

## Bachelor thesis

# Development of a cross-platform client-server architecture using HTML5 and NodeJS

Submitted by: Huang, Zili

Department: Electrical Engineering and Computer Science

Degree program: Information Technology

First examiner: Herr B.Sc. F. Anthony

Date handing out: 8<sup>th</sup> March 2020

Date handing in: 8<sup>th</sup> June 2020



(Professor Dr. Andreas Hanemann)

Head of Examination Board

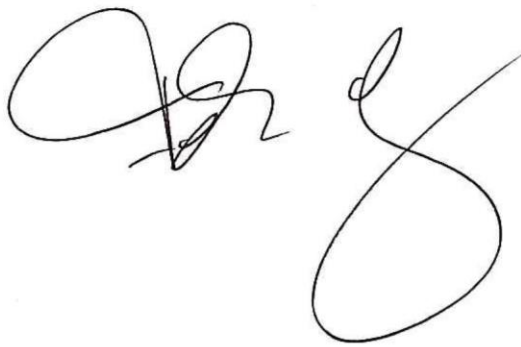
---

**Task description:**

HTML5 has been the major and recommended version of HTML since 2014 and presents features that were specifically introduced for the handling of rich media content (such as graphics, animations and audio) and real-time socket connections, which was previously only possible via web browser plugins - which, in turn, were the cause for various security concerns.

This thesis aims to provide the details of an effective cross-platform (desktop/mobile) client-server architecture by explaining the construction of a semi turned-based multiplayer soccer game exclusively featuring Javascript: a HTML5 frontend, based on the Phaser 3 engine, and a NodeJS backend, extended by the Socket.io framework. A comparison to the alternative Peer-to-Peer architecture will be made, to outline strengths and weaknesses of this approach.

Herr B.Sc. F. Anthony

A handwritten signature in black ink, consisting of a stylized 'F' followed by a large, loopy 'A'.

### Statement on the bachelor thesis

I assure you that I wrote the work independently, without outside help.

Only the indicated sources have been used in the preparation of the work.  
The text or the sense of the text are marked as such.

I agree that my work may be published, in particular that the work may be submitted to third parties for inspection or that copies of the work may be made for forwarding to third parties.

31.05.2020

Date

Huang Zili 董子立

Signature

## **Abstract**

The purpose of this thesis is to develop an effective cross-platform client-server architecture. A semi turn-based multiplayer soccer game exclusively in JavaScript was constructed to provide the details on the development. The thesis project includes three attempts on the implementation, and each of them improved from the previous attempt. A questionnaire was distributed to usability testers to evaluate the complete application and raise areas for enhancement. And the collected data showed a high review on the network structure of the game, specifically for a cross-platform web application based on web browsers. The results provide some support on the convenience of the architectures with NodeJS and HTML5 developed in the thesis, especially in the practical use of a low-cost cross-platform project that demands interaction or cooperation between users.

# Table of Contents

<b>Task Description .....</b>	<b>2</b>
<b>Declaration of the Candidate .....</b>	<b>3</b>
<b>Abstract .....</b>	<b>4</b>
<b>Table of Contents .....</b>	<b>5</b>
<b>1 Introduction .....</b>	<b>7</b>
1.1 Background .....	7
1.2 Introduction .....	8
1.3 The techniques used in the thesis .....	9
1.3.1 Phaser 3.....	9
1.3.2 Express.....	10
1.3.3 Socket.IO .....	10
1.3.4 jsdom and node-canvas.....	10
1.3.5 Path and datauri .....	11
1.3.6 Electron, electron-packager, and OS .....	11
1.4 Thesis structure .....	11
<b>2 Design Phase .....</b>	<b>13</b>
2.1 Design.....	13
2.1.1 Client Design .....	13
2.1.2 Server Design.....	13
2.1.3 Game Design .....	14
2.1.4 Alternative Plan .....	16
2.2 Feasibility test .....	16
2.2.1 Chat room .....	16
2.2.2 Arcade Physics from Phaser 3 .....	18
<b>3 Implementation.....</b>	<b>20</b>
3.1 First Attempt.....	20
3.1.1 Server programming .....	20
3.1.2 Client programming.....	23

3.1.3	Issues and solution .....	26
3.2	Second Attempt .....	27
3.2.1	Server programming .....	27
3.2.2	Client programming.....	31
3.2.3	Issues and solution .....	32
3.3	Third Attempt .....	32
3.3.1	Server programming .....	32
3.3.2	Packaged application .....	35
3.3.3	Issues.....	35
<b>4</b>	<b>Evaluation .....</b>	<b>37</b>
4.1	Usability test.....	37
4.2	Questionnaire .....	37
4.2.1	Test Result.....	37
4.2.2	Data Analysis .....	39
4.3	Further Improvement.....	40
<b>5</b>	<b>Conclusion and Outlook .....</b>	<b>41</b>
5.1	Conclusion.....	41
5.2	Outlook.....	41
	<b>Acknowledgements.....</b>	<b>43</b>
	<b>References .....</b>	<b>44</b>
	<b>Appendix .....</b>	<b>46</b>
	Socket.IO Events Flowcharts.....	46
	List of Figures.....	49
	Usability Test: Instructions .....	50
	Usability Test: Questionnaire.....	51

# 1 Introduction

## 1.1 Background

With the growing network construction in modern life, the demand for using fragmented time with mobile devices is raised. Increasing IO games, whose names end in “.io”, on Google Play or websites and various mini programs on WeChat represent the trend of making the most of fragments of time on the Internet. On the one hand, IO games enable users to start the game within few steps and learn it simply according to the website ‘iogames.space’ [1], which allow users to spend spare time on online games easily on desktop web browsers or installed apps. There are already more than four hundred IO games listed on ‘iogames.space’ and over two hundred on Google Play as of June 2020. Many developers are even testing their Artificial Intelligence (AI) algorithms with some of the IO games due to their reasonable gameplay and high popularity [2] [3]. On the other hand, WeChat mini programs provide different kinds of applications without installing them on smartphones. They use less space on storage and normally cost less time on loading while they still have close functionality compared to installed apps [4]. As a result, users can shop online, play single-player games, even track their risk on infecting COVID-19, etc. in different mini programs via one WeChat application, and switch among programs without effort.

Most IO games on web sites are using completely JavaScript as the programming language or its APIs, e.g. Web Graphics Library (WebGL), as middleware to provide excellent performance together with HTML5 on the modern web browsers. In the meantime, in the light of the official WeChat mini programs’ document, “The Mini Program provides its own view layer description languages WXML and WXSS, and a logic layer framework based on JavaScript.” [5]. In which cases, WXML and WXSS are like HTML (Hypertext Markup Language) and CSS (Cascading Style Sheets) in behaviors, but adjusted for WeChat intentionally. As can be seen from the above, the combination of HTML and JavaScript still plays an important role in the development of web applications, and thus this thesis was determined to focus on topics around HTML and JavaScript.

At present, IO games on Google Play, mini games on WeChat, or other web games on similar platforms have already had a stable structure that allows users to play online during the fragmented time with mobile devices effortlessly. However, a large part of these mobile games is multiplayer game that need installed apps or desktop browsers, or single-player games run by Web Viewer or alike apps, such as WeChat mini programs or web browsers. Few online multiplayer games can be simply accessed by browsers across platforms with great experiences, and fewer of them handle relatively much data processing. In conclusion, only a handful of those online games can achieve multiplayer and running via the cross-platform web browser at the same time. Online board games can be a cross-platform multiplayer game that run on not only installed apps but also various browsers. For example, “Majsoul”, a Japanese Mahjong game, can be played by either using web browsers on PC and mobile phones, or app on mobile phones, which is a customized and packaged web browser. However, “Majsoul”, and many other cross-platform online multiplayer games, does not contain complex data computation, e.g.

physics engine and rich media transmission. Under these circumstances, it is a potential attempt for using existing techniques to implement a cross-platform and browser-based web application. In specific, developing a reasonable architecture for an online multiplayer game across platforms on browsers that contains physics calculation could be a suitable choice for a bachelor thesis topic.

As the current mainstream HTML version, the 5th version of HTML (HTML5) supports more types of element content, which allows it to accept more external frameworks or plugins, enabling more complex operations and programs in the webpage than the previous generation. Besides, it is also optimized for low-performance devices, which lays the foundation for applications to be used across platforms. Apart from that, since a cross-platform application requires a smooth run on less powerful devices as well as a multiplayer game could use a server to prevent cheating or broadcast messages, a client-server architecture from distributed systems can be applied to reduce the workloads of data processing on clients. And it thus enables the application to run across platforms well. To simplify the programming process during the thesis by coding JavaScript on both client and server, NodeJS was selected. With this feature, it is very convenient to convert client-side code into a server-side one to reduce the workloads on clients when encountering a client-data-processing bottleneck, which greatly reduces the time on refactoring codes comparing to using two or more different languages for client and server. In conclusion, the combination of HTML5 and NodeJS with a client-server architecture is very likely to enable weak-computing-power devices to run cross-platform project without much effort and present similar performance as desktop web browsers.

Besides these popular techniques, Phaser 3, a 2D game framework, was chosen to implement an HTML5-based game. There is one project that has a close objective with this bachelor thesis topic, whose topic is “Phaser 3 - Real-Time Multiplayer example with Physics” [6]. It is one of the rare and shared examples that focus on calculating physics with Phaser 3 on the server. The differences between this thesis and that project are the programming language and multiplayer setting. The thesis project will use JavaScript and allows different users to interact through physical collision instead of using TypeScript and being a single-player-like multiplayer game.

## **1.2 Introduction**

This thesis aims to provide the details of an effective cross-platform (desktop and mobile) client-server architecture by explaining the construction of a semi turned-based multiplayer soccer game exclusively featuring JavaScript: an HTML5 frontend, based on the Phaser 3 engine for building an HTML5 game, and a NodeJS backend, extended mainly by the Socket.IO framework to establish bidirectional network communication. The clients are supposed to base on web browsers from both PC and mobile phones, and users can access the client by simply entering the address in the URL bar of the browser. In this way, downloading the full installation or compressed package is not necessary, which simplify the usage of the whole application. A comparison to the alternative Peer-to-Peer architecture will be made, to outline the strengths and weaknesses of this approach. Through the multiplayer game developed and tested on this thesis project, a better understanding of whether a webpage application based on HTML5 and used client-server architecture is capable of cross-platform usage can be comprehended. Issues



met during the thesis and corresponding solutions can be useful examples or one of the possible methods for further topic-related projects.

The HTML5 has been the major and recommended version of HTML since 2014, which improves the ability to handle graphics elements and real-time communication by introducing the canvas element and WebSocket protocol. As of June 3, 2020, 82.9% of all websites are using HTML5 by W3Techs.com [7]. According to the introductions on WHATWG, “The canvas element provides scripts with a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, art, or other visual images on the fly.” [8]. It allows devices with low computing performance, such as mobile phones, can run relatively complex 2D graphs or 3D graphics with the help of WebGL on an HTML5-supported web browser without plugins. In this situation, increasing frameworks based on HTML5 can be run on both mobile and desktop devices. Meanwhile, the WebSocket interface “enables web applications to maintain bidirectional communications with server-side processes” [9], which saves the server resources and bandwidth and allows more real-time communication.

Node.js® is a JavaScript runtime environment that allows developers to write code for servers using JavaScript instead of learning another server-side language. By using the V8 JavaScript engine from Google Chrome, NodeJS performs great and is well compatible with the ECMAScript standard, which standardizes JavaScript. On an account of being an open-source and cross-platform environment, Node.js® owns a massive number of libraries that are free to use, which in turn makes programming server-side JavaScript codes more convenient. [10] Implementing server with NodeJS lowers the requirements for making an independent and small web project by only mastering in JavaScript.

The distributed system still is an essential part of web application deployment. “Distributed Systems: Concepts and Design” (2005) describes its main motivation as sharing the resources in a network system which leads to the concurrency of components, the lack of a global clock and the independent failures of components [11]. Comparing to peer-to-peer architecture, client-server architecture does not require clients to share resources with other clients, while it also has a comparatively simpler network structure than n-tier architecture. Its clients are mainly responsible for displaying data requested from the server and sending input data collected from users back to the server. As a result, the applications using client-server architecture have less load on clients, and thus are more capable of operating on various devices, especially devices with weak computing power.

## **1.3 The techniques used in the thesis**

For the purpose of programming a cross-platform multiplayer game, despite HTML5 and NodeJS mentioned and introduced in the previous chapter, other libraries or frameworks are used in the final application as well. Brief introductions are provided in the following sections.

### **1.3.1 Phaser 3**

Phaser 3 is a game framework built with HTML5 techniques, whose goal is to build powerful and cross-platform HTML5 games on browsers with ease. With the help of HTML5, Phaser 3

makes great use of the benefits of modern web browsers, especially on mobile. [12] According to Phaser 3 official website, it specifically focuses on the performances of mobile browsers, since any low-performance features will not be added into the nucleus. The Phaser 3 game will be rendered with either Canvas or WebGL, based on the support of current web browser, which increases the compatibility for various browsers on both desktop and mobile. Besides, the pointer class of Phaser 3 allows inputs from both touch-screen and mouse to work nearly the same, and even swap in the mid-game, which are perfect for a cross-platform game. Additionally, Phaser 3 has two built-in physics systems: a highly light-weight AABB (Axis-Aligned Bounding Box) library for perfect performance on low-power devices called Arcade Physics, and a full-body third-part system for vast but precise physical phenomena simulation named Matter Physics. [13] However, the HEADLESS renderer mode is the most important feature used in the thesis that allows Phaser game to run without rendering, although the support of canvas is still necessary.

### 1.3.2 Express

Express is a concise and flexible NodeJS web framework that provides a series of features and HTTP methods for developers to build various web applications. Express can handle various HTTP verbs in separate routes, use plenty of middleware packages maintained by Express team or third party simultaneously to solve numerous web development issues, and create dynamical HTML pages for browsers to display. Through calling the only built-in middleware, 'express.static', Express can server different static files, such as JavaScript files, images, and audio files, from multiple directories with an optional path prefix, which allows web applications to use local files.

### 1.3.3 Socket.IO

Socket.IO is a library that achieves real-time, bidirectional, and event-based communication between the browser and the server. It consists of a NodeJS HTTP server and a JavaScript client library for the browser. Main features used in the thesis include reliable bidirectional connection, auto-reconnection and disconnection support, and wide browsers support. Although Socket.IO uses WebSocket as a mode of transport when possible, some metadata is added to each packet for efficient transmission, yet it means that a WebSocket client cannot connect to a Socket.IO server, vice versa. In return, compatibility is massively increased by starting a connection as xhr-polling, and switching to WebSocket later if the protocol is supported by current web browsers. [14] Socket.IO allows developers to customize the events for triggering and responding from both sides along with serializable data, as well as take advantages of WebSocket. Despite several predefined event names, this feature makes communication events more scalable and distinguishable.

### 1.3.4 jsdom and node-canvas

jsdom is a JavaScript library of the WHATWG DOM and HTML Standard particularly for the utilization in NodeJS. Its target is to "emulate enough of a subset of a web browser to be useful for testing and scraping real-world application". By instantiating a JSDOM object, dozens of attributes and functions from the DOM APIs can be used in a NodeJS application. Apart from

that, jsdom also allows running internal and external JavaScript codes by using `<script>` tag or methods from DOM with the same effect. [15] The most notable capability of jsdom during the thesis is to simulate as a visible browser and provide canvas tag via including other canvas packages. Under this circumstance, node-canvas, which is an implementation of the Web Canvas APIs based on Cairo (a 2D graphics library) for NodeJS, is chosen [16].

### 1.3.5 Path and datauri

The Path module in NodeJS allows the usages of file or directory paths in the application. [17]

datauri is a NodeJS library used to generate a data URI scheme to import data to web pages like external resources [18]. It is required in the application for compensating the unimplemented Blob URLs methods in jsdom. In this case, `'URL.createObjectURL()'` and `'URL.revokeObjectURL()'` are necessary for Phaser 3 to build instances with strings containing a URL, such as images and sprites in Arcade Physics.

### 1.3.6 Electron, electron-packager, and OS

Electron is a framework made by GitHub, which allows the developer to build desktop applications with web technologies, such as JavaScript, CSS (Cascading Style Sheets) and HTML. Many famous programs like Atom, Visual Studio Code and WhatsApp are mainly or partially made with Electron. The NodeJS libraries and the Chromium kernel are integrated into Electron for running and building across platforms, including Windows, Mac, and Linux. [19] A simple web application can be turned into a native program by adding electron libraries into node modules, changing a few codes for starting an application with Electron instead of NodeJS and adjusting the structure to suit the requirement of Electron.

electron-packager is one of the official recommended tools for packaging the Electron applications with selected node modules through command line or JavaScript codes. It allows developers to “generate executables or bundles” for Windows, macOS or Linux. [20]

The OS module in NodeJS provides functions and properties that are relevant to the current operating systems [21]. For example, it is used to get the local IP address of the server with methods `“os.networkInterfaces()”`, which works similar to `“ipconfig”` on the command line.

## 1.4 Thesis structure

The main content of the thesis includes five chapter. The first one, which is this chapter, mostly introduces the background and information around the thesis topic along with a detailed introduction on techniques used in the thesis project. The following chapter focuses on designing the application to achieve the thesis target and testing the feasibility of the design. The third chapter involves the process of implementing the application and the issues met during the implementation. Each of the three attempts tries to solve the raised problems from users' tests or previous attempt and improve the quality of the application. The fourth chapter evaluates the application from the second implementation through a usability test with thirteen applicants, thus improving the third implementation with its feedback. The fifth chapter

concludes the thesis and looks ahead for further enhancement on the thesis project and more practical usages on the thesis topic. The chapters following the main content are acknowledgements, references, and the appendix involved details of used figures and usability test.

## 2 Design Phase

### 2.1 Design

For full preparation on the topic, a simple design phase was processed before the programming codes. It covered four parts, including designing a webpage that can be operated via various web browser on different devices, designing a server that supports client-server architecture and bases on NodeJS, formulating a basic game template and its logical rules, and researching an alternative plan for an unexpected situation. To shorten the contents, the instances of football and players in the game will be referred to ‘objects’ when discussing on the client.

#### 2.1.1 Client Design

Since the purpose is to make a cross-platform webpage game, the game framework running on the client needs to be performed well on both mobile and desktop browsers. Fortunately, Phaser 3 has fully considered and utilized the advantages of HTML5 on weak computing devices, and can adapt to distinct calculating power on different platforms. By configuring the configurations when declaring a new Phaser 3 game instance, Phaser 3 allows the game to resize with defined ratio and center itself to adapt various sizes of the screen. This eliminates the hassle of setting separately in the HTML code for different types of web browsers, and thereby enhances the compatibility with various devices.

Phaser 3 framework includes two physics systems, which are Arcade Physics and Matter Physics. On the one hand, Arcade Physics is implemented by the developer of Phaser with AABB and bounding spheres algorithms. It only supports simple physical calculations, such as velocity, acceleration, friction, and collision computing of rectangles and circles, but it has a very fast calculation speed as a compensation. On the other hand, Matter Physics is a third-party physics engine, named Matter.js, integrated into Phaser 3. It has more powerful physical algorithms, making the physical performance of objects very close to the real world. As a result, it runs slower, but can achieve many realistic physical actions that Arcade Physics is not possible to do, e.g. impact of spinning on movement, elastic deformation and collision, and polygonal physical body [22]. Since the game only needs the collision calculation of circles and rectangles, Arcade Physics was chosen in this project to save computing power.

#### 2.1.2 Server Design

After researching diverse tutorials relevant to Phaser 3 game implementation, the server decided to utilize Express to help create an HTTP server and use Socket.IO to achieve real-time communication between clients and server. The collaboration of NodeJS, Express, and Socket.IO is so mature that enough tutorials and discussions can be referred to when programming and encountering server-related issues. Web Real-Time Communication (WebRTC) protocol, which allows establishing peer-to-peer communications between RTCPeerConnection instances in different browsers [23], was considered as a possible solution. Additionally, there are many libraries on npm that allow NodeJS application to establish a peer-to-peer connection. However, comparing to Socket.IO, which mainly uses WebSocket protocol

to establish communications, WebRTC-relevant libraries lacks both tutorials or discussions and stable compatibility. In some cases, a forwarding server that requires Socket.IO library is still necessary to start the connection between peers on browsers. As a result, the learning cost of the codes is higher than building a client-server architecture.

In addition to the determination on necessary techniques, the server will also store logic for a non-character player (NPC) to take over the player's control if there are not enough users to control all players. It should be able to "attack" or "defend" according to the situation on the football field, while it may not shoot the football so precisely that users on the opponent team have little chance to beat NPC. The detailed logic is introduced at the end of chapter 3.1.1.

### 2.1.3 Game Design

The game is mainly composed of two parts, one is the game area in the center of the screen, presenting the main game content, and the second is the user interface (UI), used to place buttons and display game-related text and images.

Before every design on the game starts, basic game rules had to be set up. Traditional soccer games used to be a real-time game that both users control their team at the same time, but it would be impossible for mobile and desktop users to compete against each other. So, the target of the thesis is to make a turn-based game which allows mobile devices to have the same performance as a desktop device. In this case, the game is similar to the billiards but with the rules close to soccer, and hence the shape of players is simplified to a circle. The player should be able to control the movement direction by dragging an arrow to visualize user's input remarkably, since it is not convenient to notice the value of input under the finger on a touch screen (Figure 1 & Figure 2). Concerning the number of users, the game is determined to let four users control two players each in one game as a result of the desire to test the potential of client-server architecture, the limited screen size of many mobile devices, and the probability of having more fun with more choices in every round. Finally, the goal is decided to be inside the football field due to the limitation of the impossible polygonal physical body of Arcade Physics. Because the football field can be predicted to be small, the goal should not be much larger than the player. For the same reason, the teams of the first two users who started the round should be placed away from the football to prevent scoring with the first shot.

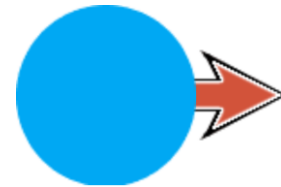


Figure 1: The circular player with an arrow in initial direction

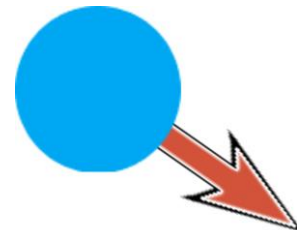


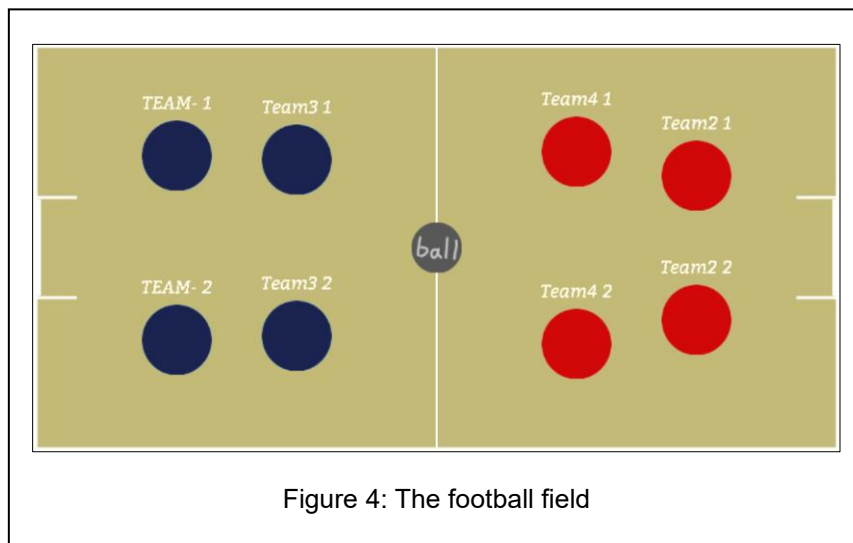
Figure 2: The arrow is dragged to another direction



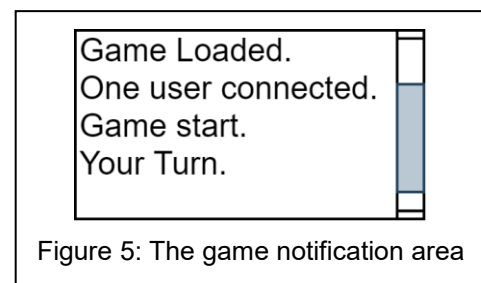
Figure 3: The circular football

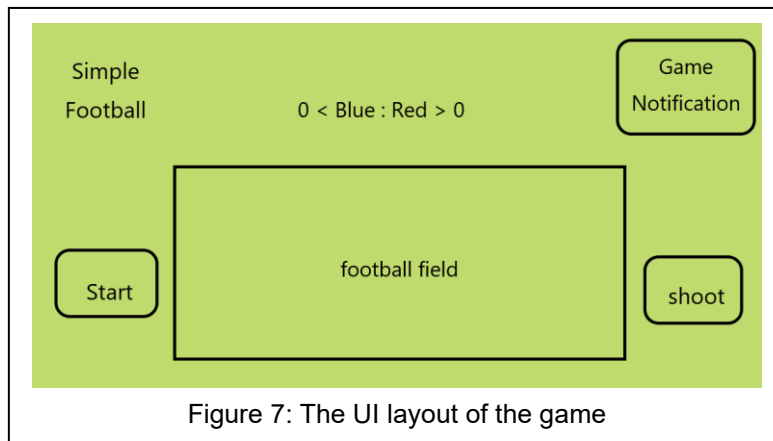
The size of the game had to be decided after the design of game rules. At present, the most popular ratio of the screen on the desktop is still 16:9 and many all-screen smartphones tend to have a longer ratio than 16:9 to make up the notch. Considering the size of the address bar above the web browser, a ratio longer than 16:9 would be a good choice to make full use of the

space on browsers. In this case, the ratio with 2:1 and the resolution of 1200\*600 is determined. With the help of Phaser 3 setting, the game can zoom in and out to fit the size and resolution of current browsers. As for the football field, which is the main input area, its size cannot be so big that GUI would not fit, while a small size could make it hard to input the proper command. In the end, the football field is chosen to be 800 pixels wide and 400 pixels high after many drafts and is placed in the lower middle. To achieve the goal of having enough moving space for eight players and one football, the player is set to be seventy pixels in radius, and the football is relatively smaller, which is fifty pixels in radius. In this case, the width of the goal is fixed to be 100 pixels. After applying the rules to player's positions and plenty of minor adjustments, the final design on the football field is made and shown in Figure 4.



Finally, the rest of the space is used to place the UI. The notification area at the upper right corner should include useful information for users. It needs to display the status of the users like “A user connects” or “A user disconnected”, and the notices of game status such as “Game starts” or “Your turn” (Figure 5). The top of the screen will display the current score, while the upper left corner will show the title of the game. On both sides of the screen will lay the buttons for uploading input and starting the game, which may also show the number of current connected users. And the result of all components placed in the game could be like Figure 7. As for the animation for celebrating the goal, it will show in the middle of the screen when any user goal. Figure 6 exhibits one frame in the whole animation.





#### 2.1.4 Alternative Plan

Since the consistency of the physical calculations with Arcade Physics on different clients cannot be guaranteed, and the probability of successfully eliminating errors on positions with location synchronization solution is not sure, a backup plan is designed to replace the original one as fast as possible. In the alternative, the server will not broadcast every user input, but directly broadcast the specific location of all movable objects in each game update loop. Then the clients will straightly place or move corresponding objects on received positions. To reach this effect, the HEADLESS mode of Phaser 3, which is used to run Phaser 3 game without rendering graphics, is necessary. And detailed researches [24] [25] suggest that more libraries are required to meet the demand of specific variables for Phaser 3 to run at a server. Additional libraries include node-canvas, jsdom, and datauri, as mentioned in chapter 1.3. Meanwhile, most parts of the codes will be reused to save time and reduce the difficulties in refactoring and rebuilding.

## 2.2 Feasibility test

In order to test whether the design will work, the feasibility test is planned. The test involves two parts. The first one focuses on the data transmission capability of the server built on NodeJS and its libraries. A chat room server will be built with NodeJS, Express, and Socket.IO and tested with its transmission latency and speed in sending rich-media files to verify how NodeJS performs under local area network (LAN). The second test concentrates on the performance of Arcade Physics from Phaser 3. After the prototype game is completed, the test will be carried on by sending the same input to different clients through the server at the same time, and the target is to analyze whether odd results from the physics computing, which could be an excessive difference in objects' positions, exists. This test will determine how clients and server communicate with each other.

#### 2.2.1 Chat room

A completed chat room is built with two parts, one is the server and the other is the webpage as the clients. The server is made with NodeJS, Express, Socket.IO, body-parser, and multer, and is responsible for broadcasting messages between different clients and storing pictures sent by



the clients. Among those libraries, body-parser allows the HTTP server to parse POST request, and multer enables the client to send image files to the server. The client webpage mainly uses Ajax methods from jQuery and Socket.IO methods to send and receive messages. Each client is distinguished by setting its cookie with its creation time. And the layout of the webpage is shown in Figure 11.

### ServerTest/

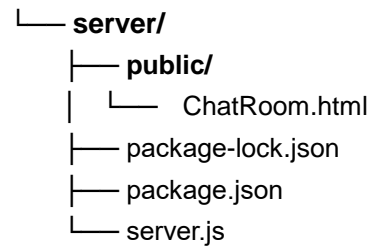


Figure 8 The file structure of chat room app  
(The file structure diagrams in this thesis were all built with 'tree-node-cli' module [26].)

```

<1> var fs = require('fs');
<2> var bodyParser = require('body-parser');
<3> var multer = require('multer');
<4> app.use(bodyParser.urlencoded({ extended: false }));
<5> app.use(multer({ dest: '/tmp/' }).array('image'));
  
```

Figure 9: Required modules and used methods for transmitting images

```

<1> app.post('/sending', function (req, res) {
<2>     var time = req.body.time;
<3>     var text = req.body.content;
<4>     var id = req.body.pageid;
<5>     console.log(text);
<6>     var serverDate = new Date();
<7>     var serverTime = serverDate.getTime();
<8>     var latency = serverTime - time;
<9>     io.emit('messages', [latency, serverTime, id]);
<10>    res.end();
<11> });
  
```

Figure 10: Codes of using Ajax method to receive messages from clients and using Socket.IO method to send message back to client

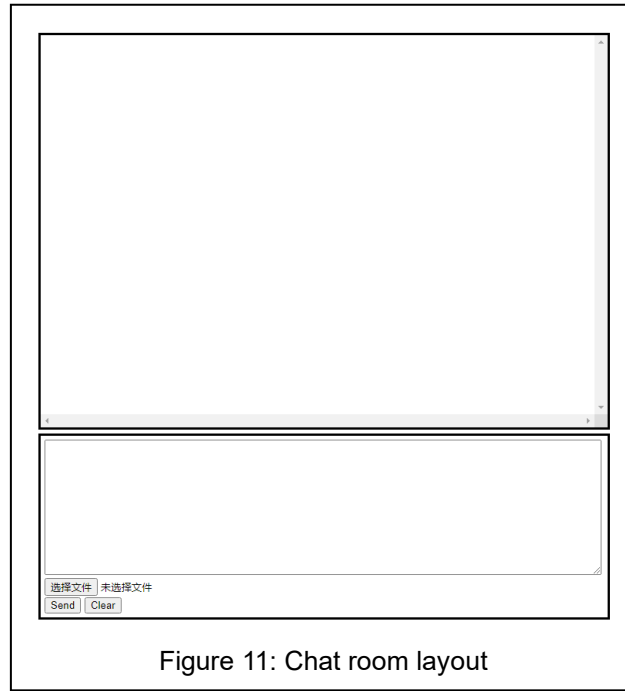


Figure 11: Chat room layout

The test method is to calculate the time required for the client to send a picture or texts to the server and the time required for the server to return the results to one client. By uploading the data to chat room server, including the picture, texts, a customized cookie, and a timestamp that recorded the uploaded time, via Ajax POST method, the server calculates the time gap between uploaded time and received time and send it back to clients via Socket.IO methods with sent time and received cookie. Then each client will calculate its second time gap. and display the cookie, the first time gap, and the second time gap on the webpage. Each message is displayed in an ordered list with the following order: the client's cookie, the first time gap, and the second time gap (Figure 12).

```
1. createtime=1591017117706: 13, 2
2. createtime=1591017117706: 11, 1
3. createtime=1591017117706: 8, 1
4. createtime=1591017117706: 7, 2
5. createtime=1591017117706: 207, 1
6. createtime=1591017117706: 118, 1
```

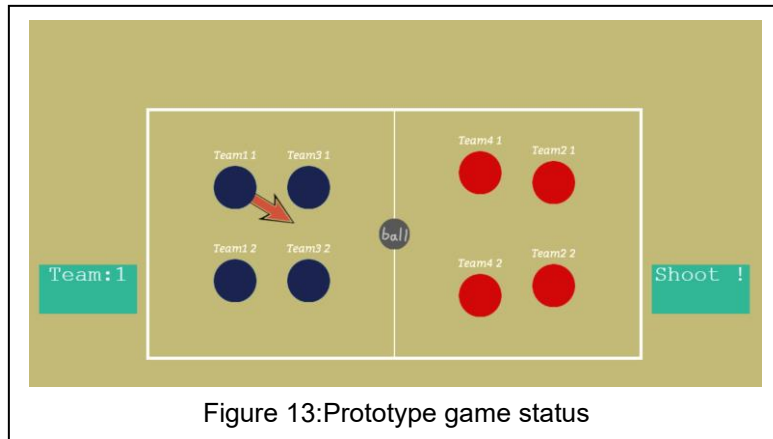
Figure 12: The output of the time gaps  
between server and client  
(The first is Ajax, the second is Socket.IO)

As the screenshot in Figure 12 shows, the two time gaps between sending and receiving data without rich-media files are small, especially when using Socket.IO methods. Meanwhile, the time gap with a big picture (approximately 2 MB) is relatively larger due to the network limitation. And the most important thing is no packet loss during the whole test. The result is cheerful since no continuous media transmission is required in the game design, which further increases the reliability of message changing.

### 2.2.2 Arcade Physics from Phaser 3

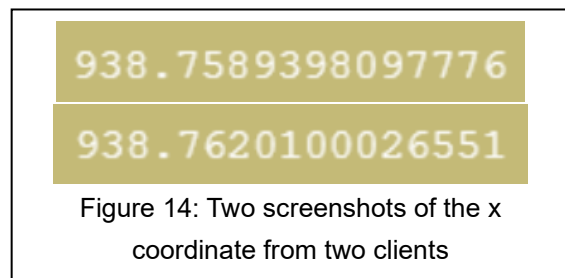
When the test was executed, the game had completed the basic construction of the server and simple game setup. The server was able to identify and record the login data and operation of different users, which could calculate the initial velocity of the user-selected player according to user's input on client and broadcast the result to all clients. Meanwhile, the client was able to collect user's input on player's direction and moving force to send to the server, and use

returned result to set the initial velocity of the player. The game area could not detect the event of goal, but only the collision among players, ball, and the bound of the game area.



The test methods include the following steps. First, after the clients run the result from the server and detect that all objects stop moving, they send a message to the server, which is an array that contains all positions of players and ball. Then the server will compare all clients' positions and output the result (true/false) in the command line. At the same time, the client will show the first player's location at the top-left corner of the display area, which could be used to contrast the locations of all clients directly.

The test result indicates that the differences in final objects' positions between two clients on different devices are quite small. Though most of the outputs on the command line are false, the divergence is always tiny and happens after two decimal digits such as the coordinate in Figure 14. In this case, adding a simple function for computing the average positions of all connected clients and forcing them to synchronize before every round starts would be enough for the tiny deviation in this test. As a result, the communication method between clients and server was determined to be exchanging game status before and after a round to reduce transmission frequency, since Arcade Physics can output close and stable positions with the same input after a series of physical computing.



Although Arcade Physics of Phaser 3 showed very close performances to the physical computing results on different clients in this test, as the client needs to process more data with growing game content, the differences between client's results enlarged. This indicates that Phaser 3 still cannot reproduce every physics calculation results among clients with different calculation capabilities. The conclusion of this test may not take the impact of heavy load on the game update loop into consideration. Regardless of whether the same browser opens different webpages, different browsers open different webpages, or different devices open different webpages, there is a high probability that the differences in positions are so large that it cannot be eliminated by simply computing the average position.

## 3 Implementation

### 3.1 First Attempt

The structure of the whole files is shown in Figure 15. Among those files and folders, ‘package-lock.json’ and ‘package.json’ are the configuration or instruction for using NodeJS that makes the projects reproducible for others, while ‘node\_modules’ folder contains both basic libraries and added frameworks of NodeJS required in the application. These three files remain the same usage in the following attempt and will not be mentioned in the following introduction.

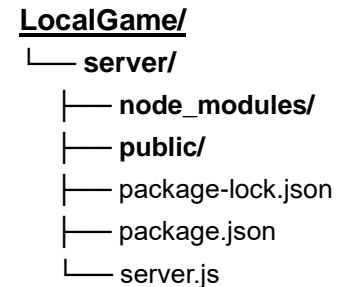


Figure 15: The file structure of the first attempt

#### 3.1.1 Server programming

The server is programmed in file ‘server.js’. The code could be separated into several parts, including requiring NodeJS libraries, declaring essential variables, setting socket events with Socket.IO methods, and finally starting the server. In the first part of the codes, packages of Express and Socket.IO are required. Then the folder path of local files for clients to access and the file path of the main page are set.

```
<1> var express = require('express');
<2> var app = express();
<3> var server = require('http').Server(app);
<4> var io = require('socket.io').listen(server);
```

Figure 16: Required modules for a HTTP server

```
<1> app.use(express.static(__dirname + '/public'));
<2> app.get('/', function(req, res) {
<3>   res.sendFile(__dirname + '/index.html');
<4> });
```

Figure 17: Define the file path for client to access

Some global variables are declared to record users’ data and game status:

- ‘userNum’ is an integer with an initial value of 0, and records the number of clients currently connected to the server.
- ‘gameStatus’ is an object with four properties to record the status of the game, including ‘gameStart’ for whether the game starts, ‘gameTurn’ for the number of game rounds, ‘gameScore’ for the game score and ‘waitNextTurn’ for whether it is waiting for the client to respond.

- ‘shootSpeed’ defines half of the maximum initial speed that every player can move, which is 800 pixels per second, at server side. It prevents the possibility that modified client files cause all selected players on clients to exceed the speed limit.
- ‘users’ is an array that records all users’ data from connected clients that are currently playing the game. The array item is an object with seven properties, and is made up of the socket id of the connected client, user’s nickname, user’s team number, whether the client is ready to start the game, whether the client has ended the current round, which team achieves goal, and the position of all objects in this round.

```

<1>  var users = [];
<2>  for (var i = 0; i < 4; i++) {
<3>      users.push({
<4>          userSocketID: null,
<5>          userName: ('Team' + (i + 1)),
<6>          teamNum: i,
<7>          startReady: false,
<8>          turnReady: false,
<9>          goalMessage: null,
<10>         position: null
<11>     });
<12> }

```

Figure 18: The properties of ‘users’

The server contains a few events of Socket.IO for the communication between clients and server. Flowcharts for explaining the detailed order of these events are listed in the appendix, and the corresponding number of the figure that contains the introducing event will be noted at the end of each following paragraph. The flowcharts for the first attempt are Figure 49, Figure 50, Figure 51, and Figure 52.

- ‘connect’ is a reserved event predefined by Socket.IO and will be fired upon a connection from the client. This event takes charge of the data update when a new client is connected. If this is the fifth or later connected client, the server will send an event called ‘over4’ to it; if not, the server will first update the number of connected clients, and then will trigger the ‘loginData’ event to the current client as well as send the present game condition. According to the contents returned by the client, the server will update the corresponding user nickname of users first. Then, If the current client is the first to connect, the server will additionally set all ‘position’ properties in ‘users’ to the position received from the client. Finally, the server will fire ‘currentStates’ event to the current client with complete values of ‘users’, and ‘newUser’ event to rest connected clients with current client’s data in ‘users’. (Figure 49)
- ‘disconnect’ is a reserved event predefined by Socket.IO and will be fired upon disconnection from the client. This event will initialize every data of the disconnected client in ‘users’, and broadcast the event called ‘playerDown’ to the rest client(s). If the current server has no client connected, ‘gameStatus’ will be initialized; if not, the server will examine whether the disconnected client has unfinished steps, and complete them if they exist. (Figure 49)
- ‘toStartGame’ will be fired when a client is ready to start a game. The server will set its corresponding ‘startReady’ property in ‘users’ to true, and check whether all connected clients have sent this signal. If this is the case, the server will broadcast the ‘startingGame’ event and reset all ‘startReady’ properties in ‘users’ to false. If not, the server will broadcast

the 'waitingGame' event with the number of ready clients and not assigned teams for clients to display. (Figure 49)

- 'toShootBall' will be fired when the client, which is operating in the current round, determines to move the player. The data received from the client contains the socket id, assigned team number, the player number and vectorized speed and force in a vector object of Phaser. The server multiplies the vector with defined 'shootSpeed' and broadcasts it with other received data to all client within the 'shootingBall' event. Then the 'waitNextTurn' property of 'gameStatus' is set to true for stating the status of waiting for clients' responses. (Figure 49 & Figure 50)
- 'toNextRound' will be fired when all objects on one client have the velocity of zero. The server will set its corresponding 'turnReady' property in 'users' to true, as well as update the matching 'position' property. Then whether all connected clients have sent this signal will be checked. If this is the case, the server will check whether all received positions from clients are the same, and update all 'position' properties and broadcast 'syncPosition' event with the computed average position supposing the result is false. Afterward, the 'goNextRound' event is broadcasted with the values of 'gameStatus' and all 'turnReady' properties in 'users' and 'waitNextTurn' in 'gameStatus' are set to false. Additionally, the 'gameTurn' in 'gameStatus' will increase by one, and in case next turn has no user to operate, the function for simple NPC logic will be called. (Figure 49)
- 'toGoal' will be fired when the client detects a goal event. The server will set its corresponding 'goalMessage' property in 'users' to the received string, and check whether all connected clients have sent the same content. If this is the case, the server will increase the score of the scoring group at 'gameScore' in 'user', broadcast the 'afterGoal' event with the values of 'gameStatus', and reset all 'goalMessage' properties in 'users' to false. (Figure 50)

Furthermore, the server also has a function called 'simpleAI' for simulating users' action to control the teams that have no user assigned. The basic logic of this simple NPC involves three steps. First, a random player number (0 or 1) will be selected, and the location of the player, the football, and the goal is obtained from the 'position' property of current turn's team in 'users'. Next, according to the relative position of the ball and the chosen player, the movement of the player is determined and the moving direction is calculated.

For the first condition, if the player is between the goal and the football, the NPC will try to move the player closer to own goal (Figure 19).

```
<1>    playerToTarget = Math.PI -  
<2>    Math.atan2((doorPos.y - playerPos.y), (doorPos.x - playerPos.x));
```

Figure 19: Code for computing the shooting angle in condition 1

```

<1>   shootAngle = Math.atan((ballPos.y - doorPos.y), (ballPos.x - doorPos.x));
<2>   targetPos = {
<3>       x: ballPos.x + Math.cos(shootAngle) * 55,
<4>       y: ballPos.y + Math.sin(shootAngle) * 55
<5>   };
<6>   playerToTarget =
<7>       Math.atan2((targetPos.y - playerPos.y), (targetPos.x - playerPos.x));
<8>   }

```

Figure 20: Code for computing the shooting angle in condition 2

As for the second condition, the NPC will decide to kick the football, if the ball lies between it and the goal. Under this circumstance, the coordinate of the ideal destination is computed by finding the point on the half-line, started at goal and passed the football, that is fifty-five pixels (the approximate sum of the radiuses of the football and the player) away from the football. Finally, the initial velocity is computed by sine and cosine functions and the selected player number and calculated velocity are broadcast to all clients in one object.

```

<1>   var speed = {
<2>       x: gameSetting.shootSpeed * Math.cos(playerToTarget),
<3>       y: gameSetting.shootSpeed * Math.sin(playerToTarget)
<4>   };

```

Figure 21: Compute final output velocity with trigonometric functions

With everything constructed properly, the server was able to start through the command line and display the server port on it.

```

D:\GitHub\BachelorThesis\LocalGame\server>node server.js
Listening on 8081

```

Figure 22: The message in command prompt

### 3.1.2 Client programming

The files of the client are stored under ‘public’ folder, and the detailed structure is shown in Figure 23.

- The ‘assets’ folder holds all the font files, images, and JavaScript files that build up the game area.

In ‘bitmapfont’ folder, there is a texture file called “bitter.png” and an XML file called “bitter.xml”, which are used to render less flexible and faster texts objects in the game as the font is determined. The free font “bitter” is used to create these bitmap text datafiles.

In ‘control’ folder, four JavaScript files control creating the customized Phaser 3 objects. “football.js” and “player.js” separately extend the “Phaser. Physics. Arcade. Image” class to customize an image with a circular Arcade Physics body to act as a football or a player. “playGround.js” creates a class to set up a background of the football field as an image and two goals as Arcade Physics images. “team.js” creates a class by instantiating two players as a team and adding an arrow image for controlling the force and direction when moving the players. The class records the assigned client’s socket id and team number and has the only function that fires ‘toShootBall’ event to move the player.

In ‘image’ folder, eight images in png format are stored for clients’ use in the game.

In ‘scene’ folder, “login.js” and “play.js” extend “Phaser. Scene” class and compose the game. Both enable users to input with mouse or touch on defined buttons, areas, and objects. “login.js” is a scene that forms a login page for user to type in their five-character nickname with twenty-six lowercase letters and hyphen and passes the nickname to “play.js” when starting it as the main game scene (Figure 24). “play.js” is the main game world that instantiates four teams, one football and the football field, as well as enables and tracks the collisions among them (Figure 25). If the football is overlapped the goal, the ‘toGoal’ event is triggered. It also records the game status of this client and uses Socket.IO methods to communicate with the server. In the update loop, the client will continuously check if all objects are stopped when the game starts.

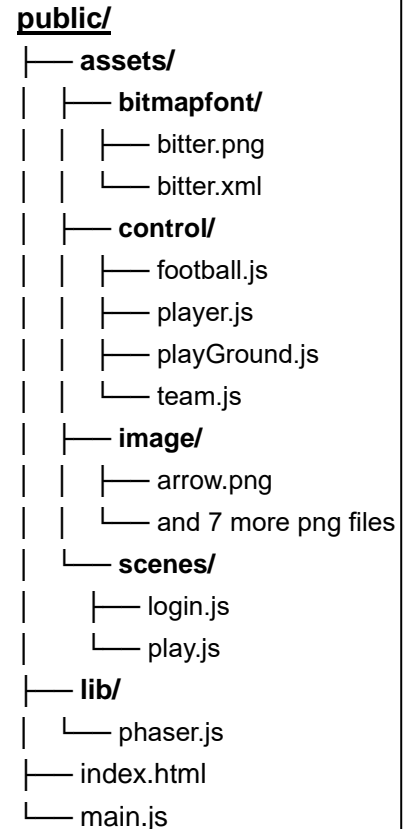


Figure 23: The file structure of ‘public’ folder

*Simple  
Football*

*Please Enter Your Nickname*

*T E S T \_*

*A B C D E F G H I  
J K L M N O P Q R  
S T U V W X Y Z -*

Delete

Confirm

Figure 24: Login Page



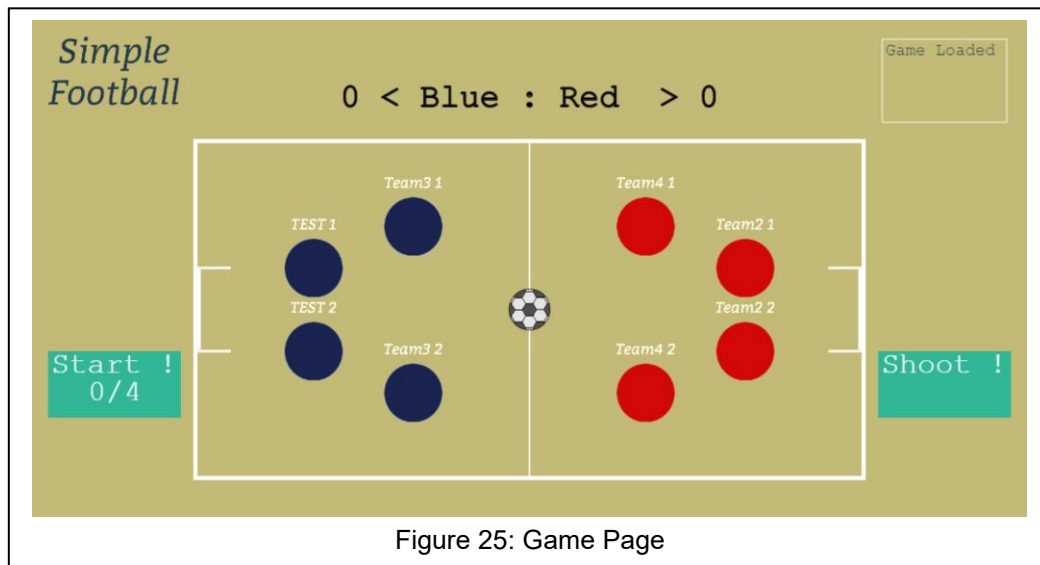


Figure 25: Game Page

- The ‘lib’ folder contains the build file of Phaser 3, which allows user to start a local multiplayer game without the Internet.
- The “index.html” is the main page that will be displayed to the user, and the place where the game is rendered. It imports the packages of Socket.IO, Phaser and “main.js” in sequence to prepare the libraries needed for the game.
- The “main.js” is where the game instance of the Phaser 3 is declared. In this file, the initial configuration of the game is set and passed to the created game. The game is adjusted to choose Canvas or WebGL renderer automatically, have an initial size of 1200 \* 600, use Arcade Physics by default, limit the graphics update loop to twenty-five frames per second, resize and center according to the screen size automatically, and use the two given scenes to run the game. The LoginPage scene will be called first when starting the game.

```

<1>  var config = {
<2>      type: Phaser.AUTO,
<3>      width: 1200,
<4>      height: 600,
<5>      backgroundColor: 0xc4ba77,
<6>      physics: { default: 'arcade', arcade: { gravity: {}, debug: false } },
<7>      fps: { min: 25, target: 25, forceSetTimeout: true },
<8>      scale: { mode: Phaser.Scale.FIT, autoCenter: Phaser.Scale.CENTER_BOTH },
<9>      scene: [LoginPage, PlayGame]
<10> };

```

Figure 26: Phaser game configuration

User notification areas and buttons mentioned in the design phase are also programmed, and function the same as the plans. Moreover, the client contains a few events of Socket.IO in “play.js” for the communication between clients and server.

- ‘loginData’ will be fired once when the server is aware of the connection of this client. The client will update the game status and objects’ positions if it connects first with received data. Then the nickname of the user and current positions of all objects will be uploaded to the server. Additionally, the user notification will also be updated if the game has started when the client connects. (Figure 49)
- ‘currentStates’ will be fired once after the server receives and processes the data in ‘loginData’ event. The client will update the ‘userSocketID’ attributes of all four teams and the names above all players with the given values of ‘users’. (Figure 49)
- ‘newUser’ will be fired whenever the server finishes updating the information of the connected client. The client will update the ‘userSocketID’ attribute of the connected client and names above players of the corresponding team with the given data of ‘users’. (Figure 49)
- ‘waitingGame’ will be fired when part of the connected clients is ready to start the game. According to this client’s status, the game will display different notifications and show the number of ready clients. (Figure 51)
- ‘startingGame’ will be fired once when all connected clients are ready to start the game. The first team will be set to be operational, and ‘gameStart’ will be turned to true which symbolize the start of the game. (Figure 49)
- ‘over4’ will be fired when this client is the fifth or later one to connect the server. The client will notice the user for not being able to join the game. (Figure 52)
- ‘playerDown’ will be fired when a client is disconnected from the server. The ‘userSocketID’ attribute and player’s names in the corresponding team instance will be initialized. (Figure 49)
- ‘shootingBall’ will be fired when a client or the NPC at server decides to move the player and initial velocity is computed. The client will set the initial velocity of the selected player and change the game status of the client to waiting for the signal of beginning next turn. (Figure 49 & Figure 50)
- ‘syncPosition’ will be fired whenever the server asks all client to synchronize each object’s position with received coordinates. (Figure 49)
- ‘goNextRound’ will be fired when all clients are ready for next turn and the server requires all clients to start next turn. The client will synchronize the game status with server’s and change to the matching notification. (Figure 49)
- ‘afterGoal’ will be fired when the server received all goal messages and confirms the goal. The client will synchronize the game status with server’s and restart the game. (Figure 50)

### 3.1.3 Issues and solution

During the early construction of the game, it was found that the objects’ positions between

different clients were close but have tiny differences. Unlike gaps in the feasibility test (chapter 2.2.2), the x coordinates of the first player in the first team from two clients have a gap in single digits. By repeating the same input several times, the reason was revealed to be the minor divergences in computing the friction on each client, and thus the differences caused player to stop after different amounts of frames after it was moved. In order to solve the issue, a manual stop function would be called after any object speed was slower than 0.2 pixels per second.

The gap between positions had been reduced at that time. However, with the growing data computation on every client for the progressing programming, the light-weight Arcade Physics were making more glitches during the game. For instance, two players attracting together and spinning weirdly, which is the most serious one, happened if they hit each other at a slow speed. After the researching relative problems on the Internet, the solution on limiting frames per second (FPS) was found and used. The FPS of the client was limited to 30, and it also affects the number of the update loops in one second, which reduces calculation load on low-power devices as well.

After fully constructing the application in the first attempt, a small-scale test was made to examine the synchronization of the positions and application did not pass due to the Arcade Physics issues. Whatever devices were used to run the application, there was always one or more distinct gap between clients on different devices. As mentioned in the last paragraph of the chapter 2.2.2, the heavy load on the client could be the most possible reason for the gap between computing results, since both rendering and calculating took place at the same time. Under these circumstances, the alternative plan in chapter 2.1.4 was determined to be implemented.

## 3.2 Second Attempt

In the second attempt, on the one hand, the structure of the server can be divided into two sections. ‘server.js’ is responsible to build the runtime environment of the server, while files in ‘physic\_server’ folder construct a HEADLESS-mode Phaser 3 game that makes calculating physics on server possible. On the other hand, the structure of the client’s files stays almost unchanged. Figure 27 shows the structure of the application in the second attempt.

### 3.2.1 Server programming

Unlike what ‘server.js’ did in the first attempt, instead of declaring variables and setting socket event emitter, it also prepared the environment for running Phaser 3 game on the server. Despite including the Express and Socket.IO, path module, jsdom package, and datauri package are required (Figure 23), whose detailed introductions could be found in chapter 1.3. Then the paths of local files are set in the same way. In the last part of ‘server.js’, a jsdom instance is constructed from ‘index.html’ introduced in next paragraph, which can execute external scripts, pretends to be a visual browser, and returns a

```
SimpleFootball/  
├── node_modules/  
├── server/  
│   ├── physic_server/  
│   ├── public/  
│   └── server.js  
├── package-lock.json  
└── package.json
```

Figure 27: The file structure of SimpleFootball app

promise object. In the following ‘then’ method, as mentioned in 1.3.5, URL.createObjectURL and URL.revokeObjectURL are created for running Phaser 3 engine successfully. Additionally, the function called ‘gameLoaded’ is constructed for starting listening after the Phaser game on the server is fully running, and the instance of Socket.IO is passed to jsdom with a property named ‘io’. Eventually, all errors will be caught in the ‘catch’ method and output in the command line.

```
<1> const path = require('path');
<2> const jsdom = require('jsdom');
<3> const express = require('express');
<4> const app = express();
<5> const server = require('http').Server(app);
<6> const io = require('socket.io').listen(server);
<7> const Datauri = require('datauri');
<8> const datauri = new Datauri();
<9> const { JSDOM } = jsdom;
```

Figure 28 Required modules in second attempt

```
<1> (function() {
<2>   JSDOM.fromFile(path.join(__dirname, 'physic_server/index.html'), {
<3>     runScripts: "dangerously", resources: "usable", pretendToBeVisual: true
<4>   }).then((dom) => {
<5>     dom.window.URL.createObjectURL = (object) => {
<6>       if (object) {
<7>         return datauri.format(object.type,
           object[Object.getOwnPropertySymbols(object)[0]]._buffer).content;
<8>       }
<9>     };
<10>     dom.window.URL.revokeObjectURL = (objectURL) => {};
<11>     dom.window.gameLoaded = () => {
<12>       server.listen(8081, function() {
<13>         console.log(`Listening on ${server.address().port}`);
<14>       });
<15>     };
<16>     dom.window.io = io;
<17>   }).catch((error) => { console.log(error.message); });
<18> }());
```

Figure 29 Use JSDOM to make up missed methods

As for the ‘physic\_server’ folder, its structure is shown in Figure 30.

- The ‘image’ folder holds the images of player, football, goal net and goal post that will be used to compute the physical body of the corresponding object in Phaser 3, and it also shares the same original files with all clients.

- The ‘js’ folder contains all JavaScript files that are needed to run the game.

The ‘lib’ folder holds the build file of Phaser 3.

In ‘objects’ folder, ‘ball.js’, ‘player.js’, and ‘team.js’ works the same as they did in the first attempt despite that codes are refactored and all input and decoration are deleted, as nothing will be rendered. The same rule applies to ‘goal.js’, which is a simplified version of ‘playground.js’ that only instantiates the parts of two-side goals.

And ‘main.js’ is the main game world that will run on the server. It stores authoritative configurations and statuses of the game, instantiates football, players, and goals to do physics calculation with Arcade Physics, and communicate with clients with Socket.IO methods.

- The ‘index.html’ imports all JavaScript files in ‘js’ folder and will be used to construct jsdom in ‘server.js’.

#### **physic\_server/**

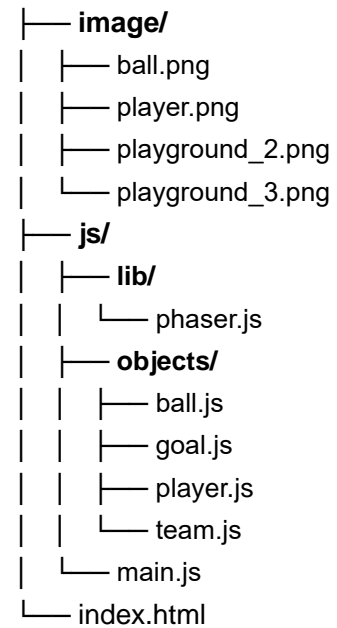


Figure 30: The file structure of ‘physic\_structure’ folder

The server-side variables in ‘main.js’ are refactored and merged with some variables of the client-side game in the first attempt. Separate variables are classified together into several objects.

```

<1>  var gameStatus = {                                var userStatus = {userNum: 0, users: []};
<2>    gameStart: false,                                for (var i = 0; i < 4; i++) {
<3>    gameTurn: 0,                                    userStatus.users.push({
<4>    gameScore: {blue: 0, red: 0},                    userSocketID: null,
<5>    waitNextTurn: false,                            userName: ('Team' + (i + 1)),
<6>    turnReady: false,                                teamNum: i,
<7>    goalReady: false                                startReady: false,
<8>  };                                                  turnReady: false,
<9>                                                  });
<10>                                                  };
  
```

Figure 31: Variables for recording the game and user status on server

‘gameConfig’ for configuring the game is nearly the same as the ‘config’ variable in ‘main.js’ of the first attempt, despite the ‘type’ property is set to ‘Phaser.HEADLESS’ for running Phaser without rendering. ‘gameSetting’ is declared for setting the positions of the game area. ‘gameStatus’ and ‘userStatus’ are similar with what ‘gameStatus’ and ‘users’ are in the first attempt, but modified the properties to fit the merge and code refactoring. Besides, ‘main.js’ also instantiates the football, four teams and each side’s goal and enables the collision detection among them, which allows them to collide and make the goal.

The number of Socket.IO events for the communication is declined to five, including ‘connect’, ‘disconnect’, ‘toStartGame’, ‘toShootBall’, and ‘toNextRound’. They work similarly as they did in the first attempt, the differences are listed below. The flowcharts for the second attempt are Figure 53 and Figure 54, and same workflows are omitted.

- ‘connect’ event simplifies the process in and after ‘loginData’ event by sending the server’s positions of all objects to the client instead of using the first client’s position.
- ‘disconnect’ does not check the goal status of the disconnected client now since the only authoritative goal event will be fired by the server due to the server-side physics calculation.
- ‘toStartGame’ remains the same.
- ‘toShootBall’ will not trigger event to move client’s players now, but directly move the client-selected player on the server. (Figure 53)
- In ‘toNextRound’ event, the server will not check all client’s positions and calculate the average to sync each client’s position anymore. (Figure 53 & Figure 54)

The update loop of the game in ‘main.js’ will trigger the ‘syncPosition’ event in every loop to broadcast current positions of all objects to connected clients and achieve the goal of using Arcade Physics only on server. The sent data contains the status of goal, current time in millisecond and all positions. Figure 32 and Figure 33 shows the structure of the sent synchronization data and position data.

```
<1> var syncData = {
<2>   goal: false,
<3>   time: Date.now(),
<4>   position: getAllObjPosition()
<5> };
```

Figure 32: Sent synchronization data

```
<1> var position =
<2>   [ [], [], // team0: player1[x,y], player2[x,y]
<3>     [], [], // team1: player1[x,y], player2[x,y]
<4>     [], [], // team2: player1[x,y], player2[x,y]
<5>     [], [], // team3: player1[x,y], player2[x,y]
<6>     [] ]; // ball: x,y
```

Figure 33: Array structure of the position

In the end, after all essential variables and functions are created, the instance of the Phaser game is declared, and then ‘gameLoaded’ method in ‘server.js’ is called to start the listening port.

With everything constructed properly, the server was able to start through the command line and display the server port and Phaser 3 information on it. Because of the internal Phaser game in the server, it starts slower at the first start-up compared to the server in the first attempt.

```
D:\GitHub\BachelorThesis\SimpleFootball>node server/server.js
Phaser v3.22.0 (Headless | HTML5 Audio) https://phaser.io
Listening on 8081
```

Figure 34: The message in command prompt

### 3.2.2 Client programming

Meanwhile, the client's files did not change significantly as Figure 35 shows. Files in folder 'bitmapfont' and 'image' stay unchanged, while all JavaScript files are moved to a new folder called 'js', which makes the whole structure closer to the server. The 'lib' folder and the Phaser build file inside, 'login.js' in 'scene' folder, 'game.js', and 'index.html' are not altered or refactored in the second attempt, which greatly reuse the existing resources. At the same time, in 'objects' folder, 'player.js' and 'playGround.js' are the brief and refactored versions of themselves in 'control' folder of the first attempt, as they remove all physics bodies to reduce the load of calculation on the client. Specifically, 'player.js' extends "Phaser. GameObjects. Image" class, which identifies a simple image object in Phaser 3, instead of "Phaser. Physics. Arcade. Image", and 'playGround.js' creates the goal nets and goalposts as simple image objects. While the remaining file in 'objects' folder, 'team.js', is optimized in structure and allows users to skip their turn now.

Finally, the 'play.js' in 'scenes' folder has changed most greatly, as it still works as the main game world. Because the game itself does not alter, the layout of the game area and notification areas remains the same. However, a few variables are changed to suit the structure of an authoritative server with the physics engine. Also, changes happen among the events of Socket.IO as well, and are listed below.

- 'loginData' will not upload the client's position to server the now, since the server has its authoritative positions before any clients can connect. Besides, the user notification update is moved to 'currentStates' event to ensure the correctness.
- 'currentStates' remains the same despite the additional user notification update.
- 'newUser', 'over4', 'waitingGame', 'startingGame' and 'playerDown' stay unchanged.
- 'syncPosition' keeps the same objective, but will be fire much more frequently, and check the timestamp before setting objects' position with received coordinates, which can help to avoid glitching movement caused by network issues. Meanwhile, it adds a method to play the animation of goal celebration when receiving corresponding values of the property 'goal'. (Figure 53 & Figure 54)
- 'nextTurnReady' is a new event that will be fired when all objects stop on the server side. It forces the client to sync positions with received coordinates and triggers 'toNextRound' event for the permit of proceeding to the next turn. (Figure 53)

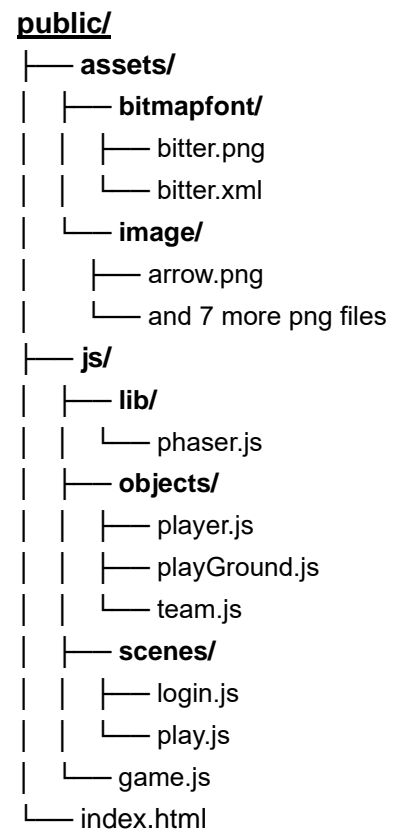


Figure 35 The file structure of 'public' folder

- ‘goNextRound’ is adjusted to prevent user, who is not supposed to operate in other’s turn, from calling out the draggable arrow for input on players, even though the user cannot upload the input result. The additional determination may avoid potential bugs. (Figure 53 & Figure 54)
- ‘afterGoal’ will be fired when the server detects the goal on server-side physic calculation without confirmation from any client and have waited for five seconds. It is no longer responsible for resetting position as ‘syncPosition’ will finish the job. In the end, it will trigger ‘toNextRound’ event for the permit of proceeding to the next turn. (Figure 53Figure 54)

### 3.2.3 Issues and solution

At the beginning of the programming in the second attempt, the way to move the players was to use the “moveTo()” method under class “Phaser. Physics. Arcade. ArcadePhysics”, which was supposed to calculate and apply the velocity according to the given distance and delta time. However, the players and football had a high possibility in moving back and forth greatly near the given coordinate or even moving out of the game area, although the outdated received coordinates were ignored according to the time stamps arrived with the positions. In this case, the method was changed to “setPosition()” under the “Phaser. GameObjects. Image” class to set the position of the objects directly with the given coordinates, and the timestamps were still applied to avoid glitches caused by possible network issues. Meanwhile, it also helped to clear all Arcade Physics related classes on the client side and thus relieved further computing pressure.

Besides, the arrow function introduced in ES6 (ECMAScript 6) was used in the final majorization among the functions for Socket.IO events in order to eliminate the use of ‘self’, which stores the value of ‘this’ as one of the solutions in ES5 for avoiding using anonymous function’s own ‘this’. With the help of the arrow function, the function syntax is shorter and looks more object-oriented.

Further issues and corresponding possible solutions are shown in chapter 4.1.3 and 4.1.4, which are collected from the usability test.

## 3.3 Third Attempt

According to the feedback in the usability test, the third attempt was planned and done to simplify the processes on installing and starting server mainly with the assistance of Electron framework. Detailed reason and related analysis can be found in chapter 4.1, and the introduction of used techniques is mentioned in chapter 1.3.6. Figure 36 shows the file structure in the third attempt, which is on the next page.

### 3.3.1 Server programming

Comparing to the second attempt, the client-side codes almost does not change except a different background color and game title of the game for differentiation. As shown in the figure, ‘public’ folder along with its stored files is completely identical, and thus the main differences



center on the structure and required libraries of the files in servers. The divergences are listed below.

- Comparing to the ‘physics\_server’ folder in the second attempt, the ‘index.html’ was moved to folder ‘server’ with a new name ‘server.html’ and would be displayed as the main window in the electron application. It additionally imported ‘server.js’ before instantiating the headless Phaser game to build a server runtime environment.
- An ‘index.js’ file was added under ‘SimpleSoccer’, which is the outermost folder. It required the Electron framework and created an application window to run the server as a desktop program. In this case, the main window content was built with ‘server.html’ in 500-pixel width and 800-pixel height. It disabled the use of top menu bar and build-in DevTools from Chromium to prevent any user from changing server-side codes during the game. After the Electron application is initialized, the main window will be created and start running the NodeJS server.

```

SimpleSoccer/
├── node_modules/
├── server/
│   ├── physic_server/
│   │   ├── image/
│   │   └── js/
│   ├── public/
│   │   ├── assets/
│   │   └── js/
│   └── index.html
├── server.html
├── server.js
├── index.js
├── package-lock.json
└── package.json

```

Figure 36: The file structure of SimpleSoccer app

```

<1>  const { app, BrowserWindow } = require('electron');
<2>  function createWindow () {
<3>    var mainWindow = new BrowserWindow({
<4>      width: 500,
<5>      height: 800,
<6>      resizable: false,
<7>      autoHideMenuBar: true,
<8>      webPreferences: {
<9>        devTools: false,
<10>        nodeIntegration: true
<11>      },
<12>      backgroundColor: '#202232'
<13>    });
<14>    mainWindow.removeMenu();
<15>    mainWindow.loadFile('server/server.html');
<16>  }
<17>  app.whenReady().then(createWindow);

```

Figure 37: Electron application configuration (index.js)

- Due to the structure of Electron apps, the Phaser 3 in HEADLESS mode now does not need to run in a virtual webpage created by jsdom, node-canvas, and datauri frameworks. In return, those frameworks can be removed from the dependencies and hence the ‘gameLoaded()’ function was retained to allow headless Phaser game fully running before

the server port is open to clients (Figure 38). Also, to completely get rid of the command prompt, the ‘getLocalIP()’ method is defined to use newly required ‘os’ module to find the correct local IP address (Figure 39) and pass it to ‘main.js’, where Phaser 3 game instance is.

```
<1> function gameLoaded() {
<2>   server.listen(8081, function() {
<3>     console.log(`Listening on ${server.address().port}`);
<4>     console.log(getLocalIP());
<5>   });
<6>   return `Server on ${getLocalIP()}:${server.address().port}`;
<7> }
```

Figure 38: Function for starting listening port of the server

```
<1> function getLocalIP() {
<2>   var networkInterface = os.networkInterfaces();
<3>   var ipAddress = 'error.error.error.error';
<4>   Object.keys(networkInterface).forEach((networkName) => {
<5>     networkInterface[networkName].forEach((ipconfig) => {
<6>       if (ipconfig.family === 'IPv4' && ipconfig.internal === false) {
<7>         ipAddress = ipconfig.address;
<8>       }
<9>     });
<10>   });
<11>   return ipAddress;
<12> }
```

Figure 39: Function for getting IPv4 address of the server

- In the third attempt, the headless Phaser game is running in the canvas of ‘server.html’. The Phaser 3 itself does not render any objects as usual, while the elements on HTML file can be accessed and created through Phaser 3 and rendered by Electron renderer process. In which case, ‘Phaser.GameObjects.DOMElement’ class is introduced to create an instance for controlling a specific HTML element to display server status (Figure 41). The server status messages are stored as an array with limited capacity, and showed via ‘array\_print()’ function. For example, the application window can display the IP address and the port of the server with the assistance of ‘gameLoaded()’ function, as well as the game status on the server. The detailed exhibition of showing the server message in an Electron message is shown in Figure 40.



Figure 40: Interface of a functioning server app

```

<1>  var serverStatus = ['Server Status :\n'];
<2>  const css = 'color: white; font-size: 20px; white-space:pre-line';
<3>  var domElement = this.add.dom(0, 0)
      .createElement('p', css, serverStatus.toString());
<4>  function array_print(text) {
<5>    if (serverStatus.length >= 21) {
<6>      serverStatus.splice(2, 1);
<7>    }
<8>    serverStatus.push('\n' + text);
<9>    domElement.setText(serverStatus.toString());
<10> }

```

Figure 41: DOMELEMENT instantiating and updating with Phaser 3 framework

### 3.3.2 Packaged application

By using the ‘electron-packager’ framework with the default configuration, the finished application was packaged into a native desktop program that can be run through opening the ‘SimpleFootball.exe’ and access the clients with any modern web browser across platforms. The URL address will be shown on the server interface after the server fully starts running (Figure 43).

#### SimpleFootball-win32-x64/

- |— locales/
- |— resources/
- |— swiftshader/
- |— [chrome\_...\_percent.pak]
- |— [... several compiler files...]
- |— [... licenses from chromium and electron...]
- |— [...resources files...]
- |— SimpleFootball.exe

Figure 42: The file structure of packaged application



Figure 43: The initial messages when server starts

### 3.3.3 Issues

Although the packaged application massively reduces the effort on operating the server, the size of either the executable or original folder is too big due to the complete Chromium rendering engine and the NodeJS runtime that come with Electron. The source files occupy 179 MB (Electron takes about half of it.), while the size of the packaged application takes 214 MB, which is acceptable but not ideal for a distributable Windows-specific desktop program. Using electron-builder, which is an alternative packaging tool for Electron, can gain a much smaller installation package, but asking users to install the server application is less convenient than a

click-and-start program.

Meanwhile, the server notifications constructed with the DOMElement instance from Phaser 3 let monitoring server status easier, but the limitation on creating and controlling DOM element through Phaser 3 methods makes it hard to continue to enhance the layout in the main window of the Electron application. For example, the HTML element cannot be controlled by internal or external scripts other than Phaser 3 methods, which leads to the restriction on stored server messages as all statuses are not possible to show on the screen together without an auto-scroll-down scrollbar that is only possible with original DOM methods.

The most serious issue in the third attempt is having the computation of server data and rendering placed in one process, which may affect responding client's request or even losing client's message when enduring heavy load. It is caused by mixing the Phaser 3 instances with Socket.IO events and using the Phaser-running HTML file as the main page of the Electron application, which results in putting all rendering elements and main server content together. One possible solution is to run a welcome page responsible for showing server status messages in the main window, while an invisible vice window running the server transmits messages across the processes. Whether it is the worthy compensation for not sending message between different windows within Electron application to avoid more potential bugs and save developing time still need to be examined in the further project.

## 4 Evaluation

### 4.1 Usability test

For the purpose of evaluating the performance and accessibility of constructed application, including the server and the client, a usability test was made. The test revolved around the experiences on game notification, gameplay, NPC performance, multiplayer performance, server start-up and server performance. Thirteen testers are invited to the usability test due to pandemic when the bachelor thesis was proceeding, and the tests were done online for the same reason.

The usability test involved the following steps:

- First, each tester received a compressed repository. It contained one folder named “SimpleFootball” including a complete application with necessary node modules preinstalled, an installation package of NodeJS in version 12.16.3, and a “readme.txt” file that stored all instructions.
- Second, Testers were requested to read the text file first before they start the test. The detailed instructions are provided in the appendix. The general test content consists of installing and starting the server, running the client with a web browser, and playing the game until three goals were earned. During this phase, the testers were able to ask for help if technical problems, like errors and glitches, or incomprehensible instructions were met and could not be solved by themselves.
- After all usability tests were over, the tester would receive a questionnaire. The entire test was over only when the tester finished all questions and sent the questionnaire back.

#### **UsabilityTest/**

##### **SimpleFootball/**

node\_modules/

server/

package-lock.json

package.json

node-v12.16.3-x64.msi

readme.txt

Figure 44: The file structure  
of 'UsabilityTest'  
compressed folder

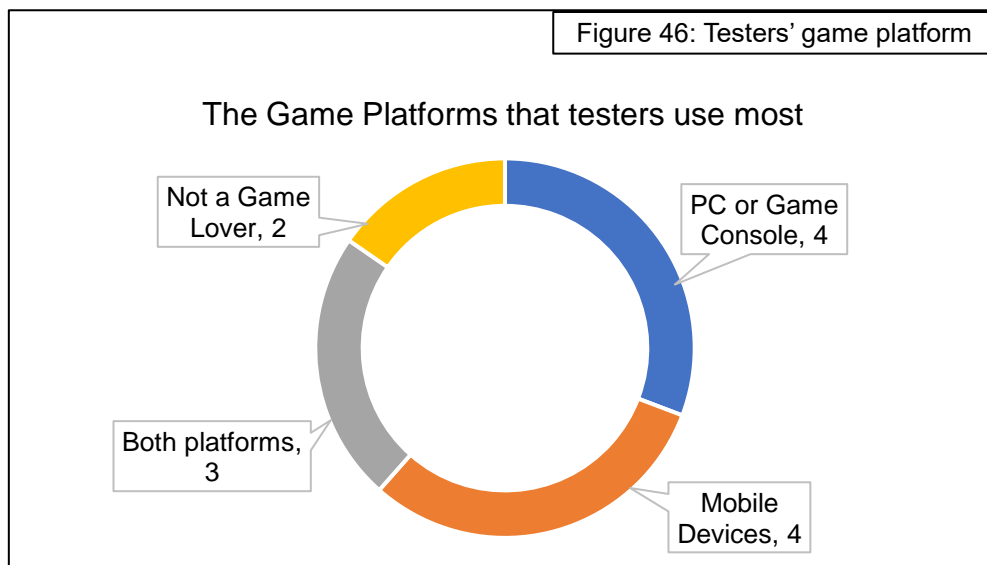
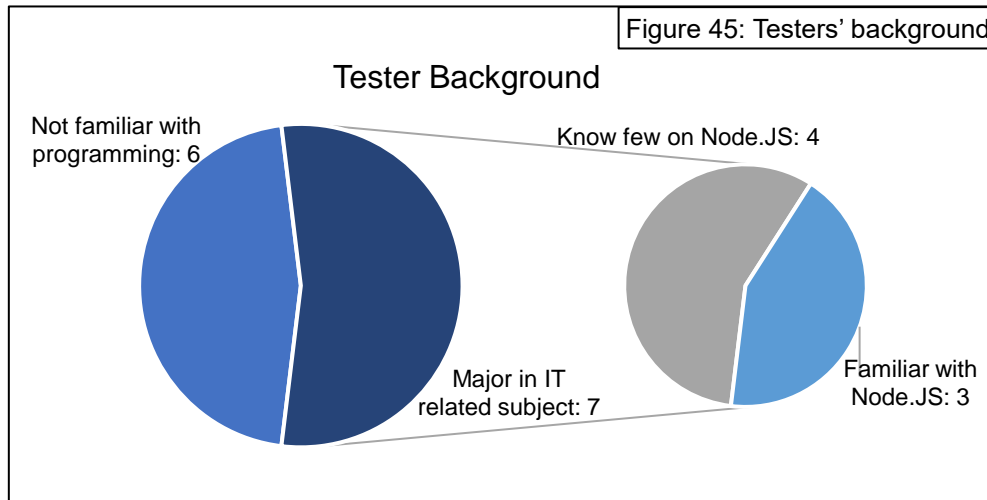
### 4.2 Questionnaire

The questionnaire had eighteen multiple choices questions and two short answer questions, and complete questions of the questionnaire are attached at the end of the appendix. Some additional questions were asked later according to tester’s answers to the questionnaire for more detailed information. The extra questions include the precise preferred game and further issues based on their opinions for the inspiration on possible improvement.

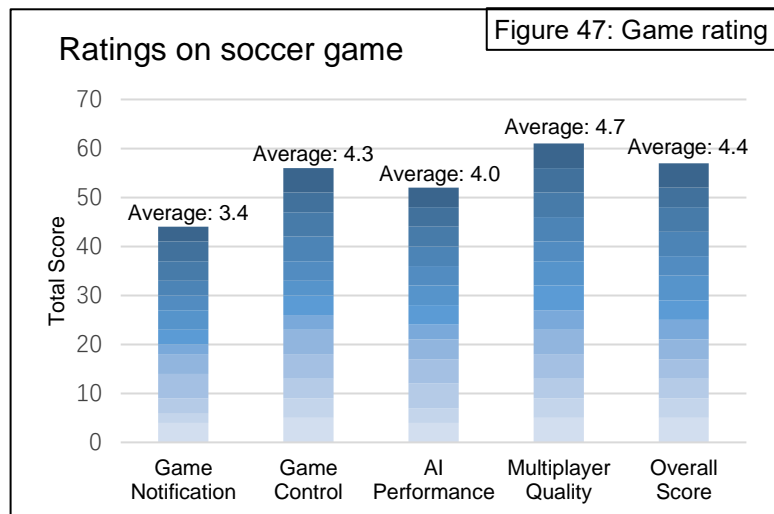
#### 4.2.1 Test Result

All thirteen participants who took the test are now in their senior year, and seven of them are studying information technology related subjects as their majors though only three people are familiar with NodeJS (Figure 45). Four testers are playing video game more on a desktop PC or a game console, while another four prefer games on mobile devices. Additionally, three of

them spends a close amount of time on both desktop and mobile games. And most of them trend to play multiplayer games, such as mobile MOBA games and board games.

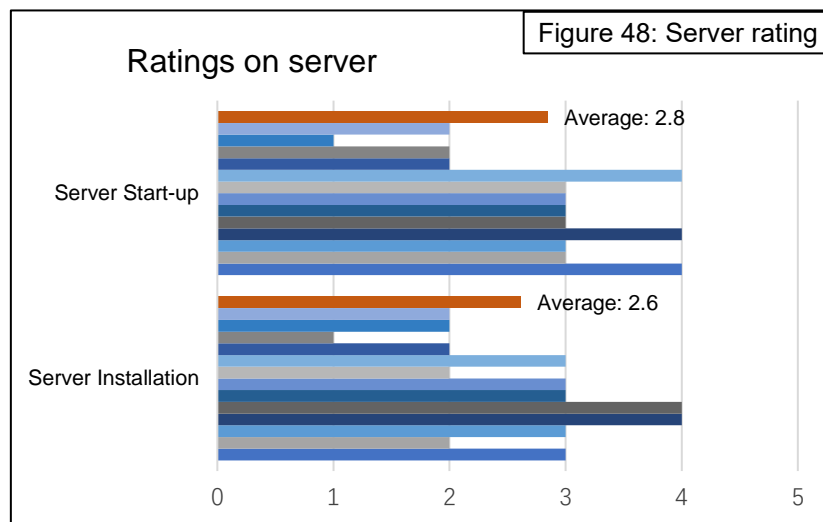


As for the experience on the soccer game, the average overall score is 4.4 on a five-point scale. The main reasons for the score are the approval of the idea of a cross-platform multiplayer game and smooth and convenient multiplayer experiences on different platforms. However, the notifications of the game are not distinct enough for users to have necessary information quickly, whose average score is 3.4 with two lowest scores which are two points. The performances on gameplay, NPC and



multiplayer scored 4 points and above that no tester thought they were bad during the game time. Three testers found observably glitches caused by Arcade Physics during the test, while only two persons felt possible network issues that not effected playing the game.

On the contrast, the test results on server installation and start-up were not satisfied, even though no serious issue that would affect the game on client happened during the test. The average scores for installing server and starting server are 2.6 and 2.8, which could be blamed on the operation with command prompt, and both got the lowest score on one point. One problem worth referring is a loading error which was caused by the Chinese characters in the file path and was solved easily by removing illegal characters.



Opinions from the testers in question twelve and twenty have shown some suggestion on the improvement of the game. The most frequently mentioned one is the improvement on using the server, as the command-line interface is not a common tool for experienced PC user. Most opinions point out that having a designed UI for the server or a one-click-start server application can greatly reduce the effort on operating the command prompt. The second one is the optimization of the game instructions as well as game notification, since both are hard to notice during the game. Other opinions focus on the NPC performance or physics performance due to their occasionally glitching movements like “shooting own goal” or “dancing around the goal post”.

#### 4.2.2 Data Analysis

The results in the last chapter indicate that the soccer game lacks the instructions or tutorials before starting, even if the gameplay is simple and interesting. Besides, the notification area is not eye-catching enough for users to notice the updated status, which could make them confused without being informed properly. Under these circumstances, the enhancement on UI should focus on adding detailed game introduction of the gameplay and the areas for game status information before starting the game, as well as highlighting the new game notification to draw users' attention in case they miss it.

The performances of the NPC logic and the Arcade Physics are fine, but there are still some areas for improvement. On the one hand, the NPC can have more decisions on controlling the

players instead of randomly picking a player and simply deciding to “attack” or “defend”. For example, the NPC can let the player block the way between the opponent and football. On the other hand, the Arcade Physics can be replaced by Matter Physics to strengthen the simulation on realistic reactions thanks to the fully server-side physics calculation. The clients will not need to share more computing resources on physics as they just move images around during the game.

In addition, few usability testers mentioned the missing winning condition and limitation on having only one game on the server at a time though they only need to earn three goals with their roommate or alone, so the winning condition and multiple game rooms are necessary. Meanwhile, an error log for recording errors automatically would be better than just showing them in the web console or command line. Another minor improvement can be forcing the webpages to be displayed in landscape on mobile phones if the auto-rotate setting is disabled.

And most importantly, the server itself, or with the client’s files, extremely needs to be packaged into an individual application, which may probably be achieved by using specific NodeJS library, e.g. Electron. Running a separate application for the server can not only simplify the operations on it, but also keep clients being accessed across platform via a web browser since the communication methods have a high chance to remain the same.

## **4.3 Further Improvement**

As mentioned in chapter 4.2.2, the application needs to package the server into a separate program to allow a much simpler way to operate on the server for inexperienced PC user. Under this circumstance, the third attempt was made to achieve that target, as well as improve the notification on the server side. Detailed implementation has written in chapter 3.3.



## 5 Conclusion and Outlook

### 5.1 Conclusion

The task of this thesis was to develop a cross-platform client-server architecture using HTML5 and NodeJS. Through three attempts on implementing a semi turned-based multiplayer soccer game with different NodeJS libraries and server structures in JavaScript, it could be concluded that the web application with HTML5-based frontend and NodeJS backend in client-server architecture is a suitable way for developers to construct a cross-platform project.

The conclusion was supported by the following two results:

- As it was explained and performed at the start of the second attempt in chapter 3.2.1, changing client-side codes to the server is much easier when using only one programming language since Phaser 3 and NodeJS can both be developed in JavaScript. This feature, on the one hand, allows extending the compatibility of clients by handing over the heavy load to the server, which is commonly much more powerful, when using the client-server architecture. On the other hand, there is a high probability to reuse as many resources as possible to save developing time while the project remains almost the same to customers, even if the structure between clients and server changes much. It could be seen from the transformation on network communication methods between the first implementation and the second one, as well as the alteration on server architecture between the second and third one.
- Comparing the normal desktop applications using a peer-to-peer architecture with the ones based on client-server architecture, peer-to-peer architecture seems to be a more reliable and flexible network architecture due to the sharing resources and unnecessary central server. However, web applications based on modern web browsers have only few protocols to set up two-way communication, which are WebSocket for client-server communication and WebRTC for peer-to-peer communication in this specific case. At present, WebRTC is a rather “young” protocols and lacks diverse instructions and tutorials for the deployment comparing to WebSocket, though it will keep ameliorating in the future. In the meantime, the heavy data processing in the thesis application could cause inconsistency on physical performance in peer-to-peer architecture more often than the client-server architecture with an authoritative server. As a result, the client-server architecture now shows more stability and simplicity in web application development that requires heavy data load and high consistency.

### 5.2 Outlook

Both the second and third implementation of the application are usable for running a cross-platform multiple-user application on web browsers, while the distinction between them is the server runtime environment. The second application structure has no user interface except the command prompt, which is more suitable for running on a host server or a cloud platform since

the user will not control the server start-up by themselves. While the third one is more favorable for users to start and monitor server by themselves, which is good for a LAN network test or entertainment.

The thesis has its limitation on comparing the WebRTC and WebSocket protocols due to the different developing progress between mature WebSocket-based Socket.IO framework and other new NodeJS environment frameworks based on WebRTC, which could lead to the bias on judging the advantages of client-server architecture. In addition, the heavy load on computation and the prevention of game cheating also probably stop thesis digging into the advantages and disadvantages analysis among various network structure. Under these circumstances, an enhancement can be made to fully analyze the architectures in distributed systems with relative frameworks based on HTML5 or NodeJS techniques.

## **Acknowledgements**

I would like to express my gratitude to my supervisor, Mr. Fabio Anthony, who provided me with plenty of useful suggestions and ideas on the thesis, as well as some daily issues. It's always a pleasure to talk to him.

I would like to thank Mrs. Lenka Kleinau, who spent her time on holding online meetings for thesis writing details during the pandemic.

I would like to thank all usability test applicants, who spent their time and effort to carry out the test with me.

## References

- [1] IOGAMES.SPACE, "io Games - Play on iogames.space," [Online]. Available: <https://iogames.space/>. [Accessed 3 6 2020].
- [2] A. O. Wiehe, N. S. Ansó, M. M. Drugan and M. A. Wiering, "Sampled Policy Gradient for Learning to Play the Game Agar. io," *arXiv preprint*, p. arXiv:1809.05763, 15 9 2018.
- [3] M. Miller, M. Washburn and F. Khosmood, "Evolving unsupervised neural networks for Slither.io," in *FDG '19: Proceedings of the 14th International Conference on the Foundations of Digital Games*, San Luis Obispo California, 2019.
- [4] X. Ma, "App Store Killer? The Storm of WeChat Mini Programs Swept Over The Mobile App Ecosystem," 23 9 2019. [Online]. Available: <https://mastersofmedia.hum.uva.nl/blog/2019/09/23/app-store-killer-the-storm-of-wechat-mini-programs-swept-over-the-mobile-app-ecosystem/>. [Accessed 3 6 2020].
- [5] Tencent, "Framework | WeChat public doc," 3 6 2019. [Online]. Available: <https://developers.weixin.qq.com/miniprogram/en/dev/framework/MINA.html>. [Accessed 3 6 2020].
- [6] Yannick, "Phaser 3 - Real-Time Multiplayer example with Physics," 15 3 2020. [Online]. Available: <https://github.com/yandeu/phaser3-multiplayer-with-physics/blob/master/README.md>. [Accessed 2 5 2020].
- [7] W3Techs, "Usage Statistics and Market Share of HTML5 for Websites, June 2020," Q-Success, [Online]. Available: <https://w3techs.com/technologies/details/ml-html5>. [Accessed 3 6 2020].
- [8] WHATWG, "HTML standard | The canvas element," 5 6 2020. [Online]. Available: <https://html.spec.whatwg.org/multipage/canvas.html>. [Accessed 5 6 2020].
- [9] WHATWG, "HTML Standard | Web Socket," 5 6 2020. [Online]. Available: <https://html.spec.whatwg.org/multipage/web-sockets.html>. [Accessed 5 6 2020].
- [10] Node.js®, "Introduction to Node.js," OpenJS Foundation, 24 12 2019. [Online]. Available: <https://nodejs.dev/learn>. [Accessed 8 5 2020].
- [11] G. F. Coulouris, J. Dollimore and T. Kindberg, "Characterization of Distributed Systems," in *Distributed systems: concepts and design*, Pearson Education, 2005, p. 1.
- [12] R. Davey, "Making your first Phaser 3 game," Photon Storm Ltd., 20 2 2018. [Online]. Available: <https://phaser.io/tutorials/making-your-first-phaser-3-game/part1>. [Accessed 20 5 2020].
- [13] Photon Storm Ltd., "Phaser - A fast, fun and free open source HTML5 game framework," Photon Storm Ltd., [Online]. Available: <https://phaser.io/>. [Accessed 9 5 2020].
- [14] d. T. K. and S. , "Socket.IO — Docs," Automattic, 17 4 2020. [Online]. Available: <https://socket.io/docs/>. [Accessed 19 5 2020].
- [15] "jsdom," jsdom, 9 1 2020. [Online]. Available: <https://github.com/jsdom/jsdom/blob/master/README.md>. [Accessed 26 5 2020].

- [16] "node-canvas," Automattic, 27 4 2020. [Online]. Available: <https://github.com/Automattic/node-canvas/blob/master/Readme.md>. [Accessed 16 5 2020].
- [17] Node.js®, "Node.js v14.3.0 Documentation: Path," Node.js®, 23 5 2020. [Online]. Available: <https://nodejs.org/api/path.html>. [Accessed 24 5 2020].
- [18] H. Santana, "datauri: Module and CLI to generate Data URI scheme.," 12 12 2016. [Online]. Available: <https://github.com/data-uri/datauri/blob/master/readme.md>. [Accessed 26 5 2020].
- [19] Electron, "Writing Your First Electron App," GitHub, Inc., 3 2 2020. [Online]. Available: <https://www.electronjs.org/docs/tutorial/first-app>. [Accessed 28 5 2020].
- [20] M. Lee, "Electron Packager," GitHub Inc., 26 5 2020. [Online]. Available: <https://github.com/electron/electron-packager/blob/master/README.md>. [Accessed 28 5 2020].
- [21] Node.js®, "Node.js v14.3.0 Documentation: OS," Node.js®, 26 5 2020. [Online]. Available: <https://nodejs.org/api/os.html>. [Accessed 28 5 2020].
- [22] I. M. and K. Boudot, "Matter.js is a JavaScript 2D rigid body physics engine for the web," 15 9 2019. [Online]. Available: <https://brm.io/matter-js/>. [Accessed 14 5 2020].
- [23] C. Jennings, H. Boström and J.-I. Bruaroey, "WebRTC 1.0: Real-time Communication Between Browsers," W3C Candidate Recommendation, 13 12 2019. [Online]. Available: <https://www.w3.org/TR/webrtc/>. [Accessed 14 5 2020].
- [24] P. Sletvold, "Running Phaser 3 on the server," 4 4 2018. [Online]. Available: <https://medium.com/@16patsle/running-phaser-3-on-the-server-4c0d09ffd5e6>. [Accessed 28 4 2020].
- [25] S. Westover, "Creating A Simple Multiplayer Game In Phaser 3 With An Authoritative Server – Part 1," 16 1 2019. [Online]. Available: <https://phasertutorials.com/creating-a-simple-multiplayer-game-in-phaser-3-with-an-authoritative-server-part-1/>. [Accessed 28 4 2020].
- [26] Y. Tay, "tree-node-cli," 5 4 2020. [Online]. Available: <https://www.npmjs.com/package/tree-node-cli>. [Accessed 26 5 2020].

# Appendix

## Socket.IO Events Flowcharts

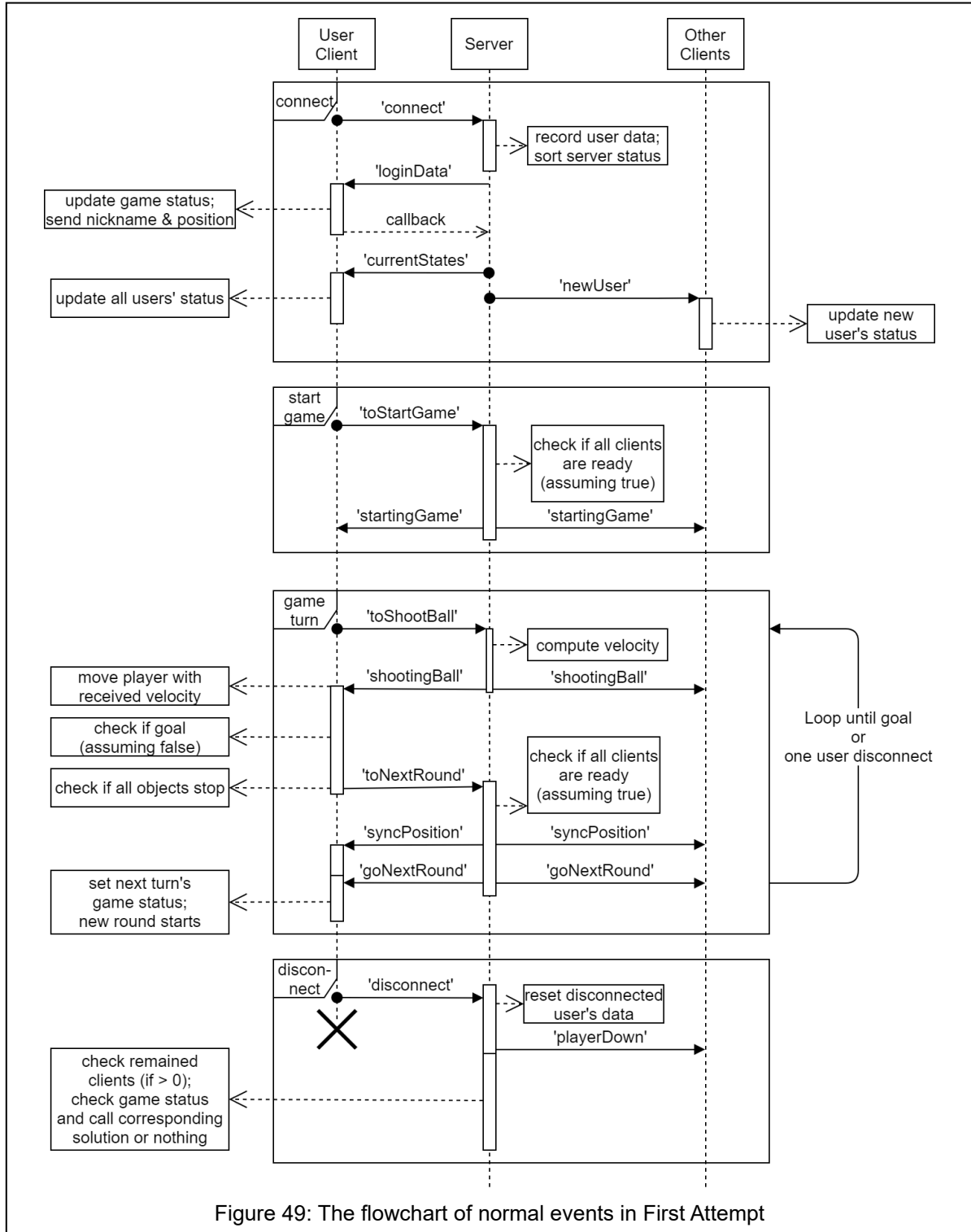


Figure 49: The flowchart of normal events in First Attempt

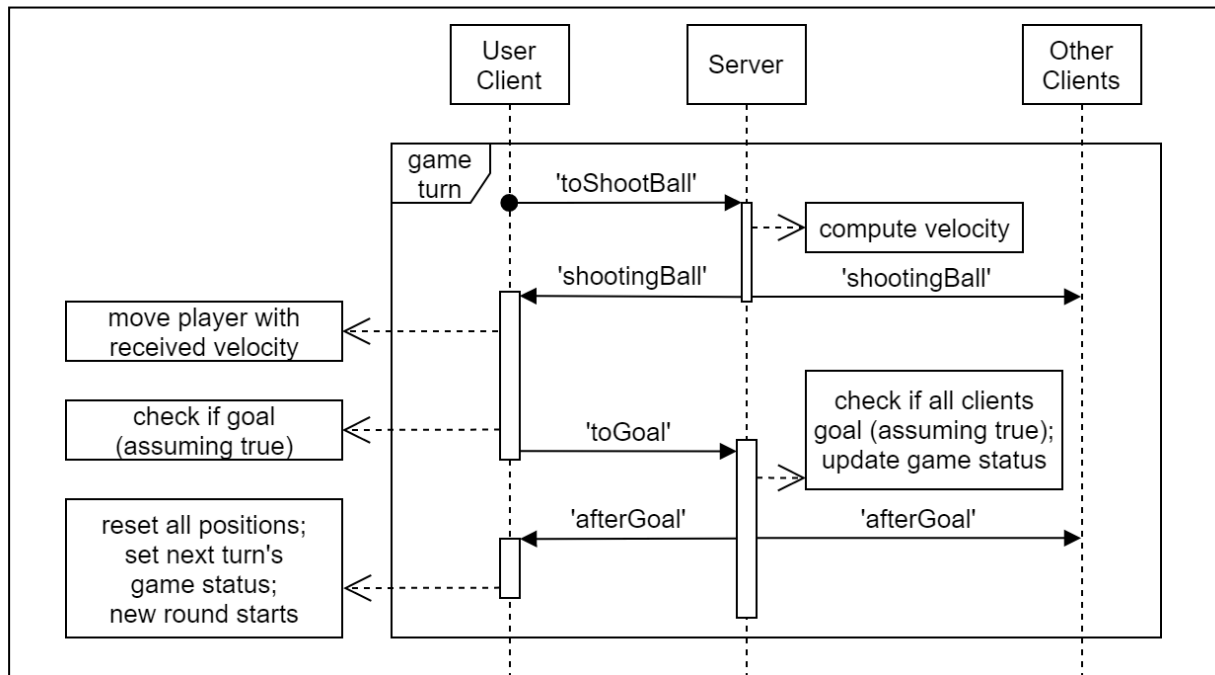


Figure 50: The flowchart of goal events in First Attempt

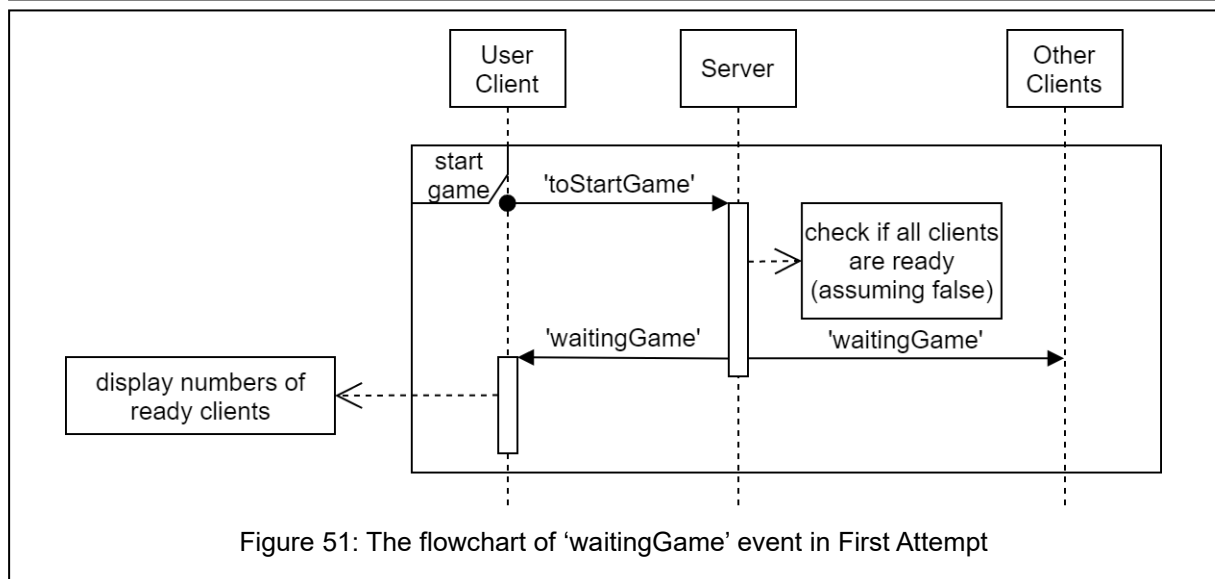
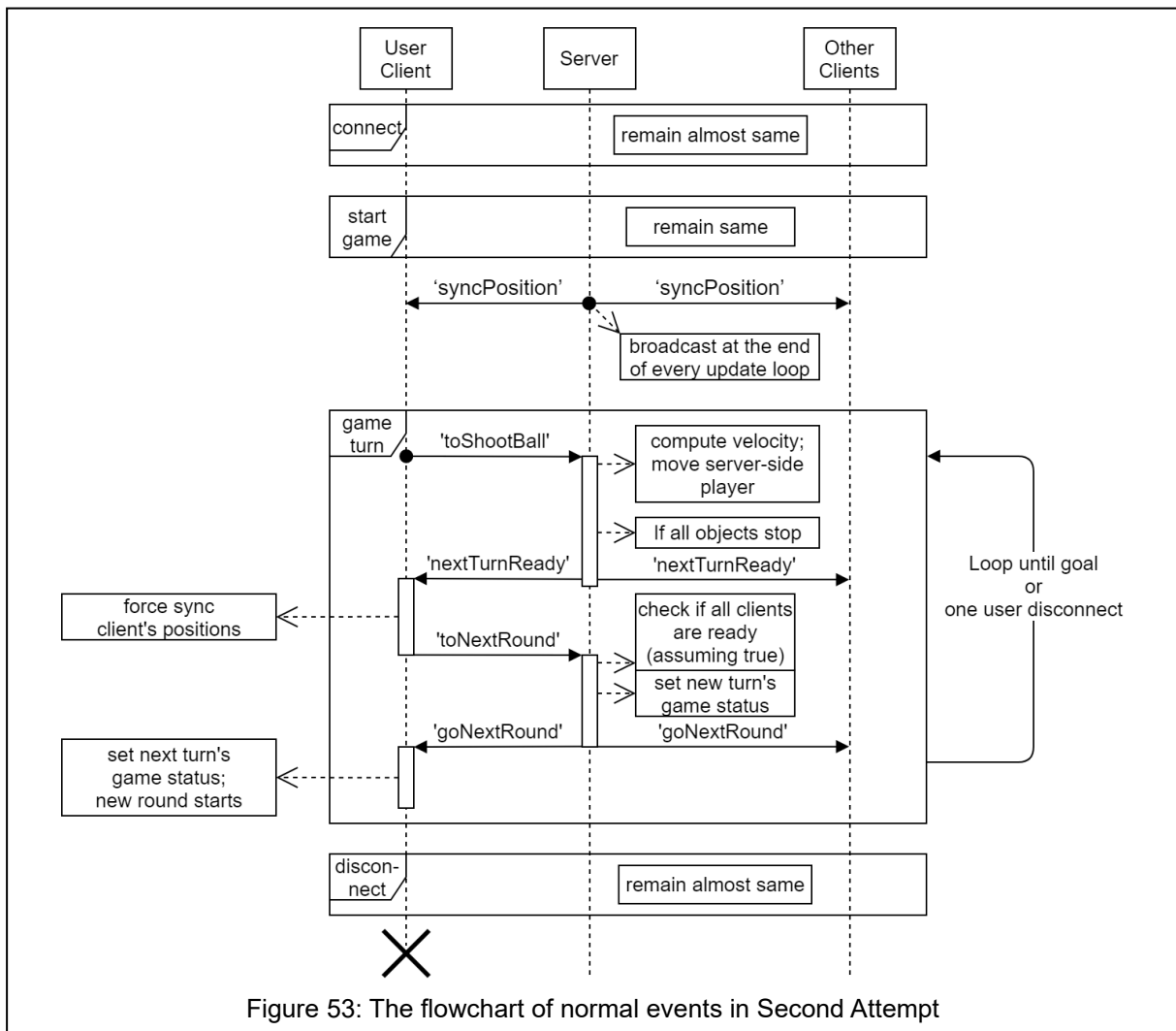
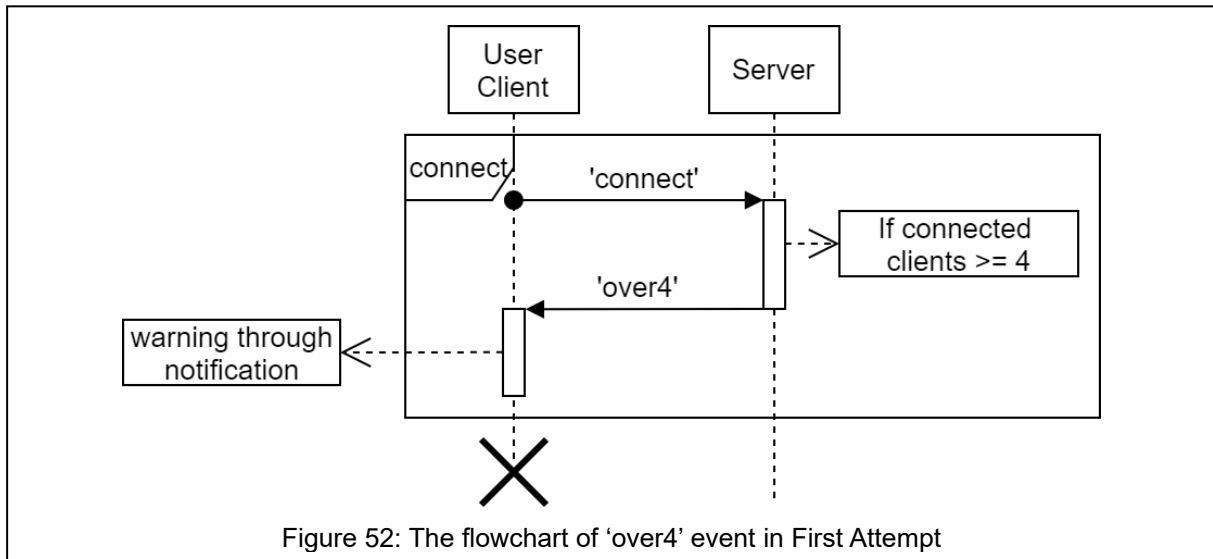
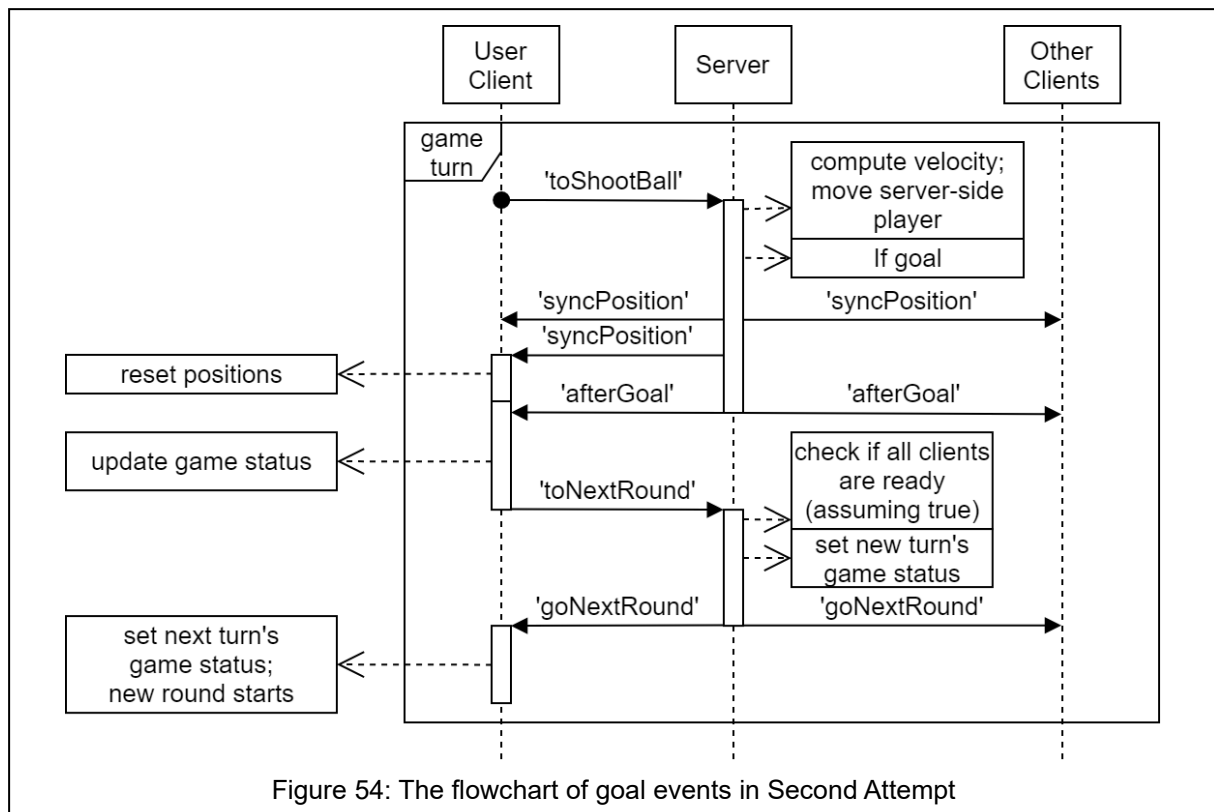


Figure 51: The flowchart of 'waitingGame' event in First Attempt







## List of Figures

Figure 1: The circular player with an arrow in initial direction .....	14
Figure 2: The arrow is dragged to another direction.....	14
Figure 3: The circular football.....	14
Figure 4: The football field.....	15
Figure 5: The game notification area.....	15
Figure 6: One frame in goal animation .....	15
Figure 7: The UI layout of the game.....	16
Figure 8 The file structure of chat room app .....	17
Figure 9: Required modules and used methods for transmitting images .....	17
Figure 10: Codes of using Ajax method to receive messages from clients and using Socket.IO method to send message back to client.....	17
Figure 11: Chat room layout.....	18
Figure 12: The output of the time gaps between server and client .....	18
Figure 13:Prototype game status .....	19
Figure 14: Two screenshots of the x coordinate from two clients .....	19
Figure 15: The file structure of the first attempt.....	20
Figure 16: Required modules for a HTTP server.....	20
Figure 17: Define the file path for client to access.....	20
Figure 18: The properties of 'users' .....	21
Figure 19: Code for computing the shooting angle in condition 1 .....	22
Figure 20: Code for computing the shooting angle in condition 2 .....	23
Figure 21: Compute final output velocity with trigonometric functions.....	23

Figure 22: The message in command prompt.....	23
Figure 23: The file structure of 'public' folder.....	24
Figure 24: Login Page.....	24
Figure 25: Game Page.....	25
Figure 26: Phaser game configuration.....	25
Figure 27: The file structure of SimpleFootball app.....	27
Figure 28 Required modules in second attempt.....	28
Figure 29 Use JSDOM to make up missed methods.....	28
Figure 30: The file structure of 'physic_structure' folder.....	29
Figure 31: Variables for recording the game and user status on server .....	29
Figure 32: Sent synchronization data .....	30
Figure 33: Array structure of the position .....	30
Figure 34: The message in command prompt.....	30
Figure 35 The file structure of 'public' folder.....	31
Figure 36: The file structure of SimpleSoccer app.....	33
Figure 37: Electron application configuration (index.js).....	33
Figure 38: Function for starting listening port of the server .....	34
Figure 39: Function for getting IPv4 address of the server .....	34
Figure 40: Interface of a functioning server app.....	34
Figure 41: DOMEElement instantiating and updating with Phaser 3 framework .....	35
Figure 42: The file structure of packaged application .....	35
Figure 43: The initial messages when server starts .....	35
Figure 44: The file structure of 'UsabilityTest' compressed folder.....	37
Figure 45: Testers' background.....	38
Figure 46: Testers' game platform.....	38
Figure 47: Game rating .....	38
Figure 48: Server rating.....	39
Figure 49: The flowchart of normal events in First Attempt.....	46
Figure 50: The flowchart of goal events in First Attempt.....	47
Figure 51: The flowchart of 'waitingGame' event in First Attempt.....	47
Figure 52: The flowchart of 'over4' event in First Attempt .....	48
Figure 53: The flowchart of normal events in Second Attempt.....	48
Figure 54: The flowchart of goal events in Second Attempt.....	49

## Usability Test: Instructions

The instructions were written in the 'readme.txt' files contained in the 'UasbilityTest' compressed folder.

1. Unzip the compressed package under the path without Chinese characters.
2. Install the NodeJS: If you have not installed the Node.js<sup>®</sup> before or your NodeJS version is under 12, please double click 'node-v12.16.3-x64.msi' to install NodeJS. Please use default configuration during the installation in case any errors occur.

3. Open the command prompt: Press 'win + R', type in 'cmd' in pop-up "Run" application and click "OK".
4. Find and write down local IP address: Type in 'ipconfig' on the command line and press 'Enter'. Look for "IPv4 Address" in the section who has a "Connection-specific DNS Suffix" called "localdomain". Record the IPv4 address. (For example: 192.168.1.30)
5. Navigate to 'SimpleFootball' folder: If the folder is not under local disk (C:), type in the corresponding letter with a colon on the command line, and press 'Enter'. For example, if the folder is under disk (D:), type in 'd:' or 'D:'. Type in the path of the 'SimpleFootball' folder with a prefix 'cd ', and press 'Enter'.  
  
(For instance: 'cd D:\GitHub\BachelorThesis\SimpleFootball')
6. Enter 'node server/server.js' on the command line. If Windows asks for permission on the network, allow NodeJS to do the following process to proceed. Wait until "Phaser [.....] Listening on 8081" shows in the command prompt.
7. Open the client with a web browser (except IE): Open your web browser on cellphone or desktop. Type in 'your IPv4 address:8081' in the URL bar, such as '192.168.1.30:8081'. The webpage should show up in a few seconds.
8. Enter your nickname and start the game. Keep playing until you earn 3 goals.
9. Reply to me when you finish step 8. A questionnaire will be sent, and needs to be filled and sent back to me in any format.
10. If you meet any issues that cannot be solved alone, please contact me as soon as possible.

## Usability Test: Questionnaire

1. Your gender:  
☐ Male.                                      ☐ Female.                                      ☐ Other.

---

2. Your age:  
☐ < 18                                      ☐ 18-22                                      ☐ 22-26                                      ☐ > 26

---

3. Do you often play video games?  
☐ Yes, often on a desktop PC or a game console.                                      ☐ Yes, often on mobile devices.  
☐ Yes, on both sides.                                      ☐ No, not often.                                      ☐ Never.

---

4. Which kind of game do you play most?  
☐ Multiplayer games.                                      ☐ Single-player game.                                      ☐ I don't play games.

---

5. How clear is the notification of the game?  
☐ 1 (very bad)    ☐ 2 (bad)                                      ☐ 3 (average)                                      ☐ 4 (good)                                      ☐ 5 (very good)

---

6. How easy is it to control the player?  
☐ 1                                      ☐ 2                                      ☐ 3                                      ☐ 4                                      ☐ 5

---

7. How well did NPC perform during the game?  
☐ 1                                      ☐ 2                                      ☐ 3                                      ☐ 4                                      ☐ 5

---

8. How is the experience of multiplayer?

	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
9. Did glitch exist during the game?					
<input type="checkbox"/> Yes, it effected game seriously.			<input type="checkbox"/> Yes, but it disappeared later.		
<input type="checkbox"/> Maybe, I don't notice it.			<input type="checkbox"/> No, I never saw it.		
10. Did the network problem exist during the game?					
<input type="checkbox"/> Yes, it effected game seriously.			<input type="checkbox"/> Yes, but it disappeared later.		
<input type="checkbox"/> Maybe, I don't notice it.			<input type="checkbox"/> No, I never saw it.		
11. Overall score on the game:					
<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5	
12. Any opinion towards the game?					
13. Is your major related to Information Technology?					
<input type="checkbox"/> Yes.	<input type="checkbox"/> No.	<input type="checkbox"/> No, but I've learned programming by myself.			
14. Are you familiar with Node.js®?					
<input type="checkbox"/> Yes.				<input type="checkbox"/> No.	
15. How convenient it is to install the sever?					
<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5	
16. How convenient it is to start the sever?					
<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5	
17. Did the error occur during the running period?					
<input type="checkbox"/> Yes, it effected game seriously.			<input type="checkbox"/> Maybe, I don't notice it.		
<input type="checkbox"/> No, I never saw it.					
18. Can you provide a detailed error message?					
<input type="checkbox"/> Yes, _____	<input type="checkbox"/> No.	<input type="checkbox"/> I have no error.			
19. Was it solved?					
<input type="checkbox"/> Yes, it was solved after restart.				<input type="checkbox"/> No.	
<input type="checkbox"/> Yes, it was solved by following other's instructions.				<input type="checkbox"/> I have no error.	
20. Any opinion towards the server?					