UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Richardas Keršis

Player emotional behavior dependency on fair video game design factors and external conditions

Bachelor's Thesis (9 ECTS)

Supervisor: Margus Luik, MSc

Tartu 2018

Player emotional behavior dependency on fair video game design factors and external conditions

Abstract:

The current thesis describes the problem of anger and aggression in video games and proposes a solution specifically for action video games where players oppose each other. Losing at those games may cause players to become angry or aggressive. As one of the factors, such anger comes from a feeling of being cheated on.

During the thesis, an action genre video game that provides equal opportunities for the opposing sides for winning was developed. An experiment which consists of playing the developed game and filling out the questionnaire by a specific group of people was conducted. Participants of the experiment were treated equally regarding game-external conditions. It turned out that fair video game design factors and external conditions have a positive influence on player emotions.

Keywords:

PvP (Player versus Player) action video game, fair gameplay, player emotions.

CERCS:

P170 (Computer science, numerical analysis, systems, control)

Mängija emotsioonide sõltuvus ausatest mängu disaini faktoritest ja välistingimustest

Lühikokkuvõte:

Käesolev bakalaureusetöö kirjeldab seda, miks videomängude mängijad muutuvad mõnikord mängides agressiivseks ja vihaseks ning pakub sellele probleemile lahendust *action* žanri videomängude puhul, kus mängijad mängivad üksteise vastu. Kaotamine nendes mängudes võib muuta mängijaid vihasteks ja agressiivseteks, mille üheks põhjuseks on mängijate kahtlus, et neid ei kohelda võrdselt ja teine võidab pettusega või talle mängu poolt loodud eelissituatsiooni abil.

Töös loodi *action* žanri katseline videomäng, mis pakub võrdseid võimalusi võitmiseks mõlemale poolele. Töö käigus korraldati uuring, kus katseisikud mängisid seda mängu ja pärast täitsid uuringu ankeedi. Selgus, et ausad mängu disaini faktorid ja välistingimused mõjuvad positiivselt mängijate emotsioonidele.

Võtmesõnad:

PvP (Mängija vastu Mängija) action žanri videomäng, aus mängu protsess, mängija emotsioonid.

CERCS:

P170 (Arvutiteadus, arvanalüüs, süsteemid, kontroll)

Contents

Introduction

With today's progression of electronics and communications more and more people are getting involved in online video gaming as computers and Internet access are becoming easier to acquire. In the year 2016 there were 1.8 billion gamers in the world [1]. Which was roughly 25% of the world population at that time [2]. More and more games get digitized and there are almost no real-life games left that are not yet digitally available. Nowadays video gaming industry has got lots of different game types, game genres and their realizations – ranging from educational digital board games to virtual reality action video games. Playing games is essential for human beings as they help us create relationships, get educated, learn to work together, get to know how it feels to win and lose around each other and more [3].

According to the statistics, the best-selling and most played are action video games [4, 5]. This is the reason the action genre is focused during the thesis. People play games for various reasons [6, 7]. Current thesis considers two of them: competition and achievements. Competing against other people and the competition itself means "the activity or condition of striving to gain or win something by defeating or establishing superiority over others [8]." In other words, someone wins and someone loses. However, despite all the good, action games bring one major problem.

Most PvP (Player versus Player) video games that provide competition have some sort of inequality or imbalance and there are many factors that can potentially cause this effect. Be it different starting positions or simply who has got the first turn. These factors tend to raise questions among the players like "is the game fair?", "do I have the same opportunity to win as my opponent?" or "do I even have a chance to win?". They may or may not be asked consciously meaning that players may doubt in a game's fairness even without thinking about it.

Looking at the players who treat action games seriously and sometimes get angry when losing at a game, we can see that they mostly tend to behave like that because consciously or subconsciously they think that the game was unfair to them [9-12]. As a result of their defeat, they might behave aggressively, break things and scream blaming someone or something else for their loss [9-12]. Some people claim that this is due to excessive violence in those games [13]. However, a recent research states that "it is not violence in games that makes people aggressive, it's their incompetence [14]."

Which brings us to the next set of questions. What if there is nothing to blame about a video game? What if the players knew that the only deciding factor of a game's outcome is their skill? Would players then not get angry or aggressive and gracefully accept their defeats? Or maybe it would only enhance negative emotions in people knowing they are alone to blame?

Thereby, the following hypothesis was formulated:

"Players would not feel anger if they knew that a played PvP action game was fair".

The purpose of this thesis is to give clear answers to those questions and the hypothesis by conducting an experiment. A specific group of people (further: research participants) have played a fair fast-paced action PvP video game with equal starting conditions. After that, research participants were interviewed, and their answers were analyzed. To ensure that the played game provides fair gameplay the decision was made to develop a simple action game. “Game fairness” and “fair gameplay” notions for the current thesis are described and discussed at the beginning of the next chapter.

The first chapter of this thesis describes the background of the conducted research in detail. In the second chapter, detailed game description and mechanics, as well as parts of the program code are presented. The third chapter provides information about the conduction of the experiment and the playtests. Finally, research results and conclusions are made and discussed.

1 Background of the conducted research

To approve or disprove the hypothesis and give clear answers to the raised questions, a game that provides fair gameplay was played by the research participants and their reactions were examined.

The target group of researched participants consisted of people who are acquainted with action genre video games to some extent. Moreover, they were placed in equal game-external conditions for the experiment to be as fair as possible. Those conditions include latency, computer power, and peripherals which are described in detail later in the thesis. To ensure that the played game was fair, a decision was made to develop a simple fast-paced action PvP video game called Swordshot. Description of the game is presented after the definition of "game fairness" and "fair gameplay" notions.

1.1 Game fairness and fair gameplay regarding Player versus Player games

As far as the definition of the word fair goes ("Treating people equally without favoritism or discrimination [15]"), this remains true when describing "fair gameplay" notion in this thesis. For a PvP game to provide "fair gameplay" it has to offer equal opportunities to the opposing sides for winning. Note: this does not mean that opponents have equal chances of winning, as their skill level might differ.

The chess game is considered as a good example of an arguable gameplay fairness. Chess is a popular board game that is played by two opponents on a checkered board. Players have equal sets of specially designed pieces of contrasting colors, commonly white and black. Players alternate turns to move one piece at a turn in accordance with fixed rules, where white moves first. They attempt to force the opponent's principal piece, the King, into checkmate - a position where it is unable to avoid capture. Chess fairness is being argued [16]. Reason being is that someone (a player that has got the white pieces) takes the first turn in the game. Statistically, this gives an advantage as more games are won with white pieces rather than black. Usually, players do not play one game at a time - they play series of games and switch sides meaning that pieces of each color will be played at least once by both players. But if the first game in the series is lost by a player with black pieces, it may cause him to play differently further and lose the series because of that. Hence, there is no perfect balance in chess.

Current thesis implies that if a game does not meet the requirement of providing equal opportunities to the opposing sides for winning, it is considered unfair.

Important note: "game fairness" and "fair gameplay" notion definitions may not be the same or even true outside the framework of this thesis. They were defined this way inside the current thesis only. They are used without quotation marks further.

1.2 Developed game description

The developed game was named Swordshot. The game is an isometric PvP fighting game (see **Figure 1**). The confrontation takes place on a flat square plane (further: map) somewhere in the sky. Both players are playing as stickman characters.

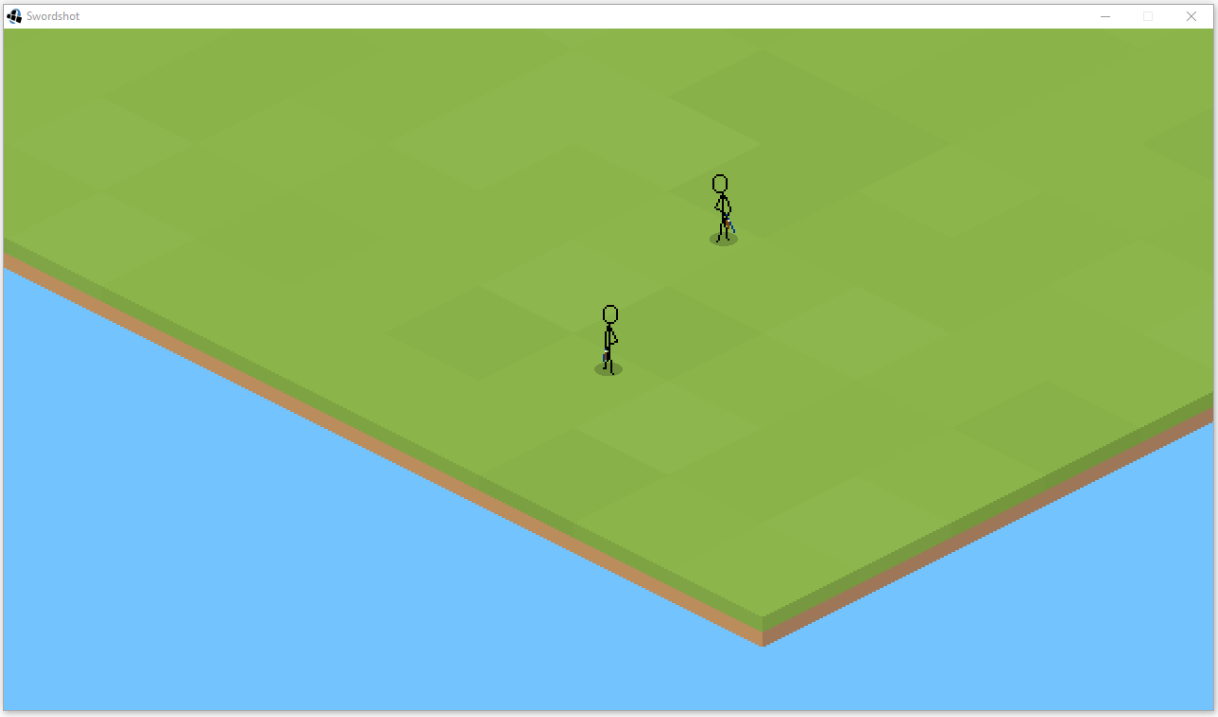

Figure 1: Swordshot in-game screenshot.

Stickman characters are able to:

1. Walk on the map;
2. Fall off from the map;
3. Execute 2 attacks: sword attacks and pistol shots;
4. Kill another player's character by executing attacks;
5. Get killed by another character attacks.

A player's character is always being centered in the game window. This is done by transitioning all game objects in accordance with a player character's position. The game implies 8-directional character movement. It can move in any of the cardinal directions (North, South, East, and West) as well as in directions in between (North - West, North - East, South - West, and South - East).

1.2.1 Input devices

Character manipulations are controlled by specific key presses (see **Figures 2** and **3**).

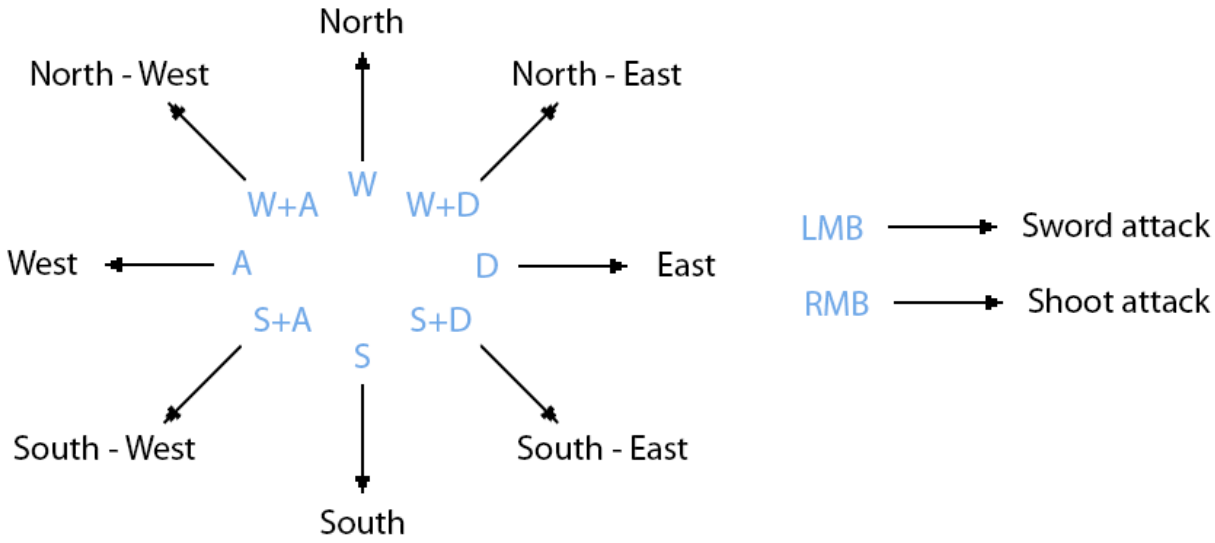

Figure 2: Character action key bindings. Character movement directions as well as attacks and corresponding controlling keys. Keys are in blue color (where W, A, S, D are keyboard keys and LMB and RMB are right and left mouse buttons respectively). Directions and attack names are colored black.

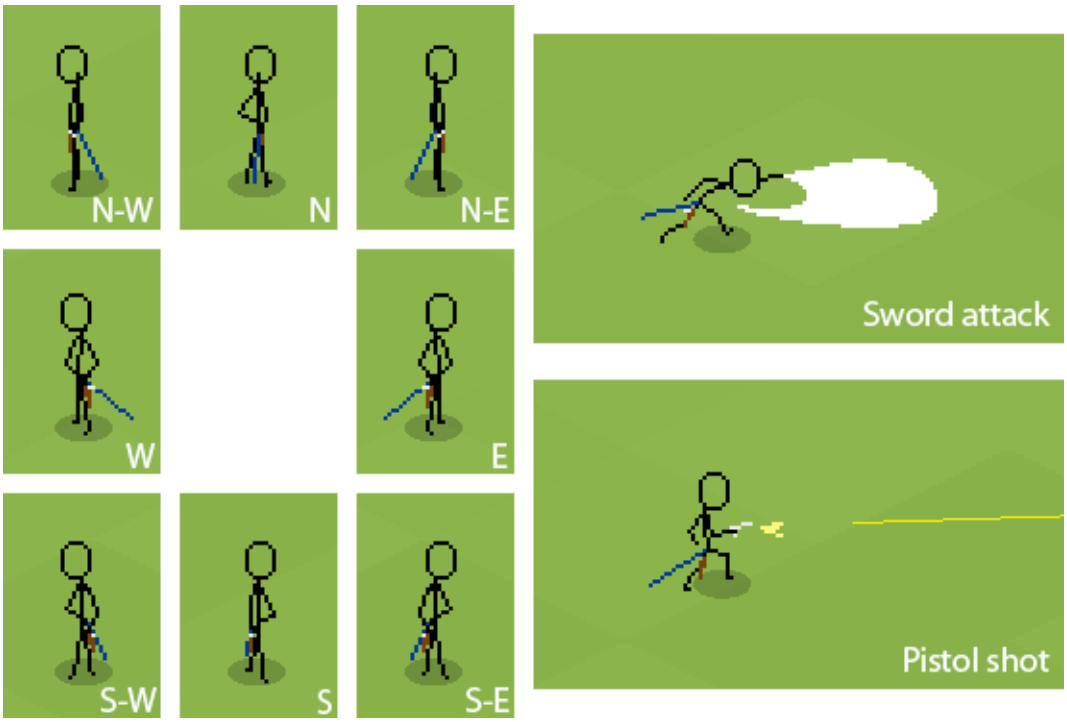

Figure 3: Character attacks and facing directions depiction. Directions North, East, South and West are shortened as N, E, S, W respectively.

In games where players must aim it is easier to do with a mouse rather than a game controller. Game controllers use analog sticks for aiming compared to the mouse which can be precisely and accurately positioned on screen requiring much less effort [17]. Hence, mouse and keyboard control were chosen over the game controllers. But as only one mouse cursor is allowed per computer, the game is made to be played through a network. This way each player can control their character using an individual computer and peripherals (i.e. input devices).

1.2.2 Game logic

Game mechanics and logic are calculated from a top-down 2D (two-dimensional) perspective of the game. What player sees is an angled isometric projection of this from the side (see **Figure 4**).

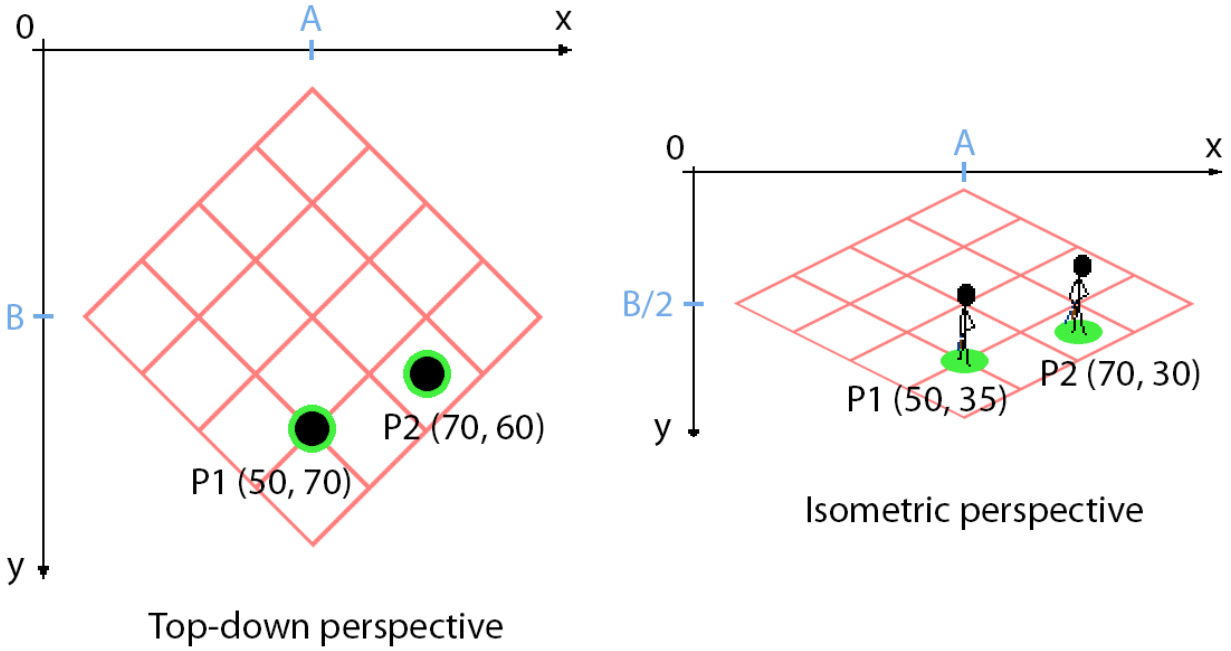

Figure 4: Top-down and isometric demonstrative perspectives of the game. A, B and B/2 are arbitrary positions on the x-axis and y-axis that show coordinate conversion upon perspective change. There are two characters at positions P1 and P2. Their logical coordinates are depicted in Top-down perspective picture whereas visual coordinates are depicted in Isometric perspective picture. Character positions are colored green.

Object positional coordinates in the game space are maintained using x-axis and y-axis. The third axis is used only for character fall simulation. Besides the character falling process, the third axis is not used. Objects have logical and visual coordinate types. Logical coordinates are the actual coordinates of an object in the game space. Those form a top-down perspective. Visual coordinates are those used to draw objects with their representative images onto the user's screen. Those form an isometric perspective. To convert logical coordinates to visual coordinates, y coordinate is divided by 2. X coordinate is not changed. This way, a three-dimensional view is simulated.

1.2.3 Spawning

When the game starts, each character is spawned at one of the four determined spawn locations. Each of them is near a separate corner of the map. (see **Figure 5**).

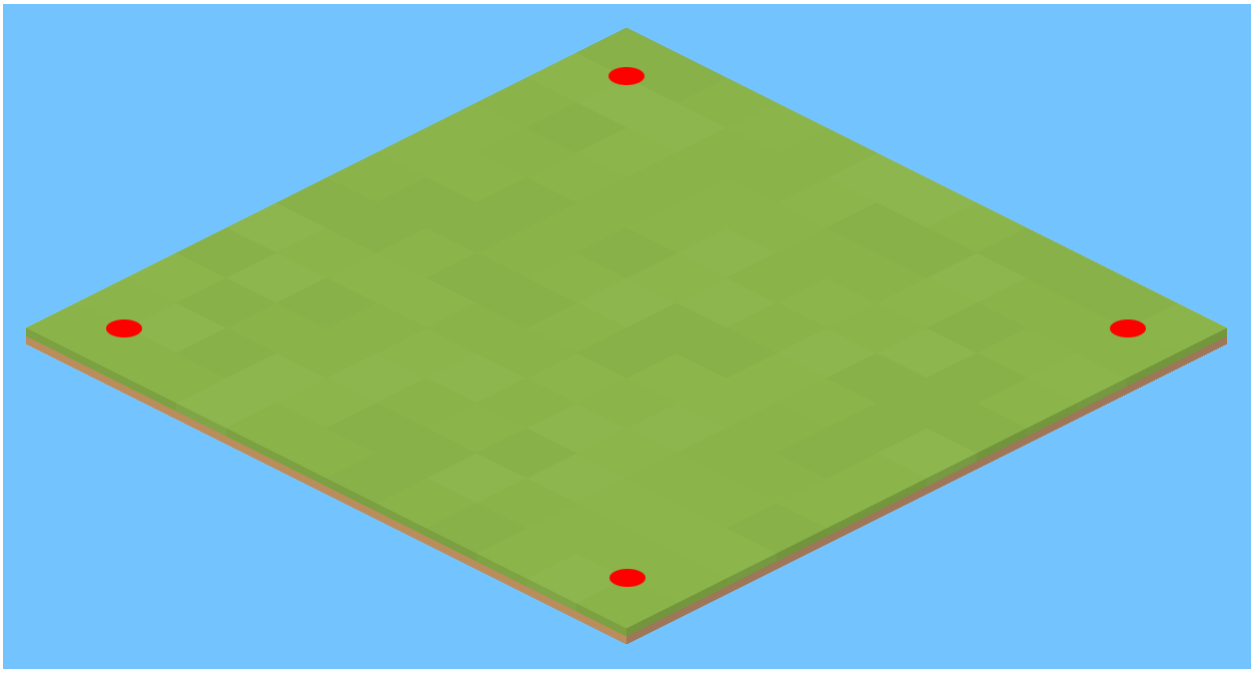

Figure 5: Swordshot game map with character spawn locations. Spawn locations are represented by red ellipses.

Respawn location is chosen by the game. A character will not respawn at the closest location to another character. It will respawn randomly at one of the other three leftover locations. Characters also respawn after dying. They die after getting hit by another character's attack or crossing the map boundaries. On death, the character becomes disabled for two seconds. The decision to respawn the character is made by the game.

1.2.4 Score system and winning

The first player to reach the score of 20 wins the game, the other player loses the game at that point. Both players start the game with 0 score points. To gain 1 point a player's character must kill other player's character. It can be done by either hitting a character with a single sword attack or a pistol shot. Falling off the map subtracts 1 score point from the current player's score and kills their character. When a player's score is changed their updated score appears on their screen for 2 seconds.

1.2.5 Combat

Characters can interact with each other only by attacking. To kill a character a player must execute a successful attack. For an attack to be successful it needs to hit another player's character. Attacks and character bodies have their areas of effect called hitboxes. The hit is registered if an attack

hitbox of one character intersects the body hitbox of another character. Attack and body hitbox shapes vary depending on the attack type (see **Table 1** and **Figures 6-8**).

Table 1: Hitbox shapes depending on the attack type

	Attack hitbox shape	Body hitbox shape
Sword attack	Rectangle	Rectangle
Pistol shot	Line	Ellipse

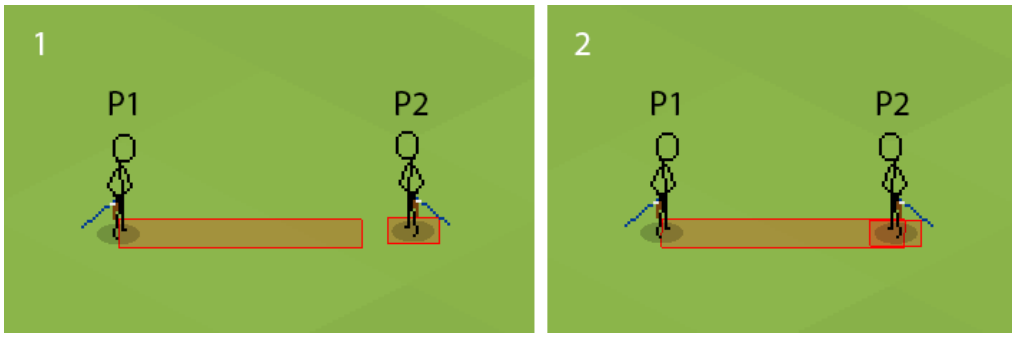

Figure 6: An East-directed sword attack hitbox of the character P1 as well as the body hitbox of the character P2 in the isometric projection. A sword attack from the character P1 in picture 1 would be unsuccessful whereas in picture 2 it would hit the character P2.

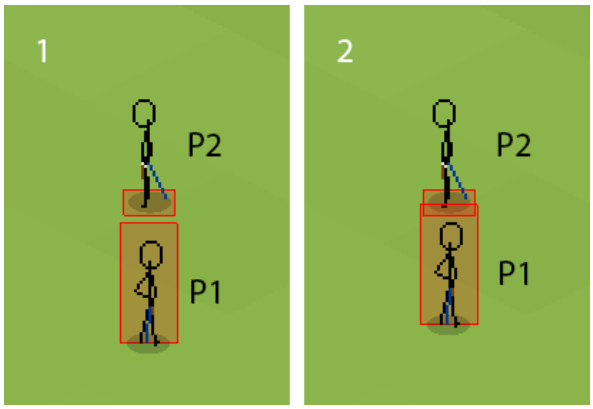

Figure 7: A North-directed sword attack hitbox of the character P1 as well as the body hitbox of the character P2 in the isometric projection. A sword attack from the character P1 in picture 1 would be unsuccessful whereas in picture 2 it would hit the character P2.

Sword attack hitbox size is visually changed in isometric projection depending on the direction it is executed at. However, as the game logic and mechanics take place from the top-down perspective, hitboxes for the sword attacks are of equal size no matter the direction. They are also visually invisible to the players during the game.

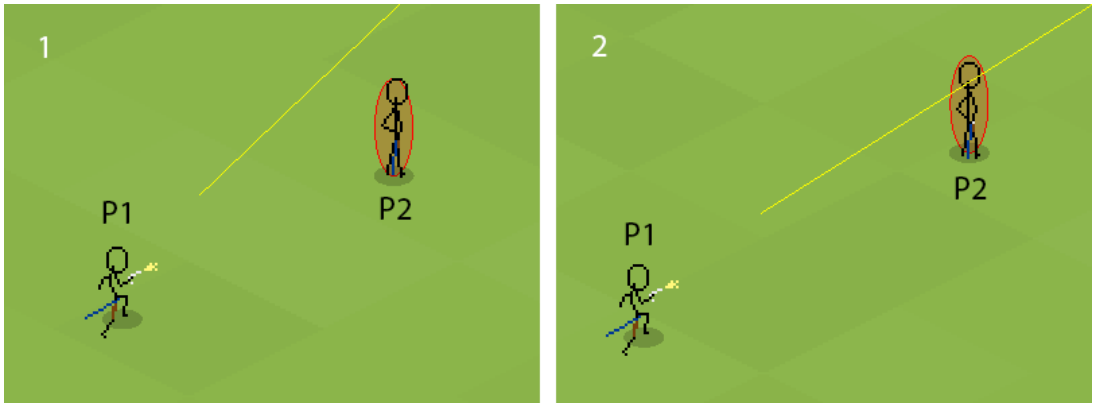

Figure 8: A pistol shot executed by the character P1 as well as the body hitbox of the character P2 in the isometric projection. Picture 1 depicts an unsuccessful pistol shot execution by the character P1. The pistol shot executed by the character P1 in picture 2 is successful and hits the character P2.

The mechanics and logic for pistol shots take place in the isometric perspective being one and only exception. Hitboxes are created and maintained using visual coordinates of the players. The exception was made for the pistol shots to be consistently represented visually. Otherwise, if the mechanic and logic took place in the top-down perspective, the shots hitting a character's head visually would not be considered as successful.

During the game, player characters are given one bullet on their spawn / respawn. Meaning, that they do not get another bullet unless they die.

1.3 Game design factors and external conditions that affect game fairness.

The next two sub-sections describe and give examples of some of the most notable game design factors and external conditions that can affect a game's fairness. The information on how each one of them is handled during the experiment and Swordshot game design is given right after their brief explanation.

1.3.1 Game design factors

Game design factors are factors that affect in-game gameplay directly. Game design is what makes up the game and its rules [18].

1.3.1.1 Position and environment

In environmental action PvP games, positioning is crucial. If an environment has got any sort of surface irregularities like hills or walls, the position a player is in affects player's chances for winning. **Figure 9** shows an example of imbalanced gameplay in first-person shooter game type.

Player A and player B are the opposing players that are placed on the same plain field. From a Player A perspective, the player B has half of their character body hidden behind a wall. This way, Player A can roughly see only the upper body part of the opposing character while player B sees the whole enemy character body. There are better chances for player B to hit player A with a gunshot than player A to hit player B.

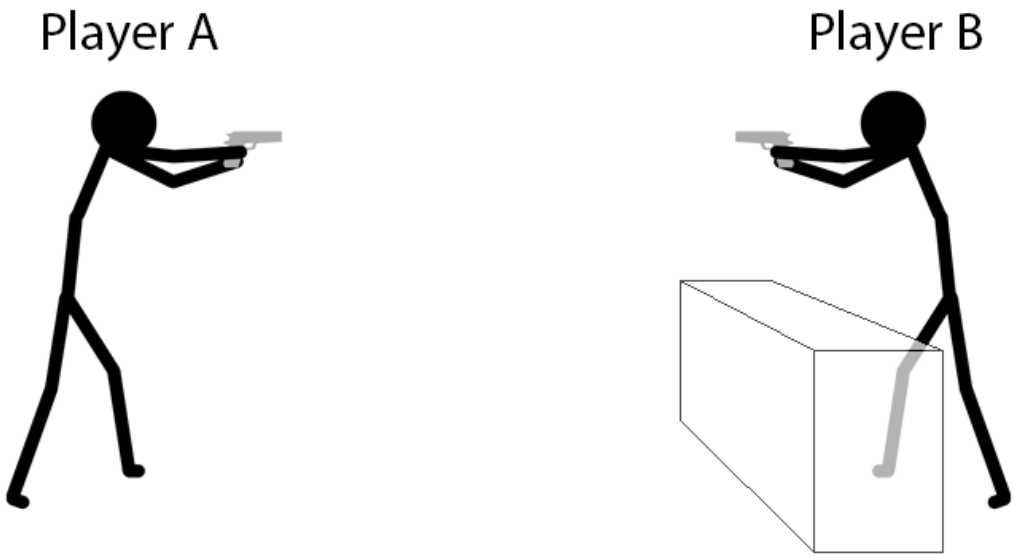

Figure 9: First person shooter game type example of environmental imbalance.

In Swordshot, positional advantages cannot occur as the environment of the game is a completely flat symmetrical rectangular surface.

1.3.1.2 Different character classes

Characters that differ from each other by an aspect are from different classes. They may differ by every aspect, such as size, movement speed, abilities and more. Different character classes may influence the outcome of a game.

An example of this would be a fighting game, where one character can move faster but has less health than another character. A perfect balance of power between those characters cannot be achieved as they are stronger or weaker in different aspects.

In Swordshot, the character design is classless. Each player is playing one and only character in the game with specific abilities.

1.3.1.3 Chance and probability

Some games include chance and probability that influence a player's winning chances. A lesser-skilled player might win a game against a skilled player because of random number generation.

Let us consider a fighting game that involves critical attacks. A critical attack in this game would be an attack that deals double the amount of normal attack's damage with some probability. In such a game, a player who has more "luck" and deals more critical attacks – has an advantage.

In Swordshot, chance and probability are completely excluded.

1.3.2 External conditions

Besides game design factors there are also game-external conditions that might affect a game's fairness and player experience. More information on how the experiment was held is available under the "Experiment conduction and playtests" chapter.

1.3.2.1 Latency

Latency in networks is the time interval it takes for a packet to travel from one designated point to another [19]. Network-communication fast-paced action games are sensitive to latency. In most games of the action genre, players with lower latency usually have an advantage over players with higher latency. Latency differences between players are caused mainly by their geographical location and Internet connection speed and reliability. As online games are played from different locations and with different Internet connection qualities, latency differences occur causing issues between player interactions.

The experiment conducted in this thesis was held locally. Computers of the participants were connected to the same local area network. By this, latency factor influence was minimal being as equal between players as possible. There are other methods to compensate latency issues between players. For example, by extrapolating player actions, using client-server side predictions and many more [20, 21]. These methods partly mitigate latency issues and do not equalize them for players as good as the method used in the conducted experiment.

1.3.2.2 Computer power

Resource-demanding games might require players to use powerful computers to be played without performance issues. Such games might experience performance issues if ran on a computer not powerful enough to handle them. Having performance issues affects gameplay, its speed, and user experience. As different people have different personal computers, games may get unfair when players with computers of different power oppose each other.

All the research participants were given equally powerful computers on which the game performance was as good as intended. This suppressed the computer power factor during the experiment.

1.3.2.3 Peripherals

Peripheral devices such as keyboards or mice might also differ by their reaction speed on user's actions. All research participants were given equal keyboards and mice for this factor to be balanced.

1.4 Achievements

People play games for various reasons, two of which (the competition and the achievements) were considered as mentioned before. The game Swordshot allows people to compete by playing against each other. As an achievement, winning players were complimented by the author. Moreover, a chocolate bar was promised and given to winners as an incentive to try harder and play better. This was done in order to make the research participants more motivated to win the game.

2 Game development

A game called Swordshot that meets the requirement of fair gameplay was developed for the research participants to be played. A server application was also developed to complement the client-server game architecture and enable players to interact with each other over a network connection. Game and server application code and releases along with the requirements on computer hardware and instructions are available in the Bitbucket repository [22]. Detailed descriptions of the structure of the game and server application are given in this chapter.

2.1 Technologies used

Although popular game engines for 2D game development (Unity [23], GameMaker: Studio [24]) were considered, the author decided to go with a clearer approach building the game up with Java [25] and libraries that utilize Java native tools, due to being more familiar and experienced with it. By this, it was also ensured that the game did not obtain any hidden mechanics from an engine that could cause it to run in an unexpected and uncontrollable way.

The game was developed using Java Development Kit version 1.8. This means the game is compatible with machines that have Java Runtime Environment version 8. The game can technically be run successfully on any of the major computer operating systems like Windows, Mac OS, and Linux. It was tested only on Windows 10.

2.1.1 Slick2D library

Slick2D library [26] is a set of tools and utilities that utilize the graphics card if possible. It is wrapped around LWJGL (Lightweight Java Game Library) [27] which is a lower-level language library. Slick2D makes Java game development easier by providing higher-level language development tools that utilize LWJGL tools. Libgdx [28], which is a similar to Slick2D library, has more features than Slick2D and is being developed by this day as an advantage. But it does not have as consistent documentation as Slick2D does which made Slick2D the preferred choice over Libgdx. Also, the Libgdx variety of tools was excessive for the developed game.

2.1.2 Kryonet library

Kryonet library [29] is based on *java.net* Java package and provides simple client-server communication tools and utilities. Kryonet simplifies and generalizes the use of java networking tools making them easier to use. It was used to make Swordshot run successfully on a network using client-server communication architecture. Swordshot game instance runs a client program that connects to a running server program at a specified address. On successful client connections, server program handles them and enables clients to communicate with each other through it. Client-server communication is described further in this chapter.

2.2 Game structure

The game Swordshot is created using Slick2D library and its container system. The container system provides the functionality that allows creating standalone application windows (i.e. the containers). *Org.newdawn.slick* package and its sub-packages will be further referred to as simply *.slick* for readability purposes. Games created using Slick2D that use the container system should implement the main game interface *.slick.Game*. The direct needed implementation of that interface is the class *.slick.StateBasedGame*. This class allows easy manipulation of different isolated, yet interchangeable game states (for example menu state, in-game state, etc.).

2.2.1 Game class

Slick2D provides an easy way to create a standalone application window for the game as an object which then has to be started:

```
AppGameContainer appgc = new AppGameContainer(new
Game("Swordshot"), SCALED_WIDTH, SCALED_HEIGHT, false); //
Create the game container
appgc.start(); // Start the game
```

.slick.AppGameContainer is the main part of the container system mentioned before and is the first thing that is called in the main method of the game Swordshot. Its constructor takes 4 parameters: an instance of a *.slick.Game* interface implementing class (in this case, a custom *Game* class that extends *.slick.StateBasedGame* class), the width of the container, the height of the container, and a boolean for the fullscreen option. The main method of the game resides inside the *Game* class. Its constructor is called when `new Game("Swordshot")` instance of it is created:

```
public Game(String name) {
    super(name); // pass to .slick.StateBasedGame constructor
}
```

The *name* parameter is what the application window of the game is named. This parameter is passed to and handled inside the *.slick.StateBasedGame* class constructor. Also, an abstract method *initStatesList* of that class is overridden in the *Game*'s class:

```
@Override
public void initStatesList(GameContainer gameContainer) throws
SlickException {
    addState(new Menu(STATE_MENU));
    addState(new InGame(STATE_INGAME));
}
```

It creates and initializes interchangeable game states using *addState* method provided by *.slick.StateBasedGame* class. This is also called as a state pattern. *addState* method takes an instance of *.slick.BasicGameState* class as an argument. *Menu* and *InGame* custom classes were

created for this purpose. These, upon instantiation, are given an ID (identification number) (***STATE_MENU*** and ***STATE_INGAME***) to switch between them. States are the "stages" or "scenes" in an application. To "enter" a game state means executing the *Game.enterState(int state_id)* method. When this method is executed, the current state is abandoned, calling its *leave* method and the state with a specified *state_id* is entered, calling its *enter* method. When a state is entered, it becomes the current state that the game is operating and the user is interacting with.

2.2.2 Game states

The game application has two custom state classes: the *Menu* class and the *InGame* class. These classes are called as states further. Game states are class-extensions of a *.slick.BasicGameState* class. This class is an implementation of the interface *.slick.GameState*. This interface generalizes game states giving them several methods described in **Table 2**.

Table 2: *.slick.GameState* interface methods.

Method	Usage
getID()	Returns the ID of the state. Used only to get the state to switch to.
init(GameContainer container, StateBasedGame game)	Called when the state is created and added to the game in *initStatesList* method discussed above. Used for functionality that needs to be initialized only once.
enter(GameContainer container, StateBasedGame game)	Called every time the state is entered. Used for functionality that needs to be executed every time the state is entered or re-entered.
leave(GameContainer container, StateBasedGame game)	Called every time the state is left. Used for functionality that needs to be executed every time the state is left.
update(GameContainer container, StateBasedGame game, int delta)	The state's logic processes are updated here (e.g. user input, character movement etc.). Ideally is called 60 times per second. Call frequency might fluctuate.
render(GameContainer container, StateBasedGame game, Graphics g)	Used to render the state to the game container. To render means to draw and create an image that is displayed on the user's screen. Ideally is called 60 times per second. Call frequency might fluctuate.

Both *Menu* and *InGame* states implement the methods described above.

2.2.3 Menu state

Menu state is what the user is presented with at the game's launch (see **Figure 10**).

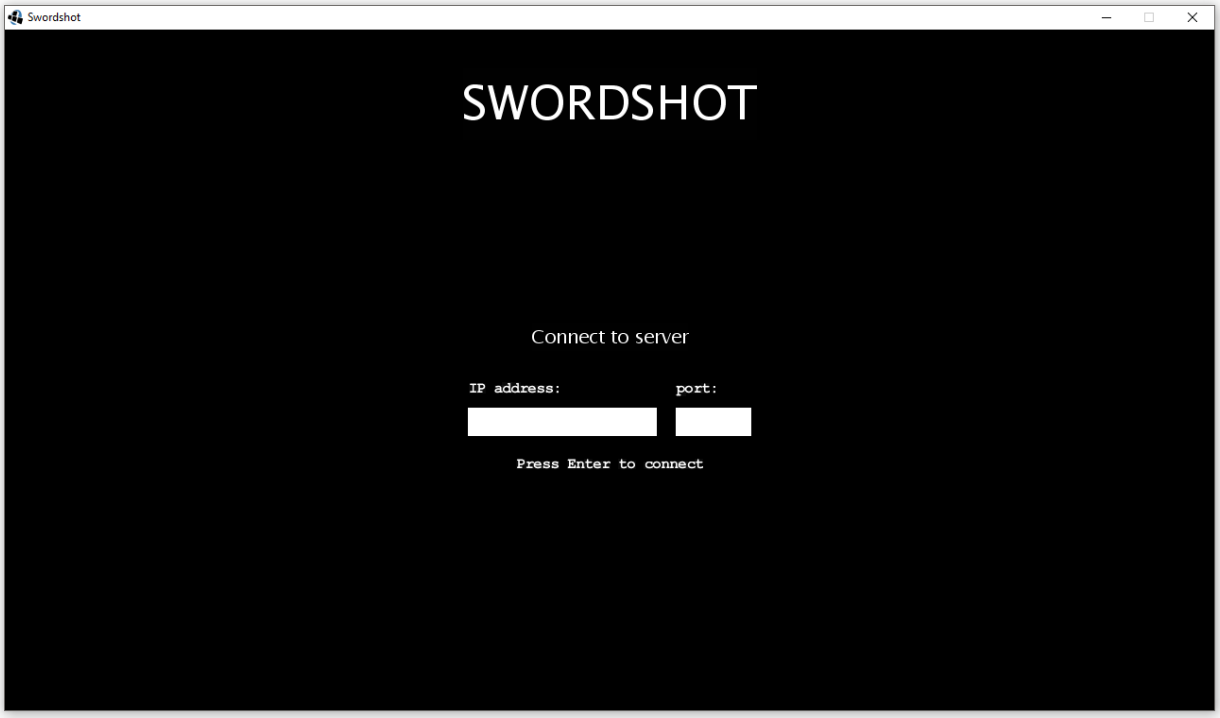

Figure 10: Swordshot *Menu* state screenshot.

This is the first state entered by the game. To play the game a user must connect to a running server application. *Menu* state provides functionality to enter an IP (Internet Protocol) address and a port number at which the desired server application is running. By pressing the "Enter" key an attempt to connect to a server with entered server address takes place. On failure, the game state is not switched, and the user is notified with an error message. On a successful execution, the game connects to the server application establishing client-server communication and the *InGame* state is entered.

2.2.4 InGame state

The *InGame* state is depicted in **Figure 1**. *InGame* state is where the actual gameplay takes place. The player gets to control their character and the game's interface using specific keyboard and mouse keys (see **Figure 2** for character controls). A player can also press the keys "I" to toggle game info statistics and "Esc" to switch to the *Menu* state and disconnect from the server application.

2.2.4.1 InGame render method

The game is rendered using the *render* method. Sky, game map, all the characters and GUI (Graphical User Interface) are drawn and displayed to the user's screen by rendering them where and when needed. Renders occur 60 times per second as set by the game. This number is called FPS (Frames Per Second). Each new frame everything is re-rendered to present a clean image of the game to the user constantly.

First of all, the sky is rendered. The sky is a solid color that the application window is refilled by. Everything after is rendered using graphical resources. All of the graphical resources reside inside the game's "res" folder (path: ../<Game folder>/res). The game map consists of sectors (called tiles) that are rendered using a file "map.txt" and picture resources located in the "default_map" folder (path: ../res/maps/default_map). The file "map.txt" contains the information about the map which is read once by the game. The game uses the information to draw specific tile pictures at specified positions. After that, characters of the players are drawn using sprite sheets.

Sprites in computer graphics are two-dimensional bitmaps (i.e. pictures). Sprite sheets are the collections of sprites. In Swordshot, sprite sheets are used to store different stages of the character animations. The game animates characters using custom-made character sprite sheets (see **Figure 11**).

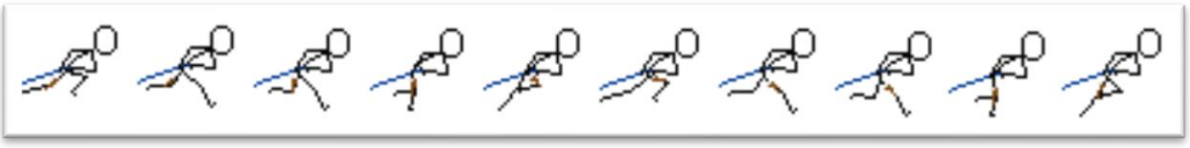

Figure 11: A part of character running animation sprite sheet.

The GUI (Graphical User Interface) is rendered last. GUI consists of text strings and the crosshair, which are rendered on the screen when necessary.

Everything is rendered using the *.slick.Graphics* type parameter *g* of the *render* method.

2.2.4.2 InGame update method

Game's logic is updated inside the *update* method. Time, character movement, physics, player inputs and interactions with the game, as well as communication with the server are handled and calculated in there. Updates also occur 60 times per second. This number is called UPS (Updates Per Second).

First, the current update time is saved. After that, the network update is executed when needed (network updates are described at the end of this section). This is then followed by game logic calculations which are player input and interactions with the game, character movement and physics. Updated character information is then sent to the server. Further time calculations

conclude the update cycle. Performing rendering and updates 60 times per second provides players with smooth gameplay experience. The lower the render and update frequencies are, the less-smooth is game experience.

Rendering and update frequency fluctuations

Render and *update* methods are not executed in a specific sequence. They are executed independently of each other. It is preferred for them to run at a constant frequency of 60 times per second as described above. Though, rendering and update frequencies (further: RF and UF) may fluctuate. Renders and updates are not always executed at a constant pace and their frequency may vary depending on computer power and performance (e.g. at a moment, UF may drop to 50 UPS; the same can happen to the RF). RF fluctuations cannot cause severe problems, compared to UF fluctuations. These might cause the game to be updated improperly while being in *InGame* state. *Menu* state is not affected by the fluctuations as it is a simple user interface without much graphics and mechanics.

UF fluctuations may cause character movement to become improper and imbalanced. A character, who moves, one unit at an update will move 60 units per second at 60 UPS and only 20 units per second at 20 UPS. This problem is mitigated by multiplying every physics calculation by the parameter *delta* of the *update* method. *Delta* is an integer number that indicates how much time in milliseconds has passed since the previous *update* execution. As a result, calculated values are approximately the same with different UF.

Network updates

There are also network updates which were mentioned before that are executed 20 times per second inside the update method. This number is called NUPS (Network Updates Per Second). Network updates perform client-server communication by sending and receiving data packets (small static classes that contain various information like character positions, actions, messages, etc.) as well as handle those data packets. Performing network updates only 20 times per second as opposed to 60 conserves network bandwidth usage. This does not affect gameplay and its fairness. Network updates are visually compensated by network interpolation technique. The algorithm takes an object's current position and last position and interpolates between them, calculating positions in between which are rendered. As a result, the user sees only smooth movement of the game's objects. The drawback of this technique is that the user is shown only the previous network update, seeing enemy characters slightly "in the past". Fairness is not affected since all players execute network updates at the same rate of 20.

2.3 Client-server communication

Client-server communication is realized using Kryonet library. *com.esotericsoftware.kryonet* package and its sub-packages will be further referred to as simply *.kryonet* for readability purposes. Custom classes *ClientProgram* and *ServerProgram* that are responsible for a successful client-server communication were built. The *ClientProgram* and the *ServerProgram* are used by the game and the server application respectively.

To start the server, it is bound to a specific port and is then started: (`sp` is the *ServerProgram* instance)

```
sp.server.bind(port); // Bind the server to a port
<..>
sp.server.start(); // Start the server
```

To connect the client to the server the following sequence is executed:

1. *connectToServer* method is called from the *Menu* state when the user presses the "Enter" key:

```
((InGame) game.getState(Game.STATE_INGAME)).connectToServer(ip,
port);
```

2. *connect* method of the *ClientProgram* instance `cp` is called:

```
public void connectToServer(String ip, int port) throws
Exception { //throw further into the Menu
    cp.connect(ip, port);
}
```

3. *connect* method is executed:

```
public void connect(String ip, int port) throws Exception {
    client.start(); // Start the client
    try {
        client.connect(5000, ip, port); // 5000 is the timeout
value in milliseconds
    } catch (Exception e) { // In case of failure
        client.stop(); // Stop the client
        throw e; // Throw the exception into the InGame
    }
}
```

The major difference between the two (the *ClientProgram* and the *ServerProgram*) is that the *ClientProgram* creates an instance of *.kryonet.Client* `client` and the *ServerProgram* - an instance of *.kryonet.Server* `server`. These two instances are also *.kryonet.Endpoint* interface implementations and thus, are called endpoints. Moreover, `client` instance can connect to only

one `server` instance and communicate only with it. `server` instance is able to have multiple `client` instances connected to it, with a possibility to communicate with all of them.

Being an extension of *.kryonet.Listener* class enables *ServerProgram* and *ClientProgram* instances to listen for incoming data packets and handle (further: serve) them upon receipt. Thus, they are referred to as listeners later. `client` and `server` instances are given those listeners by calling

`client.addListener(this);` and `server.addListener(this);`

respectively as they form separate threads. Dividing the program into several threads allows to execute multiple processes concurrently and independently. Meaning, that the data packets can be received, served and sent at the same time without these processes interrupting each other.

Endpoint *.kryonet.Client* and *.kryonet.Server* instances must be familiar with the structure of the data packets they send and receive in order to work properly. For that purpose, the *Network* class was built.

Network class is a container for data packet classes. The static *register* method of that class is used by endpoints to register networked data packets (i.e. to familiarize with them) and is called in both *ClientProgram* and *ServerProgram* constructors:

`Network.register(client);` and `Network.register(server);`

respectively.

The following example of a data packet is the *PacketUpdateX* class which contains a player's ID and their character's x coordinate:

```
static public class PacketUpdateX {
    public int id;
    public float x;
}
```

Both listeners use TCP (Transmission Control Protocol) [30] to send and receive data packets. TCP is a protocol that is used for reliable network conversations. There is also a UDP (User Datagram Protocol) [31] which allows transferring packets at a higher rate at the cost of reliability. TCP was chosen over UDP because the game played throughout the experiment had to provide fair gameplay to the players. Thus, the reliable delivery of player decisions and actions was chosen over the speed of the communication.

ClientProgram and *ServerProgram* listeners also have a queue of received data packets that need to be served. When a `client` or a `server` instance receives a data packet the *received* method of the corresponding listener is called. This puts the packet into the `receivedPackets` queue.

```
receivedPackets = new ConcurrentLinkedQueue<>(); // Received
packets are put in here
```

As packets are served outside of the listeners, they both implement a public method to retrieve the `receivedPackets` queue. Queued packets are polled out of the `receivedPackets` queue and served one by one until the queue becomes empty. This process is similar for both the game and the server application. However, the ways the game and the server application are operating with data packets are different.

Client data packet handling

The game handles data packets inside the *InGame* state instance when network update occurs (20 times per second). Clients do not communicate with each other directly. Instead, clients send data packets only about their character data to the server and receive data packets containing other data like messages, other character positions, commands, etc. from the server. First, all the received packets are served. After that, data packets containing updated character information are sent to the server. To send a packet to the server the following `client` method must be executed:

```
client.sendTCP(packet);
```

Server data packet handling

The server application handles packets inside the *ServerLoop* class which is described in the next section. This process also occurs 20 times per second. Server application's responsibility is to deliver updated information to all the clients connected to it. Right after the packet has been served, clients are informed about updated information on server processes. This is what makes client and server data packet handling different. Three `server` methods are used for sending packets to clients depending on the functionality needed:

```
server.sendToTCP(client, packet); // Send the packet to a
specified client
server.sendToAllExceptTCP(client, packet); // Send the packet to
all clients except the specified client
server.sendToAllTCP(packet); // Send the packet to all clients
```

2.4 Server application structure

The server application consists only of four classes:

1. *Network* - a container class for data packet classes which is used to register networked data packets as mentioned before (an identical class is used for the game's `client` instances);
2. *ServerReceivedPacket* - simple helper class for better data packet structuring;
3. *ServerLoop* – the class, where the main gameplay and server logic take place;

4. *ServerProgram,* where the main method of the server application resides. How it creates and runs the `server` instance to receive and send packets was described earlier. To handle client connections and disconnections the two methods (*connected* and *disconnected*) of the `server` instance are called respectively. They pass the information about the client connection or disconnection further to the *ServerLoop* instance thread `serverLoop` which handles them. `serverLoop` instance is started as a separate thread inside the *ServerProgram* class before the `server` instance: (`sp` is a *ServerProgram* instance)

```
sp.serverLoop.start(); // Start the server loop
sp.server.start(); // Start the server
```

2.4.1 ServerLoop class

ServerLoop is responsible for handling client connections, disconnections and interactions among each other, serving client data packets as well as distributing client and server information among the clients and managing gameplay modes. The reason why this class is called a "loop" is that its main functionality is running in an endless controlled *while* loop until the server application is stopped. This *while* loop is responsible for updating the server logic at the frequency of 20 times per second. Server logic updates are performed by calling the *update* method.

2.4.1.1 ServerLoop update method

First, all the received packets are served and the updated information about the game process is distributed among clients. During this, player character information excluding attacks is registered and simply distributed to other clients without processing. Attacks need to be processed and verified as the *ServerLoop* is responsible for the kill confirmation and the killing of player characters. Sword attacks and pistol shots have different mechanics.

Sword attack mechanics

Sword attack mechanics and calculations happen from a top-down perspective of the game and use logical character coordinates. When a player executes a sword attack their client sends a specific packet containing the angle information at which the sword attack was executed at. This angle is read by the server program as the "direction" of the attack. A *.slick.Shape* object is created on the server which is a rectangle representation of the sword attack hitbox at a specific position beside player character. It is then rotated clockwise by the angle from the received packet around the player character position to match the actual direction of the attack.Whether the attack hitbox intersects with any player's character body hitbox is checked next. Character body hitbox for sword attacks is a *.slick.Shape* object as well which is a rectangle around the character. **Figure 12** shows demonstrative sword attack hitboxes.

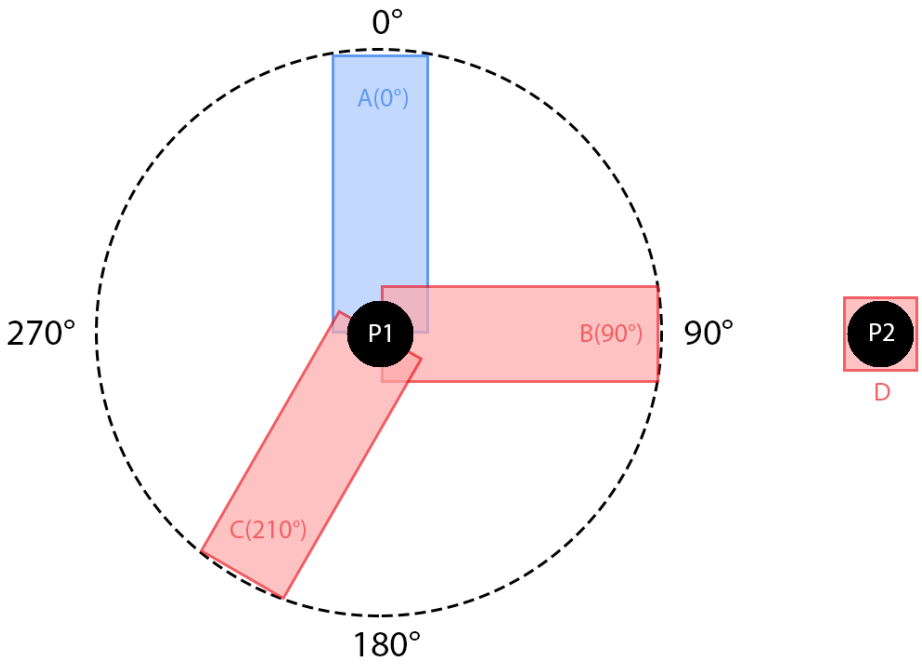

Figure 12: Attack hitbox and its rotation (on the left) as well as character body hitbox for sword attacks (on the right).

The blue rectangle A, the rotation of which is 0 degrees, is the initial sword attack hitbox created when a sword attack is executed. It is then rotated clockwise by the angle from the received packet around the character P1 for the attack to be facing at the proper direction. For example, the hitbox B is rotated by 90 degrees around the character P1 and the hitbox C is rotated by 210 degrees around the character P1. D is the body hitbox of the character P2 for sword attacks. If the sword attack hitbox of the character P1 intersects with the body hitbox of the character P2, then the sword attack is successful. If a successful sword attack from the character P2 is executed in reply to the character P1 attack shortly after, the sword clash occurs and both characters are pushed aside from each other without dying. Otherwise, the character P2 is killed.

Pistol shot mechanics

Unlike sword attacks which mechanics happen from top-down perspective, pistol shot mechanics happen from the isometric perspective and use visual character coordinates. When a player executes a pistol shot their client sends a specific packet containing the vector at which the pistol shot was executed at. A *.slick.Shape* object is created on the server which is a line representation of the pistol shot hitbox. It starts at the character's visual position and follows the vector. The line length is equal to the map size in units meaning that a pistol shot can reach any character from any position on the map. The character body hitboxes for sword attacks and pistol shots are different. The character body hitbox representation for the pistol shot is a *.slick.Shape* object as well which is an ellipse of fixed size around the character model. If the pistol shot hitbox of the character P1

intersects with the body hitbox of the character P2, then the pistol shot is successful. As a result, the P2 character is killed.

The server uses *.slick.Shape* instances because they provide necessary and convenient tools to check for intersection of two shapes.

Gameplay modes

After the packet handling, the major gameplay mode check is performed. It controls whether the gameplay mode needs to be changed or not. Gameplay modes are different game stages that are possessed by the server.

Note: gameplay mode checks are also performed after events that might cause the gameplay mode to change (e.g. a player getting a score point by killing another player).

There are 4 gameplay modes:

1. **Lobby mode:** This is the first gameplay mode the server enters. While in lobby mode, players can connect to the server, they have unlimited ammo and they can interact with each other not gaining or losing any score points. Connecting to the server is permitted until it becomes full (two players have connected to the server). When the server is full further connections are then rejected and connected players are prompted to press the "R" key to indicate that they are ready for the game to get started. When both players are ready the game starts with the server entering the countdown mode;
2. **Countdown mode:** This mode is an intermediate mode for players to get prepared for the game. It kills all the characters giving players 5 seconds to prepare for the game start. After 5 seconds have passed the game mode is entered;
3. **Game mode:** This is the main gameplay mode which represents the game process itself. Players get the ability to kill each other's characters for score points. After one of the players reaches the score of 20 the game is considered to be finished and the endgame mode is entered;
4. **Endgame mode:** Player that has first reached the score of 20 wins the game and is notified by "YOU HAVE WON!" message. Other player characters die and are not respawned until the gameplay mode is changed. Endgame mode lasts for 10 seconds after which the lobby mode is entered.

Detailed gameplay mode workflow chart is present below (see **Figure 13**).

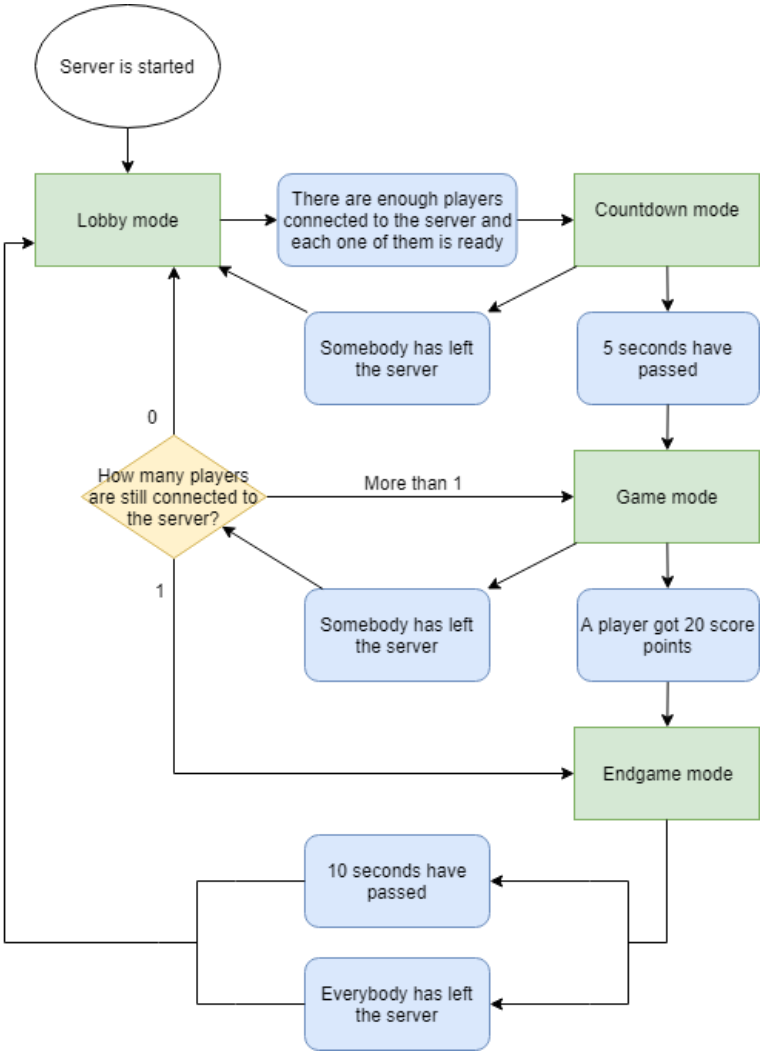

Figure 13: Gameplay mode workflow chart.

Lobby gameplay mode is the only gameplay mode which is not considered as a game-in-process mode like other modes. This means that players can connect to the server only if it is not full and only if it is in lobby mode. In other cases, new client connections are rejected and closed by the server.

Rest of the update method

SeverLoop update method is concluded by two further checks:

1. Check for sword clash event occurrence. Controls whether two characters have performed successful sword attacks on each other at almost the same time.
2. Check for dead player timer expiry. Controls whether the dead player characters need to be respawned.

The *update* method is executed inside the *run* method of the *ServerLoop* thread which contains the main while loop. It is responsible for updating the server logic at the frequency of 20 times per second as mentioned before. This is accomplished by checking whether a certain amount of time has passed after the previous update execution or not. If not, the thread sleeps for 1 millisecond in order to reduce the load on computer processing units. This means that the server program stops working for 1 millisecond and then continues where it left off.

This concludes the developed game and server application descriptions. The experiment conduction and its results, as well as the research analysis, are presented further in the thesis.

3 Experiment conduction and playtests

The experiment was conducted in a classroom of the Institute of Computer Science of the University of Tartu. The classroom presented a total of 15 computers which had the same hardware, operating systems, and peripherals (see **Figure 14**). All of them were also connected to the same local area network. There were eight research participants. Their age was ranging from 19-23 years.

Figure 14: The classroom where the experiment was held in.

3.1 Environment and preparations

Players were put in pairs of two, making a total of four pairs. Paired research participants sat next to each other while their server application ran on the adjacent separate computer. This way, opposing players were connected to a third computer. Running a server application on a separate computer equalized the client-server communication latency for both paired players. When the research participants came into the classroom, the games on their computers were already running and connected to the needed server. Players were seated and instructed the game controls and rules. After the explanations, they were given indefinite time to explore the game and get used to the game controls being in the lobby gameplay mode. Players started playing the game by pressing the “R” key when they were comfortable and ready.

3.2 Observational results

The research participants were observed during the playtests. Notes about their emotions, behavior, and questions were made. These results are only observational and should not be treated as statistically accurate.

Players looked confident while playing the game. There were very little questions about the gameplay and the game rules and only from those players who did not listen to the explanations carefully at the start of the experiment. During the gameplay, the research participants mostly had smiles on their faces. After the games were finished, pairs announced the winners themselves. None of the paired players shook hands after the games. The players did not express any negative emotions either.

Judging by the observed results, it looked like players enjoyed the process, were excited and motivated to win the game. It is worth mentioning that nobody showed disagreement on the outcome of the game. The results might have been influenced by the fact, that the participants were friends of the author. However, personal responses show their actual emotions and feelings better.

4 Questionnaire results

Research participants were asked to fill out and submit a questionnaire (**Appendix I**) about the game, the feelings they got from it, their experience with similar games and the experiment fairness. Note, that players were told that the game they were playing was fair. Also note, that players were not given the definition of gameplay fairness. What was fair was up to them to decide. The questionnaire was made with the Google Forms web application [32].

4.1 The impressions of the experiment, the game and its fairness

Overall, research participants liked the game and its idea. They had fun playing the game and were motivated to win at it. Judging by the questionnaire answers the game did an average job of providing a good challenge level to the players (see F**igure 15**). Players stated that the external conditions were pleasing and had a rather positive influence on the Swordshot gameplay than neutral (see **Figure 16**). It was also made sure that all the research participants are acquainted with and have played some of the other action PvP video games.

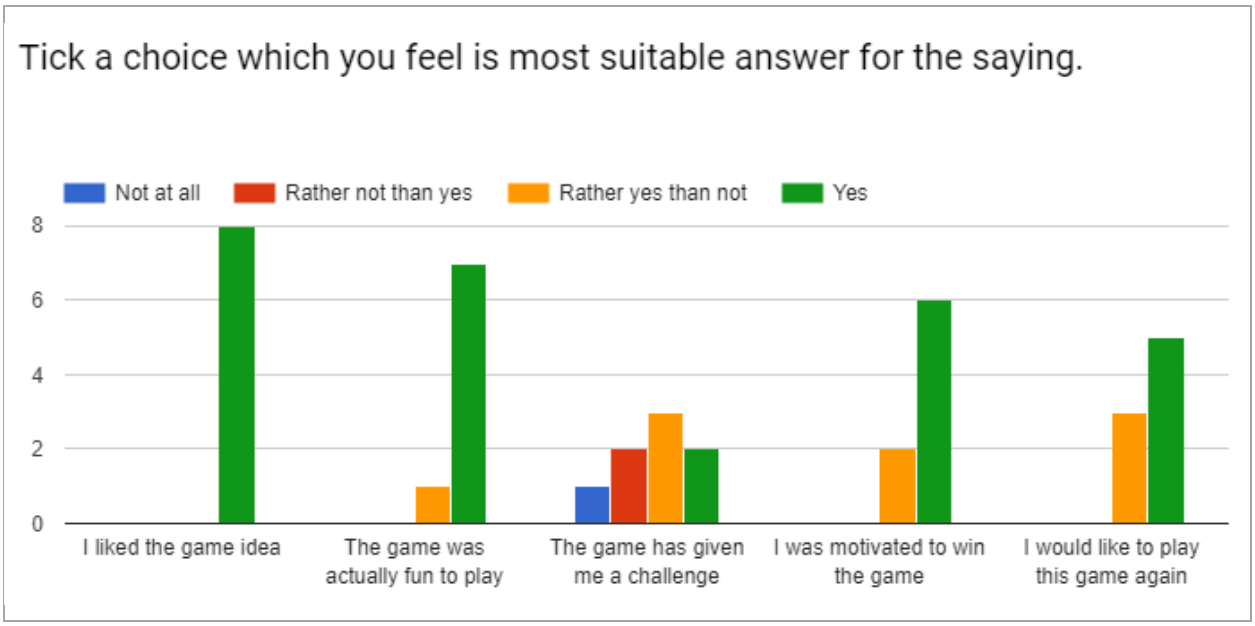

Figure 15: Questionnaire results for the general question.

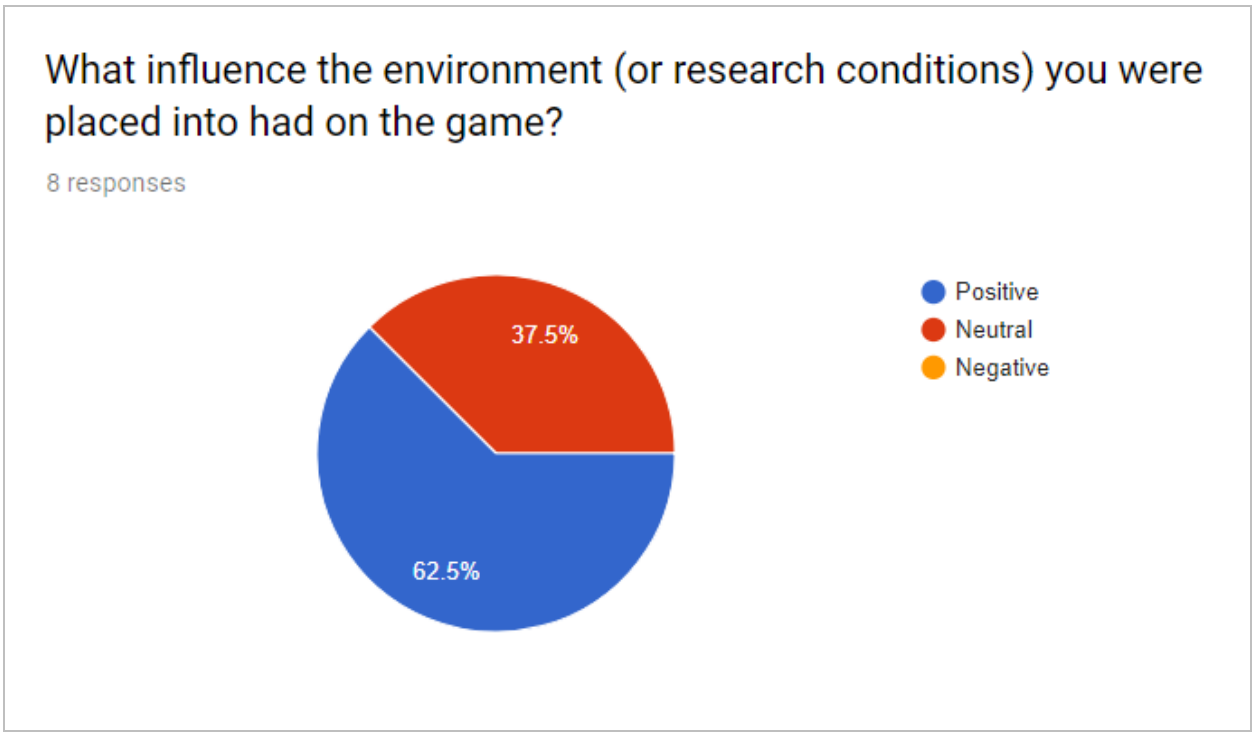

Figure 16: Questionnaire results about influence the external condition had on the game.

The results of the questionnaire showed that players sometimes feel that the action games they are playing are unfair. Half of the research participants have also mentioned that they sometimes think games are unfair only towards them (see **Figure 17**).

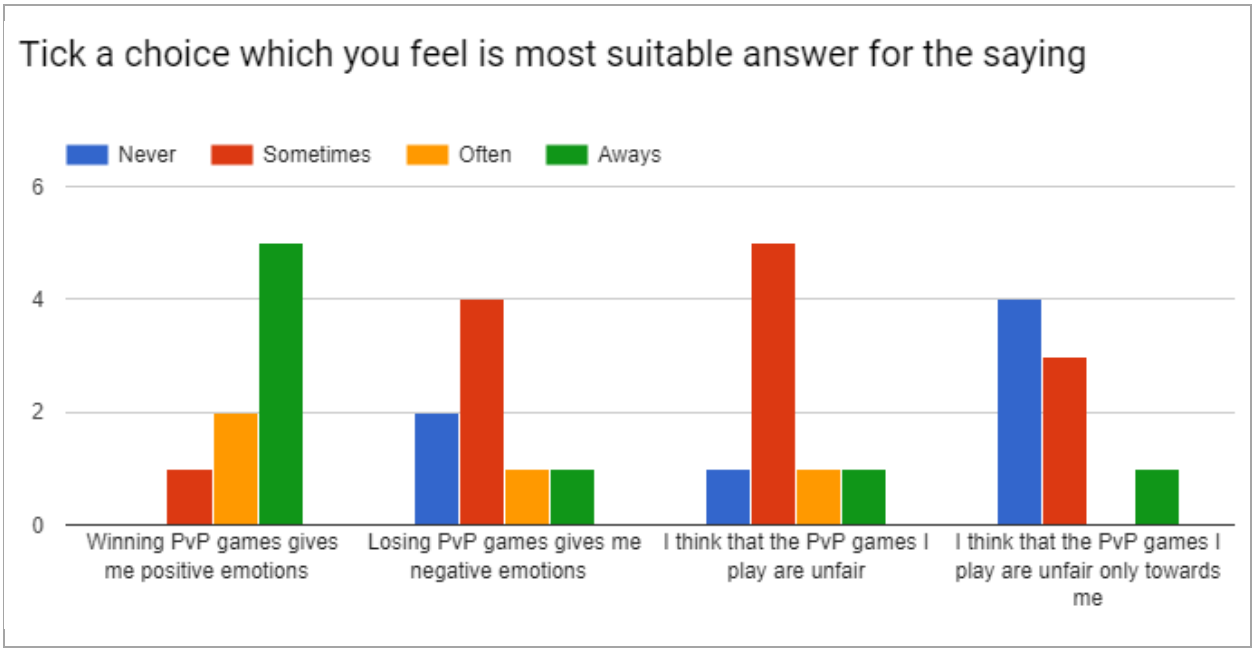

Figure 17: Questionnaire results on previous emotional experience and thoughts about other action PvP video games.

When players compared Swordshot gameplay fairness with other games, they noticed the difference and their responses were in favor of the Swordshot. They said that either the game was completely fair or somewhat fairer than other games. One player stated that the difference in gameplay fairness was unnoticeable. However, Swordshot received no negative responses regarding gameplay fairness compared to other action PvP video games (see **Figure 18**).

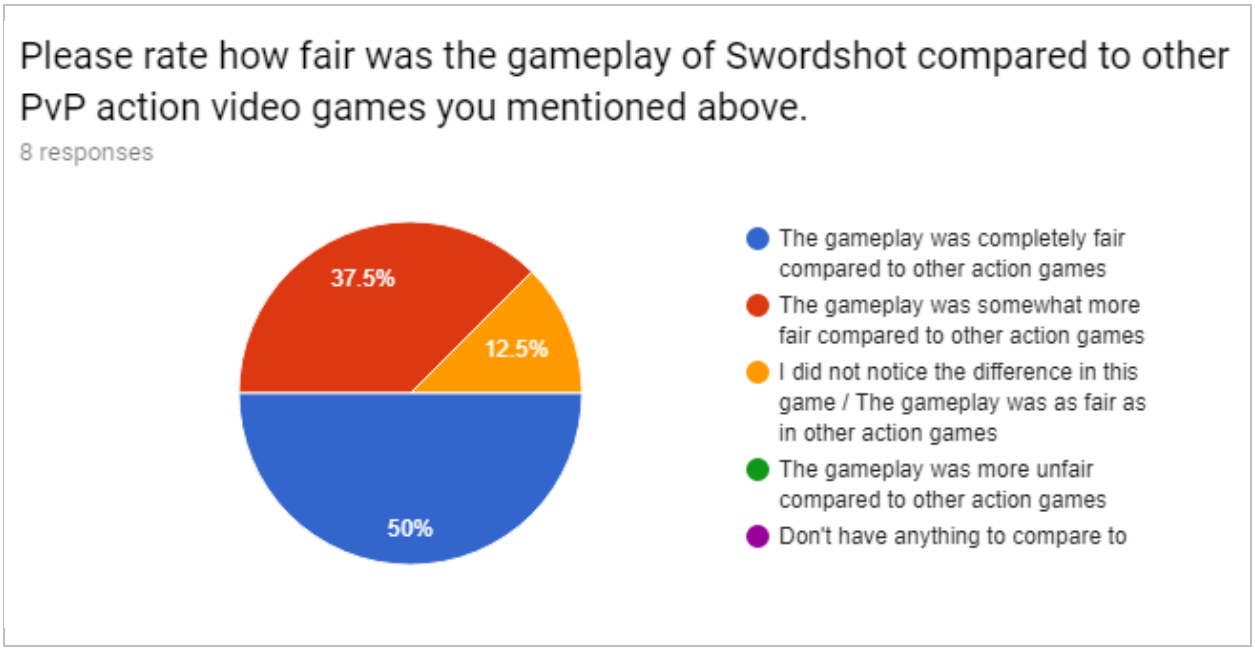

Figure 18: Questionnaire results on Swordshot gameplay fairness comparison to other action PvP video games.

Some people felt that the gameplay was unfair in an aspect. They referred to the different skill levels of the players that were divided into pairs. Stating, that overall gameplay during the experiment was fair, but assuming, that their opposing player had an advantage over them in terms of being more acquainted with the genre of action video games. As all research participants were familiar with some of the other action PvP video games, it was impossible to balance their previous skill level in this genre. Hence, these responses only supported the idea that the outcome of the Swordshot game is heavily dependent on a player's skill.

4.2 Emotions of the research participants

On average, research participants stated that they usually experience positive emotions when winning at other action PvP video games while losing at them sometimes brings up negative emotions (see **Figure 17**). During the experiment, all of the players had experienced positive emotions like happiness, joy, elatedness, and satistafction (see **Figure 19**). This shows that Swordshot managed to bring positive emotions to the research participants providing an average level of challenge. They were unconfident about the gratitude emotion. As for negative emotions (see **Figure 20**), most of the players felt no anger at all and no players felt hate. The minority, who felt anger were those exact people mentioned before, who referenced different player skill levels in action games. These people have lost their games during the experiment. Some research participants were unconfident whether they did experience anxiety or fear, but nobody felt sadness, sorrow nor shame.

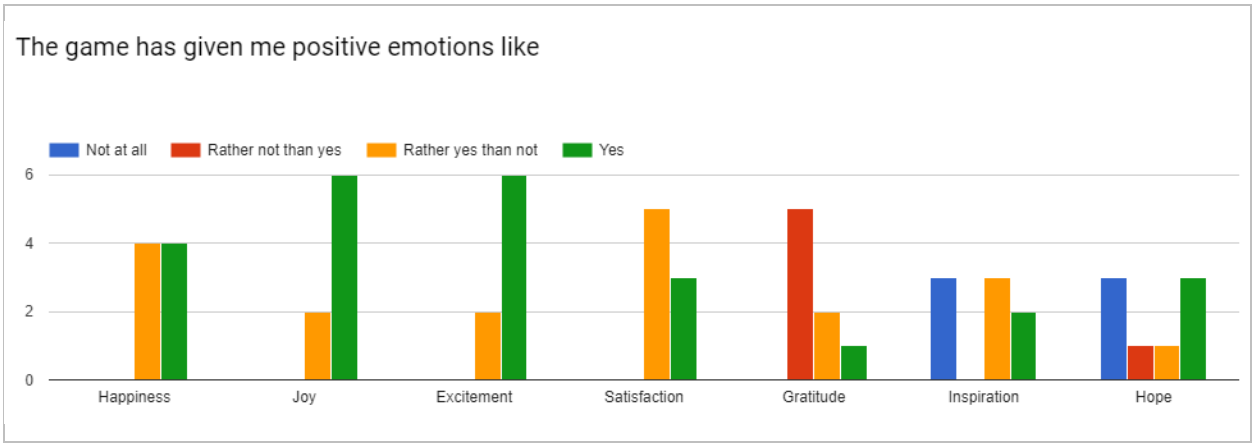

Figure 19: Questionnaire results on positive player emotions during the experiment.

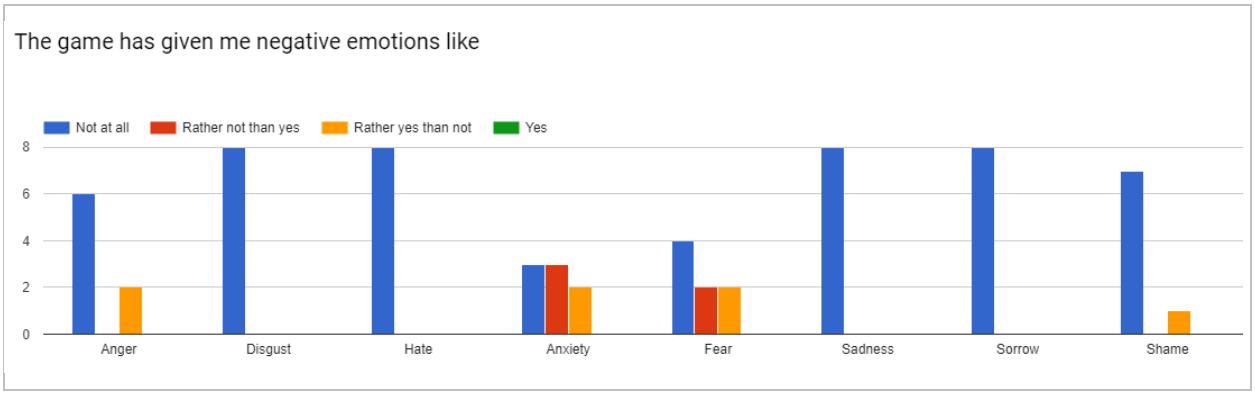

Figure 20: Questionnaire results on negative player emotions during the experiment.

The feedback for the experiment environment to be even more compelling to the game was received. Players stated that more participants or even a tournament would be really appreciated. Also, there could have been series of the games organized rather than one game per pair.

One player has proposed a solution for compensating latency issue in online games. Which is emulate equal latency for all players by creating additional latency in accordance with the real latency of the players. This would possibly mean that the allowed player latency would be limited by the server.

4.3 Conclusions

Going back to the formulated hypothesis *"Players would not feel anger if they knew that a played PvP action game was fair"*, experiment results show that players who considered Swordshot gameplay as fair did not feel anger or hate. This partially proves the hypothesis. The hypothesis cannot be completely proven as during the experiment it was revealed that Swodshot provided only an average level of challenge to the research participants. They were intended to be motivated by the challenge and the achievements. Those, who were only mildly challenged or not at all had possibly lower motivation to win the game and surpass their opponent. Being less motivated may have led to lower emotional activity from the players during the experiment. The fact that nobody felt neither sadness nor sorrow, can potentially mean that players had nothing to lose while playing the game, which is also a motivational factor. It is worth mentioning, that the research participants were friends of the author. All of this could suppress negative emotions in research participants and hence, the hypothesis cannot be completely proven.

Another reason why the hypothesis was proven only partially is that other incentive factors which might cause players to feel anger (e.g. cultural, racial, phobic, etc.) were not considered and addressed in the current thesis. Further testing for all possible incentive factors would be required to fully test the hypothesis.

It can be certainly stated that fair game design factors and external conditions did not enhance anger in the players. Experiment results showed that only the minority of people who had complained the experiment fairness uncertainly felt anger. Moreover, the external conditions might have suppressed the anger emotion levels in the players, due to the fact that multiplayer online games are often played anonymously while being disconnected from the actual people. The disconnection and anonymity may influence negative emotions of the players.

Judging by the results, people were unconfident whether they felt gratitude or not. Players did not express gratitude but did not argue with the outcome of the games. This might possibly be due to low motivation for winning during the playtests.

5 Discussion

The author is happy with the results accomplished in the framework of this thesis. The Swordshot game was fully developed and the experiment was conducted successfully. The game is playable and provides joyful gameplay experience to the players without inducing negative emotions. The game is fully functional and operates using client-server communication. The game is totally free-to-play and can be played by anyone. Simple server application code modifications can enable more than 2 clients to be handled by the server. Game and server application code and releases along with the requirements on computer hardware and instructions are available in the Bitbucket repository [22].

During the research clear answers were given to the raised questions. The formulated hypothesis was partly approved considering that Swordshot provided only an average level of challenge to the players. Looking from another perspective, action games might not even need to be challenging to be enjoyed. But the longevity of the joyfulness is questionable, as such games might become boring quickly.

The biggest drawback in the research was the size of the researched group and their motivation. The author was unable to involve more people in the research as well as provide better motivation for the research participants. Having the low number of participants, experiment results were inaccurate. The topic of Player emotional behavior on PvP games is very relevant considering how many people play them. Continuation of the topic is proposed by the author and further research should be considered.

Thanks to the University of Tartu for providing the knowledge, tools, and the equipment and all the people who took part in the conducted research. Special thanks go to Alo Peets, Anne Villems and the supervisor Margus Luik for helping in the realization of the current thesis.

References

[1] Jamie M. There are 1.8 billion gamers in the world, and PC gaming dominates the market. 2016, 4. https://mygaming.co.za/news/features/89913-there-are-1-8-billion-gamers-in-the-world-and-pc-gaming-dominates-the-market.html (12.05.2018)

[2] World population statistics archive Worldometers. http://www.worldometers.info/world-population/world-population-by-year

[3] Greg H. How are games useful to us? 2017, 4. https://www.quora.com/How-are-games-useful-to-us (12.05.2018)

[4] Ian D., Meena K. 10 Best Selling Video Games in 2018. 2018, 4. https://www.insidermonkey.com/blog/10-best-selling-video-games-in-2018-658164/?singlepage=1 (12.05.2018)

[5] Statistics portal Statista. https://www.statista.com/statistics/251222/most-played-pc-games (12.05.2018)

[6] Nikola V. Dominant video game genres – why are they so appealing? 2013, 4. https://blog.gfk.com/2013/04/dominant-video-game-genres-why-are-they-so-appealing (12.05.2018)

[7] Chirlien P. Understanding Gamer Psychology: Why Do People Play Games? 2017, 1. https://www.sekg.net/gamer-psychology-people-play-games (12.05.2018)

[8] Oxford English dictionary definition for competition. https://en.oxforddictionaries.com/definition/competition

[9] Commentary discussion on topic "Why do some ppl get so angry in games?" https://www.gamespot.com/forums/games-discussion-1000000/why-do-some-ppl-get-so-angry-in-games-31018366/ (12.05.2018)

[10] Commentary discussion on topic "Why do people tend to get angry when they lose in a game?" https://www.quora.com/Why-do-people-tend-to-get-angry-when-they-lose-in-a-game (12.05.2018)

[11] Commentary discussion on topic "I get mad when I play competitive games!" https://www.reddit.com/r/truegaming/comments/48d6rj/i_get_mad_when_i_play_competitive_games/ (12.05.2018)

[12] Commentary discussion on topic "How Can You Accept Defeat Gracefully?" https://boards.na.leagueoflegends.com/en/c/player-behavior-moderation/ErJB3LXA-how-can-you-accept-defeat-gracefully-and-wth-is-happening-to-riot (12.05.2018)

[13] Jean M. T. Yes, Violent Video Games Do Cause Aggression. 2012, 12. https://www.psychologytoday.com/us/blog/our-changing-culture/201212/yes-violent-video-games-do-cause-aggression (12.05.2018)

[14] Andrew K. P., Edward L. D. Scott R., Richard M. R. Competence-Impeding Electronic Games and Players' Aggressive Feelings, Thoughts, and Behaviors. *Journal of Personality and Social Psychology*, 2014, Vol. 106, No. 3, 441– 457.

[15] Oxford English dictionary definition for competition. https://en.oxforddictionaries.com/definition/fair

[16] Commentary discussion on topic “Is chess a fair game?” https://www.quora.com/Is-chess-a-fair-game (12.05.2018)

[17] Commentary discussion on topic “Is the advantage of mouse and keyboard over gamepad in shooters a myth?”. https://www.reddit.com/r/truegaming/comments/28dsy0/is_the_advantage_of_mouse_and_keyboard_over (12.05.2018)

[18] Keith B. Game Design Theory: A New Philosophy for Understanding Games. CRC Press. 2012

[19] WhatIs definition for latency. https://whatis.techtarget.com/definition/latency

[20] Online gaming: The real-time illusion created by client-side prediction. https://www.nbnco.com.au/blog/entertainment/the-real-time-illusion-of-client-side-prediction.html (12.05.2018)

[21] Glenn F. What Every Programmer Needs To Know About Game Networking. 2010, 2. https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking (12.05.2018)

[22] Swordshot game and server application Bitbucket repository. https://bitbucket.org/riks1233/swordshotloputoo

[23] Video game engine Unity. https://unity3d.com (12.05.2018)

[24] Video game engine GameMaker: Studio. https://www.yoyogames.com/gamemaker (12.05.2018)

[25] Java. https://www.java.com/en/ (12.05.2018)

[26] Slick2D Java library. http://slick.ninjacave.com (12.05.2018)

[27] Lightweight Java Game Library. https://www.lwjgl.org (12.05.2018)

[28] Libgdx Java library. https://libgdx.badlogicgames.com (12.05.2018)

[29] Kryonet Java library. https://github.com/EsotericSoftware/kryonet (12.05.2018)

[30] Nadeem U. TCP (Transmission Control Protocol) Explained. 2018, 2. https://www.lifewire.com/tcp-transmission-control-protocol-3426736 (12.05.2018)

[31] Bradley M. User Datagram Protocol. 2018, 4. https://www.lifewire.com/user-datagram-protocol-817976 (12.05.2018)

[32] Google Forms software. https://www.google.com/forms/about (12.05.2018)

Appendices

I. Questionnaire

Swordshot playtest questionnaire

If you are filling out this form, chances are, you have just completed a play test on a game Swordshot. If you have somehow stumbled upon this form and did not play Swordshot, please leave. Otherwise, please be honest and sincere with your answers. Thank you for your participation.

PvP - Player versus Player

* Required

Tick a choice which you feel is most suitable answer for the saying. *

	Not at all	Rather not than yes	Rather yes than not	Yes
I liked the game idea	○	○	○	○
The game was actually fun to play	○	○	○	○
The game has given me a challenge	○	○	○	○
I was motivated to win the game	○	○	○	○
I would like to play this game again	○	○	○	○

(**appendix I** continued)

The game has given me positive emotions like *

	Not at all	Rather not than yes	Rather yes than not	Yes
Happiness	○	○	○	○
Joy	○	○	○	○
Excitement	○	○	○	○
Satisfaction	○	○	○	○
Gratitude	○	○	○	○
Inspiration	○	○	○	○
Hope	○	○	○	○

The game has given me negative emotions like *

	Not at all	Rather not than yes	Rather yes than not	Yes
Anger	○	○	○	○
Disgust	○	○	○	○
Hate	○	○	○	○
Anxiety	○	○	○	○
Fear	○	○	○	○
Sadness	○	○	○	○
Sorrow	○	○	○	○
Shame	○	○	○	○

(**appendix I** continued)

What was the outcome of the game for you? *

- ◯ I have won
- ◯ I have lost

If you feel like you or your opponent had an advantage(-s) over another, please specify them. (Leaving the text field blank means "Me and my opponent were given equal opportunities to win the game").

Your answer

Please name a few of the other action PvP games (that involve fighting or shooting) that you have played yourself. *

Your answer

Tick a choice which you feel is most suitable answer for the saying *

	Never	Sometimes	Often	Aways
Winning PvP games gives me positive emotions?	◯	◯	◯	◯
Losing PvP games gives me negative emotions?	◯	◯	◯	◯
I think that the PvP games I play are unfair	◯	◯	◯	◯
I think that the PvP games I play are unfair only towards me	◯	◯	◯	◯

(**appendix I** continued)

Please rate how fair was the gameplay of Swordshot compared to other PvP action video games you mentioned above. *

- ○ The gameplay was completely fair compared to other action games
- ○ The gameplay was somewhat more fair compared to other action games
- ○ I did not notice the difference in this game / The gameplay was as fair as in other action games
- ○ The gameplay was more unfair compared to other action games
- ○ Don't have anything to compare to

What influence the environment (or research conditions) you were placed into had on the game? *

- ○ Positive
- ○ Neutral
- ○ Negative

What would you do differently in order to make the research conditions more favorable to the game?

Your answer

If you have any suggestions on how to improve game fairness in PvP action games please leave them here.

Your answer

If you have any other comments about the research please leave them here.

Your answer

License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Richardas Keršis**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,
   Player emotional behavior dependency on fair video game design factors and external conditions
   supervised by Margus Luik
2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 12.05.2018