School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

Final Report

Remote Chess Playing

Anthony Moran

Submitted in accordance with the requirements for the degree of
BSc Computer Science and Mathematics

2022/2023

COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (02/05/23)
Link to Online Repository	URL	Section 2.1 & Sent to Supervisor and Assessor (02/05/23)
Link to Video Demo	URL	Section 4.3 & Sent to Supervisor and Assessor (02/05/23)
Link to the website	URL	Section 2.2.2 & Sent to Supervisor and Assessor (02/05/23)
User Consent Forms	PDF file	Uploaded to Minerva (02/05/23)
User Feedback	URL	Section 3.2 & Sent to Supervisor and Assessor (02/05/23)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) _____

Summary

The main objective of this project is to build a web app that allows users to play chess over a network, with an emphasis on online communication.
The back end of the system will be programmed in python due to experience with the language and the front end will be built from scratch using the basic web languages: html, css and javascript.

Acknowledgements

<The page should contain any acknowledgements to those who have assisted with your work. Where you have worked as part of a team, you should, where appropriate, reference to any contribution made by other to the project.>

Note that it is not acceptable to solicit assistance on 'proof reading' which is defined as the "the systematic checking and identification of errors in spelling, punctuation, grammar and sentence construction, formatting and layout in the test"; see
`https://www.leeds.ac.uk/secretariat/documents/proof_reading_policy.pdf`

Contents

Chapter 1

Introduction and Background Research

Knowledge of chess is not necessary to understanding this report. Despite chess being the forefront of this project, we will soon see that there is a lot more involved when building an online application other than the game itself. Throughout the report, any complex rules of the game will be explained to the degree in which is necessary to understand the current context.

1.1 Introduction

Chess is a board game that has existed since the 7th century [25] however it doesn't look exactly as we know it today. The modern game is played by two people, each one in control of an army of equal strength; it is up to the player's logical reasoning and deduction to conquer the board. For a long time, chess could only be played in person, or perhaps through the post. Internet Chess Club was founded by Danny Sleator in 1992 [1], and he lead a small team of programmers to develop the first dedicated chess server. This was the introduction to playing chess over the internet and it allowed people to play chess together, regardless of the distance between them. It wasn't until 1995, where the first web based chess server was launched by Caissa [2], which featured a graphical user interface. This most likely contributed to a higher adoption of playing chess online because it gave users a friendlier interface, which was more intuitive and approachable than what was previously available. Since then, many similar services have been created such as Chess.com and Lichess and it goes to show that there is quite some variety in the way this service is implemented. This report will outline our attempt to develop a service of our own.
It has only been in recent years that chess' popularity has started to rise, a big factor on this would be Netflix's "The Queen's Gambit", which released in October of 2020 [3]. We can see in figure 1.1 that the popularity for chess spiked in November 2020, which would correlate to the release of the TV show. The timing of these events, would suggest that the Show's success reignited people's interest in the game as well as attracting new players. The figure also shows a gradual rise in popularity with chess to this day and reinforces the demand for chess software. Whilst this software already exists, we will soon see that different software offers different features and in general, competiton in the market always leads to innovation, which is beneficial to the end user.

1.2 Objectives

By the end of this project, we will have created a fully functional web app that allows two users to connect over the internet to play a game of chess together.
On top of the functionality, we will also attempt to create a comfortable user experience, because the app will not be a success if no one knows how to interact with it.
Although it is not a priority, the general asthetic of the app will be considered and we will try to design a front end that is visually pleasing.

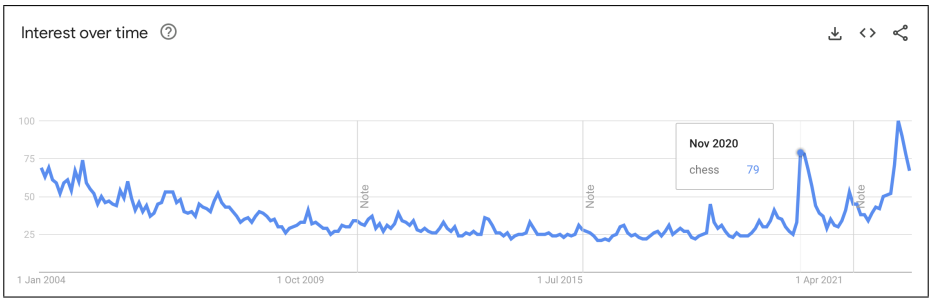

Figure 1.1: Google Trends Graph depicting the relative interest in the search term "chess"

1.3 Existing Solutions

1.3.1 Caissa

As mentioned in section 1.1, Caissa is the first *web based* chess server. The website is named after Caissa, the "patron goddess of chess players" [4].
Being accessible from the web allows for people to play chess with each other regardless of the distance between them. Caissa uses http (hypertext transfer protocol) for its communication [2], to put this into context, http was only developed by 1991 [5]. Caissa adopted this protocol while it was in its infancy and took a risk to use this over the standard telnet connection of the time.
[6] Games can either be played in real time or otherwise. The latter is referred to as a *Correspondence Game* and it allows users to play even if they are not online at the same time. Each person can log in and if it is their turn, they can make a move. Users can befriend the users they play against and this can be used to initiate games with them in the future with ease. Alternatively, if the user wants to play a game and no one is available, they can play against a computer instead, with varying difficulties (measure by elo; elo is a metric used to measure someone's relative skill in chess). There is also an additional computer type, listed as "Coaches", who have fixed elos and are helpful to users who are interested in practicing chess at their own level.
Caissa supports chess as we know it, but it also provides two alternate versions, namely Chess960 and Apollo Chess. In chess 960, the position of pieces is randomised and both sides of the board are mirrored to keep everything fair. The rules of castling change because the king and rooks are not in their standard positions, otherwise the rules are the same. The latter, being Apollo Chess plays like a normal game with the exception that forces pawns to only move one space; the standard rules allow pawns to move 2 spaces on their first move, which is removed from Apollo Chess. This version disrupts the opening theory for the game and can be a refreshing experience for those who have a lot of experience with the game. As a side note, there does not appear to be much information about this variation of chess outside of Caissa, and it must be unique to them.
By starting a new game and looking into the developer tools, we can see that the DOM (document object model) contains a canvas element, it is used as an overlay to draw circles and arrows, most presumably to help players plan their moves. We can look at the event listeners for the page and see that a mouse click is converted into a row and column by dividing the

event's offset x and y values by the size of the squares as seen in figure 1.2. The website uses a separate javascript file to handle the chess logic, which is a good example of "separation of concerns", where each file has a dedicated task.

```
let fn = parseInt(e.offsetX / squaresize),
rn = 7 - parseInt(e.offsetY / squaresize),
x,
y;
```

Figure 1.2: Extracted from the source code in [7] within the second script tag in the "released" function. "fn" referring to the file (column) on the board, and "rn" for the rank (row).

1.3.2 Chess.com

[8] Chess.com was released in 2005, which is 10 years after Caissa was released. This website is by far the most popular chess site in the world, hosting over 10 million chess games per day.This popularity must have come about for a reason, so we will investigate the way chess.com has been made to make it so successful.
Chess.com shares most of the same features as Caissa (except apollo chess). It builds on top of these features, including its chess variants, which there is a countless number of. These changes can be subtle like 3-Check chess where you must check the opponent's king 3 times to win or completely shake things up like Horde where one player starts off with four and a half rows of pawns [9]. Another variant, which has its own category is 4 player chess, as well as its own variants. As the name suggests, 4 people can play these variants at a time and is a fun way to play the game.
After attempting to look under the hood of the website in the DOM, and following the click event, it appear that the files have been minified. This is a process to reduce the size of files by removing comments, white space and condensing variable names to be small (sometimes only one character). This process is done to reduce the amount of bandwidth required to send the data. Bandwidth is a measure of how much data can be sent at a time, for example 80Mbps or 80 megabits per second. Minified files contain the exact same functionality as the original version but contain less data and can be sent faster as a result. The issue for us is that these files are not human friendly to read due to the optimisations in size. The functionality of the site is hidden so we cannot analyse it, unlike with Caissa.
[10] Something unique to this chess website is that it offers subscriptions to unlock more features. The most notable upgrades include removing the restriction to already accessible features like puzzles, which initially allow for 3 games a day. The subscriptions are split into 3 tiers and offer more features, the more expensive the subscription becomes. The higher tiers offer features that will help aspiring chess players become better at the game, by offering reviews of games they have just played as well as insights to help them improve next time.
In general, Chess.com offers many educational resources to support its player become better at the game (some of these locked behind paywalls) as listed here:
`https://www.chess.com/learn`. Once a user has become more confident with the game they can start participating in tournaments, which are also hosted on the platform. With addition of

the news and social sections of the site, chess.com manages to achieve its goals [8] of creating a close-knit community.

1.3.3 Lichess

[11] Our final example is Lichess. This site was released in 2010, making it the most recent out of the examples we are analysing. The website is open source, which means the source code can be viewed by anyone and contributions can be made (after being reviewed) to fix bugs or add new features. It is also completely free to use; it does not display ads or lock features behing subscriptions. Due to this commitment to remain free and a lack of ads, Lichess relies on crowdfunding to finance its development. While hosting 5 million game daily, Lichess certainly attracts a crowd, and potential supporters.
With the number of features included, this comes as no surprise. Although it is not as expansive as Chess.com, Lichess has 8 other variants of chess (including chess960, as mentioned in section 1.3.1). There is also an option to learn with step by step tutorials and puzzles, that will help beginners and intermediate players learn to play and develop better strategies. As with Chess.com, for those wanting more of a challenge, Lichess also offers the ability to create and partake in tournaments. Finally the website also has a section dedicated to community, much like Chess.com, where people can chat and ask questions. It's amazing that this website offers almost as much as Chess.com but can offer it all for free.
The one feature Chess.com surpasses Lichess on, would have to the game review and analysis. When a game is finished in Lichess, we can have the website generate an analysis of the game and it will point out moves in 4 categories: neutral, inaccuracy, mistake and blunder. Each category would translate to an increasingly worse move from the last. Chess.com expands from this to also include textbook moves during the start of the game that are well documented and understood by the community and then it also reinforces postive move by stating how good they were from a scale of good to brilliant. The disadvantage to this however is that Chess.com only allows one game to be reviewed per day, where as Lichess' more limited review can be used an unlimited number of times.
When starting a new game, we can look under the sources tab, as seen in figure 1.3, and see that the website uses websockets to communicate moves between the server and client. Whilst websockets made an appearance in chess.com's source files, they did not seem to be used for sending data for movements or any important data at all, it would send a number and receive a number at approximately 7 second intervals.

1.4 Stockfish

In the previous two sections, we have mentioned game review/analysis but how do these websites know what a good and bad move is? This functionality is outsourced to an external engine named Stockfish. [12] Originating from the Glaurung engine, Stockfish was forked from the codebase in 2008 and as of October 2020, become the "highest-rated chess engine according to CCRL (computer chess rating list)" [13]. As of writing, Stockfish version 15 has an elo of 3535 [14], and the highest elo for a human is held by Magnus Carlsen, with an elo of 2853 [15]. This goes to show that the engine is much further ahead than any human and that its

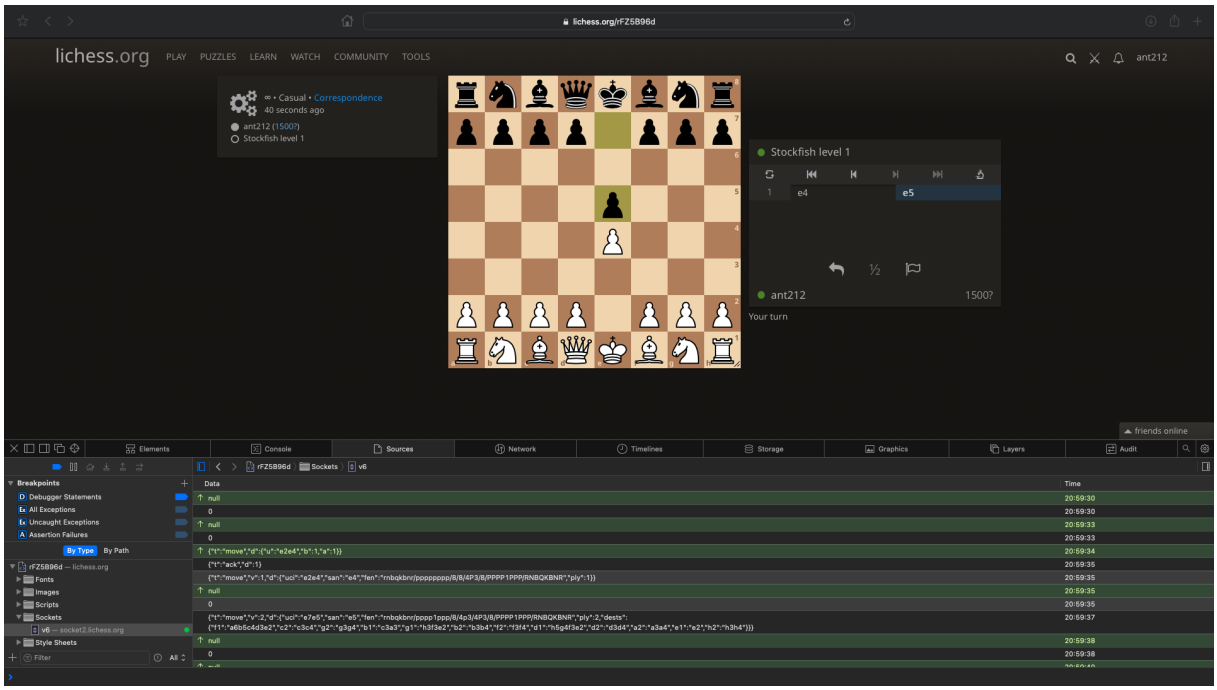

Figure 1.3: Viewing the sources tab in the developers tool during a game

judgement should be trusted when it is being used for a game review. However we wouldn't be doing Stockfish justice if we didn't also mention that it doesn't only out perform humans but also other engines! In Season 23 of the TCEC (Top Chess Engine Championship), Stockfish has come out on top [24] and also for the other 5 seasons before this! It is no wonder that chess software chooses to implement Stockfish as their engine of choice. In addition to the fact that it is open source and free, it is the obvious choice for those who develop chess software.

1.5 Background Experience From The Author

In the past I, the author, have designed two web services, called DS Club (Distributed Systems Club) and Monster Maker. The first service is a social media service that manipulates a database and can store posts, which can be requested, edited and deleted. Posts have an associated user, can include a caption and images and can also disable replies from other users. The second service, also uses a database to store monster designs. Similar to the first service, they can be requested using their unique IDs as well as being edited or deleted. The monsters could be created by customising the colour, facial expression and the type of hat they were wearing, an example of which can be seen in figure 1.4 (a happy, blue monster with a pirate hat). The two web services were composed with twitter to create a larger system and it requires the services to communicate with each other. This is where we reach the difference between this system and the app that is discussed in this report; services that communicate with each other and an application that communicates with a user. The web services could be accessed in a terminal or command line using the curl command, however this is not accessible to users that are not familiar with programming and this is why a user interface is important for the app we are creating. The back end will function very similarly to the services previously described.

Figure 1.4: A Twitter post that has been shared from DS Club, and an attached image, generated by Monster Maker

Chapter 2

Methods

2.1 Version Control

All of the code developed for this project has been managed by git and pushed to a GitHub repository with can be accessed by the link:

`https://GitHub.com/Anthony-Moran/Final-Year-Project`

The history will show the various stages of this project and how we started off with a blank workspace and ended up with this product now.
When implementing the online features, the commit messages became less detailed because the changes were extremely minor (usually one line or printing variables to the console) and changes were being made rapidly for debugging. This was required because the tools that we have used to deploy the app require the changes to be committed in order to come into effect. This differs from the earlier commits where the solution could first be tested locally and then committed only when the changes were implemeted correctly.

2.2 Back End

The back end of a web app is often referred to as the "brains" of the whole operation. This part is responsible for all of the computation, storing of information and is where the server resides. The complement of this is the front end, which receives information from the server and displays the results in a way that is visually pleasing to the end user. These two components are tied together by some form of communication, and there are multiple ways this can be achieved.
The decision for how to distribute computation between server and client can be made by considering multiple factors. For example, computationally high tasks (like cloud gaming) are better suited to the back end because dedicated hardware can process information faster than the average computer or mobile device. However this speed can be capped by the user's bandwidth. Therefore if a calculation can be performed on the client side, that should be preferred so we can cut out the communication entirely. Now the app we are developing is not computationally expensive, so we should process data on the client side right? While this would work, it may not be the best solution for this probem because it would mean duplicating data, for the two players in the game and potentially other spectators. This is common in distirbuted system design and it certainly works but in this case it would be better to store all information on the server so that there is a single point of truth. This ensures that all participants are synced and if the client needs information, the server is responsible for sending it.

2.2.1 Python Server

During early development, a simple python server was used to load my web app, it can be created using the following command in the terminal:

```
python -m http.server 8000 --bind 0.0.0.0
```

This requires python to be downloaded in order to run. The command creates a server that can be accessed on the *local* network by searching for the device's IP address followed by the port number, which is 8000 in this example. For clarity, the url would be *http://123.4.5.678:8000*, replacing the "123.4.5.678" with the IP address of the device calling the command. Notice that it is only accessible to user's on the same network, this is due to the fact that most home networks are not accessible from the internet. While this is helpful for testing during development, we will need to use something else to make our chess app truly remote.

2.2.2 GitHub Pages

This brings us to GitHub Pages. Using this service, we can host our web app on the internet directly from our online repository. What this means for us is that any time we push our changes, GitHub will redeploy the app automatically. The benefit of using this over another hosting service is the fact that GitHub Pages is built into GitHub and requires no addtional setup. However, if any experimental features are needed to be developed, they can be done so on a new branch and this ensures that only the stable version of the app is accessible online. The link to the website is: `https://anthony-moran.github.io/Final-Year-Project/`

2.2.3 Chess Logic

To implement the chess logic, we are using a python module called chess. This module contains many helpful classes and functions that we have used to develop our app. We have only used a portion of these features but there is certainly room to expand in the future. In this app, I used the Board class to store the current state of a game. Most interactions with the chess module occur through this class because the server needs to process and send data related to the board; this is specifically important when multiple games are running simultaneously.
The positions of pieces are recorded in a FEN (Forsyth-Edwards Notation) [22] string. This method of denoting positions originated in the 19th by David Forsyth and was later adapted by Steven Edwards to make it compatible with chess software. The FEN for the initial board is **rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1**. The first section represents the board; each row is separated by a slash and each letter character represents a chess piece. The numbers represent the number of consecutive blank spaces. The next letter will be either "w" or "b" and denote whether it is white's turn or black's turn respectively. The next four characters denote what castling moves are available and the next section denotes the space behind a pawn that has just moved two spaces or a dash if this is not applicable (this is necessary for an en passant move [16]). The final two numbers represent the halfmove clock and fullmove number and are used in end game conditions, however we only consider checkmate and stalemate in this app.

Some custom methods were required for the purpose of extracting unique data and/or formatting to make data more consistent for sending. A function that was surprising not to see in the chess module was for the ability to get the legal moves for a given piece. There was however a function that generated all legal moves, so we can wrap it in a new function and then extract only the moves from the space that we are interested in. In terms of data formatting, a client could ask for the name of a piece on a given square and the current chess function would return either a string or the value None. Therefore we can wrap that in a new function too and return the string if there is a piece there, otherwise we send the empty string.

2.2.4 Game Management

Current games are stored in volatile memory on the server, this means data is lost when the server shuts down. This app does not store games on a database and for that reason, games should only be played if they are intended to be finished within the same session. A database would be out of scope for this project, especially because it would require adding the ability to create accounts and associating game IDs with user IDs. Instead, the server uses a dictionary (a hash map) to store game keys to a tuple containing a chess.Board object and a set of connected users.
A customised function has been written to generate random game keys. It concatenates a random combination of letters and numbers, and then makes sure the key doesn't already exist before returning the value. For a length of 4 characters, there are $(26 + 10)^4$ possible game keys that can be generated, which is well over 1 million, and should be more than enough for the scale of this project. However the program has been designed, so that changing one constant value, which is clearly labelled JOIN_KEY_LENGTH, can be changed and increase the total number of available keys as a result. It may be tempting to run an infinite while loop to generate keys until a unique key is found but there is no guarantee that a key will be generated in a reasonable time frame. Therefore it is recommended to run a for loop for a fixed number of times, in this program it is 1000 times. However if no key is found, a runtime error is raised and handled appropriately.
When a game has ended or both users have disconnected, the game is removed from the dictionary.

2.3 Communication

Our server is able to process all of this information but now we need a way for it to send this data to our clients.

2.3.1 Flask/FlaskRESTful

The initial solution to this was to use flask and flaskRESTful, making use of the GET and POST requests over our already established http(s) connection. This solution did work but because we are making a game that takes place in real time, therefore the board needs to update every time a user makes a move. A http server however only sends messages when requested. Therefore the clients must continuously check the server to see if the opponent has moved. After some research, this is a recognised process, called "polling". This is not very

efficient because most requests will not return data and it is a waste of bandwidth. Surely there is a better method of communicating, instead of this request/response model?

2.3.2 Websockets

Websockets are the answer to this question. After discovering this new bi-directional form of communication, there was a helpful guide [17], which aided in the conversion from http requests to websocket communication. The guide was making Connect 4, so the final solution varies from the finished code in the guide [18]. The code uses the same functions as defined in the guide, excluding the replay and watch functions. A replay function was not necessary because we can send the board as a fen string when we need a full representation of the board. And then due to time constraints, the ability to watch a game was not implemented. The business logic inside each function is different for the reasons given prior. However the formatting of the send/receive data shares some similarities. They both send data as a JSON string and include a "type" key that indicates what type of data is being sent. Similar to the server code, the client code follows the same structure, using an init , receiveHandler and sendHandler function, which are all called once the DOM elements have loaded. We have also added an additional event, which detects when a websocket connection is closed and how the event should be handled. The server establishes a new websocket connection and as we haven't provided an explicit host, "all interfaces are assumed" [19]. We are interested in the websocket connection with address ws://123.4.5.678:8001, assuming we are using the same IP address in 2.2.1 and listening on port 8001. We notice that this is very similar to the browser address but http is replaced with ws (and using a different port); this is because websockets do not use the hypertext transfer protocol. The websocket protocol grants us the benefit as mentioned earlier, of bi-directional communication, so we no longer have to poll for data. The server will send data as soon as it is available, without a request being necessary. This is also solves another issue with the previous method, as now, when a user makes a move, all users (including the user who has made the move) can receive the update in parallel via a broadcast. Broadcasting is a method of sending data to all recipients at once and is more efficient instead of sending data to each user individually [20]. This ensures that all connected users will update simultaneously (excluding bandwidth and other networking factors).
It is important however to address the flaws with websockets and this mainly comes down to their short lifetimes, primarily on mobile devices. For mobile users, exiting the browser or locking the device will break the connection, which could happen in a real life scenario for someone using our app. Earlier in this section, there was an addition added, called the connection handler, and this is an instance where it is used. The page will be reloaded with a query string of "reconnecting=true" and will load a page that features a link named "Rejoin" that will send the user back to their game after being pressed. If in between this time, someone else were to join the game with the same key/link, this new user would take their place and the original user would not be able to join. This is a minor problem, and is outside the scope of the project, so it will remain that way. Additionally, during development, it was noticed that refreshing the page with only one player in the game would cause the game to disappear. This was due to an unforeseen consequence of deleting the game when both players leave. In this niche case, when the alone user refreshes, they temporarily break their websocket connection

which causes the server to kill off the game. When the page finishes refreshing and the websocket tries to initialise, the server returns with an error stating that the game no longer exists. To combat this we have added a small amount of buffer time to keep the game alive longer so if someone refreshes the game will continue to exist. Alternatively, if no one joins during the buffer time, the game is erased from memory.
Assuming that we do not run into these issues, the back end and front end will send messages back and fourth until an end condition (in the game) is met, at which stage the connection is safely closed because communication is no longer necessary.

2.3.3 Heroku

The current implementation of websockets only operates on the non-secure websockets protocol and is only accessible to those on the same network, for the same reasons as the http server. Therefore the last step to making our app remote, is to host our server online. Heroku is a cloud platform as a service. This service virtualises its hardware and allows us to use a portion of it to host our websocket server. After some amendments to the code, our clients can now connect to this websocket over wss (websocket secure protocol). In addition to our GitHub Pages website using https (hypertext transfer protocol secure), all of the communication between our clients and server is now encrypted and accessible from the internet.
For the scale of the project, we are using the eco option for our configuration. This will cover what we need, however a feature of this option is that the server will automatically sleep after 30 minutes of inactivity. A user that tries to access the server while it is asleep will have to wait an initial few seconds before the data is received. Future requests are received almost straight away.

2.4 Front End

We have now looked at the data we are interested in and a means of communicating it to our clients. The next step is to develop a friendly interface for our users to interact with our system.

2.4.1 The Welcome Page

The welcome page, as seen in figure 2.1, features the title at the top of the screen and then two boxes underneath, with the options for starting or joining a game. The interface is minimal and this design makes it easier for users to digest what they are looking at and make an informed decision for how they would like to proceed. For most users, they will select the new game link, which points to the chess.html file and queries it for a new board. As mentioned earlier in 2.2.4, there is a possibility that a board cannot be generated. In the unlikely event that this occurs, the server will send an error message, which the browser will alert to the user, using the in-built alert function, and they will be requested to try again. Alternatively, a user may want to join an existing game, in which case they should shift their attention to the second box, to see a textbox where they can enter a join key. If a user enters a non-existing key, the server will redirect them to the welcome page with a query parameter of "badRequest". If this parameter is used, the welcome page will add red text underneath the textbox to alert the user that the key does not exist. This is necessary for users who mistyped the code. The text is red

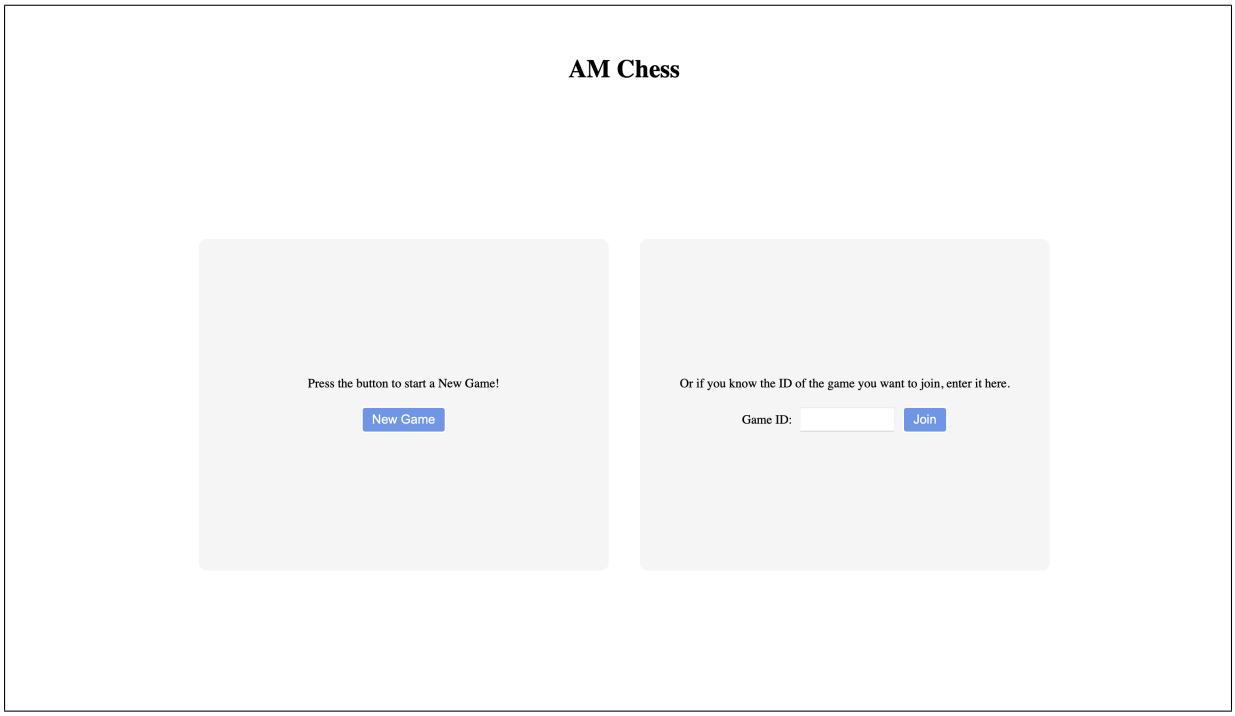

Figure 2.1: The welcome page of the app as seen on a desktop

specifically because it stands out from the other colours on the page and is typically associated with erroneous messages.

2.4.2 The Chess Page

The chess page is more complex than the previous page so we will break down its components from top to bottom.
We start with the navigation bar. In its current condition it only has one element which is the title of the game, and it acts as a way for users to navigate back to the welcome page. However, if the program were to expand, the design can accommodate for other options, but more on this in section 4.2.
Secondly, we have the text box. This is a div in the html file that stores all text a user might need to see. This includes the game link, game code and the current player's turn. In addition to the player's turn, it will also indicate if the user is in check. During development, it would sometimes look like the program was incorrectly not allowing certain moves to be played, when in reality it was because the board was in a state of check. For that reason, it felt necessary to include this in case real user's also faced the same issue. When the game ends, this prompt is replaced by the condition by which the game has ended (i.e. Checkmate or Stalemate) because the information for the current turn is no longer important.
Most importantly, we have the canvas element and this is where we draw our chess board. When the page first loads, the checkered pattern is drawn on the board and once the sprite sheet [23] has loaded, the pieces are then drawn in the positions as provided by the server[1]. For the user that plays as black, the x and y positions of all pieces needed to be flipped to mimic a

[1]The sprite sheet is licensed under creative commons and is therefore allowed to be used in this work

Figure 2.2: The chess page of the app as seen on a desktop

rotation of 180 degrees. This creates a more immersive experience for the player because it feels like the board is facing them, like it would be in real life.

From this point on, the board is only updated where changes have been made. This was to reduce the volume of computation required between moves, as the majority of the board stays the same. However this didn't come with some setbacks. In general the only spaces that change is the space that the moving piece has come from (which becomes empty) and the space that it is moving to (which will now contain the piece). During development there were three cases where this rule did not apply, namely castling, en passant and promotion, however these will be discussed in detail in section 3.1.1.

These moves make up a minor part of the game and do not occur very often compared to the other moves a player could make. To make a move, the user can select a square with a piece on it using their mouse or touchscreen. This triggers a click event on the canvas and the coordinates of the event are converted to a row and column on the board, similar to figure **??** The chess module stores the squares in a one dimensional array so we need to map the row and column to a single index. The index goes from left to right on board and then down to the next column. This can be mapped by the following:

$$index = row * 8 + column$$

Once we find the index, we can send this value to the server to indicate that we want to select this square. The square will highlight yellow and available squares will be highlighted blue, as can be seen in figure 2.3. The client is made aware of the available squares because the server sends these after receiving a "select" message. A user can select any blue square to move the piece. If a user attempts to move while it is not their turn they will not be allowed, and the browser will alert them with a message.

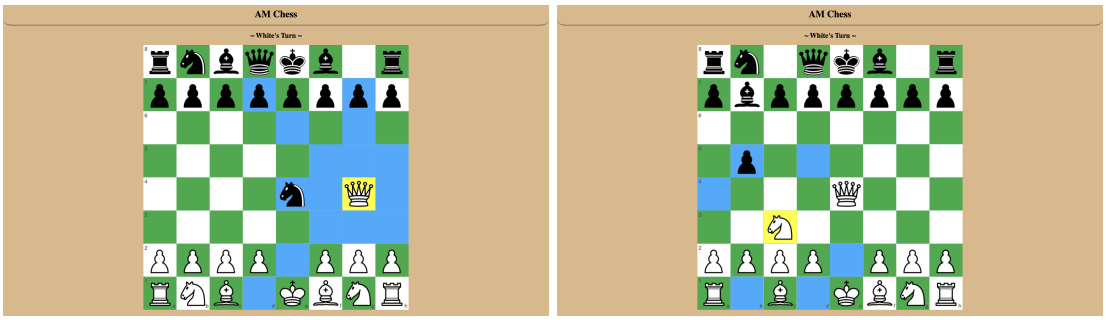

Figure 2.3: Examples of moving the queen (left) and knight (right)

It should be worth mentioning that all of this is only possible if the user has javascript enabled on their browser. The script is responsible for opening a websocket connection and communicating with the server. Without javascript, there is no data to process and the page will be blank.

2.4.3 Responsive Design

When designing anything on the internet, it's important to remember that users can be accessing resources on any device and when available, the flow of the page should reflect that. Usually this would include css media queries for the size of the screen but this app doesn't contain many elements, so this isn't necessary. We will focus on the orientation of the device because this is what will stop groups of elements from being awkwardly shrunk. We can see in figure 2.1 and 2.4 that the landscape orientation displays the two boxes next to each other. If we continued this for the portrait orientation, the widths of the boxes would have to squish to fit both of them on the screen. For this reason they are on top and bottom of each other and it also has the benefit of filling otherwise empty space.
The benefits of responsive design can also be seen on the chess page. Specifically on computers, if a user resizes the window, the board also changes size. The board will never grow larger than the width and height of its container, so its size is always the minimum of these values. Due to the way that the drawing has been implemented, when resizing the board, the whole thing is drawn again and this requires knowing what pieces are to be drawn on top. Therefore the server is queried every time the window resizes, this includes the duration of dragging the window. This is of course not very efficient and so we can implement a method called debouncing [21] to reduce the number of times this resize request is made to the server. Debouncing works by setting a timer with a callback to the function you want, in our case requesting the pieces on the board and redrawing the board. However if the the event is called again before the timer has finished, the timer is reset to 0 ticks. Therefore the resize event is only called when the user has finally stop adjusting the size of the window for a period of time.

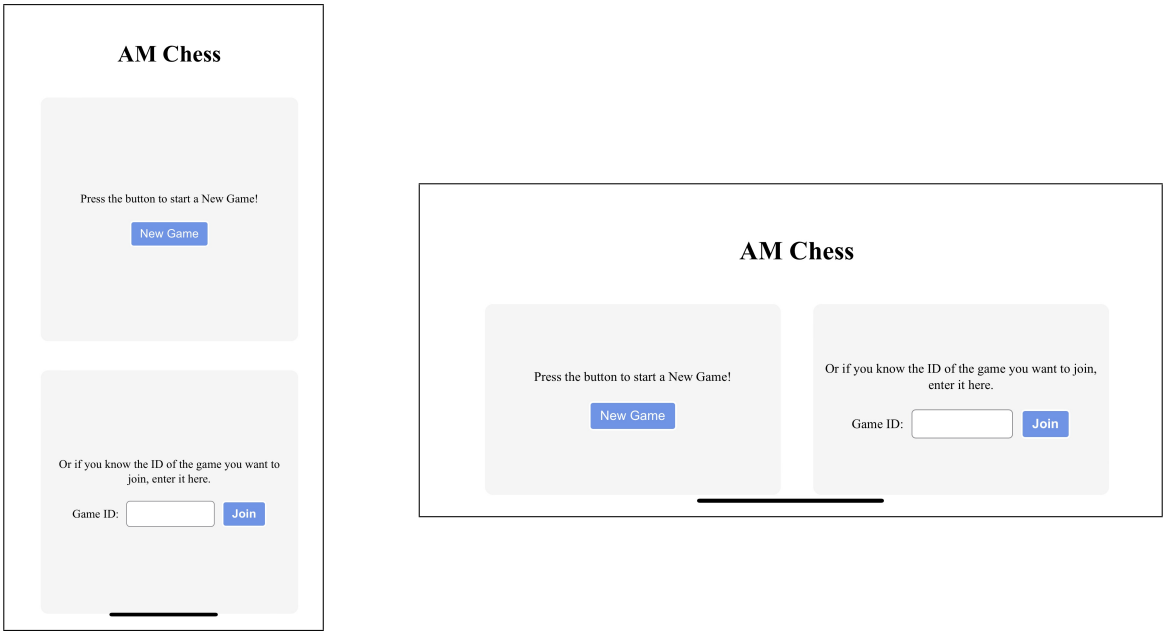

Figure 2.4: The welcome page, as seen on a mobile device in both vertical and horisontal orientations respectively

Chapter 3

Results

3.1 Alpha Testing

3.1.1 Special Moves

This was briefly mentioned in section 2.4.2 but now we will look into the process that went into testing the special moves that you can make in chess. Each one required new systems to be built into the back and front end in order to function correctly, which is what makes these moves so special.
We will start with promotion. This is the act of moving your pawn to the very end of the board, at which point you can choose to "promote" it to another piece (choosing from Queen, Rook, Knight and Bishop). In this case, the new square should instead show the piece selected by the user. The user can select this piece through a menu that we have designed, which features four icons representing the four pieces. Using the mouse or the touch screen, a user can select the piece and the data is sent by the client to the server, to inform it of the decision. The server sends back a generic message that will update the board as expected.
Moving onto the second case, we have en passant. In this case, the piece that we capture (take) is not on the same space that our piece moves to. Before we changed the code, the piece would be removed from the board on the back end but it would look like it is still there on the front end. To fix this, the server sends an additional message after the "play" message to clear the space of the pawn we have just captured.
Finally we have castling, which is the most complex of them all. When the king attempts to castle, we first need to work out the rank and file (row and column) of the rook that the king is castling with and the end position for it. We send the normal "play" message for the king and then a second "play" message for the rook. This had an unforeseen consequence of toggling the turn prompt twice on the front end so it would look like the user could move again after castling. This however is only a visual glitch and the server continues to hold the correct turn value on the back end. The reason for this is because the front end toggles the turn to switch between white and black when a user makes their move, which is usually what we want. To resolve this, we can add a flag, called "contribute turn", that alerts the front end whether this move should toggle the turn prompt. This value will default to true because this is the behaviour we would expect in most cases, however when we send the second play for the rook, we will set this flag to false.

3.1.2 Responsive Design

A minor problem for the responsive design arose when testing on an Apple iPhone, using the Safari browser. The css for the body element was set to 100vh[1], as this works fine on desktop

[1] vw/vh represent 1% of the viewport's width and height respectively, therefore 100vh should, in theory, equate to 100% of the viewport.

computers, however for iPhone, this value is not very intuitive. After testing, it appeared that the graphical user interface was included in the view port (such as the search bar), so certain elements would be hidden behind Safari's interface. This article [**?**], written by Trzciński M., provided some useful code, which was implemented in our own codebase. This addition was able to fix the issue and allowed all the elements to display correctly on the page and adapt when the GUI (graphical user interface) was minimised.
An additional problem, which is difficult to replicate (and therefore difficult to fix) occurred when using the developer tools during a chess game. When opening the developer tools on the side, the content of the page would half in size and it caused problems when mapping mouse clicks to a square on the board. However, by trying to force it to happen and having no success is a good sign because it means its a rare bug, but it would be better to know the root cause and deal with it accordingly but it is beyond the scope of this project.

3.1.3 The End Game

A benefit of using the python-chess module is the ability to provide the board class with a fen string to initialise the pieces on the board; this was especially useful for quickly testing the stalemate condition. The fen we are using to test/demonstrate stalemate is *5b2/8/8/8/8/n7/PP6/K7 b*[2]. As we can see in figure 3.1, we move the bishop from f8 to g7 and as we expect, a stalemate is reached. The text at the top of the page also reflects this.
The other end game is Checkmate and this can be attained quickly from the initial position. This is achieved by a set of moves called the "Fool's Mate", which has been recognised since the 17th century [**?**]. There are minor variations in the way this Checkmate can be achieved, in figure 3.1, the moves are as follows:

1. f4, e5
2. g4, Qh4#

Similar to the stalemate, by making these moves, we can reach and endgame and the text at the top will indicate that the condition by which the game has ended is indeed "Checkmate". This used to be the only thing that happened at the end of the game, however an extra visual has been added as a result of user feedback, more detail will be provided in section 3.2.2.

3.1.4 No Game

When testing this app, we have to start two servers, the http server to send web pages and supporting files, and the websocket server to handle requests and game logic. However, the fact that the websocket server might not be running was never considered. This led to a confusing series of events, when staring at a blank page and wondering "why it was working the day before but not today?". As one could guess, it's because the websocket server was not running. This revealed a hole in our codebase and it needed to be plugged. Before we dive into the solution, it's important to know that an event is triggered by a websocket when it closes; when a socket tries to connect to a server and fails, this is included as a close. Therefore when the

[2]This is not a full fen string but the module provides defaults for the remaining values, which is fine, because we only care about the positions and turn for this example

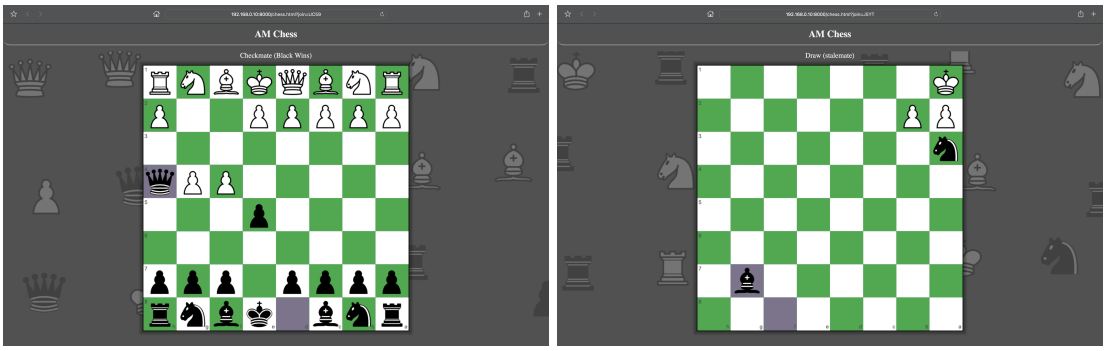

Figure 3.1: Two end states, Checkmate via "Fool's Mate" in 2 moves on the left and a Stalemate on the right

program first starts, if it detects a close event, it will alert the user that the server is not operating and will set a timeout that will send the user back to the homepage.

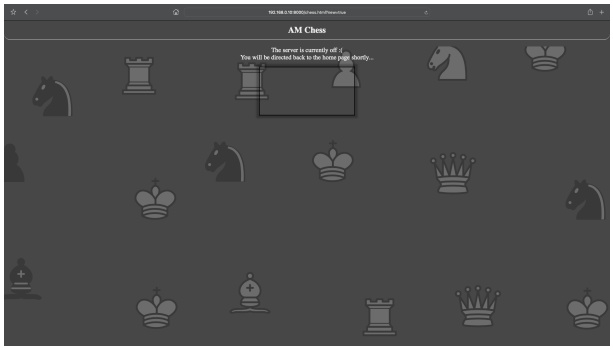

Figure 3.2: The chess page when the server has crashed or switched off

Another way in which a user could be redirected to the homepage is by manually typing the url in the search bar. In general the chess.html page takes one of two query parameters:

1. new=true
2. join=JOIN_KEY

Where we replace JOIN_KEY, with a valid 4 character code. However if a user intentionally enters a url that does not follow this structure, the browser will alert them and send them back to the home page. By testing all these edge cases we are able to evaluate the current responses, which in this case is an empty page, and make a decision as to what should happen instead. This refinements are what will lead to a smoother experience for the user.

3.1.5 Connection Issues

As an aside, I had an interesting experience while the app was in early stages of testing, when I was traveling by train and would occasionally lose my connection to the internet while going through tunnels. My app was able to survive this disturbance and using the reconnect button as described in section 2.3.2, I was able to continue testing the app as normal. Even though it was developed for other circumstances, it was nice to see it also supporting other situations as well.

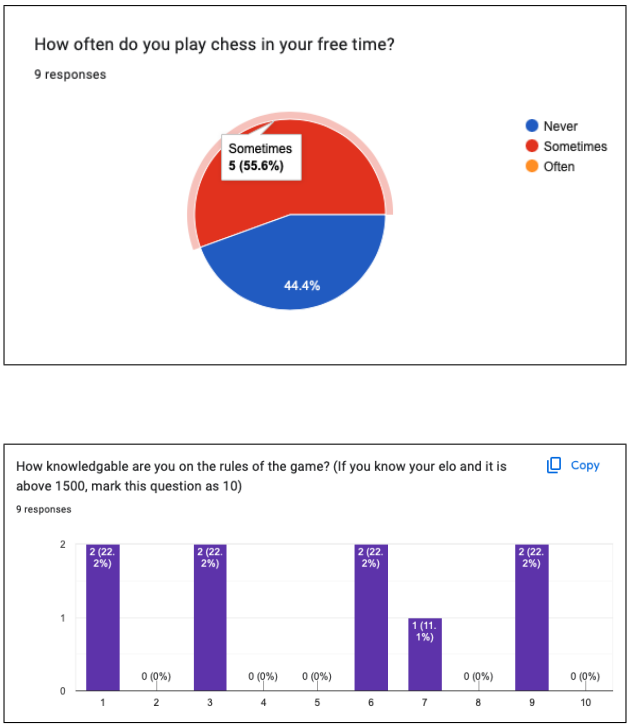

Figure 3.3: A pie chart depicting the frequency of how often users play chess and a bar char showing how they would rate their knowledge on the game.

3.2 User Testing and Feedback

User feedback was collected through google forms and the automatically generated spreadsheet can be accessed via the following link:

`https://docs.google.com/spreadsheets/d/1DjMKbAfgBAj0ksE6rK4WPJBflKK6LeDs_MvxAArTZa0/edit?usp=sharing`

There is only so much testing we can do as developers before we need to outsource our tests to other, consenting users. A side effect of being involved in the development is that we are unable to simulate the experience of a new user. We know exactly how the app functions and can navigate with ease, but can we say the same for people visiting our app for the first time? We will discuss this in the following sections.
The first part of the feedback form helps us gauge the diversity of our sample, by asking questions about the user's background with chess. By looking at figure 3.3 we can see that we have a mix of experienced chess players and those who are new to the scene. This is beneficial to us because different people want different things, and all of this feedback can be processed and implemented to create a program that is enjoyed by everyone. For example, the experienced players want features that they are used to, like the ability to draw on the board in order to plan their next move. Meanwhile, new users want features that will help them play the game, by giving instructions on the rules and even possibly hints on what move they should make.

3.2.1 Functionality

The most important statistic to consider though, is the percentage of players who were able to start and finish a game of chess... Which was 100% of the participants! Despite the range of people who tested our app, every single person was able to play at least one game of chess until the end, which was the main objective of this project! We will now look into the other aspects, all of which are important for improving the user experience.

Where did you move?

A common question that was said during the tests involved asking the opponent what piece they had just moved. Depending on who you play against, they may not be so willing to answer in order to gain an slight advantage. To eliminate this problem, we can learn from the feedback and implement a means of visualising the most recent move that was made. The feature has subtly made an appearance already in figure 3.1, through the form of translucent pink squares. These are drawn where the piece has moved from as well as where it has moved to. For reference, this would be the queen from d8 to h4 on the left and bishop from f8 to g7 on the right.

Responsiveness

As an online application, responsiveness is an important factor to consider when thinking about the user experience. When deciding where the location for the websocket server, Europe was chosen as that is the closest location to where all users will be testing. The responsiveness condition was clearly met when looking at the user feedback where 88.9% of users rated the responsiveness as 5/5. The one tester who rated 4/5, also commented in the optional box that the responsiveness was "perfect", so there may have been some misunderstanding when they gave their rating. This test only considers participants in Europe, unfortunately there was no one reliable to test with outside of this area, so we can only say for certain that it is responsive in Europe and nothing concrete can be said for continents outside of this area.

Game Keys

It was surprising to see that no one had mentioned having issues with the game keys. It wasn't considered until testing had commenced but the characters 0 and O are very similar and this could be confusing to those who are trying to join a game. Even though no one had commented on this as an issue, it would be wise to remove one of these characters to avoid the potential for confusion in the future.

3.2.2 Visuals

A goal of the project was not only for it to be functional but also visually pleasing; some of the changes may already be noticeable from the figures earlier in this chapter. After taking some feedback into consideration, the website now looks more polished and cohesive. The colour palette has changed to monochrome and this pairs well with the black and white pieces. Additionally, the neutral colours allow for important text like hints to stand out; this also has

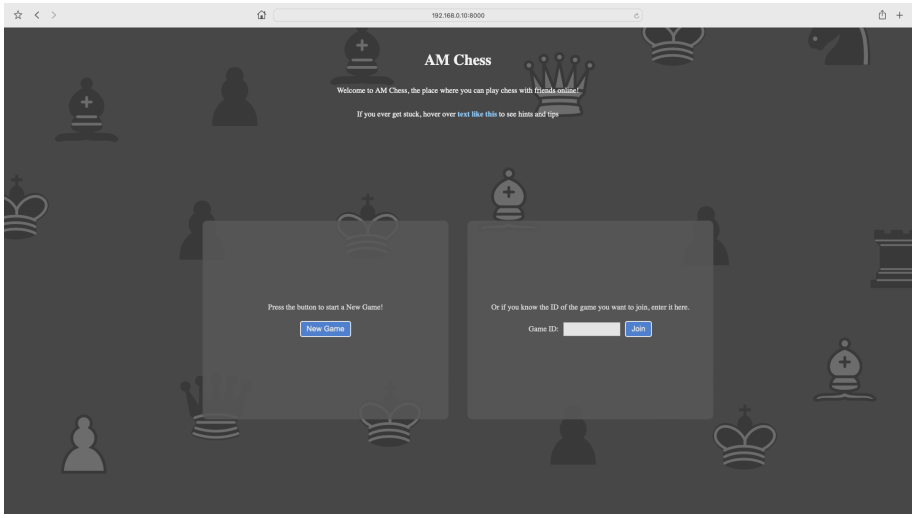

Figure 3.4: The welcome page with updated visuals, welcoming text and important information relating on how to interact with the hint system

the same effect for the board. Additional effects, such as a border and shadow, were also implemented for the board and this gives it more depth and makes it stand out even more. After all, the board is the most important part of this app, so it should have the main focus. The board not maintains more of its colour too because the highlighting for available moves is now translucent, which means you can still see a part of the original colour underneath.
The background of the website has also been updated and it reuses assets we have already implemented for the game. Upon entering the website, the program pseudo-randomly draws chess pieces to the background. From previous experience, when "programmer art" is left to complete random, it is never usually pleasing to look at. Hence why the positioning is only random, to a degree. The way the position algorithm works, is by diving the background into cells and only considering every other cell. We then generate a random value to offset the position, this is capped to ensure that no two pieces overlap. Finally the background has its opacity lowered so it is not confused with the foreground elements. This change was suggested by one of the testers on the questionnaire and it definitely improves the overall feel and quality of the app. The effects can be seen in figure 3.4.
An interesting visual feature was hinted to earlier and it's something that occurs if someone *wins*, and only if someone wins. This feature is a confetti canon that shoots multicoloured confetti around the screen. This was developed using a basic, custom made particle system. Being completely separate from everything else, this code was developed in its own file and is imported in the "game.js" file and is executed during the "win" function. This was added to provide some recognition to a user's win because multiple people commented on the anti-climatic feeling of completing a game. A figure will not be provided for this feature because a still picture would not do it justice. A demonstration of the fanfare can be seen in the demo video, which can be found in section 4.3.

3.2.3 Audio

Despite it appearing in all 3 examples in section 1.3, sound was not considered for this project. This was a mistake because sound plays a big role in immersion as well as indicating that a move has been made. As mentioned in the previous section, there is now an added visual for the last move that has been made. This is definitely a step in the right direction but after first hand experience and looking at the user feedback, it is clear that an auditory que would be beneficial. Unlike the text, that could go unnoticed, a sound is much more likely to alert a user that it is their turn. From observation, a lot of time was spent for both people waiting for each other to make a move, hopefully adding sound would make this less of an issue.

3.2.4 Inexperienced Players

Finally we will look at the feedback given by users who are new to chess. One of the testers said that they didn't know what check was and why they were not allowed to move certain pieces during this time. After hearing this feedback, we can include elements with certain traits that can offer advice. This is implemented with bold text in a unique colour, to indicate that it is interact-able. A user can either hover over it with their mouse on a computer or press it on their touchscreen device. This will cause the corresponding text to explain the current situation. So in this example, when a user is in check, if they hover over the "(In check)" text, they will see a prompt that reads: "Check means that a piece is attacking the king and he needs to get out of danger". A user is made aware of this functionality from the welcome page, where it explains explicitly how to view hints and tips. There is even an example on the page to reinforce the learning. After accessing the hint, it remains on the screen for 5 seconds. It used to appear for less time because there isn't a lot to read. However, from previous experience, this is a naïve thought to have because it is quicker to read text when you've written it yourself and already know what it says. Therefore, it will stay on the page for longer. Even though some users may be able to read it with lots of time spare, it is better to implement it this way so that it is accessible for more users.
Due to both the welcome page and chess page using hints, we have written the functionality in a new file that is included in both web pages. The file contains an entry function that accepts a "hint anchor" element - the text we hover over, and a "hint text" element - the text that includes the hint. The function then assigns the necessary event listeners to the anchor, including mouseenter and mousemove. To add some life to the prompt, if a person uses their mouse, the prompt will follow the mouse (so long as the mouse continues to hover over the anchor text). The ability of encapsulating the code to make something "hover-able" (to have an effect when hovered over), means it can be used multiple times but only written once, which aligns with the DRY (don't repeat yourself) principles. It means we can also assign it to elements easily during runtime. The best example of this is the turn text. This element is not usually hover-able, but if a user attempts to move when it is not their turn it becomes red and hover-able. This was also the result of a change from user feedback, where a user did not like receiving a browser error message for an in game error. So now when a user hovers over the (red) turn text, it will emphasise that it is not their turn. The text will eventually go back to normal after sometime, meaning it is also no longer hover-able anymore. This is managed by a

Figure 3.5: Demonstration of using the hover hint feature

second function that does the opposite of the first function and simply removes the event listeners and the class attributes that give it the hover-able look (e.g. bold font weight).

Chapter 4

Discussion

4.1 Objectives

4.2 Features to be added in the future

4.3 Conclusion

A demo has been recorded that demonstrates all of the features we've discussed throughout this report and can be accessed via this link:

References

[1] https://www.chessclub.com/sitehelp/about. (Accessed on 27/04/2023).

[2] https://caissa.com/about.html. (Accessed on 27/04/2023).

[3] https://www.rottentomatoes.com/tv/the_queens_gambit. (Accessed on 30/04/2023).

[4] https://caissa.com/whois.html. (Accessed on 27/04/2023).

[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP#. (Accessed on 30/04/2023).

[6] https://caissa.com. (Accessed on 28/04/2023).

[7] https://caissa.com/learn/_train. (Accessed on 27/04/2023).

[8] https://www.chess.com/about. (Accessed on 27/04/2023).

[9] https://www.chess.com/variants. (Accessed on 30/04/2023).

[10] https://www.chess.com/membership?c=navbar.

[11] https://lichess.org/about. (Accessed on 27/04/2023).

[12] https://stockfishchess.org/about/. (Accessed on 30/04/2023).

[13] https://stockfishchess.org/about/. (Accessed on 30/04/2023).

[14] https://computerchess.org.uk/ccrl/4040/index.html. (Accessed on 30/04/2023).

[15] https://ratings.fide.com/top.phtml?list=men. (Accessed on 30/04/2023).

[16] https://handbook.fide.com/chapter/E012018. (Accessed on 25/04/2023).

[17] https://websockets.readthedocs.io/en/stable/. (Accessed on 25/04/2023).

[18] https://websockets.readthedocs.io/en/stable/intro/tutorial3.html. (Accessed on 25/04/2023).

[19] https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.loop.create_server. (Accessed on 26/04/2023).

[20] https://websockets.readthedocs.io/en/stable/intro/tutorial2.html#broadcast. (Accessed on 25/04/2023).

[21] https://web.archive.org/web/20220714020647/https://bencentra.com/code/2015/02/27/optimizing-window-resize.html. (Accessed on 26/04/2023).

[22] S. J. Edwards. PGN Standard. `https://archive.org/details/pgn-standard-1994-03-12`, 1994. (Accessed on 25/04/2023).

[23] jurgenwesterhof. File:Chess Pieces Sprite.svg. `https://commons.wikimedia.org/wiki/Template:SVG_chess_pieces`, 2014. (Accessed on 25/04/2023).

[24] S. M. Tcec season 23 superfinal: Leela chess zero vs stockfish. `https://tcec-chess.com/articles/Sufi_23_-_Sadler.pdf`. (Accessed on 30/04/2023).

[25] M. H. J. R. *A History Of Chess*. Clarendon Press, Oxford, 1913.

Appendix A

Self-appraisal

A.1 Critical self-evaluation

A.2 Personal reflection and lessons learned

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

A.3.2 Social issues

A.3.3 Ethical issues

A.3.4 Professional issues

Appendix B

External Material