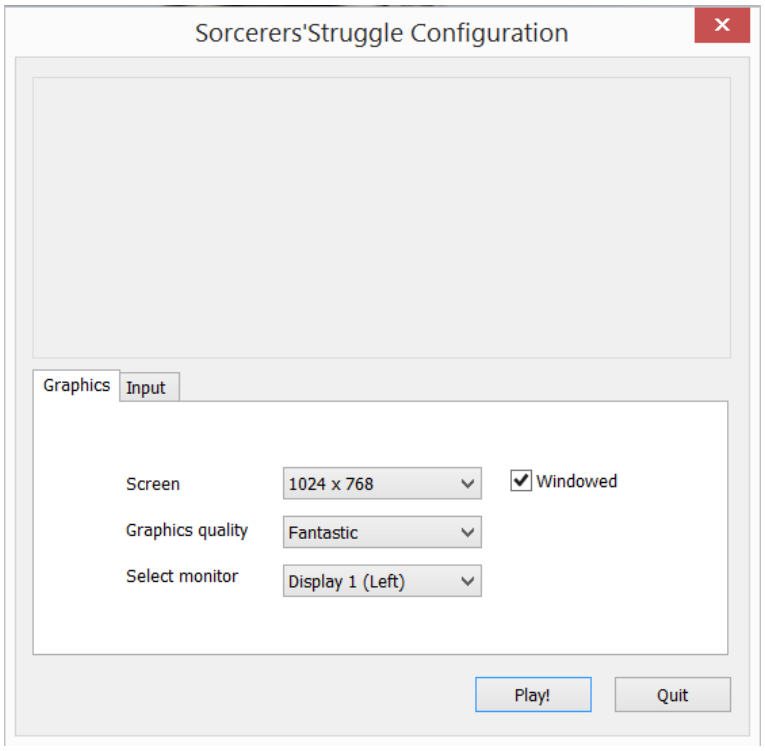

Figure A.1: Settings screen of the application.

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. With the help of the another tab called *Input*, the user can customize the controls (which are detailed later.)

After the *Play!* button is pressed, the game is launched.

A.3 Main menu - establishing the connection

After launch, the user is presented with the main menu (figure A.2).

Figure A.2: Main menu screen.

The first button (labelled *Play!*) takes the player to the play panel, which will be discussed in detail. The second button (labelled *Network Settings*) navigates to the settings panel, in which the user can modify the connection information about the server. The third button (labelled *Quit*) quits the application.

> Before continuing with the manual, please make sure that your firewall does not interfere with the application. Either specify the game as an exception in your firewall settings, or turn the firewall off entirely (not recommended).

There are 2 options a player can use to enter a game - the matchmaking option (where the player is teamed up with other player of similar skill) and the custom lobbies option (the unranked option, where teams can be chosen freely).

The play panel is displayed after pressing the *Play!* button (figure A.3). We will discuss each element's responsibility.

Figure A.3: Play panel of the game.

1. the **Back** button - takes the user back to the main menu panel
2. the **Quit** button - closes the application (after the choice was confirmed in a prompt window)
3. the **Start matchmaking** button - enters the player into the matchmaking queue (searching for a balanced ranked match).
4. the **Host option** region - this region serves as an intermediary for the player to create a new host (as a custom lobby for an unranked game). The user must enter the name of the game and optionally he can also submit a password (only users in possession of this password will be able to enter the lobby - of the field is left empty, the lobby is considered public and everyone can join). After inserting these values, with the help of the *Create server* button the host is created on the server and the host player entered the lobby.
5. the **Host list** field - the hosts currently available to be joined are listed here.
6. the **Refresh** button - updates the host list
7. **Ignore full servers** - by checking this field the host refresh mechanism will ignore and not show those hosts, which are already full
8. the **Join server** button - after a host has been selected from the host list (5), the player join the specific host with the click of this button (if the host is password protected, the user must submit the password). If the connection is successful, the client player enters the lobby.

If the host creation does not work, or if the host list does not show any hosts available (even though are registered on the server), make sure that the MasterServer is launched and its settings configuration (connection information) matches the ones specified in the game's settings.

A.4 Lobby room

The lobby room is illustrated in figure A.4.

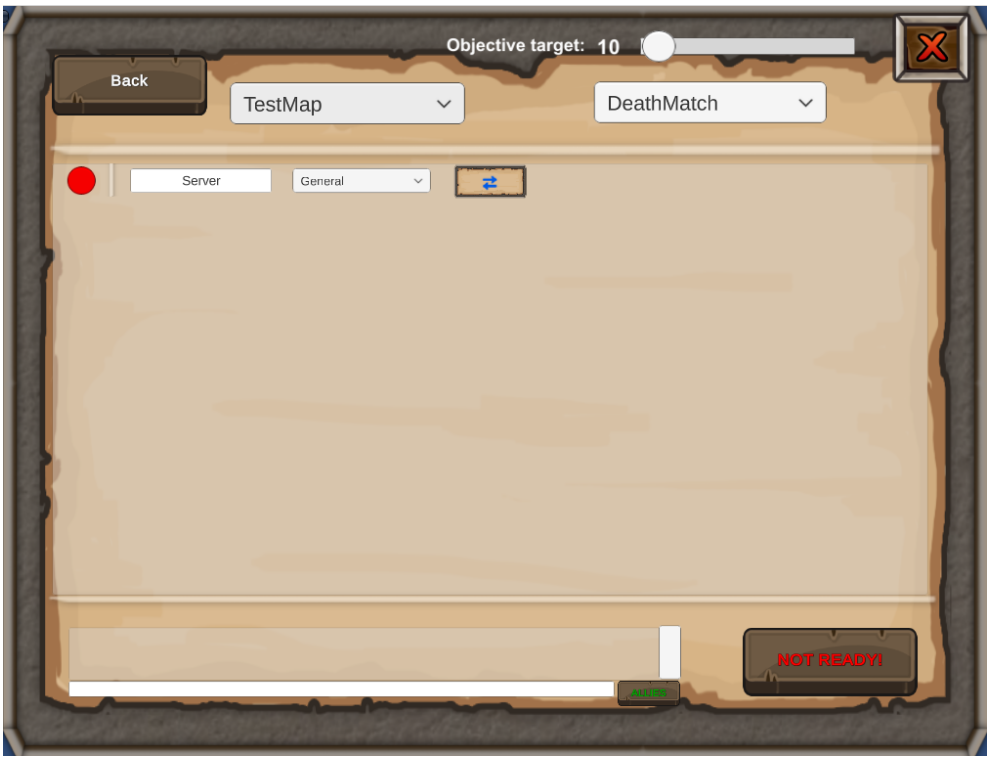

Figure A.4: The lobby room.

The panel consists of the game options (which only the host can modify) panel at the top, the player options and the chat window.

1. **Map selection** - the first of the game options, which the host can set and modify freely. The maps are read from the *StreamingAssets*
   Maps
   folder and the host can choose whichever map is available to him.

2. **Gamemode** - this property sets the style of the game and the objectives of it. The application offers 2 gamemodes: *(Team)Deathmatch* and *Capture The Flag*. The objective of the former is to eliminate other players while trying to die as few times as possible. A team earns a point if a player from the opposing side dies. The objective of the latter is to bring the enemy team's flag to your own base (or more specifically, the place where player's flag's base point is), while not losing your own flag to the enemy team (called "capturing the flag"). A points is earned for every flag captured.

3. **Objective target** - used to set the length of the game. When this objective target is reached (the objectives of the gamemodes are described above), the game is over with the winner being the team that reached it.

4. **Ready light** - signals if the player is ready to start the game (red if not, green if yes). The game can only be started (by the host user) if all players are ready.

5. **Player name** - the name of the player

6. **Character class selector** - the player can choose class of his character from this dropdown. Each class comes with a unique ability (spell), which can be used during the game. The classes are described below.

7. **Team selector** - the player can switch teams with a click on this button

8. the **Ready** button - used to set the player's state to ready (if the player changes his mind, the state can be changed back with the same button)

9. the **Chat** window - messages can be exchanged amongst the players in the lobby with the help of this window. The user can enter the message in the input field at the bottom, select the target audience for the message (can be sent to all players or just the current team he is a part of) and send the message by pressing the *Enter* button.

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 reload time of all weapons are halved (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 teammates by setting their health to 100 - additionally, if the player or 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

A.5 Game scene

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

Button	Action
W	Jump
A	Move left
D	Move right
Ctrl	Use shield
E	Use ultimate
1	Select weapon 1 (Rod)
2	Select weapon 2 (Staff)
3	Select weapon 3 (Dagger)
Mouse 0	Fire
Mouse 1	Flight
Movement of the mouse	Aim

The heads-up display is present in order to keep the user informed about the state of the game and of the character (illustrated in figure A.5).

Figure A.5: In game picture.

1. **The player object**
2. **Scoreboard** - keeps count of the points scored by the two teams (according to the objective - described above)
3. the **Quit** button - quits the application
4. **Player properties** - the sliders displayed here represent the following properties: the red **health bar**, which displays the current amount of health

points the player has - if it reaches zero, the player dies and is respawned after a short amount of time; the blue **shield bar**, which shows the time still available for the shield to be active - during the time the shield is shattered (the shield time was used up or the shield was destroyed by projectiles), the defensive mechanism is not available; the yellow **flight bar**, which shows the remaining time the player can fly for.

5. **Weapon availability** - the player possesses 3 different weapons: the first is the **Rod**, which is a chargeable weapon (where the damage dealt by the projectile is based upon the time it spent being charged by the player); the **Staff**, an automatic weapon (where there are multiple lower-damage projectiles fired in rapid succession); and the **Dagger**, which is a one-off weapon (which deals substantial amount of damage to other players, but has a longer reload time than other firearms). The selected weapon is highlighted with a frame around it. When a weapon is fired, a specific amount of time (reload time) must pass in order for it to be used again (shown by an overlay in the HUD).

6. **Spell** availability - displays the current state of the spell determined by the class of the character (described above) - if the icon is fully visible (not covered by a shade), the ultimate is ready to be activated.

There are 2 different types of neutrals (illustrated in figure A.6):

1. **Knight:** a game object that will move only horizontally and will damage 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. **Archer:** the neutral will spawn and launch a projectile in the upwards direction relative to its rotation. The projectiles are spawned after the specified time has passed.

Figure A.6: Neutral enemies in-game.

The goal of the game is determined by the gamemode and the target objective count, both of which are described above.

A.6 World Editor

A.6.1 Installation

Note that this application requires Unity3D(version 2018.3.7f1) to run.

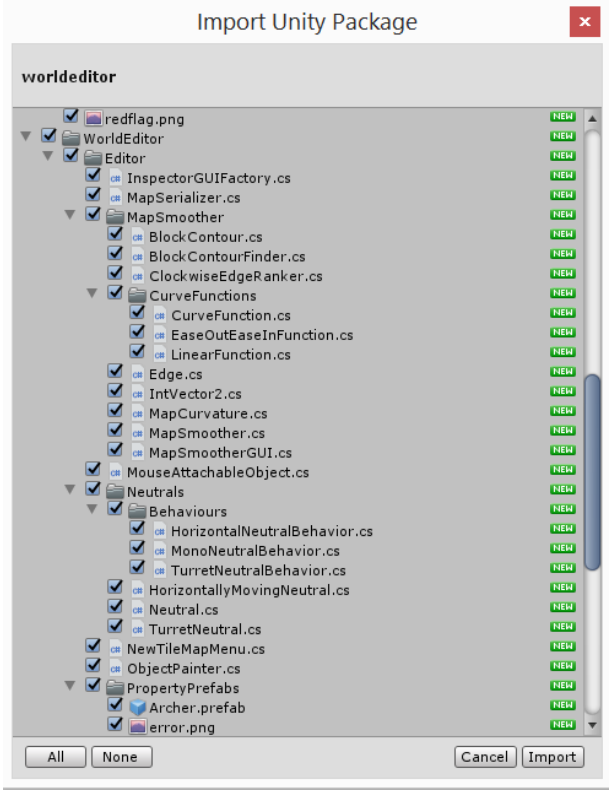

Figure A.7: Importing the world editor package to Unity.

The world editor is supplied as a Unity Package A.7. 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*.

A.6.2 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.

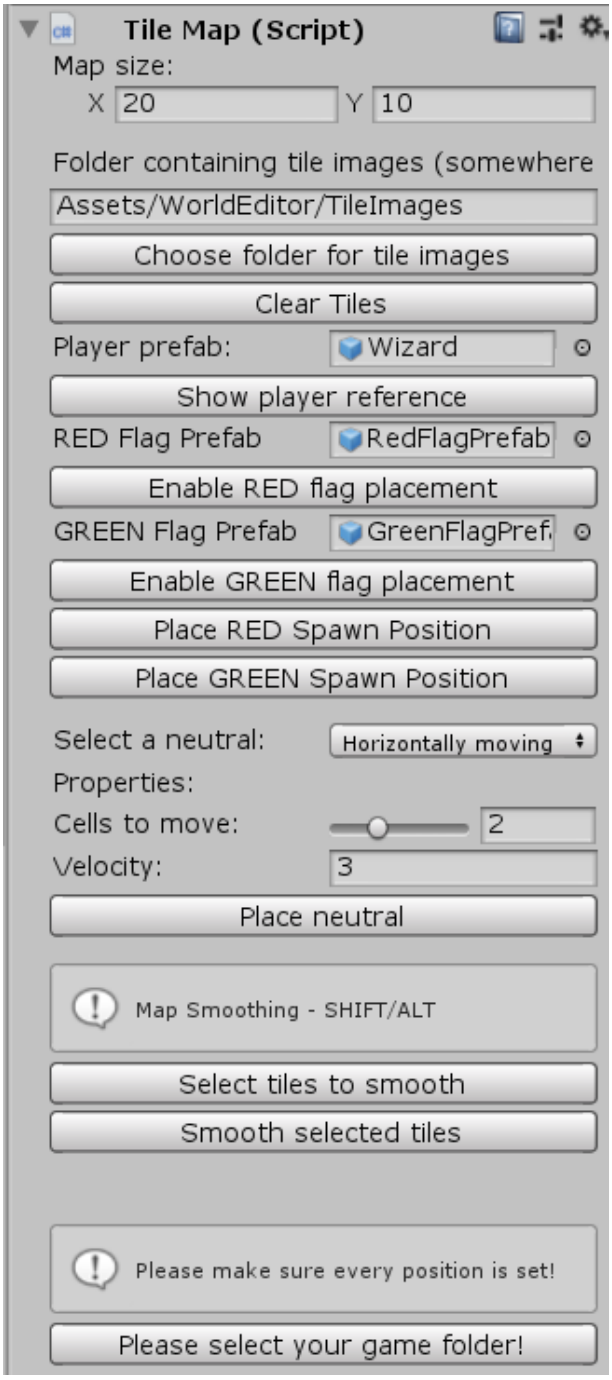

Figure A.8: WorldEditor inspector.

The settings of the map are shown in the mentioned Inspector (figure A.8). 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 map smoothing process. Using the button *Select tiles* triggers the function to select those tiles, around which the user wishes to create the polygon. After the object have been selected, the smoothing is carried out by clicking the *Smooth tiles* button.
- 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.

Figure A.9 illustrates a map currently being edited in the WorldEditor.

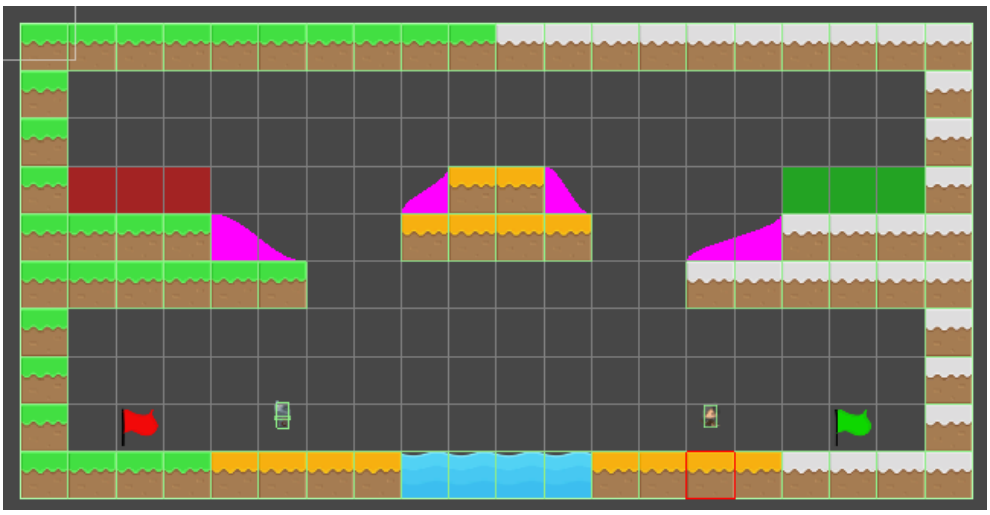

Figure A.9: Editing a level in WorldEditor.

When the user is currently not placing a property or a neutral (described below), 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).

B. Programming documentation

In chapter 4 of the thesis, we selected Unity as the game engine most suitable for our needs. Before we discuss the specifics of our application, we must look at two high-level concepts defining the structure of our implementation.

The main parts comprising a Unity application are called *Scenes*. Different scenes represent different environments and are therefore good for making the game more modular.

Additionally, we must also note the behaviour of Unity's *MonoBehaviour* class. The important feature this class supplies is that the descendants can be attached to Unity *GameObjects*. If a script is attached to an object, a few standard methods are called on them at initialization or periodically. An example of the latter is the Update() function which is called every frame and thus is essential for specifying the behaviour of the object. Via this method are the objects controlled, moved or updated.

Our application consists of 4 different scenes: *MainMenuScene*, *LobbyScene*, *GameScene* and the *WorldEditor* scene (which is not directly part of the game, which will be discussed later in this attachment). The movement between scenes is illustrated in figure B.1.

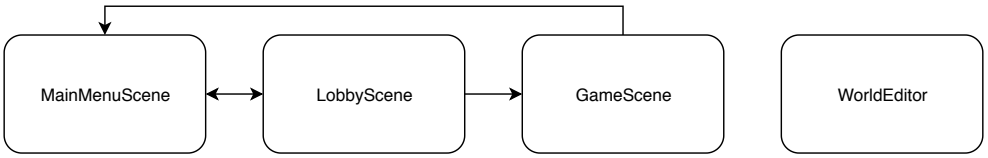

Figure B.1: High level overview of the scenes.

We will now detail each scene's responsibility in the application and we will mention and describe the scripts and prefabs used in each of them. The scripts are organized into folders depending on which scene utilizes them the most (the more frequent case is that the scene referred to by the directory's name will be the *only* scene to use them).

B.1 MainMenuScene

This is the initial scene presented when the application is launched. It has 3 different panels - the main menu panel, the settings panel and the play panel.

The first of these panels serves only as the introduction to the application typical of other video games. It presents the user with the options to play, modify settings or quit. The settings panel is also a regular component, which here can modify some default settings and connection information.

The more interesting panel is the play panel, whose structure is illustrated in figure B.2.

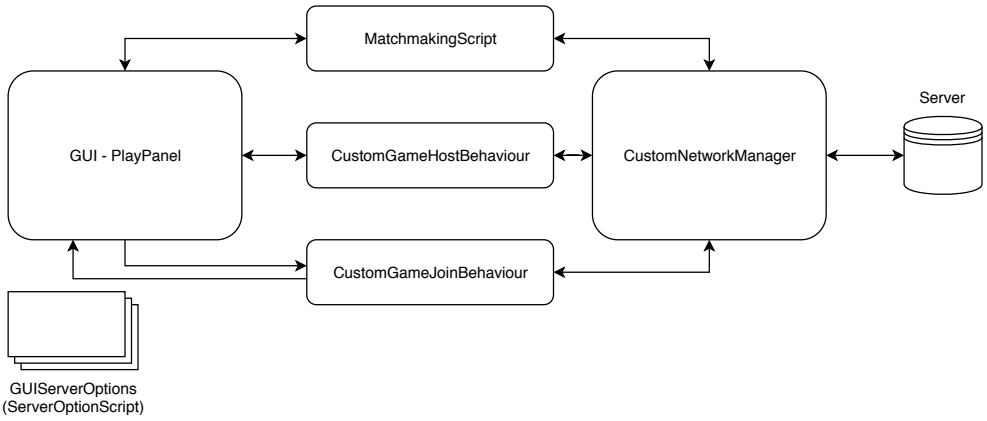

Figure B.2: Structure of the MainMenuScene scripts.

The 3 dominant scripts are:

- *MatchmakingScript* - enters the player into the matchmaking queue (note that the application benefits from matchmaking only when the number of players is sufficiently high).
- *CustomGameHostBehaviour* - serves as a bridge between the GUI and the *CustomNetworkManager* regarding hosting custom (unranked) games. Its main responsibility is to translate and forward the user chosen options for the server, but also does the TCP server initialization (*ForgeNetworking* uses the peer-to-peer model for networking, so one user is the server for the game) and requests a host to be registered on the MasterServer.
- *CustomGameJoinBehavior* - serves as a GUI bridge as well, but handles the client part of the connection process. The component performs the TCP client initialization and queries the list of hosts available on the server (through the *CustomNetworkManager*). When the response arrives and is handed over to this script, the unit processes the list of hosts by populating a scroll-down window with the host options from which the user can choose from. It also handles password inputs if necessary.

The server option objects are initialized from the prefab *GUIServerOption*. This prefab is equipped with the *ServerOptionScript* component, which handles potential property updates (like player count or gamemode) and the connection process to the given server if selected.

The class *CustomNetworkManager* server as an API between the application and the server. After requested from the above mentioned scripts it prepares the queries in the form of JSON objects which are then relayed to the server. It also handles the incoming messages which are then relayed to the scripts above.

After a game was created for the player or he decided to create or join a custom game, the application moves on to the *LobbyScene*.

B.2 LobbyScene

B.2.1 Forge Networking

Before we move onto the scene, we must first discuss a few of the networking principles of Forge.

In order for something to be networked, the game object must have a component derived from the *NetworkBehavior* class attached. This prefab is added to a list in the *CustomNetworkManager* object (which can then be used to spawn an instance of it), which naturally can not be updated dynamically during the application process and so only a networked object can be spawned and synchronized on all connected clients (figure B.3).

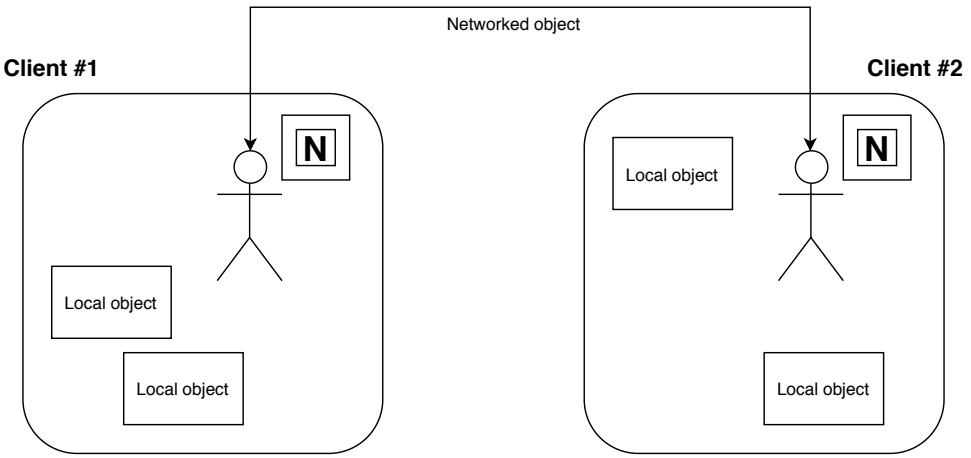

Figure B.3: Networking structure.

An object satisfying this condition can have fields and registered RPC methods to synchronize the behaviour of the object on all connected parties. The fields are useful for adjusting constantly changing properties (e.g. can be used to update the position of the object periodically), while the RPC calls are used for sporadic events (e.g. casting a spell - this happens once and then does not require constant adjusting).

To sum up, when a user wishes to change a property of his networked object, he modifies the property in the networked object which triggers an RPC call. This RPC call is relayed to all other parties (or it can targeted to be delivered only to one user) and the object instances on other machines react appropriately - usually by performing the same action as the caller.

We mark networked objects in our figures with an *N* symbol next to them.

B.2.2 Lobby rooms

The *LobbyScene* encompasses the staging area of the game, where the players can choose their teams (if the game is ranked then this option is disabled), classes and the host can set the gamemode, target objective count and the map.

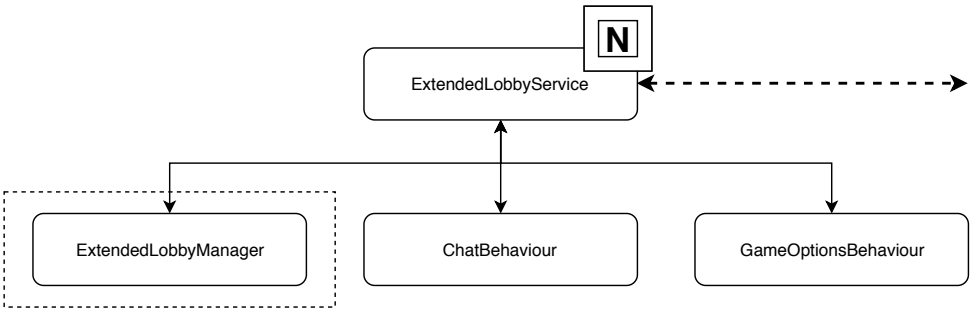

Figure B.4: High level architecture of the LobbyScene scripts.

The networked object serving as the intermediator for RPC calls is the *ExtendedLobbyService* component. This unit does the actual sending and receiving of network messages, which - if it is an incoming message - he sorts out and propagates to the interested party (figure B.4).

One of these parties is the *ChatBehaviour* script, which manages the GUI aspects of the chat system. This element reads in the local user's message and passes it to *ExtendedLobbyService* to distribute and receives other players' messages and displays them for the local user.

The second of the above mentioned parties is the *GameOptionsBehaviour*. This component handles the synchronization of the gamemode, the target objective count and the name of the map. It offers the host to change these settings freely (while other players can not interact with these GUI elements) and when he does, the unit notifies the *ExtendedLobbyService* to distribute it.

As the most important part of the scene, the component displays the list of joined players. This list of players is handled by the *ExtendedLobbyManager* component (illustrated in figure B.5).

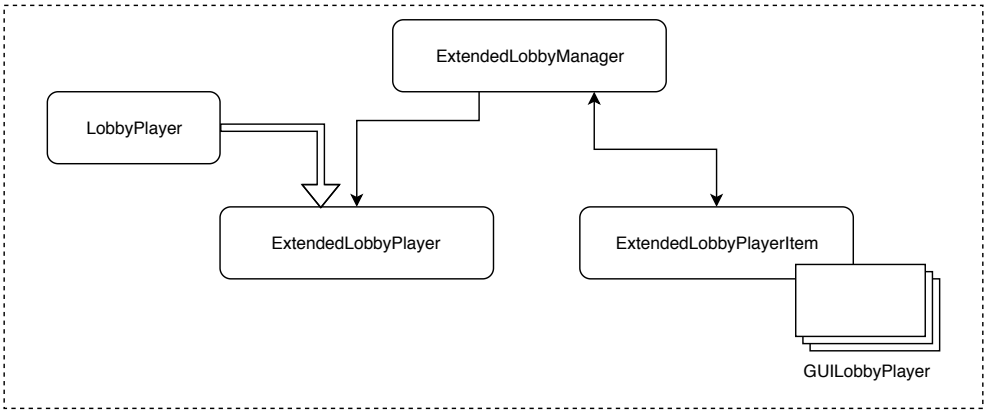

Figure B.5: Lobby player handling.

Every change regarding the properties of the players goes through this unit. Therefore, if the local player for example changes his team or class, the *ExtendedLobbyManager* registers the change and sends it to the *ExtendedLobbyService* to network the modification. On the other hand, if non local user changes a property and the *ExtendedLobbyService* receives the notification of the change via an RPC call, the *ExtendedLobbyService* calls the *ExtendedLobbyManager* unit to apply the change.

The alterations are executed in 2 places.

- One place is the *ExtendedLobbyPlayer* item, which contains the properties of the player (extended from the class *LobbyPlayer* which maintains the information about the name and the team - in the derived class we add the choice of character class and readiness information). The fields in this class are changed directly from the *ExtendedLobbyManager* class.
- The *ExtendedLobbyPlayerItem* class handles the GUI and therefore generates (if the local player made a change) or reacts to (if another player varied something which we need to display on our screen as well) the RPC call requests. The unit demonstrates the modifications on an instance of the prefab *GUILobbyPlayer*, which has a GUI element for showing each lobby player property.

We contain the general settings for various things not related to a specific game in the class *DefaultSettings*. This script holds the connection information to the MasterServer, the list of available gamemodes (however, to introduce another gamemode it is not enough to introduce a new class here, we must also add the new gamemode prefab to the *CustomNetworkManager* - because the gamemode object has to be networked - as discussed above), the list of available character classes (here it suffices adding a new class to the list, because the instantiation is done with the help of reflection), etc.

B.2.3 GameScene transition

When all players are ready and the host starts the game, we begin the transition from the *LobbyScene* to the *GameScene* (figure B.6).

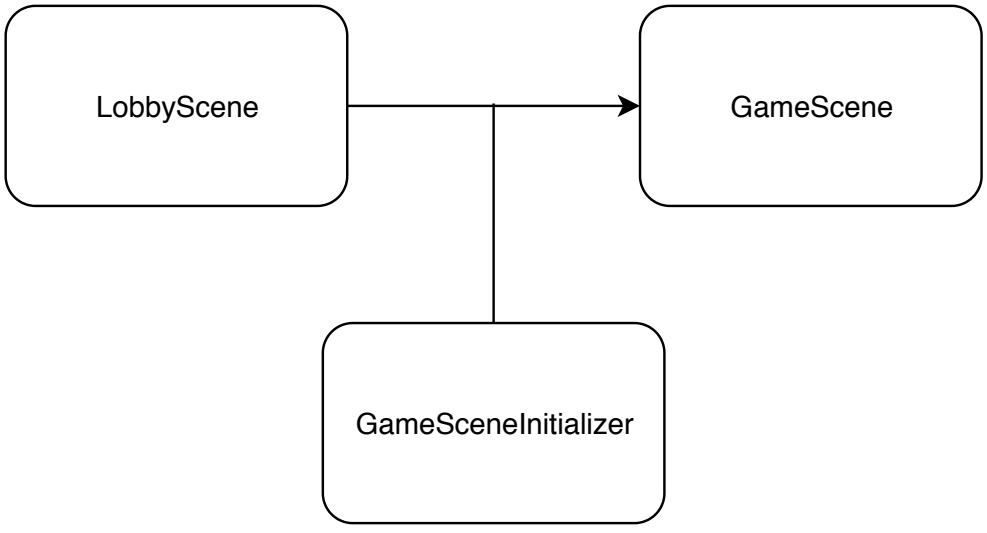

Figure B.6: GameScene transition.

When the application begins to load the *GameScene*, we start initializing the objects within. One of these is an empty gameobject, which has the *GameSceneInitializer* script attached. This script starts 3 important processes: the loading of the map, the spawn of the gamemode (both server-only activities) and the spawn

of the wizard (user object - server and client activity). The latter 2 objects are described in detail later.

The first of these processes is handled by the *MapLoader* class. This procedure consists of the deserialization of the selected map, reading the world properties (like the spawn positions of the teams and the positions of the flags), the spawning of the tiles, polygons and neutrals in the scene and the commencement of the sprite synchronization process as well. This latter operation is described in the thesis in detail, here we only specify the responsibilities of the affected classes (illustrated in figure B.7).

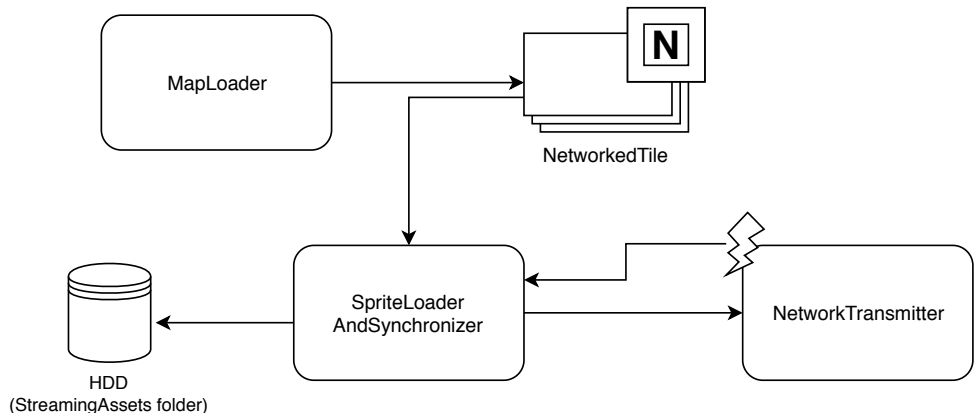

Figure B.7: Map loading structure.

The tile is a networked object created by the server holding the information about the sprite. When this tile appears on the side of the client, the *SpriteLoaderAndSynchronizer* class is requested to load the sprite from the disk or query it from the server. The file transfer is executed through the *NetworkTransmitter* class, which fires an event if a sprite arrived. The *SpriteLoaderAndSynchronizer* class reacts to this event by painting the corresponding tiles on the side of the client.

B.3 GameScene

The most important scene in the application is where the game is played. Here is where the user spends the majority of his time and where our game design and game flow are carried out. It also hosts the player's game objects which is naturally the most complex component in the application. In the next sections we detail the elements residing in this scene.

B.3.1 The player's game object

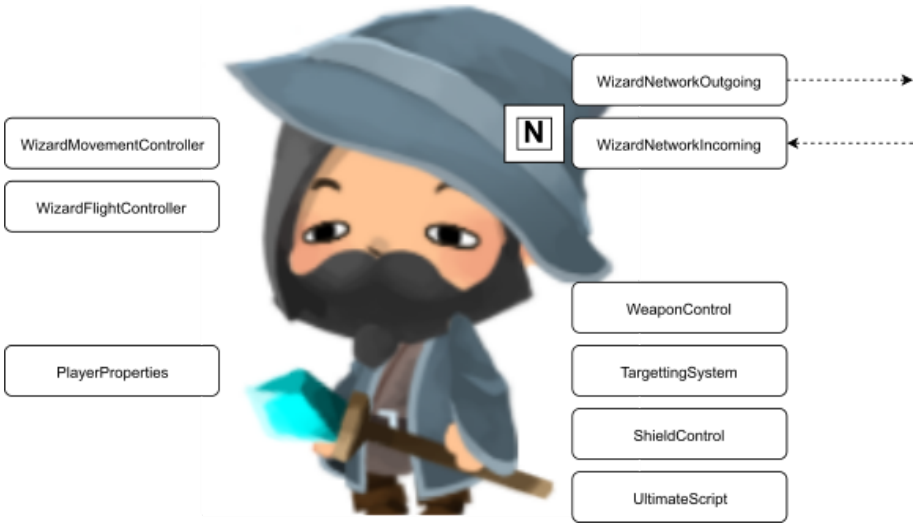

Figure B.8: Components attached to the game object.

The player's game object prefab is illustrated in figure B.8. Components inside the dash-lined rectangle are child classes of the *MonoBehaviour* class and are attached to the object as described above, therefore the Update() function is called for every one (which can monitor key presses and control command as well as update the properties of the object).

We have given each responsibility (as per the game design) to different components in order to make the structure more modular. Since we register the key strokes or mouse movement in all the attached components, we can separate each action for the responsible units.

We will now list the attached components and describe the importance of each (the relationship between them is illustrated in figure B.9).

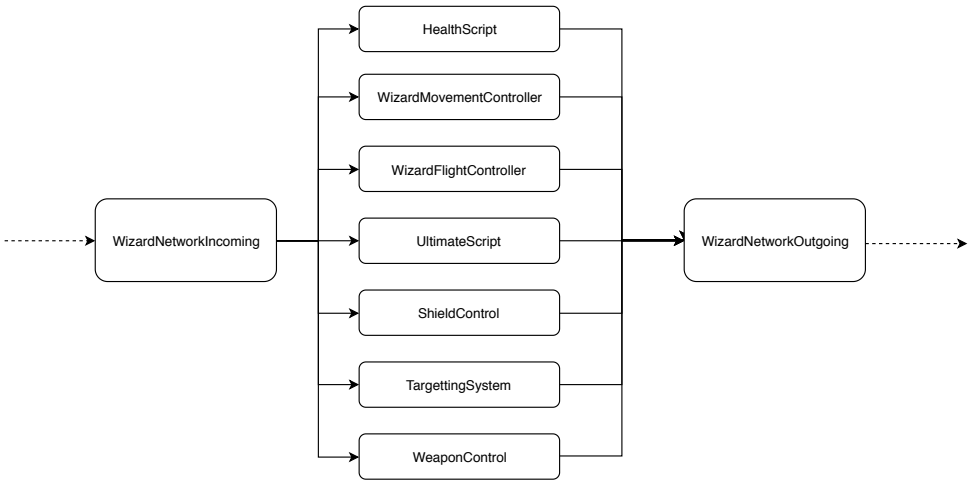

Figure B.9: Player object component networking.

- *WizardNetworkOutgoing* and *WizardNetworkIncoming* - these two classes are the networked components of the object and are thus responsible for

synchronizing the object userwide. The *WizardNetworkOutgoing* component receives requests from other attached units detailing the local player's actions and decisions and forwards them to other clients in the form of networked field updates and RPC calls. These incoming updates (on other users' machines) are then received and handled by the *WizardNetworkIncoming* component, which extract the parameters of the performed action and passes them on to the appropriate component, which then executes the update.

- *WizardMovementController* - executes the basic movement actions i.e. horizontal movement and jumping and networks the position of the object.

- *TargettingSystem* - adjusts the weapon in the wizard's hand so it always points towards the mouse's position on the screen. Networks this rotation so every player knows what the other is pointing towards.

- *PlayerProperties* - contains the characteristics of the player set up in the lobby, namely the name, network identification, team and character class. No networking is necessary here, because these values can not be changed during the game.

- *HealthScript* - handles all actions related to the health of the player (which it then networks). Deducts the health lost from the player's health points if the wizard took a hit and handles manages the player's death and - after the specified time - his respawn. Contains 4 events: one that fires when a wizard takes damage, two which fire when a wizard dies (one is when the local wizard dies, the second when any player does) and one that is executed when the player respawns.

The following components are all derived from the abstract class *BaseDeathBehaviour* (in addition the *WeaponControl* system is also a child class of this one, but it is described later in documentation). This unit contains 2 virtual methods, which are executed in the event of the player's death and respawn respectively (subscribed to the events from *HealthScript*). With the help of these methods can the other attached units specify some (component specific) behavioural logic in these events (e.g. when a player dies the *ShieldControl* component will turn off the player's shield if necessary).

- *WizardFlightController* - handles the flight logic (designed in the thesis), but networks only the the beginning and end of the flight animation (because the changing position is handled by the *WizardMovementController*). Turns off the flight in the event of death.

- *ShieldControl* - handles the shield mechanism (as designed in the thesis) and everything related (replenished by a teammate using a spell, adjusting the shield after taking a hit, etc.). Networks the size (width) of the shield.

The remaining 2 attached components we describe a bit more in detail.

UltimateScript

This component handles the castings of the spell. When the correct keystroke is registered, the unit launches an expanding circle originated from the player. We use Unity's *ParticleSystem* for the effects around the circle (the properties of the particles are specified and are different in each character class). When the circle collides with a wizard, the effect of the ultimate is applied on the player (if applicable - we make sure that the spell is intended for the collided player, e.g. beneficiary spells are only applied to teammates) and is removed after a given amount of time. This collision is handled by the *UltimateCollisionScript*, which is attached to the *ParticleSystem* component. It is also networks the casting of the ultimate, so every connected client can see the spell.

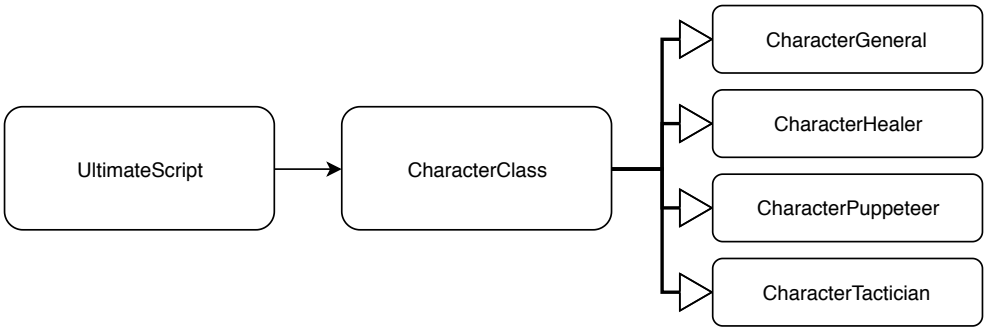

Figure B.10: The character class structure.

The spell which we cast is defined by the class of the character. All classes are children of the abstract class *CharacterClass* (as seen in figure B.10). This parent class defines the properties and abstract methods which the children then implement in the corresponding classes (such as the properties of the particles, cooldown time, ultimate effect lasting time and whether a supplied wizard is effected by the spell). Additionally, we also specify 2 events: one which is fired when the spell is cast and thus the cooldown has started; and the other when the cooldown has finished.

WeaponControl

This component handles the offensive capabilities in the game. Its responsibility is to recognize the keystrokes signalling the beginning of a firing process and the process of switching weapons.

The component contains a list of available weapons the player can choose from (this list can be extended by creating a weapon prefab and adding it to the available weapon prefabs list in addition with specifying an input key to select the new weapon and creating a HUD component). When the player intends to fire and presses down the appropriate key, the *WeaponControl* system executes the procedure according to the current weapon equipped.

This intent to fire is then networked, but the actual projectile is instantiated locally for every client (this is a major optimization technique, because after the projectile was fired, the trajectory of it cannot be affected by the players and so no networking is required). Additionally, the player switch event is networked as

well (so every player can see what weapon the other has selected) and there is also a recoil effect (which the *WizardMovementController* networks for us).

The component also contains an event which is fired when a weapon is switched (used mostly by the HUD).

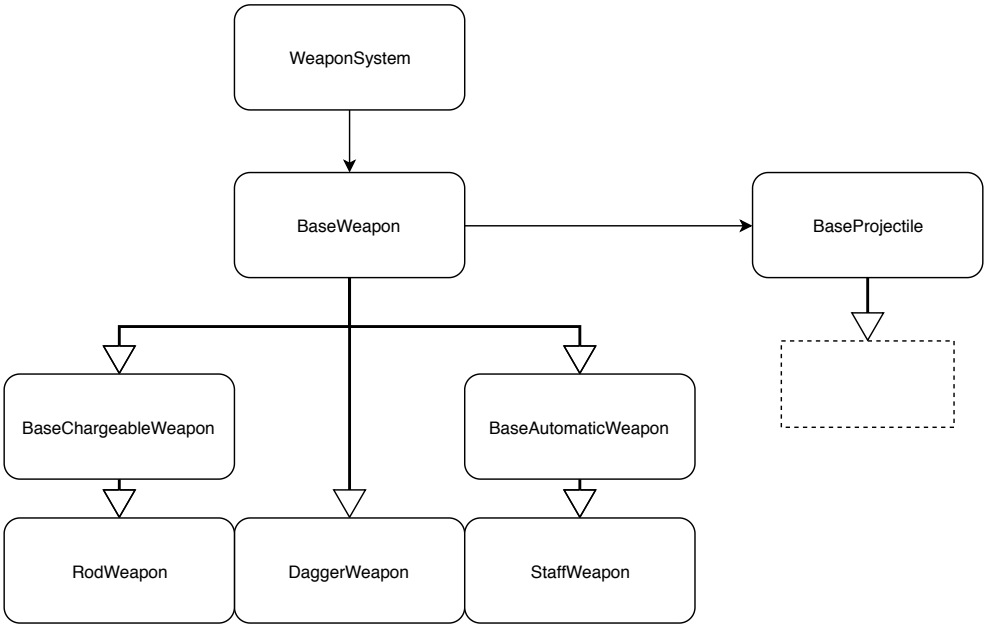

Figure B.11: Weapon system structure.

Every weapon has a common parent class called *BaseWeapon*, just as every projectile has *BaseProjectile* as a parent (displayed in figure B.11). These classes specify the basic mechanisms common for all weapons and projectiles, such as the process of reloading in the case of the former or the movement and collision control in the case of the latter. They also define virtual methods which the children can then implement specifically for their behaviour. Additionally, the *BaseWeapon* class contains 2 events, one fired when a weapon reload has started and one for when it is finished (used mostly by the HUD).

The main abstract classes derived from the *BaseWeapon* class are the *BaseChargeableWeapon* and *BaseAutomaticWeapon* which implement the techniques defined by their names. The former class allows the player to hold the instantiated projectile fixed with their weapon and charge the shot in order to deal more damage to an opponent (this is then realized by the *RodWeapon*). The latter class defines a weapon which can instantiate and fire projectiles in quick succession (in the game this is used by the weapon called *Staff*). One additional weapon is derived directly from the *BaseWeapon* class called *DaggerWeapon*, which does not require a more complex logic, but is intended as a one-off powerful projectile with a long cooldown.

The children projectile classes do not add much to the common *BaseProjectile* implementation in our case, because their properties are defined by the corresponding weapons.

The last projectile defined is the one not instantiated by a weapon and is called the *ArrowProjectile*. This class is used by the Archer neutral (described later).

B.3.2 NeutralBehaviours

The other active and moving game objects in the scene are the neutral enemies. The positions and properties of these neutrals are serialized in the map description, so during the deserialization we can initialize the specified neutral in the *WorldEditor*.

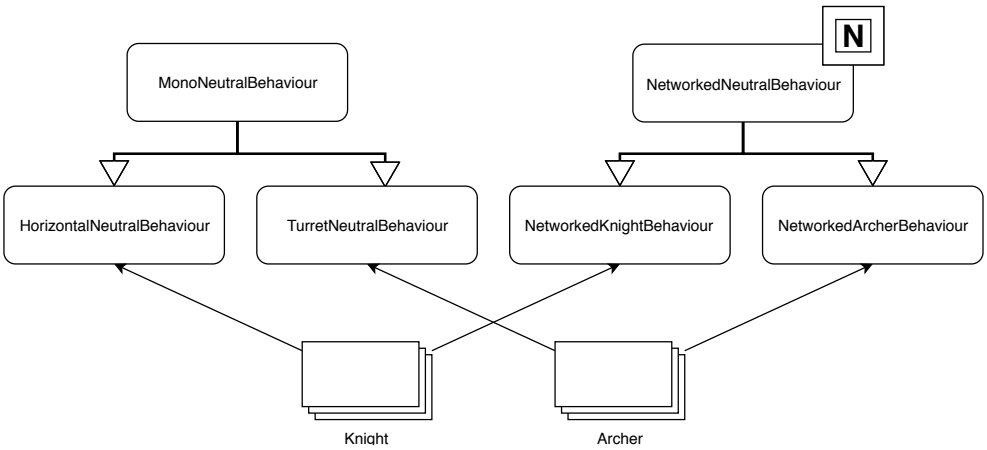

Figure B.12: Neutral structure in the game.

To each neutral object there is a component attached derived from the abstract base class *MonoNeutralBehaviour* and one child class of the class *NetworkedNeutralBehaviour* (as seen in figure B.12). The derived classes from the latter take care of the networking of the neutral (the neutrals are host owned objects). They can update the position of the object user-wide or can signal an attack via an RPC call.

The *MonoNeutralBehaviour* abstract class is a common ancestor of behaviour classes which describe the local processes. In this parent class we define the serialization and deserialization functions and outline the virtual methods the child classes can use to convey specific information about the objects. Additionally, these child classes define the movement or attack logic of the neutral.

B.3.3 Gamemodes

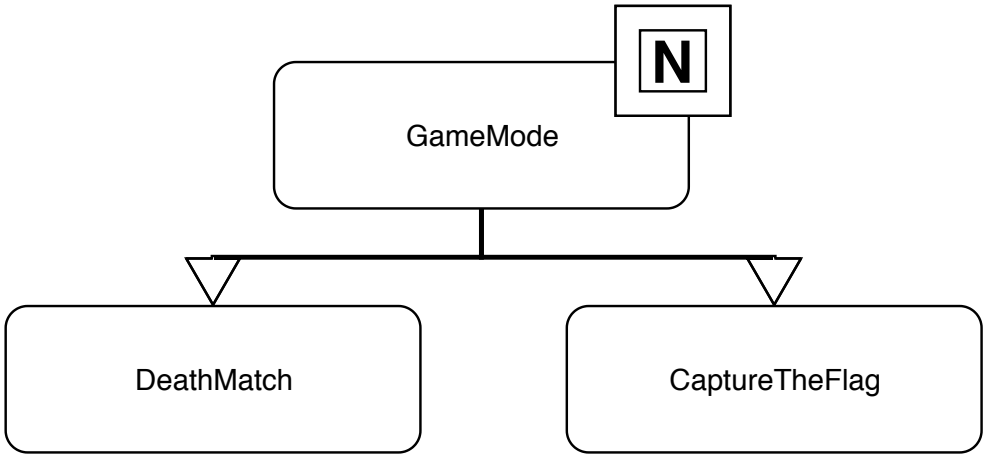

Figure B.13: Gamemode structure.

The structure of the gamemode classes are illustrated in figure B.13.

The *GameMode* abstract base class is a networked game object. This class implements the logic of the process when a team scores a point. This event is then networked and also fired (so the HUD can update the team scores). Additionally, it also handles the removal of game objects belonging to disconnected players and monitors the game state (according to the score) whether the target objective count is reached and therefore whether the game is over.

The 2 deriving subclasses each represent a gamemode. The *Deathmatch* class implements the game logic by subscribing to the event of every player's death and increasing the score of the opposing team. The *CaptureTheFlag* class follows the behaviour of the 2 flag objects and increase the score if one is captured.

B.3.4 Heads-up display

In order to inform the player about the game state and the properties of his character, we also implement a heads-up display (or HUD). The updates in the HUD are triggered solely by events and field updates so our game preserves a more modular structure.

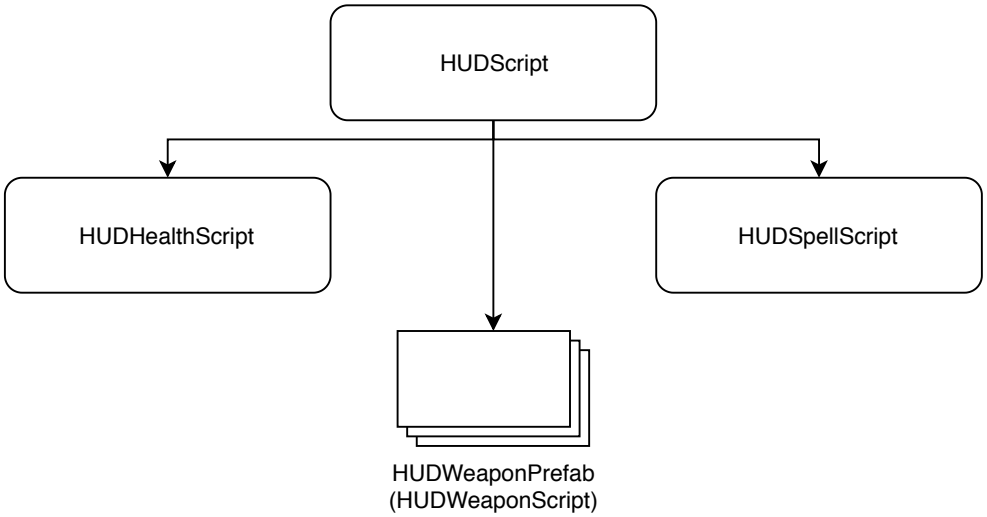

Figure B.14: HUD structure of the game.

The structure of the scripts handling the HUD is illustrated in figure B.14. The main script attached to the HUD object is the *HUDScript*, which handles the game state panel and local wizard properties like the shield or flight time remaining and initializes the other scripts presented.

The *HUDHealthScript* handles the health bar of the local wizard as well as the reactions to the player's death and respawn. The *HUDSpellScript* component reacts to the usage of the character specific spells and displays the cooldown process. The *HUDWeaponScript* attached to a *HUDWeapon* instance performs the reactions to the weapon switch and reload events.

B.4 WorldEditor

The world editor is built on the *TileMap* component. When the gameobject with the *TileMap* component gets selected in the hierarchy, the application draws a grid in the scene view. This is done by utilizing the virtual method *OnDrawGizmosSelected()* and iterating through every cell in the grid to draw it as a two-dimensional wire cube. This script will also store a few important references for us, like the size of the map, or the parent gameobject of every tile and the one of every neutral.

The editor scripting is accomplished by deriving from the *Editor* class. This is achieved in the *TileMapEditor* script. This will be our main class for defining the *Inspector* and therefore almost every function in the world editor.

One of the main functions is the drawing of the tiles. To accomplish this, we created a class titled *TileBrush*, which is a square object with the same size as the cells of the grid. The only things we need to successfully initialize the brush is the sprite of the selected tile (this is set in the *SpriteRenderer* component where we can change the sprite property at runtime) and the position of the mouse in order to move it along in the scene.

Note, that we will need to snap the tile to the grid. We can easily do this by dividing the current mouse position with the size of the cells and we get the line and column coordinate of the cell. By multiplying this with the cell size, we get the position of the upper left corner of the cell. We can use these numbers to place the tile in the correct position. This function executing the snap to grid function is used during the placing of the neutrals and properties as well.

The *TileMapEditor*'s *OnInspectorGUI()* function is the one responsible for drawing the Inspector. We can use the *EditorGUILayout* class, which offers us the GUI elements we need to design the world editor. We will have a separate class called *InspectorGUIFactory* to take care of these drawings for us. In this class we will also define every type of property and neutral as well, as to easily call their GUI functions together in one place.

Another important class is the abstract *ObjectPainter*. This class will represent the grid as a data structure and will store the references of the placed objects. We derive two classes from it, the *TilePainter*, which will remember the placed tiles and the *PropertyNeutralPainter*, which will store the properties and neutrals placed in the grid (figure B.15).

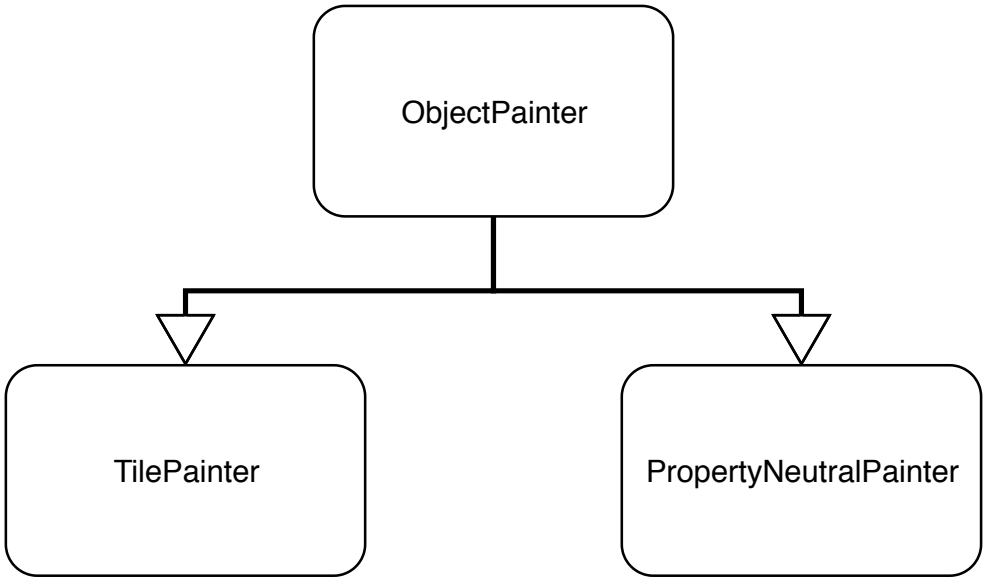

Figure B.15: ObjectPainter structure.

Although we will have to make sure that we correctly reference every object in the scene from the grid, this data structure does have benefits. It will make it easier to remove those objects that are left out of the new grid, if the user chooses to resize the map after having placed some objects. It will also make the rewriting easier (for example for the tiles we only change the sprite of the object and avoid the overhead of destroying one and creating another gameobject) and it will help us control the validity of a lamented placement.

In order to select the tile image for the *TileBrush* (which we want to draw with), we needed to implement a window, which will show us the tile images available. This window is the *TilePickerWindow*, which draws the tiles with the help of *GUI.DrawTexture*. It also uses two helper classes: the *TileHighlighter*, which is a semi-transparent dark-blue *GUI.Box* drawn around the texture which is currently selected; and the static class called *TilePickerWindowScroller* (figure B.16). This places a vertical scoller in the window in order to accumulate more pictures.

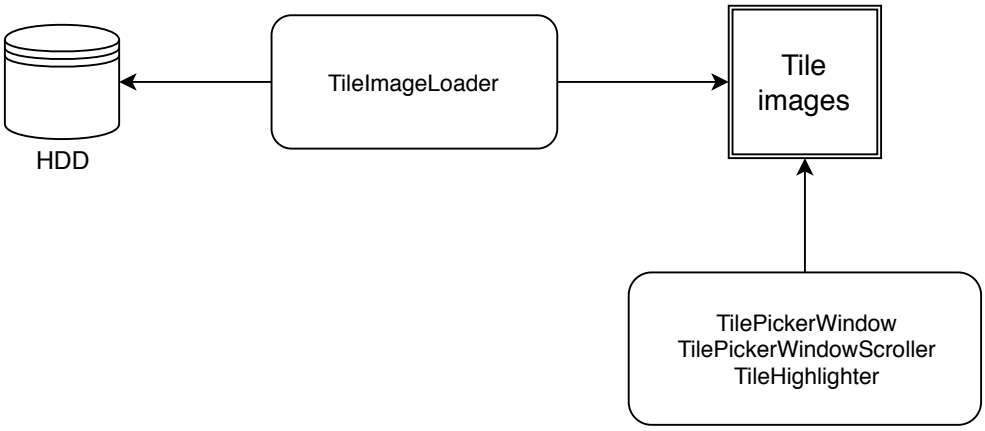

Figure B.16: Tile image loader window.

In order for this window to work, we need a method to find and import the

tile images available in the folder. This task is achieved by the *TileImageLoader* class. It is important that we recognize any changes made to the contents of the folder, but it is also important that we do not import every image every time the GUI of the window is redrawn. For this, we use a *Hashset¡Sprite¿* data structure, which remembers those sprites that have been already imported. We chose this structure because of the property of the *Add()* and *Contains()* functions, which are both $\mathcal{O}(1)$ operations.

The base class that every placable object (and the wizard size reference) derives from is the *MouseAttachableObject* (figure B.17). This class defines a behaviour, where the object has an enable placing button in the *Inspector*, which when clicked, creates an object instance by cloning the specified prefab and attaches that instance to the mouse (meaning that it will follow the movement of it).

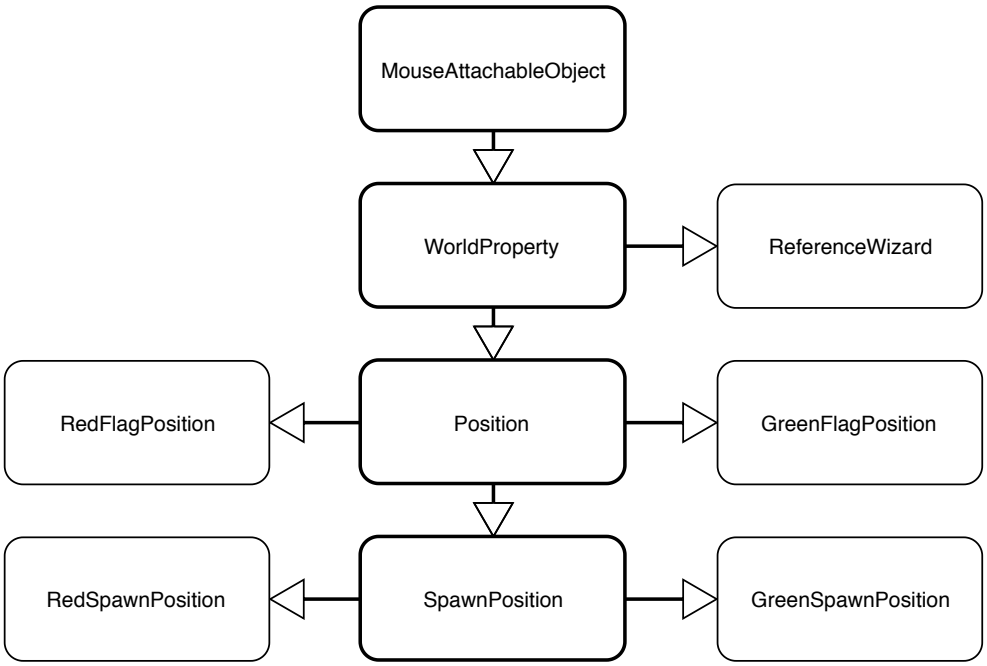

Figure B.17: Property structure in the editor.

The *FlagPosition* and *SpawnPosition* classes both derive from this class. The *Position* abstract class adds the functionality of placing and the restrictment, that only one instance of the given object may be available. This means, that if the user has already placed the red flag and wants to place the same flag, instead of instantiating a new object, the previous one snaps out of position and is attached to the mouse.

The *SpawnPosition* abstract class adds the methods for generating the prefab (which is a three tile wide object, where the position of the middle one will be the position of the first player in the team), rather than importing it from the Asset folder.

The *MapSerializer* has to work the same way as the *MapLoader* does, but in the opposite direction.

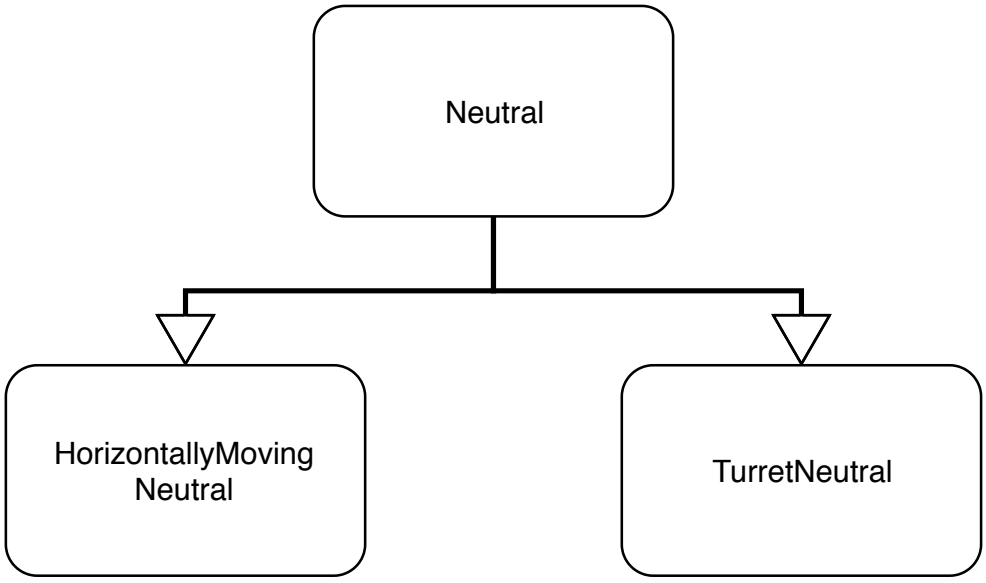

Figure B.18: Structure of neutrals in the editor.

The last not mentioned classes are *Editor Neutral* classes (figure B.18). Every type of neutral derives from the *Neutral* base class where a common logic is defined: the rotation customization of those neutrals that allow this and the actual neutral placing function. Because of this common base, every type of neutral has to declare a *ApplyCurrentPropertiesToObjectInstance()* method, which is called after placing the neutral. This method transfers the settings adjusted by the user to the actual neutral placed in the scene.

The implementation of the *MapSmoother* process closely follows the algorithm discussed in the thesis (all corresponding scripts are grouped in the *MapSmoother* directory). When the algorithm is finished calculating the points of the polygon, the *MeshCreator* class creates the object itself using the points specified.

B.5 MasterServer

The structure of the *MasterServer* is described in the thesis in detail, here we only specify the responsibilities of the affected classes and a few implementation details.

The entry point of the program is the Program class where we can specify the properties of the server and which registers the interactions as well (e.g. restart the server, list logging on/off, etc.). The class handling the incoming and outgoing messages is the MasterServer class, which is therefore characterized as the API (additionally, it also holds the list of custom unranked lobbies).

If the player wishes to join the matchmaking queue, his rank is queried from the *OngoingRankedMatches* class (if this is the first time the player wishes to play a ranked match, a new rating - an instance of the class Rating - is generated for him using the default values). This rating and the *NetworkingPlayer* instance (which holds the connection information of the player) is wrapped into a *WaitingPlayer* class instance (which has the additional property of measuring the time spent in queue) and the player is forwarded to the *PlayerSorter* class.

The *PlayerSorter* class determines the *MatchmakingBin* to which the player belongs according to his rank and enqueues him into it. The player is left waiting in queue until a *MatchmakingWorker* instance dequeues him and considers him for a game.

The *MatchmakingWorker* is a direct implementation of the unit described in the thesis. It tries to maximize match quality by considering every possible team composition, where each team consists of 5 players (we also implement the possibility of allowing imbalanced matches where one team is made up of only 1, 2, 3 or 4 players, but as a default setting we do not use this). If the h function is less than λ, then the match is created (if the match is cancelled, the players are put back into the queue). The game is added to the list of ongoing matches in the *OngoingRankedMatches* class (represented as a dictionary, where the *NetworkingPlayer* component of the server is used as a key) and the game information is sent to the players.

When the game is finished, the server reports back the result of the game and the ranks are updated by the *TwoTeamTrueSkillCalculator* defined the supplemented library.

The tests for the server performed for the thesis are implemented in the *MasterServerTests* project.