s

**CHESS**

**Candidate No: 6442**

**Centre No. 12342**

**St. Gregorys Catholic Science College**

Table of Contents

[Analysis - 3 -](#_Toc73880209)

[Problem identification - 3 -](#_Toc73880210)

[Challenges that will be faced: - 3 -](#_Toc73880211)

[Broad notion of what the system will do: - 4 -](#_Toc73880212)

[Interviews - 5 -](#_Toc73880213)

[Selected method of interview and justification: Online survey (SurveyMonkey) - 5 -](#_Toc73880214)

[Survey and results - 7 -](#_Toc73880215)

[Third parties and acceptable limitations - 11 -](#_Toc73880216)

[Observation of existing solutions - 12 -](#_Toc73880217)

[The Chess Lv. 100 - 12 -](#_Toc73880218)

[chess.com - 18 -](#_Toc73880219)

[Objectives for the proposed system - 22 -](#_Toc73880220)

[Potential solution and justification of chosen system - 24 -](#_Toc73880221)

[Documented design - 26 -](#_Toc73880222)

[Structure/Hierarchy Diagrams - 26 -](#_Toc73880223)

[Structure Diagram - 26 -](#_Toc73880224)

[Initial GUI drawings - 28 -](#_Toc73880225)

[System flowcharts - 33 -](#_Toc73880226)

[Data Flow Diagrams (DFDs) - 34 -](#_Toc73880227)

[Database design - 35 -](#_Toc73880228)

[HCI (Human-Computer Interaction)/ Screen designs - 36 -](#_Toc73880229)

[Technical solution - 37 -](#_Toc73880230)

[chess.py (!MAIN FILE!) - 37 -](#_Toc73880231)

[Single Player MODE - 44 -](#_Toc73880232)

[singleplayerENGINE.py - 44 -](#_Toc73880233)

[board.py - 57 -](#_Toc73880234)

[piece.py - 71 -](#_Toc73880235)

[ai.py - 72 -](#_Toc73880236)

[MULTIPLAYER MODE - 76 -](#_Toc73880237)

[multiplayerENGINE.py - 76 -](#_Toc73880238)

[board.py - 84 -](#_Toc73880239)

[piece.py - 98 -](#_Toc73880240)

[Chat-app Navigation Overview - 100 -](#_Toc73880241)

[server.py code - 101 -](#_Toc73880242)

[client.py code - 103 -](#_Toc73880243)

[Testing - 107 -](#_Toc73880244)

[Input & output Testing - 107 -](#_Toc73880245)

[Chess board print. - 107 -](#_Toc73880246)

[Move turn printing (singleplayer mode). - 109 -](#_Toc73880247)

[Move turn printing (multiplayer mode) - 110 -](#_Toc73880248)

[Piece movements - 111 -](#_Toc73880249)

[Taking out other pieces. - 112 -](#_Toc73880250)

[Main window GUI - 114 -](#_Toc73880251)

[Castling - 116 -](#_Toc73880252)

[Pawn exchange - 117 -](#_Toc73880253)

[Single player mode - 117 -](#_Toc73880254)

[Checkmate and stalemate display functions. - 118 -](#_Toc73880255)

[Multiplayer mode - 119 -](#_Toc73880256)

[Online mode - 120 -](#_Toc73880257)

[Chat application - 121 -](#_Toc73880258)

[Testing Outcome Screenshots - 122 -](#_Toc73880259)

[Evaluation - 132 -](#_Toc73880260)

[Comparison of Project performance against objectives. - 133 -](#_Toc73880261)

[Potential for Future Developments - 136 -](#_Toc73880262)

[Analysis of user Feedback - 136 -](#_Toc73880263)

# Analysis

## Problem identification

Currently, there is no chess application in the Microsoft store or even a website that provides users with the amalgamation of playing multiplayer (1v1) with friends/family whilst enabling them to have an intuitive chat application integrated within it with further flexibility of giving them the choice of playing with the computer (**artificial intelligence**); instead, they have to use an alternative social media application such as Discord to communicate with their peers.

Furthermore, most of the chess apps doesn’t provide the users with the option of customising the chessboard, having a bespoke system that rewards the user with points whilst also saving their game progress using their email-address.

Having an integrated chat application within the game will have many benefits such as avoiding the hassle of opening a third-party social media website/application (for instance, Discord) (which may deplete the performance of the system as the computer multi-tasks. Furthermore, it will reduce the complexities to the elderly of engaging with the computer traditionally.) Additionally, it will refine the user experience by taking it to another level, for instance, increasing user satisfaction.

### Challenges that will be faced:

Firstly, an obvious problem of this project would be the issue of administering PyGame. PyGame doesn’t provide additional features when compared to other graphics programs such as Unity which helps to build high-quality games and deploying across various other platforms tablets, phones, apple products decreasing the overall flexibility of the game.

Secondly, an algorithmic issue would include applying the minimax algorithm which is a decision rule used by the AI to minimise the worst-case potential loss; in other words, when the user selects play with computer, the computer will consider all of the best opponent responses to their strategies for minimizing maximum loss.

Thirdly, on the server side of the game, several issues such as slow servers and loading time could pose a major issue to the functioning of the game. Furthermore, additional problem would be the export of messages within the game’s in build chat application.

And lastly, making an intuitive GUI which would provide the users with the flexibility of customising the board to the colours of their likeness would also be onerous to implement, not because it is hard, but coming up with a UI that allows the user to seamlessly select the colours of their choice without hassle.

### Broad notion of what the system will do:

The features of the application that would be required for this solution would be an application with a complex server system to facilitate the game between the 2 users, a complex tree structure which will allow the application to anticipate the best move to challenge the user and all the moves countering them, and an intuitive GUI which would provide them with the flexibility of customising the board to the colours of their likeness.

## Interviews

### Selected method of interview and justification: Online survey (SurveyMonkey)



Due to disruptions caused by coronavirus, it was simply impossible to conduct interviews with my end-user(s) face-to-face. This has led to me to conduct online interviews with my end-user(s) which was onerous due to the process of designing the questionnaire in a way that eliminated/reduced bias between participants. Participants weren’t deceived and were shared with the original aim of the study; which was to find out the features they wanted in the chess game.

|  |  |
| --- | --- |
| Benefits | Drawbacks |
| * Able to talk to people directly * Being able to build a relationship more easily * Having the ability to ask questions about the project whenever a doubt which could be arranged by a meeting * More detail can be gained instantly * Can get a better understanding of my stakeholders and their problem | * Need to travel (can’t due to Covid) * They may get hard to hold of * Not being able to give as much time as hoped, meaning the meeting is rushed * Cancelling at the last-minute leading to delays in my research * Getting agitated due to the in-depth output needing to produce |

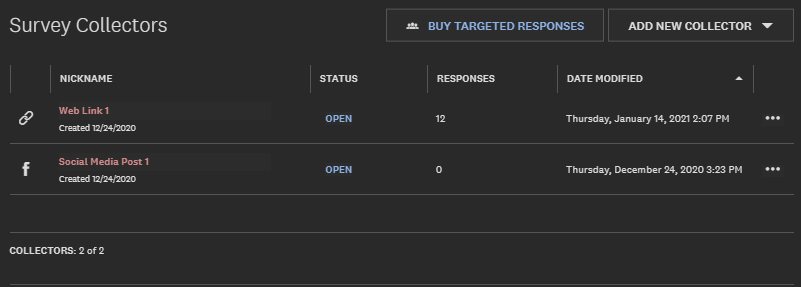
As can be seen in the table above, the benefits tend to outweigh drawbacks associated. Consequently, this would mean that some elements of the essay will miss the opportunities a physical interview will provide such as the density of details which can be gained.

A **survey** is a way of collecting data without needing having to be present. They are convenient due to the portability it gives to the user of being able to create it quickly. The benefits and drawbacks of using a survey are discussed below.

|  |  |
| --- | --- |
| Benefits | Drawbacks |
| * Questionnaire platforms such as SurveyMonkey gives you the feature of analysing data without much effort * A range of questions can be asked * Survey links can be posted on social media apps which gives the user free-will of answering the question * Being able to send to many people at once * Conducting an online survey facilitates low-cost and a fast mean of collecting data from the target population. Sending questionnaire through links are more affordable/free than face-to-face methods * Responses answered will be stored on an online database with data presented in terms of descriptive statistics in a way that is professional and convenient | * Questions cannot be clarified immediately * You may need to tell people to fill the survey repeatedly * Designing the survey takes time and there is always the possibility of bias * Survey fraud i.e. people who answer the online surveys for the sake of doing it with the desire of not contributing to the advancement of the project * Being sure of asking the right questions * Data entered by users may not always be accurate |

### Survey and results

**To examine the survey, please click** [**here**](https://www.surveymonkey.com/stories/SM-G7QV58C2/)**.**



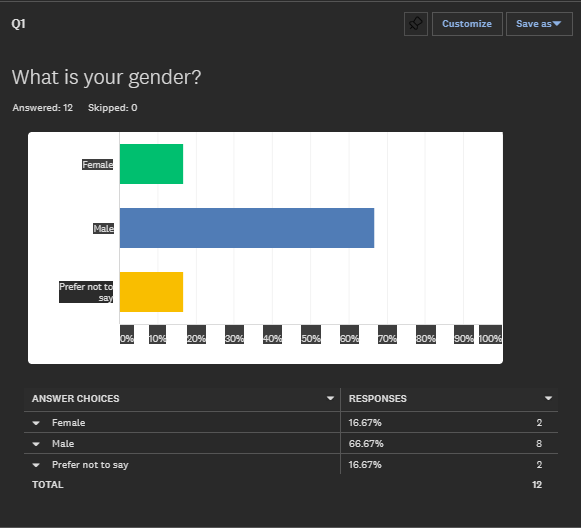
Social media applications and email clients that were used to distribute survey links were;

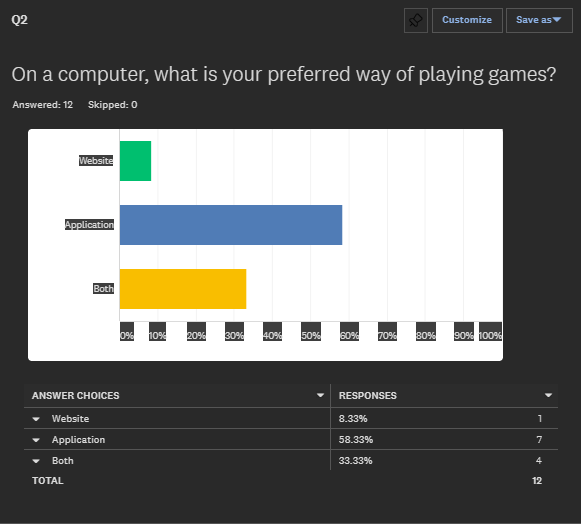
* WhatsApp
* Signal
* Google Chrome
* Outlook

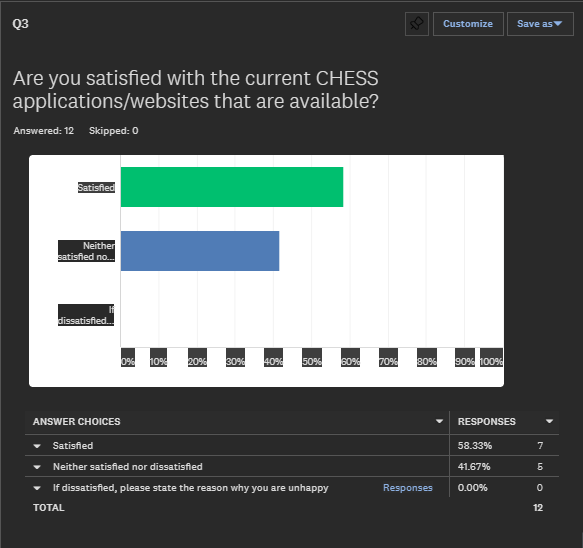
The sampling strategy used here was opportunity sampling, i.e. the sample of participants were produced by selecting people who were most speedily available at the time of the study. This technique brings its own set of strength and limitation; the strength being that the questionnaire taking less time to locate the sample than if using one of the other techniques. However, a limitation of this technique is that it is inevitably biased because the sample is drawn from a small part of the population. Results in this questionnaire was picked from a sample of a population who knew how to play chess.

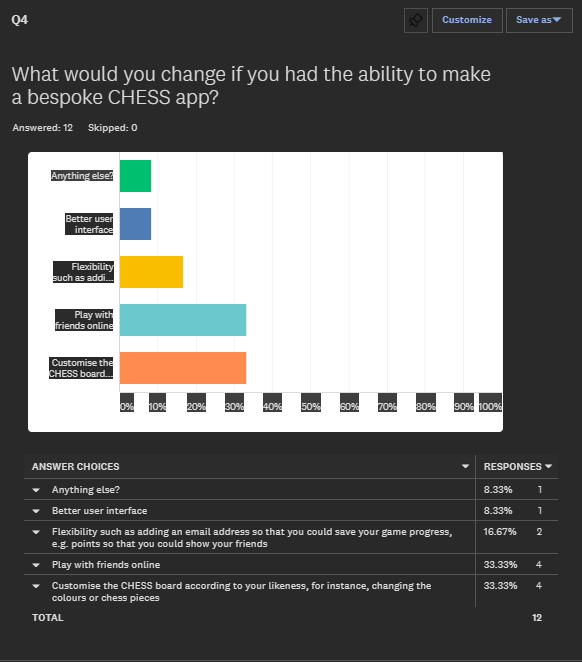
**Questions in the survey that had design flaws;**

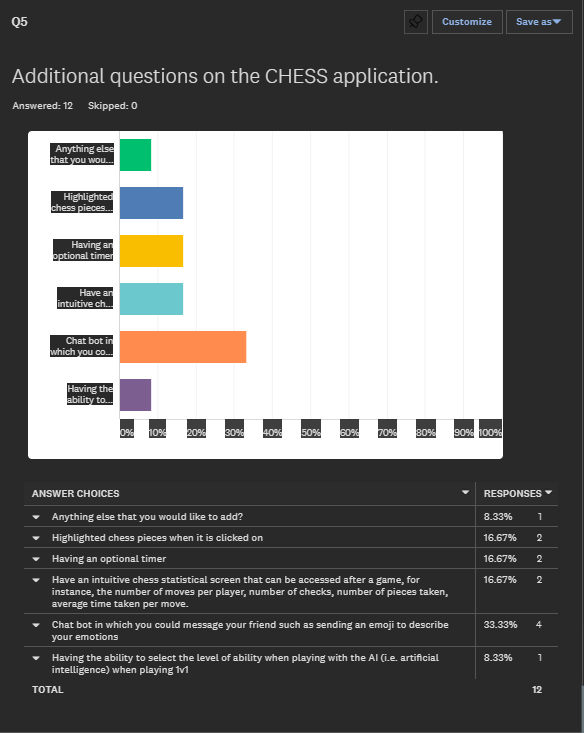
* **Q4. What would you change if you had the ability to make a bespoke a chess game.:** In Q4, the option of adding any other feature was flawed because it was a selection question, however, in turn should’ve have been a compulsory question to answer if the respondent had selected it.

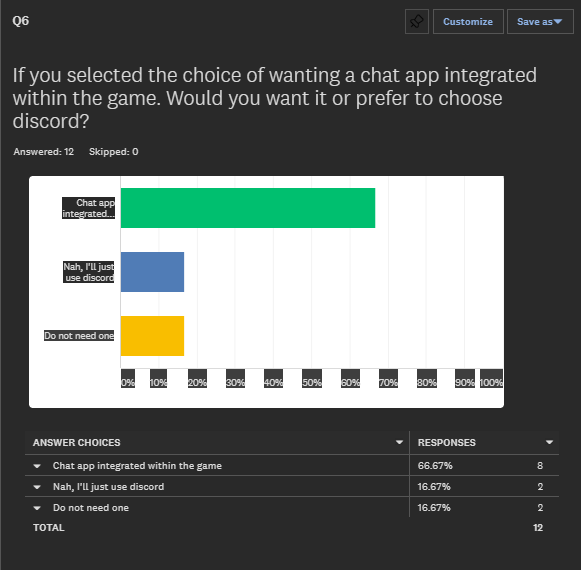


****

****

****

****

****

## Third parties and acceptable limitations

Presently, the users of this system would mainly comprise of my family and friends who comprehend the game of chess. Most of my end-users are computer literate to a level which means that they should not find it onerous when interacting with the system. Nevertheless, it is intrinsic for my end-users to spend as little as time possible tweaking the settings of the game conveniently whilst having an intuitive user experience playing and communicating, so ease-of-use is an intrinsic factor.

The third party will be directly affected by the moves played by the other user, so the chess board should be of similar format to those or will be familiar with from chess.

The limitations to my system are as follows;

* **My proficiency of coding:** Developing the AI section of the game will be too complex for me. To solve this, I will be using a chess library that will generate moves that would strategically play with the user. However, creating a server module with the project will be challenging with the amount of troubleshooting that is required in order for the user to have a stable connection whilst playing with their family/friends. I will be creating the solution with Python and also could write the solution in a variety of different paradigms.
* **Hardware and software constraints:** There is a sparse hardware that I can use, and a limited amount of software available for me to design the system. Software that will help me manage my time constraints is Notion with additional websites used across Chrome to create various graphs and diagrams.
* **Difficulty of installing a python IDLE:** Since the project would require minimum user effort. I will create a class in which when the solution is run. It will download the IDLE to run the game eliminating the need for my users to troubleshoot when they want to play the game.
* **Communication constraints:** I will be using the communication application *Discord* to communicate with my computer science group to discuss which attributes will make my game more desirable than attributes that would be inefficient and unachievable in given time constraints.
* **Time constraints:** The system needs to be completed by Easter.

## Observation of existing solutions

### The Chess Lv. 100

**Overview:**



**The Chess Lv.100** is a cross-platform chess game available in the window store for PCs or in the play store for android smartphones. It has 100 adjustable playing levels based on the engine “Crazy Bishop” which is another game that has inspired this one. It has a human vs computer feature allows the user to manage the level of difficulty giving the user the flexibility of choosing the level of difficulty he/she wants. Out of these 100 levels, it ranges from 1 being the weakest and 100 being extremely difficult (unbeatable). It has an inbuilt rating mode which scores the user on how they play with the system. It also has a human vs human mode in which the user can play with family/friends on a single computer.

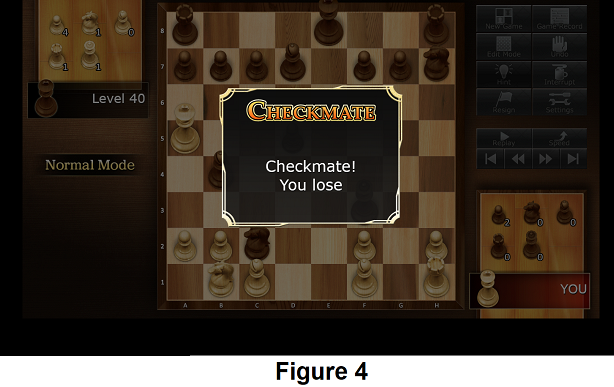
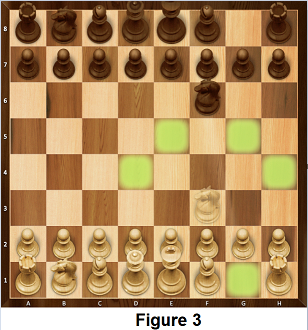
Further features include;

* Review mode
* Autosaving game progress
* Ability to view full game history in the game record and restart game from the selected move, which is practical for improving the user’s chess
* PGN file support for both reading & writing
* Hint facility (giving the user clues on which step is the most appropriate)
* Enter & analyse position you like in edit mode

The process is as follows:

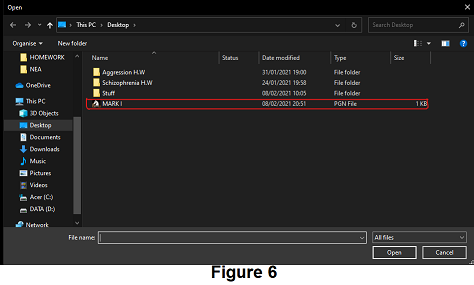
* 1. To start a game, the user is first asked to select the option: New Game. [Figure 1]
  2. The New Game Menu is opened [Figure 2] which allows the user a plethora of options starting with;
     1. **Game Mode**: There are two options available to the user; one being Rating which scores the user to manifesting their skills and the other one is normal which is basically playing with the AI part of the game.
     2. **Type of the Game:** Over here, the user is given the option of playing with the computer or with a human which works by passing the control to the next player or is basically pass and play.
     3. **Select piece colour:** If the user exclusively selects the vs Computer mode, then they are given the option to select the piece colour (black/white).
     4. **Computer Level:** Option which allows the user to select the level of difficulty it wants the computer to play with.
     5. **Time control options:** It provides the user the flexibility of adjusting the level of time each opponent gets.
     6. **Material handicaps:** With this option, the user can give some of his pieces, to the advantage of their opponent. This type of handicap has the advantage that is simple to tune, and can be easily adjusted to correspond to the difference in strength between the two players.
  3. Playing the game

1. The user selects the piece he wants to move, in which this case are only the pawns and knights that can move.
2. The system shows the moves that the user is permitted to do. **[Figure 3]**
3. The computer searches for the most appropriate move from the game tree. Theoretically, the system will examine all the moves, then will counter those moves, which takes mostly one second.
4. The system checks for a check or checkmate.
   1. If the system finds that a piece is having the chance to take the king of the opponent, then the system will display the “Check” message where the user has to guard his king or move it. Afterwards, if the system finds that the user has moved the piece, it could take out other pieces for instance
   2. If the system finds that the opponent players king is in check (threatened with capture) and there is no way to avoid the threat. Checkmating the opponent wins the game. For instance, if the user selects Level 100 difficulty level, the system will use the scholar’s mate where it will defeat the user in two moves. **[Figure 4]**
5. If the user has cleared the checkmate, the game will perform the same procedures again.



* 1. After the game has finished;

1. Following the defeat (either the system/user) the game provides the option of saving the game in the PGN format which is an abbreviation for **Portable Game Notation** which is a standard plain text format for recording chess games. PGN is structured for easy parsing and generation by computer programs. The usual filename extension is .PGN. Otherwise, the user is also able to save the game in the form of a .ucf file.
2. Game record options; **[Figure 5]**
   1. **Load game record:** This is where the user can load a pre-existing game record within the game storage which can be easily accessed without loading any pre-existing files saved on the computer. The file is stored in the .ucf format which is stored which can be even accessed if the user has logged in another device (using their Microsoft account).
   2. **Save game record:** This is where the user gets the option of saving the game in a .ucf format. It can be used in the case where the user is unable to play. The user can just save the game record and restart it from the selected move from where he played last.
   3. **Load game record (PGN):** This is where the user can open the game stored in the computer. For instance, in **[Figure 6]** I saved a game a week ago (i.e. MARK I). The game will open the file explorer which will only open files that are in the .PGN format. The file is then open **[Figure 7]**.
   4. **Save game record (PGN):** This is where the user can save the game within his computer storage.
   5. **Show game record:** This function shows the user the moves they have and the computer played already. It gives the user the flexibility of moving backwards to the number of step he/she wants.

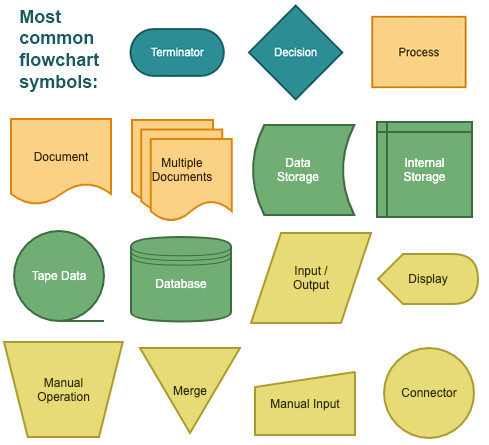




**Features that could apply to my solution:**

The concept of Chess Lvl.100 is similar to my solution, such as implementing an AI which the user can adjust to play with; however, more emphasis will be placed on how the user will be able to play with their friends/family seamlessly. This application showing the settings was very useful to me to set up the GUI which enabled me

**System Flowchart:** (Chess Lvl. 100 system flowchart on the next page)

****



### chess.com

**Overview:**

**Chess.com** is an internet chess server, internet forum and social networking website. As can be seen in the picture above, it provides two options. Play online enables the user to play with a random individual online. It matches the user to another opponent with a mini-questionnaire built-in providing options ranging from **(New to Chess - Advanced)** with additional features such as playing as a guest (i.e. not saving user progress online) and sign up/log in (i.e. saving user progress online). Similar to my project, it too provides users to play with friends but with no integrated chat application when playing with friends.

**Features that could apply to my solution: (More focus on the Play rather than the vs. computer mode here)**

**The play mode process is as follows;**

****

1. **Starting the game:** The user selects the New Game mode **[Figure 8]** which the server pairs the user with another opponent based on their skill points, for instance, in one game, my score was 1373 which the game server matched with another player with relatively close point range as shown in **[Figure 9].** The user is also required to submit the time-period they want to play with the other player.

****

1. **Playing the game:**

**2.1.** The game provides the user the option of chatting whilst playing the game. This feature can only be accessed if the user has verified their email-address.

**2.2.** The user/opponent gets to select the move based on the piece colour i.e. either white/black. White would start first followed by Black.

**2.3.** The game server records the time taken by each person when playing the game. If the user runs out of time, the other person automatically gets the win.

**2.4.** The game server checks for pieces that are taken out by the opponent and displays them with their chess notation and the time taken by the opponent/user to make the move. **[Figure 10 = right side of the screen]**

**2.5.** The system checks for a checkmate/check/stalemate**;**

**2.5.1** If the system finds that a piece is having the chance to take the king of the opponent, then the system will display the “Check” message where the user has to guard his king or move it. Afterwards, if the system finds that the user has moved the piece, it could take out other pieces for instance

**2.5.2** If the game server finds that the opponent players king is in check (threatened with capture) and there is no way to avoid the threat. Checkmating the opponent wins the game. The server will further reduce the points of the user based on the game performance. For example, in my case, due to my loss with Yuri333 the game subtracted 89 points from my net score

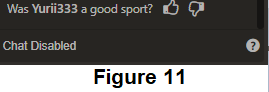
1. **After the game has finished:**

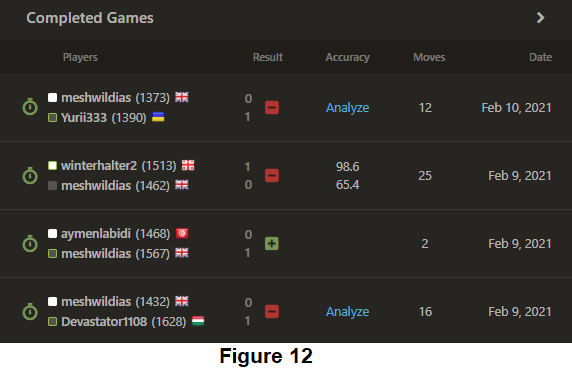
**3.1.** The server makes a game record.

**3.2.** The server than makes a detailed analysis of the game that will show the user why they lost the game.

**3.3.** In the home tab, the user then gets to check all the games they have played and gives a report of how they play and the ways they could be improvised.

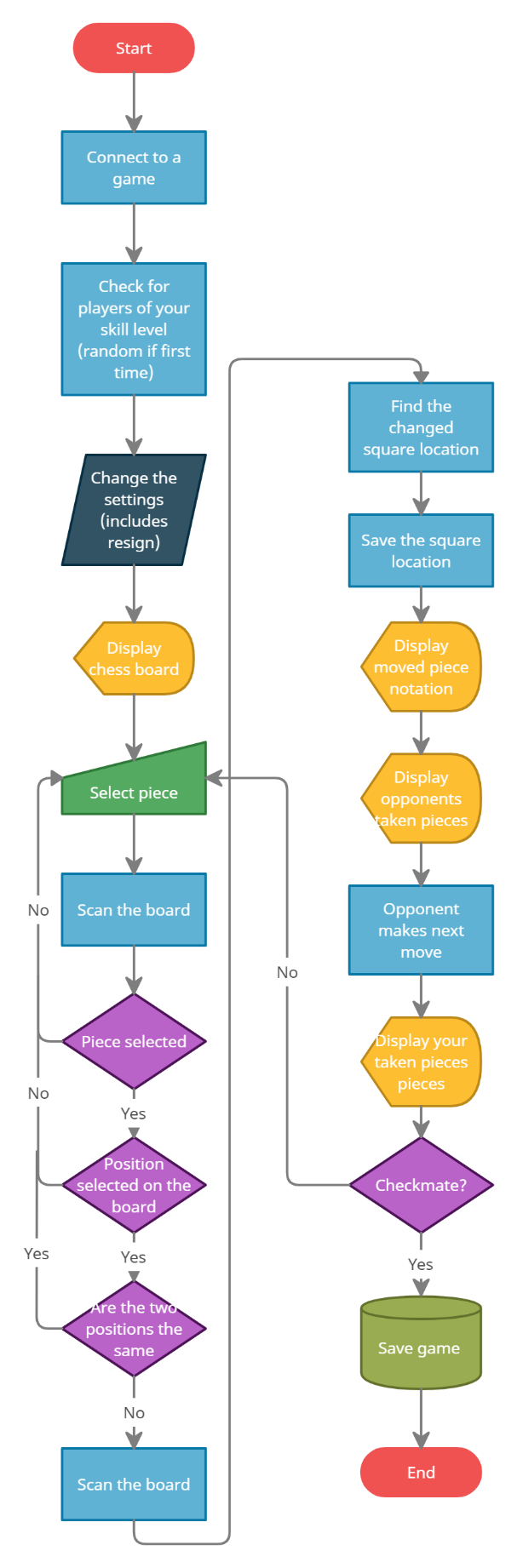




****

**System Flowchart**

This time, we will focus on the vs. player part of the chess.com part rather than the vs. computer to comprehend how the system works.



## Objectives for the proposed system

I have been able to comprehend a lot about the existing system both from observations and by actually playing the game. I was able to discuss with my friends the features they liked from the existing system and what they didn’t i.e. that the system lacked.

1. Opening the game.
   1. The user clicks on the python file named engine.
   2. The game open with the main menu displaying, it will also display tools such which will give the user the option to select whether if they want to customise the game, for example, customise the game with the timer and other features.
2. Selecting the game mode
   1. **Game mode** which will be either
      1. An **AI mode** where the user can select to play with the computer with adjustable levels of difficulty ranging from easy to hard. (PS: haven’t got that working yet. Check if can be completed within time period).
      2. A **multiplayer** where two users can play on a single computer.
      3. An **online** where the user can play with his friend on a single computer.
   2. **Time control:** This feature will only be available in all the modes where the game will deduct time of the layer whenever it’s the user’s turn to move pieces.
3. Printing the board;
   1. Pygame will print the board which is an 8x8.
   2. The piece plan will be saved in a list where each piece is saved which complies with the standard chess board.
   3. The program function will find the chess pieces and will accordingly manipulate the images specified by the dimensions which is 500x500.
   4. The program will fetch the .jpg files from a resources folder where individual pieces will be automatically printed on the board.
   5. The board will be printed with all the pieces laid out seamlessly.

Main Program part

1. Single player **(vs AI mode)**;
   1. The user will press the **Single player mode button** from the Main Menu tab**.**
   2. The user will be displayed with another tab where they will be able to customise the game.
   3. The button to select the piece colour will be displayed. 2 options will be displayed which is either black or white.
   4. The button to select a timed game will be displayed which is by default will last for 10 minutes.
   5. The button to select the **En passant mode** which is a special move in chess that can only occur immediately after a pawn makes a move of the two squares from its starting square. And it could have been captured by an enemy pawn had it advanced only one square. This function will be available in all modes. **(PS: Haven’t completed this function yet).**
   6. White piece player plays first
   7. Chess program executes as expected.
   8. The user can press the Escape button where the user will get the option to select to return back to the main menu.
2. 2 player mode **(vs multiplayer mode);**
   1. The user will press the **Single player mode button** from the Main Menu tab**.**
   2. The user will be displayed with another tab where they will be able to customise the game.
   3. The button to select the piece colour will be displayed. 2 options will be displayed which is either black or white.
   4. The button to select a timed game will be displayed which is by default will last for 10 minutes.
   5. The button to select the **En passant mode** which is a special move in chess that can only occur immediately after a pawn makes a move of the two squares from its starting square. And it could have been captured by an enemy pawn had it advanced only one square. This function will be available in all modes. **(PS: Haven’t completed this function yet).**
   6. White piece player plays first
   7. Chess program executes as expected.
   8. The user can press the Escape button where the user will get the option to select to return back to the main menu.
3. Print the moves on a separate GUI tab.
   1. There should be a separate window where the move of the user should be saved and registered.
4. Piece highlighting.
   1. When the piece is highlighted, the engine highlights the piece.
   2. Furthermore, for that specific piece, it shows the place it can move, for example, L shape for knight.
5. Demonstrates important messages such as stalemate and checkmate.
   1. When the user is in check and has no move available, the engine should print the message of which the piece colour is followed with “has won the game with checkmate”.
   2. When one player has lot most of the pieces, the engine should print the message of which the piece colour is followed with “has won the game with stalemate”.
6. Program quits as expected.

## Potential solution and justification of chosen system

**Potential solution;**

1. Write the solution in python with Pygame running the video game (e.g. computer graphics and sound libraries). The UI would involve Tkinter. Tkinter is the standard GUI library for Python. Furthermore, I am quite familiar with how Tkinter works. The game would involve a lot of classes so Tkinter would be a good choice as it offers a powerful object-oriented interface to the GUI toolkit.

* This solution is not feasible in terms of my time constraints and abilities as I would need to learn Pygame which can be inflexible when compared to other Game development platforms such as GameMaker Studio. It would take too long to design as it involves my clients interacting with other players online.
* My client would manually need to install all the libraries which would make this solution infeasible for older clients.
* I would not be able to integrate the chat application with the main Chess playing window which would mean that the clients would not have an interactive GUI to talk with other members conveniently.

1. Write the solution in GameMaker Studio which provides a lot of facilities in terms of game development. However, this would involve me learning an entirely new language which would propose detrimental problems towards the completion of the project.

* This solution is not feasible in terms of my time constraints as I would need to learn GameMaker which could affect the design aspects of the project.
* In contrast, the interactive libraries that GameMaker offers would add users with a more flexible UI as GameMaker allows games to be played on different platform.

**Chosen solution;**

1. Write the solution in python with Pygame running the video game (e.g. computer graphics and sound libraries). The UI would involve Tkinter. Tkinter is the standard GUI library for Python. Furthermore, I am quite familiar with how Tkinter works. The game would involve a lot of classes so Tkinter would be a good choice as it offers a powerful object-oriented interface to the GUI toolkit.

* This solution is not feasible in terms of my time constraints and abilities as I would need to learn Pygame which can be inflexible when compared to other Game development platforms such as GameMaker Studio. It would take too long to design as it involves my clients interacting with other players online.
* My client would manually need to install all the libraries which would make this solution infeasible for older clients.
* I would not be able to integrate the chat application with the main Chess playing window which would mean that the clients would not have an interactive GUI to talk with other members conveniently.

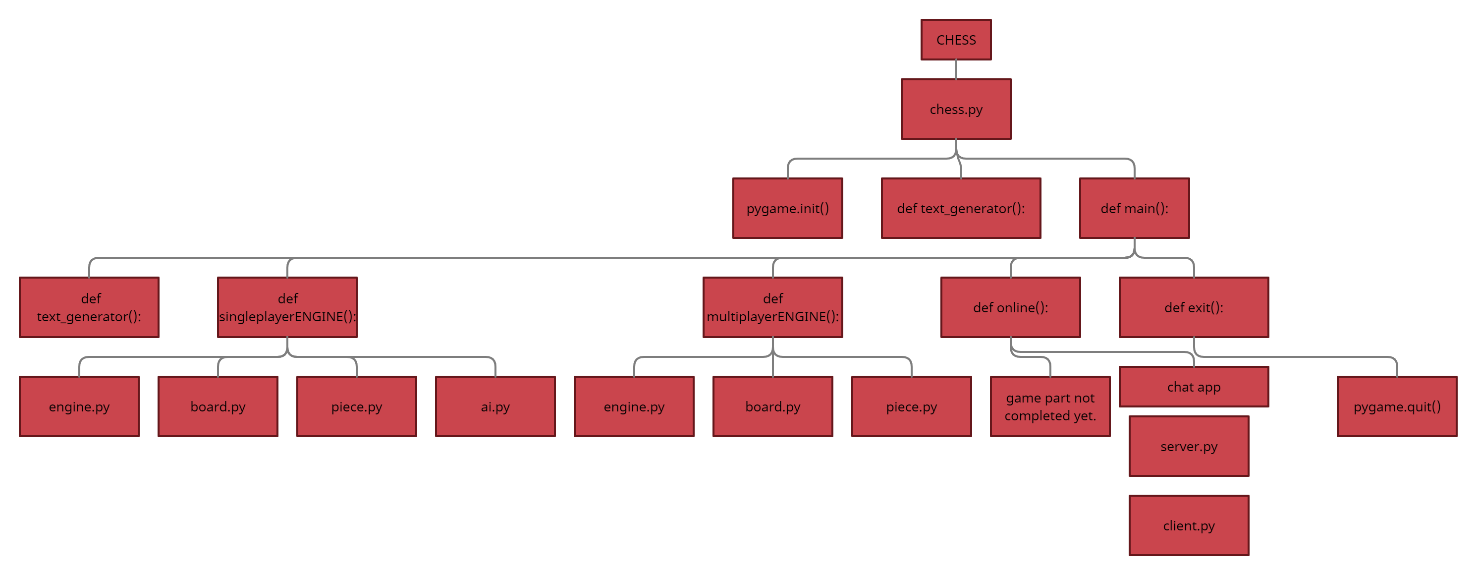
# Documented design

**Objectives;**

* **Understand the purpose of the Documented Design section.**
* **Where appropriate create the following designs.**
  + **Structure/hierarchy chart.**
  + **System flowchart.**
  + **Data Flow Diagram (DFD)**
  + **Database design**
  + **Database queries**
  + **Entity-Relationship Diagrams**
  + **Algorithms**
  + **Data structures**
  + **HCI (Human-Computer Interaction)/Screen Design**
  + **Hardware Selection**

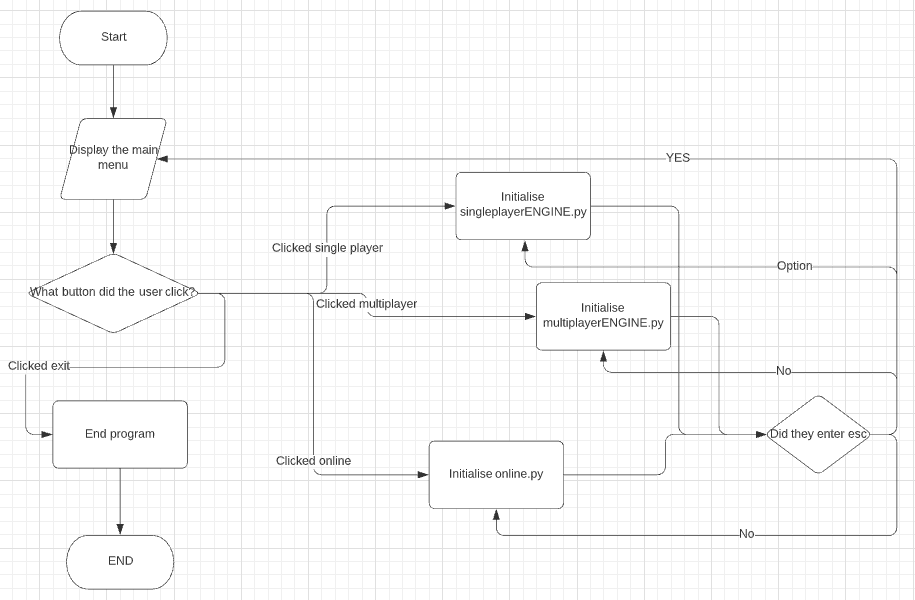
## Structure/Hierarchy Diagrams

### Structure Diagram



### System flowchart

Due to time constraints, this document hasn’t been completed as I was adding more features to my program (technical solution).



### Database

**Normalisation**

Normalisation is a database design technique that reduces data redundancy and eliminates undesirable characteristics like insertion, update and deletion anomalies. Normalisation rules divides larger tables into smaller tables and links them using relationsihps. The purpose of normalisation in SQL is to eliminate redundant (repetitive data) information and ensure data is stored logically.

Presently, there are 3 stages that are widely known when it comes to normalisation.

* First normal form (1NF)
* Second normal form (2NF)
* Third normal form (3NF)

**First Normal Form (1NF)**

Tables that are created in first normal form has a feature of containing no repeating attributes or groups of attributes. This infers that all attributes are to ‘atomic’, meaning a single attribute must not include two data items. For instance, selling price and buying price cannot coincide with each other within a single attribute. If this were to happen, it would make it near impossible to sort only on selling price or only on buying price. Sometimes, an extra table is required to join two tables together to prevent any repeating attributes

**Second Normal Form (2NF)**

When a table is in second normal form, it is to be in first normal form and contain no partial dependencies. This instance can only occur if the table’s primary key is a composite key. First of all, a primary key can be identified by its ability to uniquely identify each table record in a given attribute and table. However, a composite key is in itself a primary key but consists of two or more attributes or table columns that when linked together, uniquely identifies a table row (also known as a table occurrence). An example of this is an ID that relates to two different attribute but can still be joined to form multiple entities and dependencies. When a table is the only table with a composite key plus not having any other fields that are dependent on any part of the key, the table is indeed in second normal form.

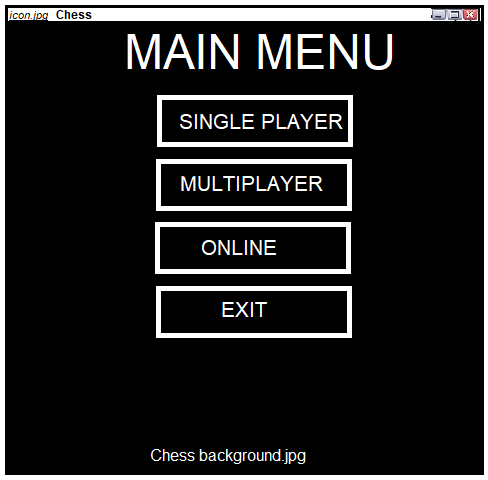
**Third Normal Form (3NF)**

Finally, third normal form databases or tables must be in second normal form and must not contain any non-key dependencies. It is considered a database schema design approach for relational databases. Each attribute must be dependent on the whole key and nothing else which allows for the prevention of duplicating data and avoiding data anomalies. Many other features include a much more simplified data management system while ensuring the database’s referential integrity.

## Initial GUI drawings

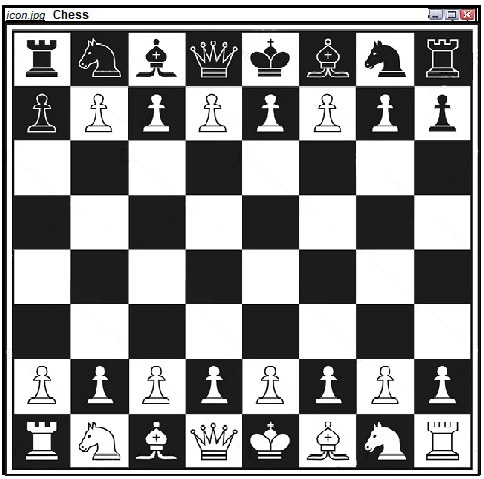
#### Single player button

These drawing are created in microsoft paint. They are my first outlines of what the user interface of the system should look like. They will most likely be the final designs as I will create this in pygame since I have experience in pygame GUI designing.



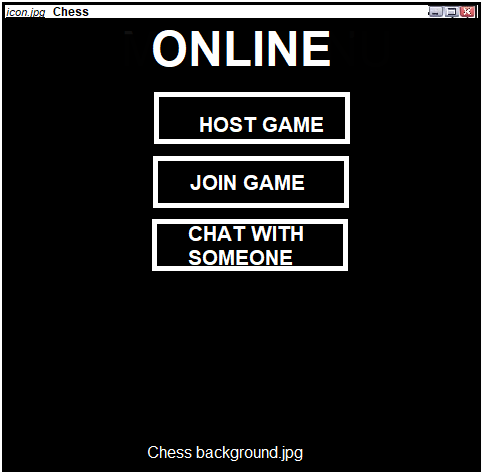
The **‘MAIN MENU’** is the pane in which all the game modes will be selected. From this pane, the user will be able to navigate to their games. After clicking the button, the user will be redirected to the game. Additionally, I will add the feature after where the user enters the button **esc** (escape), they will be redirected back to the screen. Due to the time constraints, I would be able to add a personalisations button where the user would be able to customise the game FPS settings, chess board customisations and music settings.

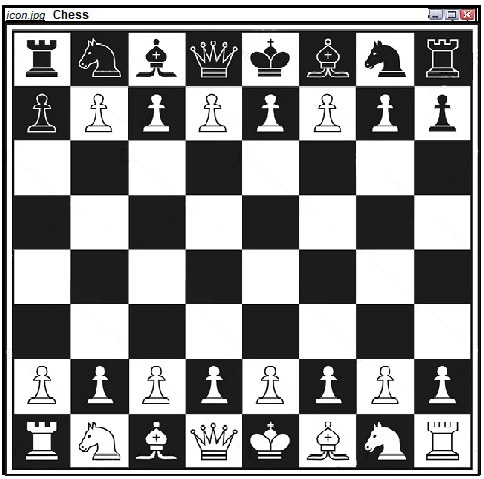
#### Multiplayer button



This is the **‘Multiplayer** Pane. The user will be directed to this page upon clicking the Multiplayer button where the user will play against each other.. If the user enters the button Z, he will be able to rewind the move. Similarly, if the user enters the button **esc** (escape), he would be redirected back to the menu. As usualy, the white player would start first, which in this case will be the user as they would be not give the choice of choosing black. This is due to the problems of time constraints as I wasn’t able to finish my personalisations button in the ‘MAIN MENU’ pane. This is just the demonstration of what I feel that my program may end up looking in the end and is not definite.

#### Online button



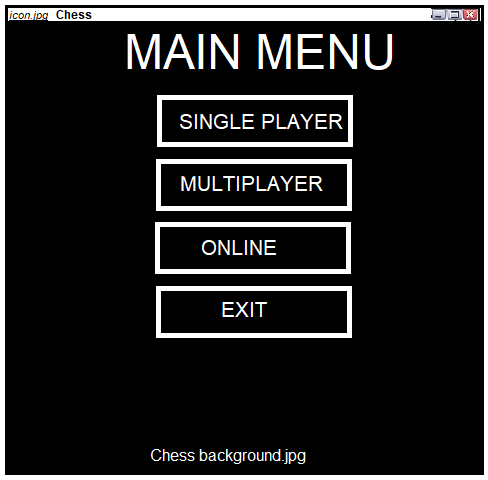


This is the ‘Online’ Pane. The user will be directed to this page upon clicking the Online button where the user will be required to enter the host IP address. There will be 3 buttons in this page;

* **HOST GAME**: The user must click on this first to activate the server.py file and then on join game to enter the chat.
* **JOIN GAME**: The user will be able to join a game if another person is hosting it.
* **CHAT WITH SOMEONE**: This will open another pane where the user will be able to communicate with their friends.

I may add another button to the ONLINE GAME MODE pane which is back to main menu where the user will be redirected to the MAIN MENU pane.

#### Exit button



Finally, we have the exit button which is self explanatory. I will be designing the game in pygame where after the Exit button is pressed. It will quit pygame. It is the opposite of pygame.init() function. It runs code that deactivates the pygame library thereby ending the loop.

## System flowcharts

## Data Flow Diagrams (DFDs)

## Database design

## HCI (Human-Computer Interaction)/ Screen designs

# Technical solution

## chess.py (!MAIN FILE!)

|  |
| --- |
| **chess.py():** In this subroutine, it will manage the entire game. It includes the GUI that will navigate the user with an intuitive chess user interface. It includes shortcuts for the user. It also includes buttons that will direct the user to the web if they don’t know how to play chess. |
| import pygame as p from pygame.locals import \* import sys as s "Importing vital pane production libraries" import webbrowser as browse """Importing webbrowser to open the url if the user does not know to play chess.""" from modes.multiplayer import multiplayerENGINE from modes.singleplayer import singleplayerENGINE "Importing the chess engine."  background = p.image.load('resources/' + 'background.jpg' ) icon = p.image.load('resources/' + 'icon.gif') p.init() clock = p.time.Clock() p.display.set\_caption("CHESS") "Setting the main pane window name as CHESS" p.display.set\_icon(icon) "Setting the pane window as a chess icon"  font = p.font.SysFont("Helvetica", 30) smallFont = p.font.SysFont("Aleo", 25) main\_font = p.font.SysFont("Aleo", 90) incomplete\_font = p.font.SysFont("Aleo", 50) "Setting font variables to produce text."  screen = p.display.set\_mode((500,500), 0, 32) "Declaring screen to be accessed by the program"  darkCyan = (23,174,165) black = (0,0,0) white = (255,255,255) "Setting colour RGB variables."  def text\_generator(text, font, colour, surface, x, y):  *"Subroutine that will generate the MAIN MENU text on screen"* textObject = font.render(text, 1, colour)  textrectangle = textObject.get\_rect()  textrectangle.topleft = (x, y)  surface.blit(textObject, textrectangle)   valid = True "Flag variable that will run the main loop"  def mainGUI():  *"Subroutine that will manage the full game"* clicked = False  "Flag variable that will check if the user clicked"   while valid:  "Infinite loop that will run the game."   screen.fill((255,255,255))  screen.blit(background, (0, 0))  mainMenu = p.Rect((70,12), (360,50))  p.draw.rect(screen, (black), mainMenu)  text\_generator("MAIN MENU", main\_font, (255,255,255), screen, 70,10)  "Producing the main pane."   mouseX, mouseY = p.mouse.get\_pos()  "Variable that will get the x and y positions of the mouse."   but1TEXT = font.render("SINGLE PLAYER", True, white)  but2TEXT = font.render("MULTIPLAYER", True, white)  but3TEXT = font.render("ONLINE", True, white)  but4TEXT = font.render("EXIT", True, white)  howToTEXT = smallFont.render("HOW TO PLAY CHESS", True, white)  basicCommands = smallFont.render("SHORTCUT COMMANDS", True, white)  "Variables that will save the button text."   but1 = p.Rect((150,100), (200,50))  but2 = p.Rect((150,200), (200,50))  but3 = p.Rect((150,300), (200,50))  but4 = p.Rect((150,400), (200,50))  howTobut = p.Rect((0,460), (180,50))  commands = p.Rect((285,460), (210,50))  "Variables that will save the button rectangles with their positions."   if but1.collidepoint((mouseX, mouseY)):  "If button 1 is clicked"  if clicked:  singleplayer()  if but2.collidepoint((mouseX, mouseY)):  "If button 2 is clicked"  if clicked:  multiplayer()  if but3.collidepoint((mouseX, mouseY)):  "If button 3 is clicked"  if clicked:  online()  if but4.collidepoint((mouseX, mouseY)):  "If button 4 is clicked"  if clicked:  p.quit()  if howTobut.collidepoint((mouseX, mouseY)):  "If the howTobutton is clicked."  if clicked:  howTo()  if commands.collidepoint((mouseX, mouseY)):  "If the commands button is clicked."  if clicked:  shortcut()   p.draw.rect(screen, (black), but1)  p.draw.rect(screen, (black), but2)  p.draw.rect(screen, (black), but3)  p.draw.rect(screen, (black), but4)  p.draw.rect(screen, (black), howTobut)  p.draw.rect(screen, (black), commands)  "Program lines that will produce the rectangle buttons."   screen.blit(but1TEXT, (155, 105))  screen.blit(but2TEXT, (170, 205))  screen.blit(but3TEXT, (200, 305))  screen.blit(but4TEXT, (220, 405))  screen.blit(howToTEXT, (0,470))  screen.blit(basicCommands, (290, 470))  "Program lines that will produce the text on the buttons."   clicked = False  for event in p.event.get():  "Loop that will manage user inputs."  if event.type == QUIT:  "If the user clicks the close button on the pane."  p.quit()  s.exit()  "Ending the game"  if event.type == KEYDOWN:  "If the user clicks a keyboard button"  if event.key == K\_ESCAPE:  p.quit()  s.exit()  "Ending the game."  if event.type == MOUSEBUTTONDOWN:  "If the user clicks a mouse button"  if event.button == 1:  "If the user clicks the mouse button once."  clicked = True  "Mouse clicked will be true."   p.display.update()  clock.tick(60)   def singleplayer():  *"Subroutine that calls the singleplayer subroutine"* singleplayerENGINE.playGame()  def multiplayer():  *"Subroutine that calls the multiplayer subroutine"* multiplayerENGINE.playGame()  def howTo():  *"Subroutine that will open the how to play chess link."* browse.open\_new("https://www.chess.com/learn-how-to-play-chess")  ####ONLINE#####  def online():  *"Subroutine that will manage the play online game mode."* BUT1ONLINE = p.Rect((150, 100), (200, 50))  BUT2ONLINE = p.Rect((150, 200), (200, 50))  BUT3ONLINE = p.Rect((150, 300), (200, 50))  "Variables that holds the button positions"   host = font.render("HOST GAME", True, white)  join = font.render("JOIN GAME", True, white)  joinchatroom = font.render("CHAT", True, white)  "Subroutine that will hold the text data to position on the buttons."   run = True  "Flag variable that will manage the loop"  click = False  "Flag variable that will manage the mouse clicks"  while run:  screen.fill((255, 255, 255))  screen.blit(background, (0, 0))  mouseXpos, mouseYpos = p.mouse.get\_pos()  "Code similar to the mainGUI() subroutine"   if BUT1ONLINE.collidepoint((mouseXpos, mouseYpos)):  "If the user clicks the first button from online mode"  if click:  incomplete()  if BUT2ONLINE.collidepoint((mouseXpos, mouseYpos)):  "If the user clicks the second button from online mode"  if click:  incomplete()  if BUT3ONLINE.collidepoint((mouseXpos, mouseYpos)):  "If the user clicks the third button from online mode"  if click:  didnotINTEGRATE()   p.draw.rect(screen, (black), BUT1ONLINE)  p.draw.rect(screen, (black), BUT2ONLINE)  p.draw.rect(screen, (black), BUT3ONLINE)  "Drawing the buttons"   screen.blit(host, (175, 105))  screen.blit(join, (175, 205))  screen.blit(joinchatroom, (200, 305))  "Drawing the text on the buttons"   click = False   for event in p.event.get():  "Managing the user inputs"  if event.type == QUIT:  "If the user clicks the close button on the pane"  p.quit()  s.exit()  if event.type == KEYDOWN:  "If the user enter a keyboard key"  if event.key == K\_ESCAPE:  run = False  if event.type == MOUSEBUTTONDOWN:  "If the user enters a mouse button"  if event.button == 1:  click = True   p.display.update()  clock.tick(60)  def incomplete():  *"Subroutine that will display that the online mode is not complete"* running = True  messageLINE1 = "This part of the"  messageLINE2 = "game has not been"  messageLINE3 = "completed yet :("   printMESSAGE1 = incomplete\_font.render(messageLINE1, True, white)  printMESSAGE2 = incomplete\_font.render(messageLINE2, True, white)  printMESSAGE3 = incomplete\_font.render(messageLINE3, True, white)  textBACKGROUND = p.Rect((50, 50), (400, 400))   while running:  screen.fill((0,0,0))  screen.blit(background, (0, 0))  p.draw.rect(screen, (black), textBACKGROUND)   screen.blit(printMESSAGE1, (120, 100))  screen.blit(printMESSAGE2, (100, 200))  screen.blit(printMESSAGE3, (110, 300))   for event in p.event.get():  if event.type == QUIT:  p.quit()  s.exit  if event.type == KEYDOWN:  if event.key == K\_ESCAPE:  running = False   p.display.update()  clock.tick(60)  def didnotINTEGRATE():  *"""Subroutine that will display that the chat app is not  integrated with the mainGUI."""* running = True  messageLINE1 = "This part of the"  messageLINE2 = "game has been"  messageLINE3 = "finished but not"  messageLINE4 = "been integrated with"  messageLINE5 = "the chess.py file."   printMESSAGE1 = incomplete\_font.render(messageLINE1, True, white)  printMESSAGE2 = incomplete\_font.render(messageLINE2, True, white)  printMESSAGE3 = incomplete\_font.render(messageLINE3, True, white)  printMESSAGE4 = incomplete\_font.render(messageLINE4, True, white)  printMESSAGE5 = incomplete\_font.render(messageLINE5, True, white)  textBACKGROUND = p.Rect((50, 50), (400, 300))  while running:  screen.fill((0,0,0))  screen.blit(background, (0, 0))  p.draw.rect(screen, (black), textBACKGROUND)   screen.blit(printMESSAGE1, (120, 60))  screen.blit(printMESSAGE2, (120, 110))  screen.blit(printMESSAGE3, (110, 160))  screen.blit(printMESSAGE4, (90, 210))  screen.blit(printMESSAGE5, (110, 260))   for event in p.event.get():  if event.type == QUIT:  p.quit()  s.exit  if event.type == KEYDOWN:  if event.key == K\_ESCAPE:  running = False   p.display.update()  clock.tick(60) #####END OF ONLINE########   def shortcut():  *"""Subroutine that will display to the user the different  shortcuts the user can use."""* running = True  pressESC = "• Press esc to go back"  pressESC2 = "to the main menu."  pressZ = "• Press Z to rewind in"  pressZ2 = "games."  pressR = "• Press R to reset the"  pressR2 = "board."    printMESSAGE1 = incomplete\_font.render(pressESC, True, white)  printMESSAGE2 = incomplete\_font.render(pressESC2, True, white)  printMESSAGE3 = incomplete\_font.render(pressZ, True, white)  printMESSAGE4 = incomplete\_font.render(pressZ2, True, white)  printMESSAGE5 = incomplete\_font.render(pressR, True, white)  printMESSAGE6 = incomplete\_font.render(pressR2, True, white)    textBACKGROUND = p.Rect((50, 50), (400, 250))   while running:  screen.fill((0,0,0))  screen.blit(background, (0, 0))  p.draw.rect(screen, (black), textBACKGROUND)   screen.blit(printMESSAGE1, (75, 60))  screen.blit(printMESSAGE2, (95, 90))  screen.blit(printMESSAGE3, (75, 140))  screen.blit(printMESSAGE4, (95, 170))  screen.blit(printMESSAGE5, (75, 220))  screen.blit(printMESSAGE6, (95, 250))  for event in p.event.get():  if event.type == QUIT:  p.quit()  s.exit  if event.type == KEYDOWN:  if event.key == K\_ESCAPE:  running = False   p.display.update()  “Updates the screen.”  clock.tick(60)  “Refreshing the screen 60 times a second”  if \_\_name\_\_ == “\_\_main\_\_”:  “Calls the main subroutine.”  mainGUI()  **THIS IS THE END OF THE *chess.py* file.** |
| In summary, the *chess.py* file is the exception handling part of the game which managers user input that helps the user navigate to various parts of the game. Granted that I could’ve used classes for this .py file, but due to time constraints I could not implement efficient code. |

## Single Player MODE

### singleplayerENGINE.py

|  |
| --- |
| In this part of my code, I will talk about the main python program that will run, import and exception handle the other libraries imported. |

|  |
| --- |
| **Importing all the other libraries.** |
|  |
| * Firstly, we import **pygame as p** as it will manufacture, lay out piece images that will initially set up the chess board. * We then need to import the **piece** from the other python file piece.py which will be in a dictionary format already rendered in piece.py. * In the game file, we also have another folder named modes which holds the data to manufacture the different game modes when taped by the button in the main GUI screen. However, it hasn’t been completed. * We also import the board.py as b which imports the most vital class that holds data about each individual piece i.e. **class piece\_manager:** * Importing the negamax algorithm saved in the **ai.py** is also important as the engine.py file will execute after the user has played their move. * The **sys** module provides information about constants, functions and methods of the **Python** interpreter. * **Multiprocessing** refers to the ability of a system to support more than one processor at the same time. Applications in a **multiprocessing** system are broken to smaller routines that run independently. The operating system allocates these threads to the processors improving performance of the system. * Secondly, we starting defining key global variables that will be used throughout the program, we start with **boardDIMENSION** which simply means that it would be used when manufacture the board. * **width/height** is used to save the overall program screen with boxheight dividing the height with the board dimension to create 8 squares of equal dimension. The **//** function is used for integer division which gets integer value and not real. * FPS is the game frames/second with the dictionary **rendered** being the library where the pictures rendered by the **RenderImage** subroutine in the **piece.py.** * Finally, **p.init()** basically means that pygame is being initialised. * **moveTURN** holds dimension for producing which player’s turn it is. |

|  |
| --- |
| **def PlayGame():** In this section of the engine.py code it will be the main driver of the chess game. Its job is to handle the clicks of the user on the board along with the task of updating the graphics. |
| def playGame():  screen = p.display.set\_mode((width + moveTURNwidth, height))  clock = p.time.Clock()  screen.fill((0,0,0))  gs = b.piece\_manager()  validate = gs.moveValidation()  """  • Screen is declared as a variable which will be used to set the display specified   by the global variable dimension i.e. width and height.  • Clock is an important variable where it will be used to animate the piece when   they are moved by the user.   • screen.fill basically fills the screen with black, i.e. specified by the RGB   values (0,0,0).  • gs is basically a call saved in engine.py which calls the class from board.py   which is GameState.  • validate is a call saved in engine.py which calls the class from board i.e.   moveValidation().  """    moveMADE = False  animations = False  pie.RenderImage()  running = True  square\_selected = ()  clicked = []  game\_over = False  aiProcess = False  move\_undone = False  move\_finder\_process = None  player\_one = True  player\_two = False  """  • moveMADE is a flag variable which is set and will change as moves are made.  • animations is a flag variable which will turn True whenever the animateMoves  subroutine is called that will then animate the piece transition.  • pie.RenderImage() we initialise the subroutine def RenderImage from the   piece.py library which initialises the process of rendering the pictures  according to the dimension. It will be passed from piece.py.  • running is a flag variable that will change after the game finishes. It's the  idea of an infinite loop that will continuously run the game.  • square\_selected tracks user clicks and saves it in a tuple.  • clicked saves the clicks data of the user when clicked at a square on the   chess board.  • game\_over is a flag variables that triggers after the game is over.  • aiProcess is a flag variable that is set to pass the command to the AI.  • move\_undone is a flag variable to check whether the user has undone a   move.  • move\_finder\_process is a set to None.  • player\_one is a flag variable which is set which will be true for the first   player otherwise false for the AI. player\_two is a similar flag variable that   is set that will pass the command to the AI.  """   while running: # Infinite loop that will run def main():  "The flag variable running being called that will run the loop."  human\_turn = (gs.white\_to\_move and player\_one) or (not gs.white\_to\_move and player\_two)  "An ADD operation where the results are saved in a variable named human\_turn."   for event in p.event.get():  "For every event in the pygame library."   if event.type == p.QUIT:  "If the event type is pygame.QUIT"  p.quit()  sys.exit()  #"Pygame and system exiting."   elif event.type == p.MOUSEBUTTONDOWN:  "Part of the code when the user clicks a mouse button."   if not game\_over:  location = p.mouse.get\_pos() # Holds the location of the mouse.  column = location[0] // BOX\_SIZE  row = location[1] // BOX\_SIZE  "• location holds the position of the mouse."   if square\_selected == (row, column) or column >= 8:  square\_selected = ()  clicked = []  """  If the user clicks a location which doesn't have to do anything   with the game, clicked and square\_selected will be set to empty."""   else:  square\_selected = (row, column)  clicked.append(square\_selected)  """  If the user has clicked on the piece and then clicks on where they  want to move it. It will append the move in the clicked tuple.  """   if len(clicked) == 2 and human\_turn:  move = b.Move(clicked[0], clicked[1], gs.board)  "Occurring after the second click and if the human is playing."   for i in range(len(validate)):  """  Loop that will execute to the number of contents in validate  i.e. validation subroutine in board.py.  """    if move == validate[i]:  "If the move matches the requirements in the validate library."   gs.makeMove(validate[i])  moveMADE = True  animations = True  square\_selected = ()  clicked = []  """  Variables being set to true if the move matches the validate   requirements."""   if not moveMADE:  clicked = [square\_selected]  "Error handling in case the user has not made any move."   elif event.type == p.KEYDOWN:  "Part of the code when pygame finds after it discovers a keyboard button click."   if event.key == p.K\_z:  "If the button click is z"  gs.undoMove()  "It will call the subroutine from board.py which is undoMove()"  moveMADE = True  animations = False  game\_over = False  """  • moveMADE will be set to True as the user will get to play again.  • animations will be False as it will have to redo as the user  has undone the move.  • game\_over will be set back to false even if the the end message  is displayed."""   if aiProcess:  "If ai Process is True"  move\_finder\_process.terminate()  "Start executing move\_finder\_process.terminate() to find the move."  aiProcess = False  "aiProcess will be set back to false then."  move\_undone = True  "move\_undone will be set back to True."   if event.key == p.K\_ESCAPE:  "If the user enters the escape key"  p.display.set\_mode((width, height))  "It will produce a pane for the mainGUI.py to continue."  running = False  "Will stop the loop execution."   if event.key == p.K\_r:  "If the keyboard button r is pressed."  gs = b.piece\_manager()  "Defines the gamestate class from board.py in engine.py."  validate = gs.moveValidation()  "Defines the moveValidation class from board.py in engine.py."  square\_selected = ()  clicked = []  """  • square\_selected i.e. the tuple will be set back to empty.  • The clicked list will also be set back to an empty list."""  moveMADE = False  animations = False  game\_over = False  """  • moveMade will be False i.e. the user will get to start again.  • animations will be false as the user has to make a move and then the  code will start again with the animation.  • if the game was previously over, it would be set back to false."""   if aiProcess:  """If the aiProcess variable is True, then this part of the code   will start executing."""   move\_finder\_process.terminate()  aiProcess = False  """  • move\_finder process.terminate will begin.  • after the AI has played, it will set it back to False for the user's move."""  move\_undone = True  "If aiProcess is not True, move\_undone will be False"    if not game\_over and not human\_turn and not move\_undone:  "Boolean selection statement used if requirements met."   if not aiProcess:  "if aiProcess if False"  aiProcess = True  "aiProcess will be set to True"  return\_queue = Queue()  move\_finder\_process = Process(target=ai.findBestMove, args=(gs, validate, return\_queue))  move\_finder\_process.start()  """This will initiate the move finding process which imports from the ai.py   and will validate it with the validate subroutine from the board.py file."""   if not move\_finder\_process.is\_alive():  "If not move\_finder\_process is False, this will activate this part."   ai\_move = return\_queue.get()   if ai\_move is None:  ai\_move = ai.findRandomMove(validate)  "If the AI doesn't find any relevant moves, it will start generating random moves."  gs.makeMove(ai\_move)  "Start executing makeMove from the ai.py file."  moveMADE = True  animations = True  aiProcess = False  "Variables start changing for the ai to perform."   if moveMADE:  "If moveMADE is True"   if animations:  "If animations is True, it will start producing the animations"  animateMove(gs.moveRegister[-1], screen, gs.board, clock)  "Piece animation begins"  validate = gs.moveValidation()  "Move validation"  moveMADE = False  animations = False  move\_undone = False  "Key variables change to facilitate change."   drawGameState(screen, gs, validate, square\_selected)  "Starts drawing the game board with the pieces and whose turn it is."   if gs.checkmate:  "if the program finds the game is in checkmate."  game\_over = True  "game\_over will be set to True"   if gs.white\_to\_move:  "If gs.white\_to\_move is True, this part will execute."  drawEndGameText(screen, "BLACK WINS THE GAME BY CHECKMATE")  """will draw the text on the screen that the game is over   by checkmate by black."""  else:  drawEndGameText(screen, "WHITE WINS THE GAME BY CHECKMATE")  """will draw the text on the screen that the game is over   by checkmate by white."""    elif gs.stalemate:  "if the program finds the game is in a stalemate"  game\_over = True  "game\_over will be set to True"  drawEndGameText(screen, "STALEMATE")  """will draw the text on the screen that the game is over by  stalemate any of the pieces."""    clock.tick(FPS)  p.display.flip()  """  • The clock will start ticking which will start FPS count, i.e. run the loop  on 60 frames.  • display.flip will update the contents of the entire display."""  **THIS IS THE END OF THE def PlayGame() subroutine.** |
| As can be seen in the code above. It is the main part of the game folder which will manage all the operations of the game which is chess piece movements, animations, highlighting, etc and a lot more. It involves various selection statements which will handle the game if the user doesn’t interact with the game as it was supposed to. |

|  |
| --- |
|  |

|  |
| --- |
| **def drawGameState():** This subroutine will draw the board. It has the control of 3 other subroutines that will be called in this subroutine. The subroutine will be called in def main(): in order to construct the chess board. |
| def drawGameState(screen, gs, valid\_moves, square\_selected):   *"Imports arguments that holds information to construct the board."* drawBoard(screen)  "#draw squares on the board"  highlightSquares(screen, gs, valid\_moves, square\_selected)  "highlight squares on the board"  drawPieces(screen, gs.board)  "# draw pieces on top of those squares"  drawTurn(screen, gs)  "Subroutine that draw whose turn it is."  **THIS IS THE END OF def drawGameState():** |
| In summary, this subroutine is responsible for manufacturing the chess board along with animation of whose turn it is. |

|  |
| --- |
| **def drawBoard():** Subroutine that will construct the squares on the board. |
| drawBoard(screen):  *"Imports arguments that holds data to construct the squares."* global colors  "Imports colors which don't require pygame to type RGB values."  colors = [p.Color("grey"), p.Color("darkcyan")]   "List that saves the colours for the variable BOX\_SQUARE"  for row in range(boardDIMENSION):   "for loop that will draw the 2D array i.e. the chess board."  for column in range(boardDIMENSION):   "# for loop that will partition the board which will result in 64 squares."  color = colors[((row + column) % 2)]   "# Board will be equal to the chess board pattern. Chequered squares."  p.draw.rect(screen, color, p.Rect(column \* BOX\_SIZE, row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE))  "# Draw the boxes."  **THIS IS THE END OF def drawBoard():** |
| In summary, this subroutine is responsible for manufacturing the checquered pattern on the board. |

|  |
| --- |
| **def highlightSquares():**Subroutine that will contains commands imported from pygame, and from the variables stored in main(): to highlight the chess pieces. |
| def highlightSquares(screen, gs, valid\_moves, square\_selected):  *"Part of the game that will handle the highlighting of the chess pieces."* if (len(gs.moveRegister)) > 0:  "If the length of the moveRegister in the board.py file is greater than 0."  last\_move = gs.moveRegister[-1]  s = p.Surface((BOX\_SIZE, BOX\_SIZE))  s.set\_alpha(100)  s.fill(p.Color('green'))  """after the piece has reached its location, it will highlight  the square green."""  screen.blit(s, (last\_move.end\_col \* BOX\_SIZE, last\_move.end\_row \* BOX\_SIZE))  "producing the highlight."   if square\_selected != ():  "if the square\_selected tuple is not empty."  row, col = square\_selected  "row, column will be equal to square\_selected as it will highlight."   if gs.board[row][col][0] == (   'w' if gs.white\_to\_move else 'b'):   "square\_selected is a piece that can be moved"  "highlight selected square"  s = p.Surface((BOX\_SIZE, BOX\_SIZE))  s.set\_alpha(100)  "transparency value 0 -> transparent, 255 -> opaque"  s.fill(p.Color('blue'))  screen.blit(s, (col \* BOX\_SIZE, row \* BOX\_SIZE))  "highlight moves from that square"  s.fill(p.Color('yellow'))   for move in valid\_moves:  "for the number of moves in valid\_moves"  if move.start\_row == row and move.start\_col == col:  "if the move.start\_row is equal to the column"  screen.blit(s, (move.end\_col \* BOX\_SIZE, move.end\_row \* BOX\_SIZE))  """on the specific row and column the screen will be blit,  the highlight will be drawn."""  **THIS IS THE END OF def highlightSquares():** |
| In summary, this subroutine is responsible for highlighting the checquered boxes on the board when the player does a move. |

|  |
| --- |
| **def drawPieces():** Subroutine that will call the rendered library which contains the images and prints them on the board. |
| drawPieces(screen, board):  *"Subroutine that will print the chess pieces on the board."* for row in range(boardDIMENSION):  "for i in boardDimension i.e. 8 because 64 boxes on board"  for column in range(boardDIMENSION):  "for j in boardDimension i.e. 8 because 64 boxes on board"  piece = board[row][column]  "saving the board rows and columns in a variable named piece"  if piece != "--":  "if piece is empty, then it will leave that space empty"  screen.blit(rendered[piece], p.Rect(column \* BOX\_SIZE, row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE))  "finally render the images on the board."  **THIS IS THE END OF def drawPieces():** |
| In summary, this subroutine is responsible for manufacturing the checquered pattern on the board. |

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **def drawTurn():** Subroutine that will manufacture which player’s turn it is. | | def drawTurn(screen, gs):  *"Subroutine that will manufacture whose turn it on the singleENGINE pane"* white = p.image.load('resources/' + 'white.jpg')  black = p.image.load('resources/' + 'black.jpg')  "Loads the pictures"    playerFont = p.font.SysFont("Aleo", 35)  box1 = p.Rect((525, 100), (150, 100))  box2 = p.Rect((525, 300), (150, 100))  player1 = playerFont.render("YOUR TURN", True, p.Color("black"))  player2 = playerFont.render("COMPUTER", True, p.Color("black"))  highlightP1 = p.Rect((515, 90), (170, 120))  highlightP2 = p.Rect((515, 290), (170, 120))  """Varibles that holds vital information to produce the text and rectangles  on the singleENGINE pane."""    if gs.white\_to\_move:  """his needs to produced first since other rectangles  will be placed over this."""    "Exception handling to handle which player is playing"  "In this case if it is white's turn."  p.draw.rect(screen, (p.Color("gold")), highlightP1)  "Highlight that will show which player't turn it is."  else:  p.draw.rect(screen, (p.Color("gold")), highlightP2)  "Highlight that will show which player't turn it is."   if not gs.white\_to\_move:  p.draw.rect(screen, (p.Color("black")), highlightP1)  "Highlight that will show which player't turn it is."  else:  p.draw.rect(screen, (p.Color("black")), highlightP2)  "Highlight that will show which player't turn it is."   p.draw.rect(screen, (p.Color("darkcyan")), box1)  p.draw.rect(screen, (p.Color("grey")), box2)  screen.blit(player1, ((528, 105)))  screen.blit(player2, ((530, 305)))  screen.blit(white, ((590, 130)))  screen.blit(black, ((590, 330)))  "Drawing the squares"  **THIS IS THE END OF THE drawTurn() subroutine.** | | In summary, this subroutine produces the gold highlight behind the computer and your turn playing. | |

|  |
| --- |
| **def drawEndGameText():** Subroutine that will handle the displaying of the end of the game such as a stalemate and checkmate. |
| def drawEndGameText(screen, text):  *"Part of the code that will print the last part of the game i.e. checkmate/stalemate"* font = p.font.SysFont("Helvetica", 25, True, False)  "font objects such as the colour and font style"  text\_object = font.render(text, False, p.Color("gray"))  text\_location = p.Rect(0, 0, width, height).move(width / 2 - text\_object.get\_width() / 2,  height / 2 - text\_object.get\_height() / 2)  "location of the font saved in text\_location"  screen.blit(text\_object, text\_location)  "produces the text\_object on the screen"  text\_object = font.render(text, False, p.Color('black'))  "location of the text\_object saved in text\_object"  screen.blit(text\_object, text\_location.move(2, 2))  "produces the text location on the board"  **THIS IS THE END OF def drawEndGameText():** |
| In summary, this subroutine is responsible for manufacturing game text on the screen. |

|  |
| --- |
| **def animateMove():** Subroutine that will contain important commands to animate the chess piece moves. |
| def animateMove(move, screen, board, clock):  *"Part of the code that will animate the chess moves."* global colors  """Imports colors which is a library which contains colours, instead of  entering RGB values, only entering string values is enough."""  rows = move.end\_row - move.start\_row  "subtracts the two variables to check the distance of animation"  columns = move.end\_col - move.start\_col  "subtracts the two variables to check the distance of animation"  FPSpieceMove = 12 # frames to move one square  "frames to move one square"  FPScount = (abs(rows) + abs(columns)) \* FPSpieceMove  ""  for frame in range(FPScount + 1):  row, col = (move.start\_row + rows \* frame / FPScount, move.start\_col + columns \* frame / FPScount)  drawBoard(screen)  drawPieces(screen, board)  # erase the piece moved from its ending square  color = colors[(move.end\_row + move.end\_col) % 2]  end\_square = p.Rect(move.end\_col \* BOX\_SIZE, move.end\_row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE)  p.draw.rect(screen, color, end\_square)  # draw captured piece onto rectangle  if move.piece\_captured != '--':  if move.is\_enpassant\_move:  enpassant\_row = move.end\_row + 1 if move.piece\_captured[0] == 'b' else move.end\_row - 1  end\_square = p.Rect(move.end\_col \* BOX\_SIZE, enpassant\_row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE)  screen.blit(rendered[move.piece\_captured], end\_square)  # draw moving piece  screen.blit(rendered[move.piece\_moved], p.Rect(col \* BOX\_SIZE, row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE))  p.display.flip()  clock.tick(60  **THIS IS THE END OF def animateMove():** |
| In summary, this subroutine is responsible for chess piece transition. |
| **And finally we have the subroutine call which calls the singleplayerENGINE game mode.** |

### board.py

|  |
| --- |
| **class GameState:** Contains all the methods that will validate the chess moves.  **def makeMove():** Subroutine that allows the players to make a move.  **def undoMove():** Subroutine that will allow the user to undo a move.  **def updateCastleRights():** Subroutine that has castle rights that is vital for validation.  **def getValidMoves():** Subroutines that handles move validation.  **def inCheck():** Subroutine that will check whether if either of the players are in check.  **def squareUnderAttack():** Determine whether if enemy can attack the players piece positions.  **def getAllPossibleMoves():** Subroutine that will trigger piece subroutines whenever a piece being clicked in the self.board state is being triggered.  **def checkForPinsAndChecks():** Keeps track of pins and checks.  **def getPawnMoves():** Subroutine that manages pawn movement.  **def getRookMoves():** Subroutine that manages rook movement.  **def getKnightMoves():** Subroutine that manages knight movement.  **def getBishopMove():** Subroutine that manages bishop movement.  **def getQueenMoves():** Subroutine that manages queen movement.  **def getKingMoves():** Subroutine that manages king movement.  **def getCastleMoves():** Subroutine that will take care of all the castling rights.  **def getKingsideCastleMoves():** Subroutine that will take care of castling rights with the king.  **def getQueensideCastleMoves():** Subroutine that will take care of castling rights with the queen.  **class CastleRights:** Class where castle rights are defined.  **class Move:** Class that will control piece movements.  **def \_\_eq\_\_():** Python method that overrides the equals method.  **def getChessNotation():** Subroutine that prints the chess notation on the moveRegister window.  **def getRankFile():** Will print the piece rank file.  **def \_\_str\_\_():** The \_\_str\_\_ method is one that should be the most human-readable possible, yet also descriptive of that exact object. |
| class GameState:  *"Main class that will handle all piece functions."* def \_\_init\_\_(self):  *""" "--" will represent the empty list, whereas every other word is a piece name directly"""* self.board = [  ["bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"],  ["bp", "bp", "bp", "bp", "bp", "bp", "bp", "bp"],  ["--", "--", "--", "--", "--", "--", "--", "--"],  ["--", "--", "--", "--", "--", "--", "--", "--"], # 2x2 array as there are 64 squares in a chess board.  ["--", "--", "--", "--", "--", "--", "--", "--"],  ["--", "--", "--", "--", "--", "--", "--", "--"],  ["wp", "wp", "wp", "wp", "wp", "wp", "wp", "wp"],  ["wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"]]  "directly imported from the pieces library in the engine.py file."  self.white\_to\_move = True  "After black, the right to move will be transferred to white."  self.moveRegister = []  'List that will keep track of all the moves.'  self.white\_king\_location = (7, 4)  self.black\_king\_location = (0, 4)  self.checkmate = False  'Flag variable that is set to see when the players are in checkmate.'  self.stalemate = False  'Flag variable that is set to see when the players are in stalemate.'  self.in\_check = False  'Flat variable that is set to see when the players are in check.'  self.pins = []  self.checks = []  self.enpassant\_possible = ()  'coordinates for the square where en-passant capture is possible'  self.enpassant\_possible\_log = [self.enpassant\_possible]  self.current\_castling\_rights = CastleRights(True, True, True, True)  self.castle\_rights\_log = [CastleRights(self.current\_castling\_rights.wks, self.current\_castling\_rights.bks,  self.current\_castling\_rights.wqs, self.current\_castling\_rights.bqs)]   def makeMove(self, move):  *"Subroutine that takes the parameter move that allows us to make the move."* self.board[move.start\_row][move.start\_col] = "--"  self.board[move.end\_row][move.end\_col] = move.piece\_moved  self.moveRegister.append(move)  "log the move so we can undo it later"  self.white\_to\_move = not self.white\_to\_move  "switch players"  "update king's location if moved"  if move.piece\_moved == "wK":  self.white\_king\_location = (move.end\_row, move.end\_col)  elif move.piece\_moved == "bK":  self.black\_king\_location = (move.end\_row, move.end\_col)   "Code that will trigger once the pieces make it to the start of the other players squares."  if move.is\_pawn\_promotion:  self.board[move.end\_row][move.end\_col] = move.piece\_moved[0] + "Q"   "Code that will check for en-passant moves."  if move.is\_enpassant\_move:  self.board[move.start\_row][move.end\_col] = "--" # capturing the pawn   "update enpassant\_possible variable"  if move.piece\_moved[1] == "p" and abs(move.start\_row - move.end\_row) == 2:  "only on 2 square pawn advance"  self.enpassant\_possible = ((move.start\_row + move.end\_row) // 2, move.start\_col)  else:  self.enpassant\_possible = ()   "castle move"  if move.is\_castle\_move:  if move.end\_col - move.start\_col == 2:  "king-side castle move"  self.board[move.end\_row][move.end\_col - 1] = self.board[move.end\_row][  move.end\_col + 1]  "moves the rook to its new square"  self.board[move.end\_row][move.end\_col + 1] = '--' "erase old rook"  else:  "queen-side castle move"  self.board[move.end\_row][move.end\_col + 1] = self.board[move.end\_row][  move.end\_col - 2] # moves the rook to its new square  self.board[move.end\_row][move.end\_col - 2] = '--' # erase old rook   self.enpassant\_possible\_log.append(self.enpassant\_possible)   "update castling rights - whenever it is a rook or king move"  self.updateCastleRights(move)  self.castle\_rights\_log.append(CastleRights(self.current\_castling\_rights.wks, self.current\_castling\_rights.bks,  self.current\_castling\_rights.wqs, self.current\_castling\_rights.bqs))  "Subroutine that is the opposite of makeMove."  def undoMove(self):  if len(self.moveRegister) != 0:  "make sure that there is a move to undo"  move = self.moveRegister.pop()  self.board[move.start\_row][move.start\_col] = move.piece\_moved  self.board[move.end\_row][move.end\_col] = move.piece\_captured  self.white\_to\_move = not self.white\_to\_move  "swap players"  "update the king's position if needed"  if move.piece\_moved == "wK":  self.white\_king\_location = (move.start\_row, move.start\_col)  elif move.piece\_moved == "bK":  self.black\_king\_location = (move.start\_row, move.start\_col)  # undo en passant move  if move.is\_enpassant\_move:  self.board[move.end\_row][move.end\_col] = "--" "leave landing square blank"  self.board[move.start\_row][move.end\_col] = move.piece\_captured   self.enpassant\_possible\_log.pop()  self.enpassant\_possible = self.enpassant\_possible\_log[-1]   # undo castle rights  self.castle\_rights\_log.pop()  "get rid of the new castle rights from the move we are undoing"  self.current\_castling\_rights = self.castle\_rights\_log[  -1]  "set the current castle rights to the last one in the list"  "undo the castle move"  if move.is\_castle\_move:  if move.end\_col - move.start\_col == 2:  "king-side"  self.board[move.end\_row][move.end\_col + 1] = self.board[move.end\_row][move.end\_col - 1]  self.board[move.end\_row][move.end\_col - 1] = '--'  else:  "queen-side"  self.board[move.end\_row][move.end\_col - 2] = self.board[move.end\_row][move.end\_col + 1]  self.board[move.end\_row][move.end\_col + 1] = '--'  self.checkmate = False  self.stalemate = False  def updateCastleRights(self, move):  *"Updates the castle rights that is given by the moves."* if move.piece\_captured == "wR":  if move.end\_col == 0: # left rook  self.current\_castling\_rights.wqs = False  elif move.end\_col == 7: # right rook  self.current\_castling\_rights.wks = False  elif move.piece\_captured == "bR":  if move.end\_col == 0: # left rook  self.current\_castling\_rights.bqs = False  elif move.end\_col == 7: # right rook  self.current\_castling\_rights.bks = False  if move.piece\_moved == 'wK':  self.current\_castling\_rights.wqs = False  self.current\_castling\_rights.wks = False  elif move.piece\_moved == 'bK':  self.current\_castling\_rights.bqs = False  self.current\_castling\_rights.bks = False  elif move.piece\_moved == 'wR':  if move.start\_row == 7:  if move.start\_col == 0: # left rook  self.current\_castling\_rights.wqs = False  elif move.start\_col == 7: # right rook  self.current\_castling\_rights.wks = False  elif move.piece\_moved == 'bR':  if move.start\_row == 0:  if move.start\_col == 0: # left rook  self.current\_castling\_rights.bqs = False  elif move.start\_col == 7: # right rook  self.current\_castling\_rights.bks = False   # Subroutine that handles move validation.  def moveValidation(self):  temp\_castle\_rights = CastleRights(self.current\_castling\_rights.wks, self.current\_castling\_rights.bks,  self.current\_castling\_rights.wqs, self.current\_castling\_rights.bqs)  # advanced algorithm  moves = []  self.in\_check, self.pins, self.checks = self.checkForPinsAndChecks()   if self.white\_to\_move:  king\_row = self.white\_king\_location[0]  king\_col = self.white\_king\_location[1]  else:  king\_row = self.black\_king\_location[0]  king\_col = self.black\_king\_location[1]  if self.in\_check:  if len(self.checks) == 1:  "only 1 check, block the check or move the king"  moves = self.getAllPossibleMoves()  "to block the check you must put a piece into one of the squares between the enemy piece and your king"  check = self.checks[0]  "check information"  check\_row = check[0]  check\_col = check[1]  piece\_checking = self.board[check\_row][check\_col]  valid\_squares = []  "squares that pieces can move to saved in a list, will be passed to other subroutines"  "if knight, must capture the knight or move your king, other pieces can be blocked"  if piece\_checking[1] == "N":  valid\_squares = [(check\_row, check\_col)]  else:  for i in range(1, 8):  valid\_square = (king\_row + check[2] \* i,  king\_col + check[3] \* i)  "check[2] and check[3] are the check directions"  valid\_squares.append(valid\_square)  if valid\_square[0] == check\_row and valid\_square[  1] == check\_col:  "once you get to piece and check"  break  "get rid of any moves that don't block check or move king"  for i in range(len(moves) - 1, -1, -1):  "iterate through the list backwards when removing elements"  if moves[i].piece\_moved[1] != "K":  "move doesn't move king so it must block or capture"  if not (moves[i].end\_row,  moves[i].end\_col) in valid\_squares:  "move doesn't block or capture piece"  moves.remove(moves[i])  else:  "double check, king has to move"  self.getKingMoves(king\_row, king\_col, moves)  else:  "not in check - all moves are fine"  moves = self.getAllPossibleMoves()  if self.white\_to\_move:  self.getCastleMoves(self.white\_king\_location[0], self.white\_king\_location[1], moves)  else:  self.getCastleMoves(self.black\_king\_location[0], self.black\_king\_location[1], moves)   if len(moves) == 0:  if self.inCheck():  self.checkmate = True  else:  self.stalemate = True  else:  self.checkmate = False  self.stalemate = False   self.current\_castling\_rights = temp\_castle\_rights  return moves  def inCheck(self):  *"Subroutine that will check whether if a player is in check."* if self.white\_to\_move:  return self.squareUnderAttack(self.white\_king\_location[0], self.white\_king\_location[1])  else:  return self.squareUnderAttack(self.black\_king\_location[0], self.black\_king\_location[1])    def squareUnderAttack(self, row, col):  *"Determine if enemy can attack the square row col"* self.white\_to\_move = not self.white\_to\_move  "switch to opponent's point of view"  opponents\_moves = self.getAllPossibleMoves()  self.white\_to\_move = not self.white\_to\_move  for move in opponents\_moves:  if move.end\_row == row and move.end\_col == col: # square is under attack  return True  return False    def getAllPossibleMoves(self):  *"Subroutine that checks what moves have taken place and will thereby trigger the appropriate piece subroutine."* moves = []  for row in range(len(self.board)):  for col in range(len(self.board[row])):  turn = self.board[row][col][0]  if (turn == "w" and self.white\_to\_move) or (turn == "b" and not self.white\_to\_move):  piece = self.board[row][col][1]  if piece == "p":  self.getPawnMoves(row, col, moves)  elif piece == "R":  self.getRookMoves(row, col, moves)  elif piece == "B":  self.getBishopMoves(row, col, moves)  elif piece == "N":  self.getKnightMoves(row, col, moves)  elif piece == "Q":  self.getQueenMoves(row, col, moves)  elif piece == "K":  self.getKingMoves(row, col, moves)  return moves  def checkForPinsAndChecks(self):  pins = []  "squares pinned and the direction its pinned from"  checks = []  "squares where enemy is applying a check"  in\_check = False  if self.white\_to\_move:  enemy\_color = "b"  ally\_color = "w"  start\_row = self.white\_king\_location[0]  start\_col = self.white\_king\_location[1]  else:  enemy\_color = "w"  ally\_color = "b"  start\_row = self.black\_king\_location[0]  start\_col = self.black\_king\_location[1]  "check outwards from king for pins and checks, keep track of pins"  directions = ((-1, 0), (0, -1), (1, 0), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1))  for j in range(len(directions)):  direction = directions[j]  possible\_pin = () # reset possible pins  for i in range(1, 8):  end\_row = start\_row + direction[0] \* i  end\_col = start\_col + direction[1] \* i  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7:  end\_piece = self.board[end\_row][end\_col]  if end\_piece[0] == ally\_color and end\_piece[1] != "K":  if possible\_pin == (): # first allied piece could be pinned  possible\_pin = (end\_row, end\_col, direction[0], direction[1])  else: # 2nd allied piece - no check or pin from this direction  break  elif end\_piece[0] == enemy\_color:  enemy\_type = end\_piece[1]  """1. orthogonally away from king and piece is a rook  2. diagonally away from king and piece is a bishop  3. 1 square away diagonally from king and piece is a pawn  4. any direction and piece is a queen  5. any direction 1 square away and piece is a king"""  if (0 <= j <= 3 and enemy\_type == "R") or (4 <= j <= 7 and enemy\_type == "B") or (  i == 1 and enemy\_type == "p" and (  (enemy\_color == "w" and 6 <= j <= 7) or (enemy\_color == "b" and 4 <= j <= 5))) or (  enemy\_type == "Q") or (i == 1 and enemy\_type == "K"):  if possible\_pin == (): # no piece blocking, so check  in\_check = True  checks.append((end\_row, end\_col, direction[0], direction[1]))  break  else: # piece blocking so pin  pins.append(possible\_pin)  break  else: # enemy piece not applying checks  break  else:  break # off board  # check for knight checks  knight\_moves = ((-2, -1), (-2, 1), (-1, 2), (1, 2), (2, -1), (2, 1), (-1, -2), (1, -2))  for move in knight\_moves:  end\_row = start\_row + move[0]  end\_col = start\_col + move[1]  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7:  end\_piece = self.board[end\_row][end\_col]  if end\_piece[0] == enemy\_color and end\_piece[1] == "N": # enemy knight attacking a king  in\_check = True  checks.append((end\_row, end\_col, move[0], move[1]))  return in\_check, pins, checks    def getPawnMoves(self, row, col, moves):  *"Subroutine that will be trigger whenever the pawn is moved, it will be saved in a list moves then."* piece\_pinned = False  pin\_direction = ()  for i in range(len(self.pins) - 1, -1, -1):  if self.pins[i][0] == row and self.pins[i][1] == col:  piece\_pinned = True  pin\_direction = (self.pins[i][2], self.pins[i][3])  self.pins.remove(self.pins[i])  break   if self.white\_to\_move:  move\_amount = -1  start\_row = 6  enemy\_color = "b"  king\_row, king\_col = self.white\_king\_location  else:  move\_amount = 1  start\_row = 1  enemy\_color = "w"  king\_row, king\_col = self.black\_king\_location   if self.board[row + move\_amount][col] == "--": # 1 square pawn advance  if not piece\_pinned or pin\_direction == (move\_amount, 0):  moves.append(Move((row, col), (row + move\_amount, col), self.board))  if row == start\_row and self.board[row + 2 \* move\_amount][col] == "--": # 2 square pawn advance  moves.append(Move((row, col), (row + 2 \* move\_amount, col), self.board))  if col - 1 >= 0: # capture to the left  if not piece\_pinned or pin\_direction == (move\_amount, -1):  if self.board[row + move\_amount][col - 1][0] == enemy\_color:  moves.append(Move((row, col), (row + move\_amount, col - 1), self.board))  if (row + move\_amount, col - 1) == self.enpassant\_possible:  attacking\_piece = blocking\_piece = False  if king\_row == row:  if king\_col < col: # king is left of the pawn  # inside: between king and the pawn;  # outside: between pawn and border;  inside\_range = range(king\_col + 1, col - 1)  outside\_range = range(col + 1, 8)  else: # king right of the pawn  inside\_range = range(king\_col - 1, col, -1)  outside\_range = range(col - 2, -1, -1)  for i in inside\_range:  if self.board[row][i] != "--": # some piece beside en-passant pawn blocks  blocking\_piece = True  for i in outside\_range:  square = self.board[row][i]  if square[0] == enemy\_color and (square[1] == "R" or square[1] == "Q"):  attacking\_piece = True  elif square != "--":  blocking\_piece = True  if not attacking\_piece or blocking\_piece:  moves.append(Move((row, col), (row + move\_amount, col - 1), self.board, is\_enpassant\_move=True))  if col + 1 <= 7: # capture to the right  if not piece\_pinned or pin\_direction == (move\_amount, +1):  if self.board[row + move\_amount][col + 1][0] == enemy\_color:  moves.append(Move((row, col), (row + move\_amount, col + 1), self.board))  if (row + move\_amount, col + 1) == self.enpassant\_possible:  attacking\_piece = blocking\_piece = False  if king\_row == row:  if king\_col < col: # king is left of the pawn  # inside: between king and the pawn;  # outside: between pawn and border;  inside\_range = range(king\_col + 1, col)  outside\_range = range(col + 2, 8)  else: # king right of the pawn  inside\_range = range(king\_col - 1, col + 1, -1)  outside\_range = range(col - 1, -1, -1)  for i in inside\_range:  if self.board[row][i] != "--": # some piece beside en-passant pawn blocks  blocking\_piece = True  for i in outside\_range:  square = self.board[row][i]  if square[0] == enemy\_color and (square[1] == "R" or square[1] == "Q"):  attacking\_piece = True  elif square != "--":  blocking\_piece = True  if not attacking\_piece or blocking\_piece:  moves.append(Move((row, col), (row + move\_amount, col + 1), self.board, is\_enpassant\_move=True))  def getRookMoves(self, row, col, moves):  *"Subroutine that will handle all the rook moves. It will be triggered whenever rook is moved."* piece\_pinned = False  pin\_direction = ()  for i in range(len(self.pins) - 1, -1, -1):  if self.pins[i][0] == row and self.pins[i][1] == col:  piece\_pinned = True  pin\_direction = (self.pins[i][2], self.pins[i][3])  if self.board[row][col][  1] != "Q": # can't remove queen from pin on rook moves, only remove it on bishop moves  self.pins.remove(self.pins[i])  break  directions = ((-1, 0), (0, -1), (1, 0), (0, 1)) # up, left, down, right  enemy\_color = "b" if self.white\_to\_move else "w"  for direction in directions:  for i in range(1, 8):  end\_row = row + direction[0] \* i  end\_col = col + direction[1] \* i  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7: # check for possible moves only in boundaries of the board  if not piece\_pinned or pin\_direction == direction or pin\_direction == (  -direction[0], -direction[1]):  end\_piece = self.board[end\_row][end\_col]  if end\_piece == "--": # empty space is valid  moves.append(Move((row, col), (end\_row, end\_col), self.board))  elif end\_piece[0] == enemy\_color: # capture enemy piece  moves.append(Move((row, col), (end\_row, end\_col), self.board))  break  else: # friendly piece  break  else: # off board  break    def getKnightMoves(self, row, col, moves):  *"Subroutine for knight pattern moving. It will trigger once the knight has moved."* piece\_pinned = False  for i in range(len(self.pins) - 1, -1, -1):  if self.pins[i][0] == row and self.pins[i][1] == col:  piece\_pinned = True  self.pins.remove(self.pins[i])  break   knight\_moves = ((-2, -1), (-2, 1), (-1, 2), (1, 2), (2, -1), (2, 1), (-1, -2),  (1, -2)) # up/left up/right right/up right/down down/left down/right left/up left/down  ally\_color = "w" if self.white\_to\_move else "b"  for move in knight\_moves:  end\_row = row + move[0]  end\_col = col + move[1]  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7:  if not piece\_pinned:  end\_piece = self.board[end\_row][end\_col]  if end\_piece[0] != ally\_color: # so its either enemy piece or empty square  moves.append(Move((row, col), (end\_row, end\_col), self.board))  def getBishopMoves(self, row, col, moves):  *"Subroutine that will trigger whenever the bishop moves."* piece\_pinned = False  pin\_direction = ()  for i in range(len(self.pins) - 1, -1, -1):  if self.pins[i][0] == row and self.pins[i][1] == col:  piece\_pinned = True  pin\_direction = (self.pins[i][2], self.pins[i][3])  self.pins.remove(self.pins[i])  break   directions = ((-1, -1), (-1, 1), (1, 1), (1, -1)) # diagonals: up/left up/right down/right down/left  enemy\_color = "b" if self.white\_to\_move else "w"  for direction in directions:  for i in range(1, 8):  end\_row = row + direction[0] \* i  end\_col = col + direction[1] \* i  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7: # check if the move is on board  if not piece\_pinned or pin\_direction == direction or pin\_direction == (  -direction[0], -direction[1]):  end\_piece = self.board[end\_row][end\_col]  if end\_piece == "--": # empty space is valid  moves.append(Move((row, col), (end\_row, end\_col), self.board))  elif end\_piece[0] == enemy\_color: # capture enemy piece  moves.append(Move((row, col), (end\_row, end\_col), self.board))  break  else: # friendly piece  break  else: # off board  break    def getQueenMoves(self, row, col, moves):  *"A queen is basically a rook as well as a bishop. It will call the bishop and rook subroutine for minimal coding."* self.getBishopMoves(row, col, moves)  self.getRookMoves(row, col, moves)    def getKingMoves(self, row, col, moves):  *"King can basically move 1 direction at a time. It will be activated whenever someone moves the king."* row\_moves = (-1, -1, -1, 0, 0, 1, 1, 1)  col\_moves = (-1, 0, 1, -1, 1, -1, 0, 1)  ally\_color = "w" if self.white\_to\_move else "b"  for i in range(8):  end\_row = row + row\_moves[i]  end\_col = col + col\_moves[i]  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7:  end\_piece = self.board[end\_row][end\_col]  if end\_piece[0] != ally\_color: # not an ally piece - empty or enemy  # place king on end square and check for checks  if ally\_color == "w":  self.white\_king\_location = (end\_row, end\_col)  else:  self.black\_king\_location = (end\_row, end\_col)  in\_check, pins, checks = self.checkForPinsAndChecks()  if not in\_check:  moves.append(Move((row, col), (end\_row, end\_col), self.board))  # place king back on original location  if ally\_color == "w":  self.white\_king\_location = (row, col)  else:  self.black\_king\_location = (row, col)    def getCastleMoves(self, row, col, moves):  *"Subroutine that will take care of all the castling rights."* if self.squareUnderAttack(row, col):  return # can't castle while in check  if (self.white\_to\_move and self.current\_castling\_rights.wks) or (  not self.white\_to\_move and self.current\_castling\_rights.bks):  self.getKingsideCastleMoves(row, col, moves)  if (self.white\_to\_move and self.current\_castling\_rights.wqs) or (  not self.white\_to\_move and self.current\_castling\_rights.bqs):  self.getQueensideCastleMoves(row, col, moves)    def getKingsideCastleMoves(self, row, col, moves):  *"Subroutine that will take care of all the castling moves with the king."* if self.board[row][col + 1] == '--' and self.board[row][col + 2] == '--':  if not self.squareUnderAttack(row, col + 1) and not self.squareUnderAttack(row, col + 2):  moves.append(Move((row, col), (row, col + 2), self.board, is\_castle\_move=True))    def getQueensideCastleMoves(self, row, col, moves):  *"Subroutine that will take care of all the castling moves with the queen."* if self.board[row][col - 1] == '--' and self.board[row][col - 2] == '--' and self.board[row][col - 3] == '--':  if not self.squareUnderAttack(row, col - 1) and not self.squareUnderAttack(row, col - 2):  moves.append(Move((row, col), (row, col - 2), self.board, is\_castle\_move=True))   class CastleRights:  *"Class where certain castle rights are defined."* def \_\_init\_\_(self, wks, bks, wqs, bqs):  self.wks = wks  self.bks = bks  self.wqs = wqs  self.bqs = bqs  class Move:  *"Class that will control the movement of the pieces."*  ranks\_to\_rows = {"1": 7, "2": 6, "3": 5, "4": 4,  "5": 3, "6": 2, "7": 1, "8": 0}  rows\_to\_ranks = {v: k for k, v in ranks\_to\_rows.items()}  files\_to\_cols = {"a": 0, "b": 1, "c": 2, "d": 3,  "e": 4, "f": 5, "g": 6, "h": 7}  cols\_to\_files = {v: k for k, v in files\_to\_cols.items()}   def \_\_init\_\_(self, start\_square, end\_square, board, is\_enpassant\_move=False, is\_castle\_move=False):  self.start\_row = start\_square[0]  self.start\_col = start\_square[1]  self.end\_row = end\_square[0]  self.end\_col = end\_square[1]  self.piece\_moved = board[self.start\_row][self.start\_col]  self.piece\_captured = board[self.end\_row][self.end\_col]  # pawn promotion  self.is\_pawn\_promotion = (self.piece\_moved == "wp" and self.end\_row == 0) or (  self.piece\_moved == "bp" and self.end\_row == 7)  # en passant  self.is\_enpassant\_move = is\_enpassant\_move  if self.is\_enpassant\_move:  self.piece\_captured = "wp" if self.piece\_moved == "bp" else "bp"  # castle move  self.is\_castle\_move = is\_castle\_move   self.is\_capture = self.piece\_captured != "--"  self.moveID = self.start\_row \* 1000 + self.start\_col \* 100 + self.end\_row \* 10 + self.end\_col   # Function that overrides the equals method.  def \_\_eq\_\_(self, other):  if isinstance(other, Move):  return self.moveID == other.moveID  return False  def getChessNotation(self):  *"Subroutine that prints the chess notation on the moveRegister\_movelog."* if self.is\_pawn\_promotion:  return self.getRankFile(self.end\_row, self.end\_col) + "Q"  if self.is\_castle\_move:  if self.end\_col == 1:  return "0-0-0"  else:  return "0-0"  if self.is\_enpassant\_move:  return self.getRankFile(self.start\_row, self.start\_col)[0] + "x" + self.getRankFile(self.end\_row,  self.end\_col) + " e.p."  if self.piece\_captured != "--":  if self.piece\_moved[1] == "p":  return self.getRankFile(self.start\_row, self.start\_col)[0] + "x" + self.getRankFile(self.end\_row,  self.end\_col)  else:  return self.piece\_moved[1] + "x" + self.getRankFile(self.end\_row, self.end\_col)  else:  if self.piece\_moved[1] == "p":  return self.getRankFile(self.end\_row, self.end\_col)  else:  return self.piece\_moved[1] + self.getRankFile(self.end\_row, self.end\_col)    def getRankFile(self, row, col):  *"Will print the piece rank file."* return self.cols\_to\_files[col] + self.rows\_to\_ranks[row]   def \_\_str\_\_(self):  if self.is\_castle\_move:  return "0-0" if self.end\_col == 6 else "0-0-0"   end\_square = self.getRankFile(self.end\_row, self.end\_col)   if self.piece\_moved[1] == "p":  if self.is\_capture:  return self.cols\_to\_files[self.start\_col] + "x" + end\_square  else:  return end\_square + "Q" if self.is\_pawn\_promotion else end\_square   move\_string = self.piece\_moved[1]  if self.is\_capture:  move\_string += "x"  return move\_string + end\_square  **THIS IS THE END OF THE board.py file** |
| In summary, the objective of the board.py file is to manage, validate the piece movements. It has function for the user to not move in a square where the piece was meant to move and does the job of printing the chess notation on the main screen. |

|  |
| --- |
|  |
|  |

### piece.py

|  |
| --- |
| **def RenderImage():** Part of the code that will be responsible for adjusting the image according to the dimension specified by the engine.py file. |
| import pygame as p  BOX\_SIZE = 500//8 "The size of the single box will be achieved by dividing the HEIGHT/WIDTH by dimension i.e. 100" rendered = {} "Dictionary where the rendered images will be saved at"  def RenderImage():  pieces = ['wp', 'wR', 'wN', 'wB', 'wK', 'wQ', 'bp', 'bR', 'bN', 'bB', 'bK', 'bQ']  "Piece saved in the form of strings."  for piece in pieces:  rendered[piece] = p.transform.scale(p.image.load("resources/" + piece + ".png"), (BOX\_SIZE, BOX\_SIZE))  """The for loop will go through the pictures in the resources library which fetch and  render the image i.e. p.transform.scale which are the images specified by the BOX\_SIZE."""  **THIS IS THE END OF THE piece.py FILE** |
| In summary, the job of the piece.py file is to import images from the resource folder and adjust them. |

|  |
| --- |
|  |

### ai.py

|  |
| --- |
| **def findBestMove():** Manager ai.py subroutines when called in main.py.  **def findMoveNegaMaxAlphaBeta():** Subroutine that will manage the negamax algorithm for the AI part of the game.  **def scoreBoard():** Subroutine that will score the code based on the level of points that other piece has. If the points for the other piece is higher than the current AI piece, it will take it otherwise it will defend.  **def findRandomMove():** This subroutine is not in use as it was used for development purposes for when the ai was being tested first. |
| import random  piece\_score = {"K": 0, "Q": 9, "R": 5, "B": 3, "N": 3, "p": 1} """Is a variable that is responsible for assigning a point value for taking out each values. The AI priorities are as below; • King is not that worthy to take out as it would be defended  thereby having a piece\_score value of 0 (low priority). • Queen is a strong opponent so taking out a queen would make the  AI go for the queen (high priority). • Rook will be set to 5 (medium priority). • Bishop will be 3 (low priority) • Knight will be 3 (low priority) • Pawn will be 1 (lowest priority)."""  pawn\_scores = [[0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.99],  [0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8],  [0.6, 0.6, 0.7, 0.7, 0.7, 0.7, 0.6, 0.6],  [0.4, 0.4, 0.4, 0.5, 0.5, 0.4, 0.4, 0.4],  [0.2, 0.2, 0.2, 0.4, 0.4, 0.2, 0.2, 0.2],  [0.25, 0.15, 0.1, 0.2, 0.2, 0.1, 0.15, 0.25],  [0.25, 0.3, 0.3, 0.0, 0.0, 0.3, 0.3, 0.25],  [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]] "Pawn will be emphasised to move forward to get more pieces i.e. the queen." knight\_scores = [[0.0, 0.1, 0.2, 0.2, 0.2, 0.2, 0.1, 0.0],  [0.1, 0.3, 0.5, 0.5, 0.5, 0.5, 0.3, 0.1],  [0.2, 0.5, 0.6, 0.65, 0.65, 0.6, 0.5, 0.2],  [0.2, 0.55, 0.65, 0.9, 0.9, 0.65, 0.55, 0.2],  [0.2, 0.5, 0.65, 0.9, 0.9, 0.65, 0.5, 0.2],  [0.2, 0.55, 0.6, 0.65, 0.65, 0.6, 0.55, 0.2],  [0.1, 0.3, 0.5, 0.55, 0.55, 0.5, 0.3, 0.1],  [0.0, 0.1, 0.2, 0.2, 0.2, 0.2, 0.1, 0.0]] "Knights will be emphasied to be in the middle as it will the strongest there." rook\_scores = [[0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25],  [0.5, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.5],  [0.0, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.0],  [0.0, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.0],  [0.0, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.0],  [0.0, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.0],  [0.0, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.0],  [0.25, 0.25, 0.25, 0.5, 0.5, 0.25, 0.25, 0.25]] "Rooks will be best at the edges of the chess board."    bishop\_scores = [[0.0, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.0],  [0.2, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.2],  [0.2, 0.4, 0.5, 0.6, 0.6, 0.5, 0.4, 0.2],  [0.2, 0.5, 0.5, 0.6, 0.6, 0.5, 0.5, 0.2],  [0.2, 0.4, 0.6, 0.6, 0.6, 0.6, 0.4, 0.2],  [0.2, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.2],  [0.2, 0.5, 0.4, 0.4, 0.4, 0.4, 0.5, 0.2],  [0.0, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.0]] "Bishops will be best at the mid section of the chess board." queen\_scores = [[0.0, 0.2, 0.2, 0.3, 0.3, 0.2, 0.2, 0.0],  [0.2, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.2],  [0.2, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.2],  [0.3, 0.4, 0.5, 0.7, 0.7, 0.5, 0.4, 0.3],  [0.3, 0.4, 0.5, 0.6, 0.6, 0.5, 0.4, 0.3],  [0.2, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.2],  [0.2, 0.4, 0.5, 0.4, 0.4, 0.4, 0.4, 0.2],  [0.0, 0.2, 0.2, 0.3, 0.3, 0.2, 0.2, 0.0]] "Queen will be generally be the most powerful piece if in the middle." """Saved in these list are the scores which will give the AI points for moving the  pieces to going to the designated destination which will make it stronger. For instance, if the AI finds that if another location has more points than the current location, it will  move the piece to the designated places."""   piece\_position\_scores = {"wN": knight\_scores,  "bN": knight\_scores[::-1],  "wB": bishop\_scores,  "bB": bishop\_scores[::-1],  "wQ": queen\_scores,  "bQ": queen\_scores[::-1],  "wR": rook\_scores,  "bR": rook\_scores[::-1],  "wp": pawn\_scores,  "bp": pawn\_scores[::-1]}  CHECKMATE = 1000 "Priority to get a checkmate will be set to 1000" STALEMATE = 0 "Priority to get a stalemate will be set to 0" DEPTH = 3 """Variable that stores the constant 3 specifying the negamax algorithm depth i.e. the AI will reduce possibilities up to 3 moves. Whichever move has the highest value. It will be executed. The more the depth, the slower the ai will think."""   def findBestMove(game\_state, valid\_moves, return\_queue):  *"Subroutine that will act as the main subroutine for the ai.py file."* "It will be called in the main.py file."  global next\_move  next\_move = None  random.shuffle(valid\_moves)  findMoveNegamaxAlphaBeta(game\_state, valid\_moves, DEPTH, -CHECKMATE, CHECKMATE,  1 if game\_state.white\_to\_move else -1)  return\_queue.put(next\_move)   def findMoveNegamaxAlphaBeta(game\_state, valid\_moves, depth, alpha, beta, turn\_multiplier):  *"""Greedy fashion. We will be looking at all the moves and find the best available move.  It will also include alpha beta pruning, the idea is as we are going through the game tree,  at some state, if we find that if we are making a move, they may counter with a move which  result in the AI losing it. Rather than going through the recursion depth, we just eliminate  the game tree search."""* "When alpha becomes equal to beta, we break out of the loop."  global next\_move  if depth == 0:  return turn\_multiplier \* scoreBoard(game\_state)  max\_score = -CHECKMATE  "Score being negative CHECKMATE"  for move in valid\_moves:  "For move in the number of valid moves."  game\_state.makeMove(move)  "Making the move"  next\_moves = game\_state.moveValidation()  "Validate the move"  score = -findMoveNegamaxAlphaBeta(game\_state, next\_moves, depth - 1, -beta, -alpha, -turn\_multiplier)  if score > max\_score:  "If the score exceeds the max\_score"  max\_score = score  "max\_score will be set to score. It will make the maximise the best move potential."  if depth == DEPTH:  "After the depth has been reached i.e. 3"  next\_move = move  "next\_move will be set to move."  game\_state.undoMove()  if max\_score > alpha:  "If the max score is already greater than alpha, it becomes our new alpha value."  alpha = max\_score  if alpha >= beta:  "When alpha becomes equal to beta, we break out of the loop."  break  return max\_score   def scoreBoard(game\_state):  *"""Subroutine that scores moves. Where there is a positive for white, and a negative score  for black."""* if game\_state.checkmate:  if game\_state.white\_to\_move:  return -CHECKMATE  #black wins here  else:  return CHECKMATE  #White wins here  elif game\_state.stalemate:  return STALEMATE  score = 0  for row in range(len(game\_state.board)):  for columns in range(len(game\_state.board[row])):  piece = game\_state.board[row][columns]  if piece != "--":  "If the box is empty."  piece\_position\_score = 0  if piece[1] != "K":  piece\_position\_score = piece\_position\_scores[piece][row][columns]  if piece[0] == "w":  "If the piece is a white piece, we will add the score."  score += piece\_score[piece[1]] + piece\_position\_score  if piece[0] == "b":  "If the piece is a black piece, will will subtract the score."  score -= piece\_score[piece[1]] + piece\_position\_score   return score   def findRandomMove(valid\_moves):  *"""Calls the random.choice subroutine with valid move generation."""* return random.choice(valid\_moves)  **THIS IS THE END OF THE ai.py FILE.** |
| In summary, the function of the ai.py file of the game is to store points for the AI which will assist when needing to decide which move is the most optimum. It will do this by checking in the score list. |

**THIS IS THE END OF SINGPLAYER MODE**

## MULTIPLAYER MODE

### multiplayerENGINE.py

|  |
| --- |
| **multiplayerENGINE.py :** The multiplayer mode follows the same principle such as the singleplayerENGINE.py. However, it contains significant changes to facilitate with a friend. In this feature, the friend can pass the mouse to his/her friends showcasing the offline playing capabilities with a friend. |
| import pygame as p from modes.multiplayer.imports import piece as pie from modes.multiplayer.imports import board as b import sys  #################################### boardDIMENSION = 8 # width = height = 500 # BOX\_SIZE = height // boardDIMENSION# moveTURNwidth = 200 FPS = 60 # rendered = pie.rendered # #################################### p.init()  """• Firstly, we import pygame as p as it will manufacture, lay out piece images  that will initially set up the chess board. • We then need to import the piece from the other python file piece.py which will  be in a dictionary format already rendered in piece.py. • In the game file, we also have another folder named modes which holds the data  to manufacture the different game modes when taped by the button in the main GUI  screen. However, it hasn’t been completed. • We also import the board.py as b which imports the most vital class that holds  data about each individual piece i.e. class piece\_manager: • The sys module provides information about constants, functions and methods  of the Python interpreter.  • Secondly, we starting defining key global variables that will be used  throughout the program, we start with boardDIMENSION which simply means that  it would be used when manufacture the board. • width/height is used to save the overall program screen with boxheight  dividing the height with the board dimension to create 8 squares of equal  dimension. The // function is used for integer division which gets integer value and not real. • FPS is the game frames/second with the dictionary rendered being the  library where the pictures rendered by the RenderImage subroutine in the piece.py. • Finally, p.init() basically means that pygame is being initialised. • moveTURN holds dimension for producing which player’s turn it is."""    def playGame():  gs = b.GameState()  global colors  screen = p.display.set\_mode((width + moveTURNwidth, height))  clock = p.time.Clock()  screen.fill((0,0,0))  validate = gs.moveValidation()  """  • Screen is declared as a variable which will be used to set the display specified   by the global variable dimension i.e. width and height.  • Clock is an important variable where it will be used to animate the piece when   they are moved by the user.   • screen.fill basically fills the screen with black, i.e. specified by the RGB   values (0,0,0).  • gs is basically a call saved in engine.py which calls the class from board.py   which is GameState.  • validate is a call saved in engine.py which calls the class from board i.e.   moveValidation().  """   moveMADE = False  animations = False  pie.RenderImage()  running = True  square\_selected = ()  clicked = []   game\_over = False   """  • moveMADE is a flag variable which is set and will change as moves are made.  • animations is a flag variable which will turn True whenever the animateMoves  subroutine is called that will then animate the piece transition.  • pie.RenderImage() we initialise the subroutine def RenderImage from the   piece.py library which initialises the process of rendering the pictures  according to the dimension. It will be passed from piece.py.  • running is a flag variable that will change after the game finishes. It's the  idea of an infinite loop that will continuously run the game.  • square\_selected tracks user clicks and saves it in a tuple.  • clicked saves the clicks data of the user when clicked at a square on the   chess board.  • game\_over is a flag variables that triggers after the game is over.  """   while running:  "The flag variable running being called that will run the loop."   for event in p.event.get():  "For every event in the pygame library."   if event.type == p.QUIT:  "If the event type is pygame.QUIT"  p.quit()  sys.exit()  #"Pygame and system exiting."   elif event.type == p.MOUSEBUTTONDOWN:  "Part of the code when the user clicks a mouse button."   if not game\_over:  location = p.mouse.get\_pos() # Holds the location of the mouse.  column = location[0] // BOX\_SIZE  row = location[1] // BOX\_SIZE  "• location holds the position of the mouse."   if square\_selected == (row, column) or column >= 8:  square\_selected = ()  clicked = []  """  If the user clicks a location which doesn't have to do anything   with the game, clicked and square\_selected will be set to empty."""   else:  square\_selected = (row, column)  clicked.append(square\_selected)  """  If the user has clicked on the piece and then clicks on where they  want to move it. It will append the move in the clicked tuple.  """   if len(clicked) == 2:  move = b.Move(clicked[0], clicked[1], gs.board)  "Occurring after the second click and if the human is playing."   for i in range(len(validate)):  """  Loop that will execute to the number of contents in validate  i.e. validation subroutine in board.py.  """   if move == validate[i]:  "If the move matches the requirements in the validate library."   gs.makeMove(validate[i])  moveMADE = True  animations = True  square\_selected = ()  clicked = []  """  Variables being set to true if the move matches the validate   requirements."""   if not moveMADE:  clicked = [square\_selected]  "Error handling in case the user has not made any move."   elif event.type == p.KEYDOWN:  "Part of the code when pygame finds after it discovers a keyboard button click."   if event.key == p.K\_z:  "If the button click is z"  gs.undoMove()  "It will call the subroutine from board.py which is undoMove()"  moveMADE = True  animations = False  game\_over = False  """  • moveMADE will be set to True as the user will get to play again.  • animations will be False as it will have to redo as the user  has undone the move.  • game\_over will be set back to false even if the the end message  is displayed."""   if event.key == p.K\_ESCAPE:  "If the button click is esc"  p.display.set\_mode((width, height))  "It will set the window for the mainGUI pane"  running = False  "Will stop executing the loop"   if event.key == p.K\_r:  "If the keyboard button r is pressed."  gs = b.GameState()  "Defines the gamestate class from board.py in engine.py."  validate = gs.moveValidation()  "Defines the moveValidation class from board.py in engine.py."  square\_selected = ()  clicked = []  """  • square\_selected i.e. the tuple will be set back to empty.  • The clicked list will also be set back to an empty list."""  moveMADE = False  animations = False  game\_over = False  """  • moveMade will be False i.e. the user will get to start again.  • animations will be false as the user has to make a move and then the  code will start again with the animation.  • if the game was previously over, it would be set back to false."""   if moveMADE:  "If moveMADE is True"   if animations:  "If animations is True, it will start producing the animations"  animateMove(gs.moveRegister[-1], screen, gs.board, clock)  "Piece animation begins"  validate = gs.moveValidation()  "Move validation"  moveMADE = False  animations = False  move\_undone = False  "Key variables change to facilitate change."   drawGameState(screen, gs, validate, square\_selected, drawTurn)  "Starts drawing the game board with the pieces."   if gs.checkmate:  "if the program finds the game is in checkmate."  game\_over = True  "game\_over will be set to True"   if gs.white\_to\_move:  "If gs.white\_to\_move is True, this part will execute."  drawEndGameText(screen, "BLACK WINS THE GAME BY CHECKMATE")  """will draw the text on the screen that the game is over   by checkmate by black."""  else:  drawEndGameText(screen, "WHITE WINS THE GAME BY CHECKMATE")  """will draw the text on the screen that the game is over   by checkmate by white."""    elif gs.stalemate:  "if the program finds the game is in a stalemate"  game\_over = True  "game\_over will be set to True"  drawEndGameText(screen, "STALEMATE")  """will draw the text on the screen that the game is over by  stalemate any of the pieces."""   clock.tick(FPS)  p.display.flip()  """  • The clock will start ticking which will start FPS count, i.e. run the loop  on 60 frames.  • display.flip will update the contents of the entire display."""   def drawGameState(screen, gs, valid\_moves, square\_selected, moveTURNfont):   *"Imports arguments that holds information to construct the board."* drawBoard(screen)  "#draw squares on the board"  highlightSquares(screen, gs, valid\_moves, square\_selected)  "highlight squares on the board"  drawPieces(screen, gs.board)  "# draw pieces on top of those squares"  drawTurn(screen, gs)   def drawBoard(screen):  *"Imports arguments that holds data to construct the squares."* global colors  "Imports colors which don't require pygame to type RGB values."  colors = [p.Color("grey"), p.Color("darkcyan")]  "List that saves the colours for the variable BOX\_SQUARE"  for row in range(boardDIMENSION):  "for loop that will draw the 2D array i.e. the chess board."  for column in range(boardDIMENSION):  "# for loop that will partition the board which will result in 64 squares."  color = colors[((row + column) % 2)]  "# Board will be equal to the chess board pattern. Chequered squares."  p.draw.rect(screen, color, p.Rect(column \* BOX\_SIZE, row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE))  "# Draw the boxes."   def highlightSquares(screen, gs, valid\_moves, square\_selected):  *"Part of the game that will handle the highlighting of the chess pieces."* if (len(gs.moveRegister)) > 0:  "If the length of the moveRegister in the board.py file is greater than 0."  last\_move = gs.moveRegister[-1]  s = p.Surface((BOX\_SIZE, BOX\_SIZE))  s.set\_alpha(100)  s.fill(p.Color('green'))  """after the piece has reached its location, it will highlight  the square green."""  screen.blit(s, (last\_move.end\_col \* BOX\_SIZE, last\_move.end\_row \* BOX\_SIZE))  "producing the highlight."  if square\_selected != ():  "if the square\_selected tuple is not empty."  row, col = square\_selected  "row, column will be equal to square\_selected as it will highlight."   if gs.board[row][col][0] == (   'w' if gs.white\_to\_move else 'b'):   "square\_selected is a piece that can be moved"  "highlight selected square"  s = p.Surface((BOX\_SIZE, BOX\_SIZE))  s.set\_alpha(100)  "transparency value 0 -> transparent, 255 -> opaque"  s.fill(p.Color('blue'))  screen.blit(s, (col \* BOX\_SIZE, row \* BOX\_SIZE))  "highlight moves from that square"  s.fill(p.Color('yellow'))   for move in valid\_moves:  "for the number of moves in valid\_moves"  if move.start\_row == row and move.start\_col == col:  "if the move.start\_row is equal to the column"  screen.blit(s, (move.end\_col \* BOX\_SIZE, move.end\_row \* BOX\_SIZE))  """on the specific row and column the screen will be blit,  the highlight will be drawn."""   # Code that will fit the chess piece image on the board. def drawPieces(screen, board):  *"Subroutine that will print the chess pieces on the board."* for row in range(boardDIMENSION):  "for i in boardDimension i.e. 8 because 64 boxes on board"  for column in range(boardDIMENSION):  "for j in boardDimension i.e. 8 because 64 boxes on board"  piece = board[row][column]  "saving the board rows and columns in a variable named piece"  if piece != "--":  "if piece is empty, then it will leave that space empty"  screen.blit(rendered[piece], p.Rect(column \* BOX\_SIZE, row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE))  "finally render the images on the board."   def drawTurn(screen, gs):  *"Subroutine that will manufacture whose turn it is on the multiplayerENGINE pane"* white = p.image.load('resources/' + 'white.jpg')  black = p.image.load('resources/' + 'black.jpg')  "Loads the pictures."    playerFont = p.font.SysFont("Aleo", 35)  box1 = p.Rect((525, 100), (150, 100))  box2 = p.Rect((525, 300), (150, 100))  player1 = playerFont.render("PLAYER 1", True, p.Color("black"))  player2 = playerFont.render("PLAYER 2", True, p.Color("black"))  highlightP1 = p.Rect((515, 90), (170, 120))  highlightP2 = p.Rect((515, 290), (170, 120))  """Varibles that holds vital information to produce the text and rectangles  on the singleENGINE pane."""    if gs.white\_to\_move:  """This needs to produced first since other rectangles  will be placed over this."""   "Exception handling to handle which player is playing"  "In this case if it is white's turn."  p.draw.rect(screen, (p.Color("gold")), highlightP1)  "Highlight that will show which player't turn it is."  else:  p.draw.rect(screen, (p.Color("gold")), highlightP2)  "Highlight that will show which player't turn it is."   if not gs.white\_to\_move:  p.draw.rect(screen, (p.Color("black")), highlightP1)  "Highlight that will show which player't turn it is."  else:  p.draw.rect(screen, (p.Color("black")), highlightP2)  "Highlight that will show which player't turn it is."   p.draw.rect(screen, (p.Color("darkcyan")), box1)  p.draw.rect(screen, (p.Color("grey")), box2)  screen.blit(player1, ((540, 105)))  screen.blit(player2, ((540, 305)))  screen.blit(white, ((590, 130)))  screen.blit(black, ((590, 330)))  "Drawing the squares and text on the pane"   def drawEndGameText(screen, text):  *"Part of the code that will print the last part of the game i.e. checkmate/stalemate"* font = p.font.SysFont("Helvetica", 25, True, False)  "font objects such as the colour and font style"  text\_object = font.render(text, False, p.Color("gray"))  text\_location = p.Rect(0, 0, width, height).move(width / 2 - text\_object.get\_width() / 2,  height / 2 - text\_object.get\_height() / 2)  "location of the font saved in text\_location"  screen.blit(text\_object, text\_location)  "produces the text\_object on the screen"  text\_object = font.render(text, False, p.Color('black'))  "location of the text\_object saved in text\_object"  screen.blit(text\_object, text\_location.move(2, 2))  "produces the text location on the board"   def animateMove(move, screen, board, clock):  *"Part of the code that will animate the chess moves."* global colors  """Imports colors which is a library which contains colours, instead of  entering RGB values, only entering string values is enough."""  rows = move.end\_row - move.start\_row  "subtracts the two variables to check the distance of animation"  columns = move.end\_col - move.start\_col  "subtracts the two variables to check the distance of animation"  FPSpieceMove = 15 # frames to move one square  "frames to move one square"  FPScount = (abs(rows) + abs(columns)) \* FPSpieceMove  ""  for frame in range(FPScount + 1):  row, col = (move.start\_row + rows \* frame / FPScount, move.start\_col + columns \* frame / FPScount)  drawBoard(screen)  drawPieces(screen, board)  "erase the piece moved from its ending square"  color = colors[(move.end\_row + move.end\_col) % 2]  end\_square = p.Rect(move.end\_col \* BOX\_SIZE, move.end\_row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE)  p.draw.rect(screen, color, end\_square)  "draw captured piece onto rectangle"  if move.piece\_captured != '--':  if move.is\_enpassant\_move:  enpassant\_row = move.end\_row + 1 if move.piece\_captured[0] == 'b' else move.end\_row - 1  end\_square = p.Rect(move.end\_col \* BOX\_SIZE, enpassant\_row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE)  screen.blit(rendered[move.piece\_captured], end\_square)  "draw moving piece"  screen.blit(rendered[move.piece\_moved], p.Rect(col \* BOX\_SIZE, row \* BOX\_SIZE, BOX\_SIZE, BOX\_SIZE))  p.display.flip()  clock.tick(60)  if \_\_name\_\_ == "\_\_main\_\_":  playGame()  **THIS IS THE END OF THE multiplayerENGINE.py FILE** |
| In summary, this part of the code is responsible for handling all the other files along with exception handling. If the user exits this file, it will automatically reroute the user to the mainGUI pane. |

### board.py

|  |
| --- |
| **class GameState:** Contains all the methods that will validate the chess moves.  **Def makeMove():** Subroutine that allows the players to make a move.  **Def undoMove():** Subroutine that will allow the user to undo a move.  **Def updateCastleRights():** Subroutine that has castle rights that is vital for validation.  **Def getValidMoves():** Subroutines that handles move validation.  **Def inCheck():** Subroutine that will check whether if either of the players are in check.  **Def squareUnderAttack():** Determine whether if enemy can attack the players piece positions.  **Def getAllPossibleMoves():** Subroutine that will trigger piece subroutines whenever a piece being clicked in the self.board state is being triggered.  **Def checkForPinsAndChecks():** Keeps track of pins and checks.  **Def getPawnMoves():** Subroutine that manages pawn movement.  **Def getRookMoves():** Subroutine that manages rook movement.  **Def getKnightMoves():** Subroutine that manages knight movement.  **Def getBishopMove():** Subroutine that manages bishop movement.  **Def getQueenMoves():** Subroutine that manages queen movement.  **Def getKingMoves():** Subroutine that manages king movement.  **Def getCastleMoves():** Subroutine that will take care of all the castling rights.  **Def getKingsideCastleMoves():** Subroutine that will take care of castling rights with the king.  **Def getQueensideCastleMoves():** Subroutine that will take care of castling rights with the queen.  **Class CastleRights:** Class where castle rights are defined.  **Class Move:** Class that will control piece movements.  **Def \_\_eq\_\_():** Python method that overrides the equals method.  **Def getChessNotation():** Subroutine that prints the chess notation on the moveRegister window.  **Def getRankFile():** Will print the piece rank file.  **Def \_\_str\_\_():** The \_\_str\_\_ method is one that should be the most human-readable possible, yet also descriptive of that exact object. |
| Class GameState:  *“Main class that will handle all piece functions.”* def \_\_init\_\_(self):  *“”” “—" will represent the empty list, whereas every other word is a piece name directly”””* self.board = [  [“bR”, “bN”, “bB”, “bQ”, “bK”, “bB”, “bN”, “bR”],  [“bp”, “bp”, “bp”, “bp”, “bp”, “bp”, “bp”, “bp”],  [“—", “—", “—", “—", “—", “—", “—", “—"],  [“—", “—", “—", “—", “—", “—", “—", “—"], # 2x2 array as there are 64 squares in a chess board.  [“—", “—", “—", “—", “—", “—", “—", “—"],  [“—", “—", “—", “—", “—", “—", “—", “—"],  [“wp”, “wp”, “wp”, “wp”, “wp”, “wp”, “wp”, “wp”],  [“wR”, “wN”, “wB”, “wQ”, “wK”, “wB”, “wN”, “wR”]]  “directly imported from the pieces library in the engine.py file.”  self.white\_to\_move = True  “After black, the right to move will be transferred to white.”  self.moveRegister = []  ‘List that will keep track of all the moves.’  self.white\_king\_location = (7, 4)  self.black\_king\_location = (0, 4)  self.checkmate = False  ‘Flag variable that is set to see when the players are in checkmate.’  self.stalemate = False  ‘Flag variable that is set to see when the players are in stalemate.’  self.in\_check = False  ‘Flat variable that is set to see when the players are in check.’  self.pins = []  self.checks = []  self.enpassant\_possible = ()  ‘coordinates for the square where en-passant capture is possible’  self.enpassant\_possible\_log = [self.enpassant\_possible]  self.current\_castling\_rights = CastleRights(True, True, True, True)  self.castle\_rights\_log = [CastleRights(self.current\_castling\_rights.wks, self.current\_castling\_rights.bks,  self.current\_castling\_rights.wqs, self.current\_castling\_rights.bqs)]   def makeMove(self, move):  *“Subroutine that takes the parameter move that allows us to make the move.”* self.board[move.start\_row][move.start\_col] = “—"  self.board[move.end\_row][move.end\_col] = move.piece\_moved  self.moveRegister.append(move)  “log the move so we can undo it later”  self.white\_to\_move = not self.white\_to\_move  “switch players”  “update king’s location if moved”  if move.piece\_moved == “wK”:  self.white\_king\_location = (move.end\_row, move.end\_col)  elif move.piece\_moved == “bK”:  self.black\_king\_location = (move.end\_row, move.end\_col)   “Code that will trigger once the pieces make it to the start of the other players squares.”  if move.is\_pawn\_promotion:  self.board[move.end\_row][move.end\_col] = move.piece\_moved[0] + “Q”   “Code that will check for en-passant moves.”  if move.is\_enpassant\_move:  self.board[move.start\_row][move.end\_col] = “—" # capturing the pawn   “update enpassant\_possible variable”  if move.piece\_moved[1] == “p” and abs(move.start\_row — move.end\_row) == 2:  “only on 2 square pawn advance”  self.enpassant\_possible = ((move.start\_row + move.end\_row) // 2, move.start\_col)  else:  self.enpassant\_possible = ()   “castle move”  if move.is\_castle\_move:  if move.end\_col — move.start\_col == 2:  “king-side castle move”  self.board[move.end\_row][move.end\_col — 1] = self.board[move.end\_row][  move.end\_col + 1]  “moves the rook to its new square”  self.board[move.end\_row][move.end\_col + 1] = ‘—' “erase old rook”  else:  “queen-side castle move”  self.board[move.end\_row][move.end\_col + 1] = self.board[move.end\_row][  move.end\_col — 2] # moves the rook to its new square  self.board[move.end\_row][move.end\_col — 2] = ‘—' # erase old rook   self.enpassant\_possible\_log.append(self.enpassant\_possible)   “update castling rights — whenever it is a rook or king move”  self.updateCastleRights(move)  self.castle\_rights\_log.append(CastleRights(self.current\_castling\_rights.wks, self.current\_castling\_rights.bks,  self.current\_castling\_rights.wqs, self.current\_castling\_rights.bqs))  “Subroutine that is the opposite of makeMove.”  def undoMove(self):  if len(self.moveRegister) != 0:  “make sure that there is a move to undo”  move = self.moveRegister.pop()  self.board[move.start\_row][move.start\_col] = move.piece\_moved  self.board[move.end\_row][move.end\_col] = move.piece\_captured  self.white\_to\_move = not self.white\_to\_move  “swap players”  “update the king’s position if needed”  if move.piece\_moved == “wK”:  self.white\_king\_location = (move.start\_row, move.start\_col)  elif move.piece\_moved == “bK”:  self.black\_king\_location = (move.start\_row, move.start\_col)  # undo en passant move  if move.is\_enpassant\_move:  self.board[move.end\_row][move.end\_col] = “—" “leave landing square blank”  self.board[move.start\_row][move.end\_col] = move.piece\_captured   self.enpassant\_possible\_log.pop()  self.enpassant\_possible = self.enpassant\_possible\_log[-1]   # undo castle rights  self.castle\_rights\_log.pop()  “get rid of the new castle rights from the move we are undoing”  self.current\_castling\_rights = self.castle\_rights\_log[  -1]  “set the current castle rights to the last one in the list”  “undo the castle move”  if move.is\_castle\_move:  if move.end\_col — move.start\_col == 2:  “king-side”  self.board[move.end\_row][move.end\_col + 1] = self.board[move.end\_row][move.end\_col — 1]  self.board[move.end\_row][move.end\_col — 1] = ‘—'  else:  “queen-side”  self.board[move.end\_row][move.end\_col — 2] = self.board[move.end\_row][move.end\_col + 1]  self.board[move.end\_row][move.end\_col + 1] = ‘—'  self.checkmate = False  self.stalemate = False  def updateCastleRights(self, move):  *“Updates the castle rights that is given by the moves.”* if move.piece\_captured == “wR”:  if move.end\_col == 0: # left rook  self.current\_castling\_rights.wqs = False  elif move.end\_col == 7: # right rook  self.current\_castling\_rights.wks = False  elif move.piece\_captured == “bR”:  if move.end\_col == 0: # left rook  self.current\_castling\_rights.bqs = False  elif move.end\_col == 7: # right rook  self.current\_castling\_rights.bks = False  if move.piece\_moved == ‘wK’:  self.current\_castling\_rights.wqs = False  self.current\_castling\_rights.wks = False  elif move.piece\_moved == ‘bK’:  self.current\_castling\_rights.bqs = False  self.current\_castling\_rights.bks = False  elif move.piece\_moved == ‘wR’:  if move.start\_row == 7:  if move.start\_col == 0: # left rook  self.current\_castling\_rights.wqs = False  elif move.start\_col == 7: # right rook  self.current\_castling\_rights.wks = False  elif move.piece\_moved == ‘bR’:  if move.start\_row == 0:  if move.start\_col == 0: # left rook  self.current\_castling\_rights.bqs = False  elif move.start\_col == 7: # right rook  self.current\_castling\_rights.bks = False   # Subroutine that handles move validation.  def moveValidation(self):  temp\_castle\_rights = CastleRights(self.current\_castling\_rights.wks, self.current\_castling\_rights.bks,  self.current\_castling\_rights.wqs, self.current\_castling\_rights.bqs)  # advanced algorithm  moves = []  self.in\_check, self.pins, self.checks = self.checkForPinsAndChecks()   if self.white\_to\_move:  king\_row = self.white\_king\_location[0]  king\_col = self.white\_king\_location[1]  else:  king\_row = self.black\_king\_location[0]  king\_col = self.black\_king\_location[1]  if self.in\_check:  if len(self.checks) == 1:  “only 1 check, block the check or move the king”  moves = self.getAllPossibleMoves()  “to block the check you must put a piece into one of the squares between the enemy piece and your king”  check = self.checks[0]  “check information”  check\_row = check[0]  check\_col = check[1]  piece\_checking = self.board[check\_row][check\_col]  valid\_squares = []  “squares that pieces can move to saved in a list, will be passed to other subroutines”  “if knight, must capture the knight or move your king, other pieces can be blocked”  if piece\_checking[1] == “N”:  valid\_squares = [(check\_row, check\_col)]  else:  for I in range(1, 8):  valid\_square = (king\_row + check[2] \* I,  king\_col + check[3] \* i)  “check[2] and check[3] are the check directions”  valid\_squares.append(valid\_square)  if valid\_square[0] == check\_row and valid\_square[  1] == check\_col:  “once you get to piece and check”  break  “get rid of any moves that don’t block check or move king”  for I in range(len(moves) — 1, -1, -1):  “iterate through the list backwards when removing elements”  if moves[i].piece\_moved[1] != “K”:  “move doesn’t move king so it must block or capture”  if not (moves[i].end\_row,  moves[i].end\_col) in valid\_squares:  “move doesn’t block or capture piece”  moves.remove(moves[i])  else:  “double check, king has to move”  self.getKingMoves(king\_row, king\_col, moves)  else:  “not in check — all moves are fine”  moves = self.getAllPossibleMoves()  if self.white\_to\_move:  self.getCastleMoves(self.white\_king\_location[0], self.white\_king\_location[1], moves)  else:  self.getCastleMoves(self.black\_king\_location[0], self.black\_king\_location[1], moves)   if len(moves) == 0:  if self.inCheck():  self.checkmate = True  else:  self.stalemate = True  else:  self.checkmate = False  self.stalemate = False   self.current\_castling\_rights = temp\_castle\_rights  return moves  def inCheck(self):  *“Subroutine that will check whether if a player is in check.”* if self.white\_to\_move:  return self.squareUnderAttack(self.white\_king\_location[0], self.white\_king\_location[1])  else:  return self.squareUnderAttack(self.black\_king\_location[0], self.black\_king\_location[1])    def squareUnderAttack(self, row, col):  *“Determine if enemy can attack the square row col”* self.white\_to\_move = not self.white\_to\_move  “switch to opponent’s point of view”  opponents\_moves = self.getAllPossibleMoves()  self.white\_to\_move = not self.white\_to\_move  for move in opponents\_moves:  if move.end\_row == row and move.end\_col == col: # square is under attack  return True  return False    def getAllPossibleMoves(self):  *“Subroutine that checks what moves have taken place and will thereby trigger the appropriate piece subroutine.”* moves = []  for row in range(len(self.board)):  for col in range(len(self.board[row])):  turn = self.board[row][col][0]  if (turn == “w” and self.white\_to\_move) or (turn == “b” and not self.white\_to\_move):  piece = self.board[row][col][1]  if piece == “p”:  self.getPawnMoves(row, col, moves)  elif piece == “R”:  self.getRookMoves(row, col, moves)  elif piece == “B”:  self.getBishopMoves(row, col, moves)  elif piece == “N”:  self.getKnightMoves(row, col, moves)  elif piece == “Q”:  self.getQueenMoves(row, col, moves)  elif piece == “K”:  self.getKingMoves(row, col, moves)  return moves  def checkForPinsAndChecks(self):  pins = []  “squares pinned and the direction its pinned from”  checks = []  “squares where enemy is applying a check”  in\_check = False  if self.white\_to\_move:  enemy\_color = “b”  ally\_color = “w”  start\_row = self.white\_king\_location[0]  start\_col = self.white\_king\_location[1]  else:  enemy\_color = “w”  ally\_color = “b”  start\_row = self.black\_king\_location[0]  start\_col = self.black\_king\_location[1]  “check outwards from king for pins and checks, keep track of pins”  directions = ((-1, 0), (0, -1), (1, 0), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1))  for j in range(len(directions)):  direction = directions[j]  possible\_pin = () # reset possible pins  for I in range(1, 8):  end\_row = start\_row + direction[0] \* i  end\_col = start\_col + direction[1] \* i  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7:  end\_piece = self.board[end\_row][end\_col]  if end\_piece[0] == ally\_color and end\_piece[1] != “K”:  if possible\_pin == (): # first allied piece could be pinned  possible\_pin = (end\_row, end\_col, direction[0], direction[1])  else: # 2nd allied piece — no check or pin from this direction  break  elif end\_piece[0] == enemy\_color:  enemy\_type = end\_piece[1]  “””1. Orthogonally away from king and piece is a rook  2. Diagonally away from king and piece is a bishop  3. 1 square away diagonally from king and piece is a pawn  4. Any direction and piece is a queen  5. Any direction 1 square away and piece is a king”””  if (0 <= j <= 3 and enemy\_type == “R”) or (4 <= j <= 7 and enemy\_type == “B”) or (  I == 1 and enemy\_type == “p” and (  (enemy\_color == “w” and 6 <= j <= 7) or (enemy\_color == “b” and 4 <= j <= 5))) or (  enemy\_type == “Q”) or (I == 1 and enemy\_type == “K”):  if possible\_pin == (): # no piece blocking, so check  in\_check = True  checks.append((end\_row, end\_col, direction[0], direction[1]))  break  else: # piece blocking so pin  pins.append(possible\_pin)  break  else: # enemy piece not applying checks  break  else:  break # off board  # check for knight checks  knight\_moves = ((-2, -1), (-2, 1), (-1, 2), (1, 2), (2, -1), (2, 1), (-1, -2), (1, -2))  for move in knight\_moves:  end\_row = start\_row + move[0]  end\_col = start\_col + move[1]  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7:  end\_piece = self.board[end\_row][end\_col]  if end\_piece[0] == enemy\_color and end\_piece[1] == “N”: # enemy knight attacking a king  in\_check = True  checks.append((end\_row, end\_col, move[0], move[1]))  return in\_check, pins, checks    def getPawnMoves(self, row, col, moves):  *“Subroutine that will be trigger whenever the pawn is moved, it will be saved in a list moves then.”* piece\_pinned = False  pin\_direction = ()  for I in range(len(self.pins) — 1, -1, -1):  if self.pins[i][0] == row and self.pins[i][1] == col:  piece\_pinned = True  pin\_direction = (self.pins[i][2], self.pins[i][3])  self.pins.remove(self.pins[i])  break   if self.white\_to\_move:  move\_amount = -1  start\_row = 6  enemy\_color = “b”  king\_row, king\_col = self.white\_king\_location  else:  move\_amount = 1  start\_row = 1  enemy\_color = “w”  king\_row, king\_col = self.black\_king\_location   if self.board[row + move\_amount][col] == “—": # 1 square pawn advance  if not piece\_pinned or pin\_direction == (move\_amount, 0):  moves.append(Move((row, col), (row + move\_amount, col), self.board))  if row == start\_row and self.board[row + 2 \* move\_amount][col] == “—": # 2 square pawn advance  moves.append(Move((row, col), (row + 2 \* move\_amount, col), self.board))  if col — 1 >= 0: # capture to the left  if not piece\_pinned or pin\_direction == (move\_amount, -1):  if self.board[row + move\_amount][col — 1][0] == enemy\_color:  moves.append(Move((row, col), (row + move\_amount, col — 1), self.board))  if (row + move\_amount, col — 1) == self.enpassant\_possible:  attacking\_piece = blocking\_piece = False  if king\_row == row:  if king\_col < col: # king is left of the pawn  # inside: between king and the pawn;  # outside: between pawn and border;  inside\_range = range(king\_col + 1, col — 1)  outside\_range = range(col + 1, 8)  else: # king right of the pawn  inside\_range = range(king\_col — 1, col, -1)  outside\_range = range(col — 2, -1, -1)  for I in inside\_range:  if self.board[row][i] != “—": # some piece beside en-passant pawn blocks  blocking\_piece = True  for I in outside\_range:  square = self.board[row][i]  if square[0] == enemy\_color and (square[1] == “R” or square[1] == “Q”):  attacking\_piece = True  elif square != “—":  blocking\_piece = True  if not attacking\_piece or blocking\_piece:  moves.append(Move((row, col), (row + move\_amount, col — 1), self.board, is\_enpassant\_move=True))  if col + 1 <= 7: # capture to the right  if not piece\_pinned or pin\_direction == (move\_amount, +1):  if self.board[row + move\_amount][col + 1][0] == enemy\_color:  moves.append(Move((row, col), (row + move\_amount, col + 1), self.board))  if (row + move\_amount, col + 1) == self.enpassant\_possible:  attacking\_piece = blocking\_piece = False  if king\_row == row:  if king\_col < col: # king is left of the pawn  # inside: between king and the pawn;  # outside: between pawn and border;  inside\_range = range(king\_col + 1, col)  outside\_range = range(col + 2, 8)  else: # king right of the pawn  inside\_range = range(king\_col — 1, col + 1, -1)  outside\_range = range(col — 1, -1, -1)  for I in inside\_range:  if self.board[row][i] != “—": # some piece beside en-passant pawn blocks  blocking\_piece = True  for I in outside\_range:  square = self.board[row][i]  if square[0] == enemy\_color and (square[1] == “R” or square[1] == “Q”):  attacking\_piece = True  elif square != “—":  blocking\_piece = True  if not attacking\_piece or blocking\_piece:  moves.append(Move((row, col), (row + move\_amount, col + 1), self.board, is\_enpassant\_move=True))  def getRookMoves(self, row, col, moves):  *“Subroutine that will handle all the rook moves. It will be triggered whenever rook is moved.”* piece\_pinned = False  pin\_direction = ()  for I in range(len(self.pins) — 1, -1, -1):  if self.pins[i][0] == row and self.pins[i][1] == col:  piece\_pinned = True  pin\_direction = (self.pins[i][2], self.pins[i][3])  if self.board[row][col][  1] != “Q”: # can’t remove queen from pin on rook moves, only remove it on bishop moves  self.pins.remove(self.pins[i])  break  directions = ((-1, 0), (0, -1), (1, 0), (0, 1)) # up, left, down, right  enemy\_color = “b” if self.white\_to\_move else “w”  for direction in directions:  for I in range(1, 8):  end\_row = row + direction[0] \* i  end\_col = col + direction[1] \* i  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7: # check for possible moves only in boundaries of the board  if not piece\_pinned or pin\_direction == direction or pin\_direction == (  -direction[0], -direction[1]):  end\_piece = self.board[end\_row][end\_col]  if end\_piece == “—": # empty space is valid  moves.append(Move((row, col), (end\_row, end\_col), self.board))  elif end\_piece[0] == enemy\_color: # capture enemy piece  moves.append(Move((row, col), (end\_row, end\_col), self.board))  break  else: # friendly piece  break  else: # off board  break    def getKnightMoves(self, row, col, moves):  *“Subroutine for knight pattern moving. It will trigger once the knight has moved.”* piece\_pinned = False  for I in range(len(self.pins) — 1, -1, -1):  if self.pins[i][0] == row and self.pins[i][1] == col:  piece\_pinned = True  self.pins.remove(self.pins[i])  break   knight\_moves = ((-2, -1), (-2, 1), (-1, 2), (1, 2), (2, -1), (2, 1), (-1, -2),  (1, -2)) # up/left up/right right/up right/down down/left down/right left/up left/down  ally\_color = “w” if self.white\_to\_move else “b”  for move in knight\_moves:  end\_row = row + move[0]  end\_col = col + move[1]  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7:  if not piece\_pinned:  end\_piece = self.board[end\_row][end\_col]  if end\_piece[0] != ally\_color: # so its either enemy piece or empty square  moves.append(Move((row, col), (end\_row, end\_col), self.board))  def getBishopMoves(self, row, col, moves):  *“Subroutine that will trigger whenever the bishop moves.”* piece\_pinned = False  pin\_direction = ()  for I in range(len(self.pins) — 1, -1, -1):  if self.pins[i][0] == row and self.pins[i][1] == col:  piece\_pinned = True  pin\_direction = (self.pins[i][2], self.pins[i][3])  self.pins.remove(self.pins[i])  break   directions = ((-1, -1), (-1, 1), (1, 1), (1, -1)) # diagonals: up/left up/right down/right down/left  enemy\_color = “b” if self.white\_to\_move else “w”  for direction in directions:  for I in range(1, 8):  end\_row = row + direction[0] \* i  end\_col = col + direction[1] \* i  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7: # check if the move is on board  if not piece\_pinned or pin\_direction == direction or pin\_direction == (  -direction[0], -direction[1]):  end\_piece = self.board[end\_row][end\_col]  if end\_piece == “—": # empty space is valid  moves.append(Move((row, col), (end\_row, end\_col), self.board))  elif end\_piece[0] == enemy\_color: # capture enemy piece  moves.append(Move((row, col), (end\_row, end\_col), self.board))  break  else: # friendly piece  break  else: # off board  break    def getQueenMoves(self, row, col, moves):  *“A queen is basically a rook as well as a bishop. It will call the bishop and rook subroutine for minimal coding.”* self.getBishopMoves(row, col, moves)  self.getRookMoves(row, col, moves)    def getKingMoves(self, row, col, moves):  *“King can basically move 1 direction at a time. It will be activated whenever someone moves the king.”* row\_moves = (-1, -1, -1, 0, 0, 1, 1, 1)  col\_moves = (-1, 0, 1, -1, 1, -1, 0, 1)  ally\_color = “w” if self.white\_to\_move else “b”  for I in range(8):  end\_row = row + row\_moves[i]  end\_col = col + col\_moves[i]  if 0 <= end\_row <= 7 and 0 <= end\_col <= 7:  end\_piece = self.board[end\_row][end\_col]  if end\_piece[0] != ally\_color: # not an ally piece — empty or enemy  # place king on end square and check for checks  if ally\_color == “w”:  self.white\_king\_location = (end\_row, end\_col)  else:  self.black\_king\_location = (end\_row, end\_col)  in\_check, pins, checks = self.checkForPinsAndChecks()  if not in\_check:  moves.append(Move((row, col), (end\_row, end\_col), self.board))  # place king back on original location  if ally\_color == “w”:  self.white\_king\_location = (row, col)  else:  self.black\_king\_location = (row, col)    def getCastleMoves(self, row, col, moves):  *“Subroutine that will take care of all the castling rights.”* if self.squareUnderAttack(row, col):  return # can’t castle while in check  if (self.white\_to\_move and self.current\_castling\_rights.wks) or (  not self.white\_to\_move and self.current\_castling\_rights.bks):  self.getKingsideCastleMoves(row, col, moves)  if (self.white\_to\_move and self.current\_castling\_rights.wqs) or (  not self.white\_to\_move and self.current\_castling\_rights.bqs):  self.getQueensideCastleMoves(row, col, moves)    def getKingsideCastleMoves(self, row, col, moves):  *“Subroutine that will take care of all the castling moves with the king.”* if self.board[row][col + 1] == ‘—' and self.board[row][col + 2] == ‘—':  if not self.squareUnderAttack(row, col + 1) and not self.squareUnderAttack(row, col + 2):  moves.append(Move((row, col), (row, col + 2), self.board, is\_castle\_move=True))    def getQueensideCastleMoves(self, row, col, moves):  *“Subroutine that will take care of all the castling moves with the queen.”* if self.board[row][col — 1] == ‘—' and self.board[row][col — 2] == ‘—' and self.board[row][col — 3] == ‘—':  if not self.squareUnderAttack(row, col — 1) and not self.squareUnderAttack(row, col — 2):  moves.append(Move((row, col), (row, col — 2), self.board, is\_castle\_move=True))   class CastleRights:  *“Class where certain castle rights are defined.”* def \_\_init\_\_(self, wks, bks, wqs, bqs):  self.wks = wks  self.bks = bks  self.wqs = wqs  self.bqs = bqs  class Move:  *“Class that will control the movement of the pieces.”*  ranks\_to\_rows = {“1”: 7, “2”: 6, “3”: 5, “4”: 4,  “5”: 3, “6”: 2, “7”: 1, “8”: 0}  rows\_to\_ranks = {v: k for k, v in ranks\_to\_rows.items()}  files\_to\_cols = {“a”: 0, “b”: 1, “c”: 2, “d”: 3,  “e”: 4, “f”: 5, “g”: 6, “h”: 7}  cols\_to\_files = {v: k for k, v in files\_to\_cols.items()}   def \_\_init\_\_(self, start\_square, end\_square, board, is\_enpassant\_move=False, is\_castle\_move=False):  self.start\_row = start\_square[0]  self.start\_col = start\_square[1]  self.end\_row = end\_square[0]  self.end\_col = end\_square[1]  self.piece\_moved = board[self.start\_row][self.start\_col]  self.piece\_captured = board[self.end\_row][self.end\_col]  # pawn promotion  self.is\_pawn\_promotion = (self.piece\_moved == “wp” and self.end\_row == 0) or (  self.piece\_moved == “bp” and self.end\_row == 7)  # en passant  self.is\_enpassant\_move = is\_enpassant\_move  if self.is\_enpassant\_move:  self.piece\_captured = “wp” if self.piece\_moved == “bp” else “bp”  # castle move  self.is\_castle\_move = is\_castle\_move   self.is\_capture = self.piece\_captured != “—"  self.moveID = self.start\_row \* 1000 + self.start\_col \* 100 + self.end\_row \* 10 + self.end\_col   # Function that overrides the equals method.  def \_\_eq\_\_(self, other):  if isinstance(other, Move):  return self.moveID == other.moveID  return False  def getChessNotation(self):  *“Subroutine that prints the chess notation on the moveRegister\_movelog.”* if self.is\_pawn\_promotion:  return self.getRankFile(self.end\_row, self.end\_col) + “Q”  if self.is\_castle\_move:  if self.end\_col == 1:  return “0-0-0”  else:  return “0-0”  if self.is\_enpassant\_move:  return self.getRankFile(self.start\_row, self.start\_col)[0] + “x” + self.getRankFile(self.end\_row,  self.end\_col) + “ e.p.”  if self.piece\_captured != “—":  if self.piece\_moved[1] == “p”:  return self.getRankFile(self.start\_row, self.start\_col)[0] + “x” + self.getRankFile(self.end\_row,  self.end\_col)  else:  return self.piece\_moved[1] + “x” + self.getRankFile(self.end\_row, self.end\_col)  else:  if self.piece\_moved[1] == “p”:  return self.getRankFile(self.end\_row, self.end\_col)  else:  return self.piece\_moved[1] + self.getRankFile(self.end\_row, self.end\_col)    def getRankFile(self, row, col):  *“Will print the piece rank file.”* return self.cols\_to\_files[col] + self.rows\_to\_ranks[row]   def \_\_str\_\_(self):  if self.is\_castle\_move:  return “0-0” if self.end\_col == 6 else “0-0-0”   end\_square = self.getRankFile(self.end\_row, self.end\_col)   if self.piece\_moved[1] == “p”:  if self.is\_capture:  return self.cols\_to\_files[self.start\_col] + “x” + end\_square  else:  return end\_square + “Q” if self.is\_pawn\_promotion else end\_square   move\_string = self.piece\_moved[1]  if self.is\_capture:  move\_string += “x”  return move\_string + end\_square  **THIS IS THE END OF THE board.py file** |
| In summary, the objective of the board.py file is to manage, validate the piece movements. It has function for the user to not move in a square where the piece was meant to move and does the job of printing the chess notation on the main screen. |

|  |
| --- |
|  |
|  |

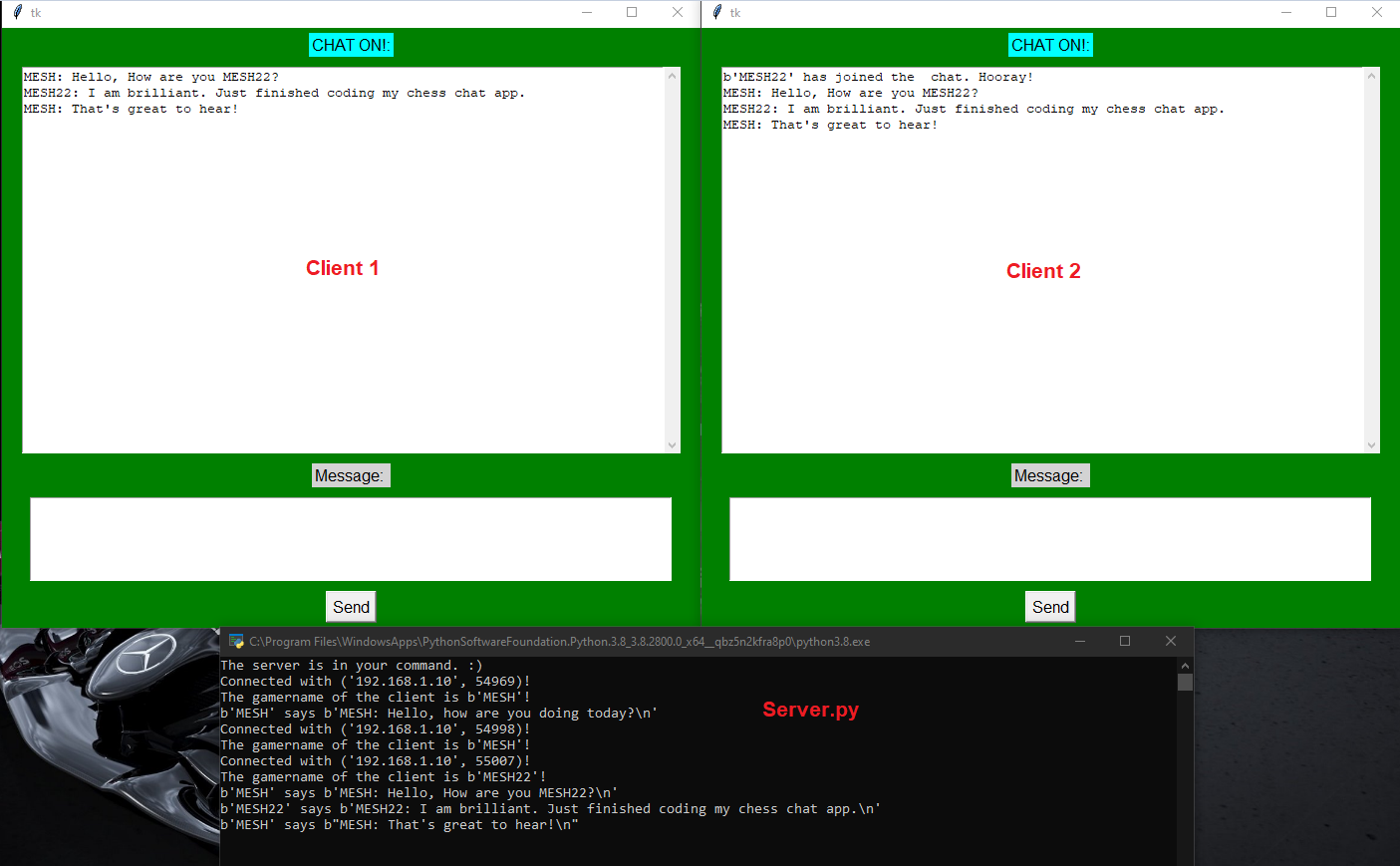
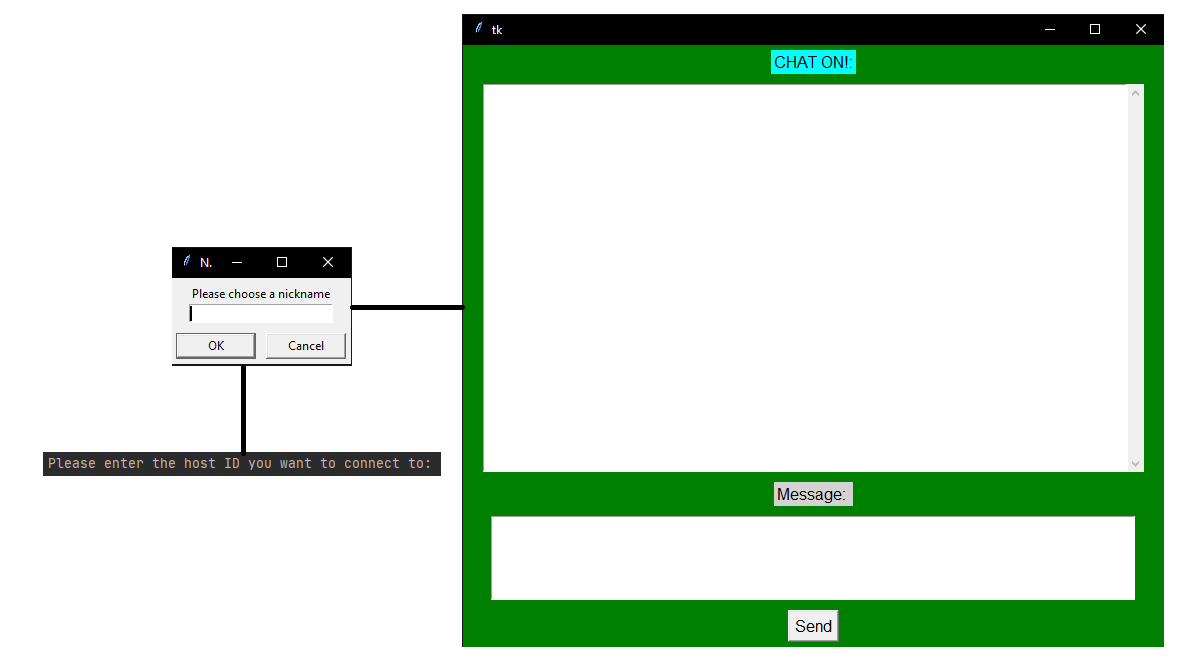
### piece.py

|  |
| --- |
| **def RenderImage():** Part of the code that will be responsible for adjusting the image according to the dimension specified by the engine.py file. |
| Import pygame as p  BOX\_SIZE = 500//8 “The size of the single box will be achieved by dividing the HEIGHT/WIDTH by dimension i.e. 100” rendered = {} “Dictionary where the rendered images will be saved at”  def RenderImage():  pieces = [‘wp’, ‘wR’, ‘wN’, ‘wB’, ‘wK’, ‘wQ’, ‘bp’, ‘bR’, ‘bN’, ‘bB’, ‘bK’, ‘bQ’]  “Piece saved in the form of strings.”  for piece in pieces:  rendered[piece] = p.transform.scale(p.image.load(“resources/” + piece + “.png”), (BOX\_SIZE, BOX\_SIZE))  “””The for loop will go through the pictures in the resources library which fetch and  render the image i.e. p.transform.scale which are the images specified by the BOX\_SIZE.”””  **THIS IS THE END OF THE piece.py FILE** |
| In summary, the job of the piece.py file is to import images from the resource folder and adjust them. |

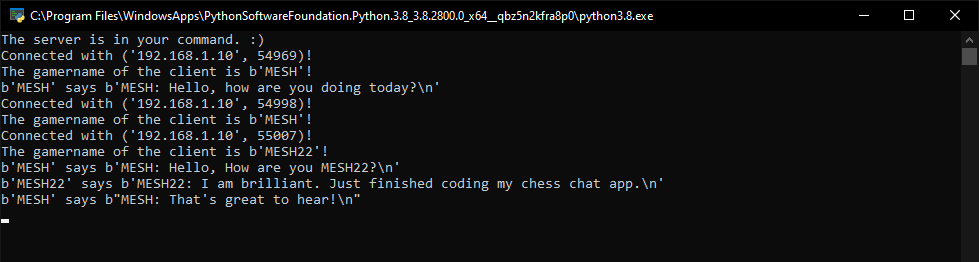
**THIS IS THE END OF MULTIPLAYER MODE**

## Chat-app Navigation Overview

The diagram below manifests how all the forms of the chat application are linked to each other.



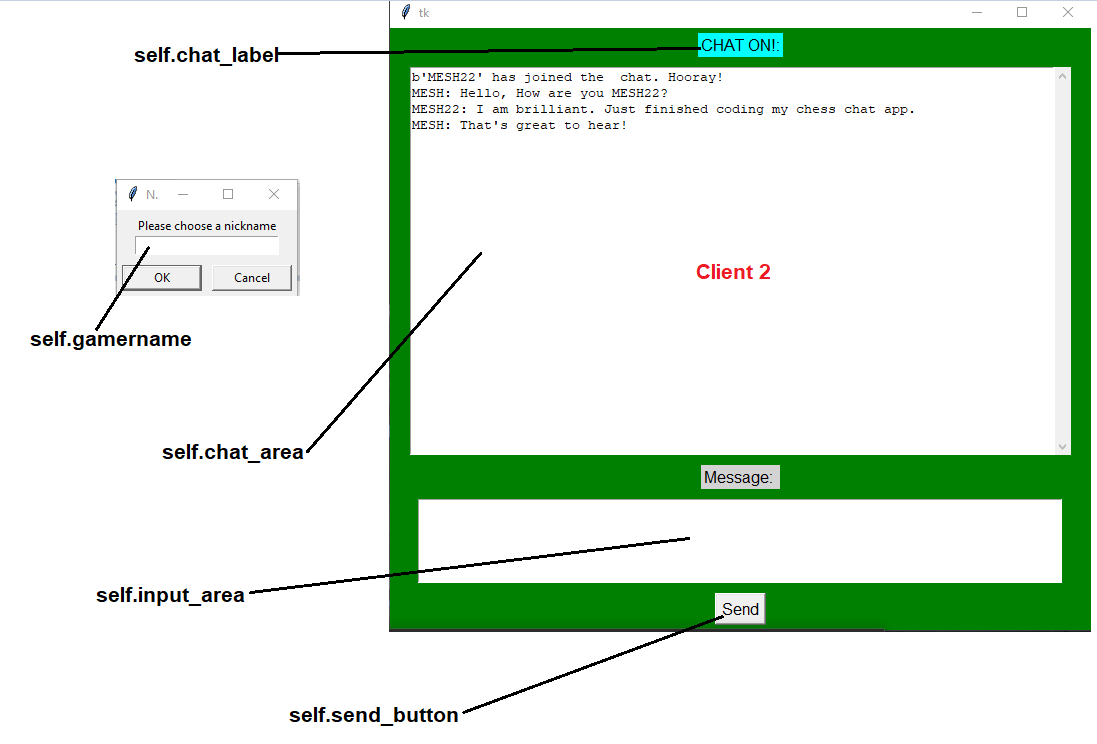
### server.py code



import threading # Library which helps the program to run multiple threads.  
import socket # Library to make the connection between the two players.  
  
PORT = 55555 # By default, the standard port to be used is set to 55555.  
HOST = socket.gethostbyname(socket.gethostname()) # Function that automatically retrieves the  
# host address if they click the server.py file.  
  
SERVER = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
#socket.socket specifies the address family and socket type. AF\_INET is the internet family address for IPv4.  
#socket.SOCK\_STREAM is the socket type for TCP, the protocol that would be used to transmit the message between 2 users  
  
SERVER.bind((HOST, PORT)) # This method will bind the HOST and PORT to the socket instance when called.  
# Held in a tuple, hence why the enclosed brackets.  
SERVER.listen() # Makes the socket listen to the specified port.  
  
clients = [] # List that will hold the  
gamernames = []  
  
def transmit(message): # Function that would transmit one message to the other client.  
  
 for client in clients: # For every client in clients, this function will send the message.  
 client.send(message) # Message send.

def operate(CLIENT): #Error handling function.  
 valid = True # Flag set  
 while valid:  
 try:  
 message = CLIENT.recv(1024) # Receive message from client, limit of 1024 bytes.  
 print(f"{gamernames[clients.index(CLIENT)]} says {message}") # To be printed on the server.  
 transmit(message) # If satisfies the condition, the message will be transmitted.  
  
 except: # Error handling part of the operate method.  
 index = clients.index(CLIENT) # Gets the index of the client.  
 clients.remove(CLIENT) # Removes the client from the list.  
 CLIENT.close() # Closing the connection.  
 gamername = gamernames[index] # Gets the index of the gamername.  
 gamernames.remove(gamername) #Removes the gamername if client disconnects.  
 break  
  
def receive(): #Function that would accept new connections in the chat and will run in the main thread.  
  
 valid = True  
 while valid:  
 CLIENT, address = SERVER.accept() # Method that will return the clients address.  
 print(f"Connected with {str(address)}!") # Message that would be displayed on the server.  
 CLIENT.send("NAME".encode('utf-8')) # The keyword "NAME" will be sent to the client for a gamername to request.  
 # If the name matches the selection of the client, they will be asked for the gamername.  
  
 gamername = CLIENT.recv(1024) # Method to receive the gamername.  
 gamernames.append(gamername) # Appended to the gamername list.  
 clients.append(CLIENT) #the client data would also be appended on the clients list.  
 print(f"The gamername of the client is {gamername}!") # The gamername of the client will be printed on the server.  
 transmit(f"{gamername} has joined the chat. Hooray!\n".encode('utf-8')) # This message would be sent to all the clients in the server.  
 CLIENT.send("Connected to the server".encode('utf-8')) # If connection successful, this message will be displayed.  
 thread = threading.Thread(target=operate, args=(CLIENT,)) # Used to execute multiple function. The ',' is included in args because it still needs to be treated as a tuple.  
 thread.start() # This will initiate the threading process.  
  
print("The server is in your command. :)")  
receive() # Receive run the entire server program.

### client.py code



import tkinter  
import threading  
import socket  
import tkinter.scrolledtext  
from tkinter import simpledialog  
# Libraries imported to handle the GUI part of the game.  
  
HOST = input("Please enter the host ID you want to connect to: ") # Input where the client has to manually type the host ID to connect.  
PORT = 55555 # Default host ID set to 55555.  
  
class CLIENT:  
  
 def \_\_init\_\_(self,HOST,PORT): # HOST and PORT being set as parameters.  
 self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # Establishes the connection with the host.

self.sock.connect((HOST,PORT)) # Initiates the connection with the host.  
  
 message = tkinter.Tk() # Window where the files would be printed.  
 message.withdraw()  
  
 self.gamername = simpledialog.askstring("Nickname", "Please choose a nickname", parent=message)  
  
 self.gui\_done = False # Flag set because after the GUI loads it will turn back to true.  
 self.running = True # Variable that if True will infinitely run the chat.  
  
 gui\_thread = threading.Thread(target=self.gui\_loop)  
 receive\_thread = threading.Thread(target=self.receive)  
  
 gui\_thread.start()  
 receive\_thread.start()

def gui\_loop(self): #GUI loop for presenting messages.  
 self.win = tkinter.Tk()  
 self.win.configure(bg="green")  
 self.chat\_label = tkinter.Label(self.win, text="CHAT ON!:", bg="cyan")  
 self.chat\_label.config(font=("Arial", 12))  
 self.chat\_label.pack(padx=20, pady=5)  
 self.text\_area = tkinter.scrolledtext.ScrolledText(self.win)  
 self.text\_area.pack(padx=20, pady=5)  
 self.text\_area.config(state='disabled')  
 self.message\_label = tkinter.Label(self.win, text="Message: ", bg = "lightgray")  
 self.message\_label.config(font=("Arial, 12"))  
 self.message\_label.pack(padx=20, pady=5)  
 self.input\_area = tkinter.Text(self.win, height=5)  
 self.input\_area.pack(padx=20, pady=5)  
  
 self.send\_button = tkinter.Button(self.win, text="Send", command=self.write)  
 self.send\_button.config(font=("Arial", 12))  
 self.send\_button.pack(padx=20, pady=5)  
  
 self.gui\_done = True  
 self.win.protocol("WM\_DELETE\_WINDOW", self.stop)  
 self.win.mainloop()  
  
 def write(self):  
 message = f"{self.gamername}: {self.input\_area.get('1.0', 'end')}"  
 self.sock.send(message.encode('utf-8'))  
 self.input\_area.delete('1.0', 'end')  
  
 def stop(self):  
 self.running = False  
 self.win.destroy()  
 self.sock.close()  
 exit(0)

def receive(self): # Used to receive messages and then decode.  
 while self.running:  
 try:  
 message = self.sock.recv(1024).decode('utf-8')  
 if message == 'NAME':  
 self.sock.send(self.gamername.encode("'utf-8"))  
 else:  
 if self.gui\_done: # If self.gui\_done is true, then the following code will take place.  
 self.text\_area.config(state='normal')  
 self.text\_area.insert('end', message)  
 self.text\_area.yview('end')  
 self.text\_area.config(state='disabled')  
 except ConnectionAbortedError: # If something is wrong with the connection, the program will stop looping.  
 break  
 except: # If anything else happens, it will too stop running the loop and will display the error message.  
 print("Error. Please close the application and try again")  
 self.sock.close()  
 break  
  
client = CLIENT(HOST,PORT) # Calls the program with arguments defined in line 8 & 9.

# Testing

**Objectives:**

* **Produce evidence of thorough testing of the solution.**
* **Cross-reference testing with the objectives.**
* **Understand the types of testing that can be used for the testing section**

## Input & output Testing

This table is the overview of the basic input and output tests conducted on the main game file in order to make sure that the expected outcome of every action is the actual outcome when using the system. Pale red rows indicate tests that were not featured/completed in the main program. Pale cyan shows the main headers.

### Chess board print.

The testing for this part of the code will in my google Drive folder for which I will paste the link.

The board printing is the same for single player as well as multiplayer mode as they inherit the same program properties: [**https://drive.google.com/drive/folders/1QmkHAL2gTq27qHfgo5p0aWXiLV-5FETm?usp=sharing**](https://drive.google.com/drive/folders/1QmkHAL2gTq27qHfgo5p0aWXiLV-5FETm?usp=sharing)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome (with evidence page number)** | **Actual Outcome** | **Improvement comments** |
|  | Printing the board with the different patterns. In this case white and dark cyan were selected. | def drawBoard() | Chequered board with white and dark cyan boxes. | As expected. | None required. |
|  | Checking for any errors when pygame is printing the board. | None. | Hello from the pygame community. | As expected. | None required. |
|  | Checking for any errors when the .png piece image files are being imported from the resources folder. | **resources file along with python code which goes through each individual image.** | Hello from the pygame community. | As expected. | None required. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome (with evidence page number)** | **Actual Outcome** | **Improvement comments** |
|  | Checking for any errors when the pictures are being rendered according to the image dimension specified by the HEIGHT and WIDTH | All the .png names saved in a list which automatically goes through a for loop for the number of piece images in the list. | Hello from the pygame community.  No errors. | As expected. | None required. |
|  | Printing the rendered piece image in the 8x8 chess board pieces seamlessly. | def drawGameState() | Hello from the pygame community.  No errors. | As expected. | None required. |
|  | Print the chess board along with the pieces rendered. | In the piece.py file. | Prints the chess board with the chess pieces rendered simultaneously in the chess board. | As expected. | None required. |
|  | Print the highlighted piece when clicked on. | def highlightSquares()  on line #163. | The user selected square is highlighted with a blue colour.  The available moves are highlighted in a yellow squares. | As expected. | None required. |

### Move turn printing (singleplayer mode).

The testing for this part of the code will in my google Drive folder for which I will paste the link.

The link is: [**https://drive.google.com/drive/folders/126U8BQARPcoCCwr1Vmv9bPtyd5ZS0Ymb?usp=sharing**](https://drive.google.com/drive/folders/126U8BQARPcoCCwr1Vmv9bPtyd5ZS0Ymb?usp=sharing)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome (with evidence page number)** | **Actual Outcome** | **Improvement comments** |
|  | Checking if the board will print whose turn it is when singleplayerENGINE.py is executed. | Automatically loads | Prints the whose turn tab next to the mane pane. | As expected. | None required |
|  | Checking if the board highlights your turn (the user which is the white piece) when starting the game. | Automatically loads | Highlights the **your turn** rectangle signifying the user must play. | As expected. | None required. |
|  | Checking if the board highlights the computer’s turn after the user has played his turn. | Automatically loads. | Highlights the computer’s turn signifying the user must wait for the computer to finish its move. | As expected. | None required. |
|  | Any errors when highlighting whose turn it is. | Playing the game. | No errors | As expected. | None required. |

### Move turn printing (multiplayer mode)

The testing for this part of the code will in my google Drive folder for which I will paste the link.

The link is: <https://drive.google.com/drive/folders/1-y00tQw4TcjNYWT4QmG_XPAE_AQS3wAK?usp=sharing>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome (with evidence page number)** | **Actual Outcome** | **Improvement comments** |
|  | Checking if the board will print whose turn it is when multiplayerENGINE.py is executed. | Automatically loads | Prints the whose turn tab next to the mane pane. | As expected. | None required |
|  | Checking if the board highlights your turn (user 1 which is white) when starting the game. | Automatically loads | Highlights the **Player 1** rectangle signifying the user must play. | As expected. | None required. |
|  | Checking if the board highlights when the second player is playing i.e. player/user 2. | Automatically loads. | Highlights the **Player 2** rectangle signifying the second user must play. | As expected. | None required. |
|  | Any errors when highlighting whose turn it is. | Playing the game. | No errors | As expected. | None required. |

### Piece movements

As usual, I have recorded the clips of testing each of the piece movements in my google Drive which will give you access. The way the chess piece moves is the same for single player as well as multiplayer as they inherit the same program characteristics.

The link is:

[**https://drive.google.com/drive/folders/1WdQ\_4NN4xWvGSuRMH1RXsX8zKA1v3nW5?usp=sharing**](https://drive.google.com/drive/folders/1WdQ_4NN4xWvGSuRMH1RXsX8zKA1v3nW5?usp=sharing)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** |
|  | Pawn Moves. | Click on the piece and then the place to where it is supposed to go. (1 tile) | Piece moves 1 tile which complies with the chess rules. | As expected. | None. |
|  | Pawn moves 2 places if starts from the first position. | Click on the piece and then the place to where the piece is supposed to go. (2 tiles) | Piece moves 2 tiles which complies with the chess rules. | As expected. | None |
|  | Rook moves in a straight line. | Click on the rook piece and then place it on the location it is supposed also considering the fact that the position is in a straight line. | Rook moves in a straight line fashion which complies with the chess rules. | As expected. | None |
|  | Knight moves in a L pattern. | Click on the Knight piece then place it on the location it was supposed to be also considering the fact that the position was in a L manner. | Knight moves in a L formation manner which complies with the chess rules. | As expected. | None |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** |
|  | Bishop moves in a diagonal manner. | Click on the piece then place it on the location it was supposed to be also considering the fact the position was diagonal | Bishop piece moves diagonally which complies with the chess rules. | As expected. | None. |
|  | King moves in any pattern 1 tile at a time. | Click on the King piece then place it on the location it was supposed to be also considering the fact that the position was in any direction 1 tile at a time. | King moves in any direction but on a single tile at a time which complies with the chess rules. | As expected. | None |
|  | Queen moves in Rook and Bishop manner. | Click on the queen then place it on the location it was supposed to be considering the position was either diagonal or straight. | Queen moves diagonally and straight. | As expected. | None |

### Taking out other pieces.

As usual, the video evidence for these tests will be the link from my google drive. In the video, I will playing the game of chess till the program has ended with a stalemate or checkmate. The video should automatically show the taking out other piece capability rather than individually testing each piece costing a lot of time and effort. Taking out other pieces work the same for single player as well as multiplayer game modes.

The link is :

[**https://drive.google.com/drive/folders/11L2RRqleGF5Gd82w3svJ1kskDVvdwFzj?usp=sharing**](https://drive.google.com/drive/folders/11L2RRqleGF5Gd82w3svJ1kskDVvdwFzj?usp=sharing)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** |
|  | Pawn takes out other pieces. | Place the pawn to the enemy piece and take it out. | Successful taking out of the enemy piece with the chess notation displayed. | As expected. | None. |
|  | Rook takes out the other pieces. | Place the pawn on the enemy piece to take it out. | Successful taking out of the enemy piece with the chess notation displayed. | As expected. | None. |
|  | Pawn takes out other pieces. | Place the pawn to the enemy piece and take it out. | Successful taking out of the enemy piece with the chess notation displayed. | As expected. | None. |
|  | Rook takes out the other pieces. | Place the rook on the enemy piece to take it out. | Successful taking out of the enemy piece with the chess notation displayed. | As expected. | None. |
|  | Knight takes out the other pieces. | Place the Knight on the enemy piece to take it out. | Successful taking out of the enemy piece with the chess notation displayed. | As expected. | None |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** |
|  | Bishop takes out the enemy piece. | Place the Bishop on the enemy piece to take it out. | Successful taking out of the enemy piece with the chess notation displayed. | As expected. | None |
|  | Queen takes out the enemy piece. | Place the Queen on the enemy piece to take it out. | Successful taking out of the enemy piece with the chess notation displayed. | As expected. | None |
|  | King taking out the enemy piece. | Place the king on the enemy piece to take it out. | Successful taking out of the enemy piece with the chess notation displayed. | As expected. | None |

### Main window GUI

**The main GUI windows has been completed and is working seamlessly which meets the objectives.**

The link for main window GUI pane testing is :

[**https://drive.google.com/drive/folders/15B\_u8\_eEf8LQNL9Qs7bevun65gttKwSE?usp=sharing**](https://drive.google.com/drive/folders/15B_u8_eEf8LQNL9Qs7bevun65gttKwSE?usp=sharing)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** |
|  | Have a main window which displays the different type of playing modes. | None | Displays the different types of playing modes. | As expected. | None. |
|  | When clicked on the single player button, the program takes the user to play against the AI. | Press the single player mode button. | Initiate the single player.py file in the modes which prints the board along with the ai.py file. | As expected. | Opens a separate pane when a user plays a move. I could’ve debugged and found the error but because of time constraints couldn’t. |
|  | When clicked on the multiplayer button, a window is opened with 2 users being able to play one after the other. | Press the multiplayer mode button. | Initiate the multiplayer.py in the modes file which prints the board which passes the control of the piece from Player 1 to Player 2. | Not as expected. | Didn’t complete. |
|  | When clicked on the online button, a window is opened with the user needing to type the IP address of the host computer or hosting the game. | Press the online mode button. | Initiate the online.py in the modes file which prints the board which asks the user whether he wants to host a game or join a game. | Not as expected. | Didn’t finish producing the online game mode. |
|  | When clicked on the exit button, it destroys pygame which ends the game. | Press the exit or close button. | Ends the game. | As expected. | None.. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** |
|  | When clicked on the shortcuts button, it opens a pane showing the different shortcuts available to the user. | Click on the shortcut commands button. | Shows the different shortcuts available to the user. | As expected. | None |
|  | When clicked on the how to play chess button, it opens a link from chess.com showing how to play chess. | Click on the how to play chess button. | Opens the link successfully. | As expected. | None. |

### Castling

I have been successfully able to implement the castling method to which I will share the code to my google drive where there is video evidence.

The link is:

[**https://drive.google.com/drive/folders/187FmPCCVZYg3xV0JPZMO65dwQwfrlvsM?usp=sharing**](https://drive.google.com/drive/folders/187FmPCCVZYg3xV0JPZMO65dwQwfrlvsM?usp=sharing)

### Pawn exchange

I also have been successfully able to implement the pawn exchange method where once the pawn enters the first row of the opponent’s area, it will automatically exchange the pawn for a queen.

The link is:

[**https://drive.google.com/drive/folders/187FmPCCVZYg3xV0JPZMO65dwQwfrlvsM?usp=sharing**](https://drive.google.com/drive/folders/187FmPCCVZYg3xV0JPZMO65dwQwfrlvsM?usp=sharing)

### Single player mode

**This is the only mode in my mode which functions properly.**

**I have recorded a video demonstrating the program’s AI capabilities. I have used the application OBS studio to record the file.**

Firstly, we will check whether if the **stalemate** AI function works. I will paste the link to the video which I have tested below.

[**https://drive.google.com/file/d/1e-DWZqzwanIk0x-16LhuuoCWfLqECXiX/view?usp=sharing**](https://drive.google.com/file/d/1e-DWZqzwanIk0x-16LhuuoCWfLqECXiX/view?usp=sharing)

Secondly, we will check whether if the **checkmate** AI function works. I will paste the link to the video which I have tested the function.

[**https://drive.google.com/file/d/1UMxef2i5-Gz8vnK6Oz\_Hp4Mmo6l-atrK/view?usp=sharing**](https://drive.google.com/file/d/1UMxef2i5-Gz8vnK6Oz_Hp4Mmo6l-atrK/view?usp=sharing)

### Checkmate and stalemate display functions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** | **Page number of the picture of testing** |
|  | Display the checkmate function when the user/opponent has no way to avoid the check. | None | Displays the checkmate message along with the piece colour. | As expected.. | None | Tested in the AI checkmate function video. (above) |
|  | Display the checkmate function when the user/opponent has no way to avoid the check. | None | Displays the checkmate message along with the piece colour. | As expected.. | None | Tested in the AI stalemate function video. (above) |

### Multiplayer mode

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** | **Page number of the picture of testing** |
|  | Multiplayer mode where the control of the pieces gets passed to the 2 players sequentially after each other’s turn. | Multiplayer button on the main window | Displays the chess board which controls starts with player 1 who controls white, then black afterwards. | As expected. | Multiplayer mode where the control of the pieces gets passed to the 2 players sequentially after each other’s turn. | Tested on the **Main window GUI section.** |

### Online mode

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** |
|  | Online mode where the control of the pieces gets passed to the 2 players sequentially after each other’s turn. They are first greeted whether if the player wants to host the game or want to join the host. | Online button on the main window | Displays the message to type the host IP address/or host the game. Controls gets passed from one partner to another. | Not as expected.. | Didn’t complete. |

### Chat application

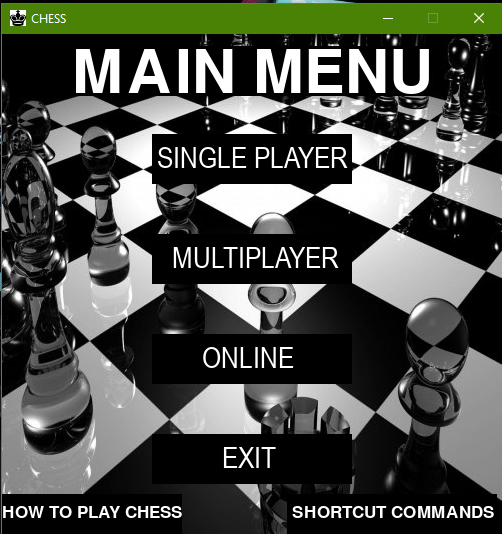
As usual, I will paste the testing of my chat application in the Google Drive which will contain evidence of successful transmission.

This link is:

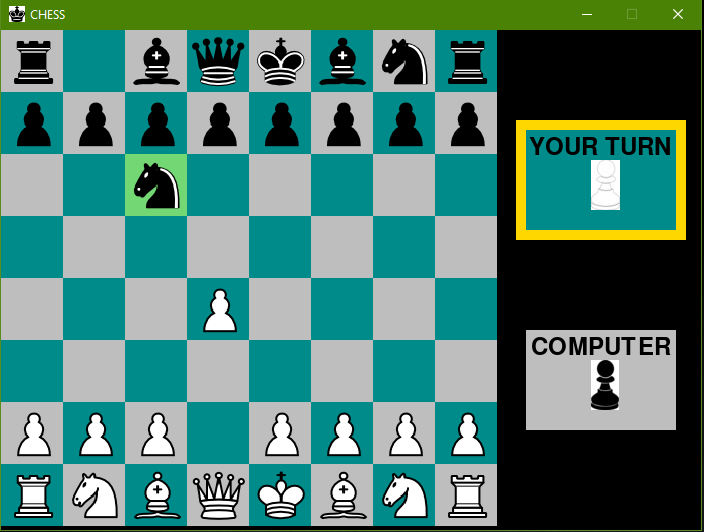
[**https://drive.google.com/drive/folders/1Bm33xKo8OL\_AkHZv5xchf6SdVk\_g566A?usp=sharing**](https://drive.google.com/drive/folders/1Bm33xKo8OL_AkHZv5xchf6SdVk_g566A?usp=sharing)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test #** | **Description** | **Input** | **Expected Outcome** | **Actual Outcome** | **Comments on testing** |
|  | Asks for the host ID if the client has executed the client.py file. If the main user is hosting, he has to run the server.py file. | **Input details only required for client;**   * Host ID * Gamername | Successful transmission of message. | As expected. | It would have been more convenient if the chat application had been integrated to the main file i.e. engine.py. |

## Testing Outcome Screenshots











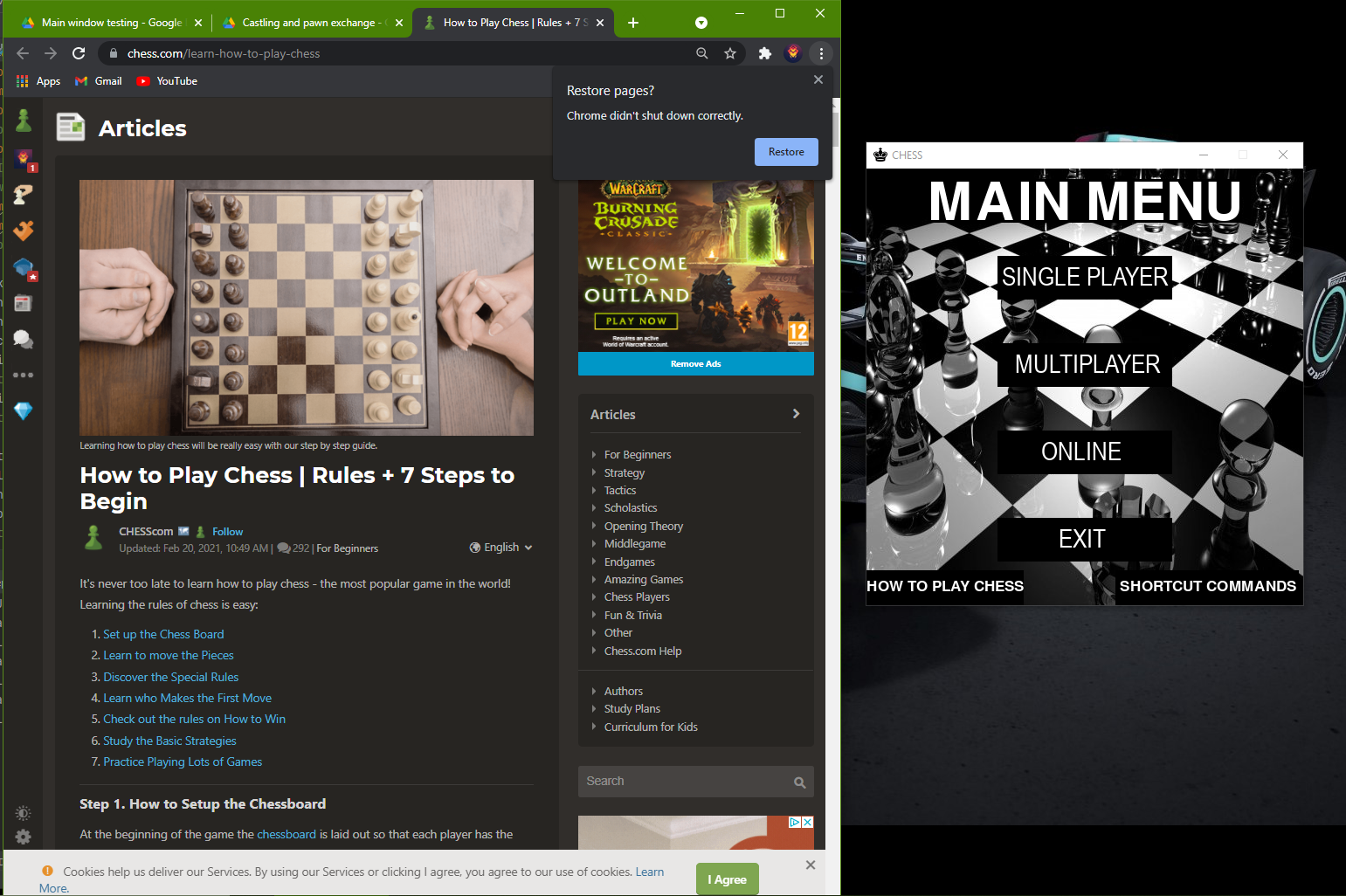


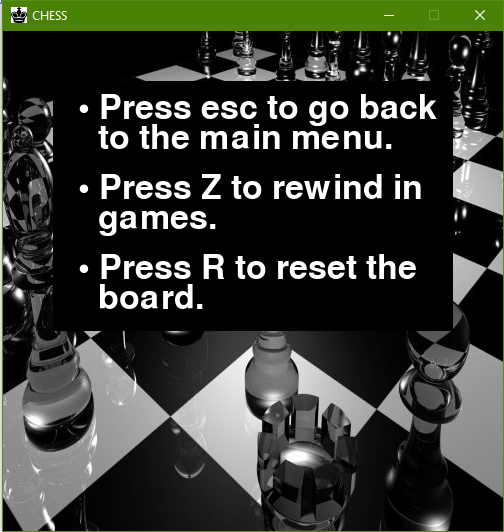












# Evaluation

**Objectives**

* **Provide evidence of how well each of the requirements is met**
* **Be able to discuss detailed improvements**
* **Obtain independent feedback**
* **Evaluate the independent feedback**

## Comparison of Project performance against objectives.

|  |  |  |  |
| --- | --- | --- | --- |
| **Objective not met** | **Objective partially met** | **Objective met** | **Objective exceeded** |

|  |  |
| --- | --- |
| **Original objective** | **Completed system** |
| The game open with the main menu displays a separate button which contains tools to customise the board with a timer and other features. | The game doesn’t have a main GUI program tab where the user can select the button and customise the game according to their likeness. |
| Having a main menu where shows the different modes the user can play chess with. | A main GUI manager.py has been completed. It has two functional game modes which are the single player and multiplayer game modes. The online game mode hasn’t been completed, however, the chess application from which the user would’ve communicated with their peers is complete. |
| Single player (AI) where the user can play with adjustable levels of difficulty ranging from easy to hard. | The single player (AI) has been implemented using the negamax algorithm with alpha-beta pruning which reaches a recursion depth of 3 to find the most optimal solution to take out the user. It calculates this as points are saved in the list for which the AI targets for more points. |
| A multiplayer mode where 2 users can play on a single computer. | I have been able to successfully complete the multiplayer part of the program and it opens and functions as expected by the objectives. |
| An online mode where the user can play with their friends over the internet. | Although the chat application had been developed using the socket library, I wasn’t able to find the most optimal solution to transfer proper moves via the internet. |
| Time control where this mode was meant to be exclusively available for modes such as **multiplayer** and **online.** | This was due to the fact that I wasn’t able to finish the multiplayer and online mode. |
| Printing the board with 8\*8 square. | The engine.py file was successfully able to import images rendered by the piece.py file. |
| The piece plan will be saved in a list where accordingly pygame will be able to print the images imported by the piece.py library. | The piece names was saved in a list named self.board in the GameState library in the board.py file. |

|  |  |
| --- | --- |
| **Original objective** | **Completed system** |
| The program will find the chess piece and will accordingly manipulate the images specified by the dimension 500x500. | The pieces were rendered accordingly to the box where it was placed properly. |
| The program will fetch the .jpg/.png files from the resources folder where individual pieces will be automatically printed on the board. | The pieces were imported from a folder named resource which was imported by a for loop. |
| The board will be printed. | Successful printing of the chess board. |
| **Single player mode** button from the main menu tab. | Wasn’t able to complete due to time constraints. |
| The user will get the choice to select the piece colour. | The customisations button in the main GUI wasn’t complete. |
| Castling move. | Working as expected. |
| Pawn exchange move. | Exchanges queen. Doesn’t return any other piece such as knight or rook. |
| White player plays/starts first. | Working as expected. |
| The user can press the Z button to undo a move. | Working as expected. |
| The user can press the escape button where the user will get the option to select to return back to the main menu. | The main menu GUI wasn’t complete. |
| Any of the multiplayer game/2 player mode working. | None. |
| Printing the moves on a separate GUI window in chess notation. | The moves of the players are saved in a self.moveRegister in class GameState in the board.py library. The program was able to beautifully print the chess notation of the game moves. |
| Piece highlighting. | The engine highlights the piece. Furthermore, it also shows the location the piece can move towards. |
| Demonstrates vital game messages such as stalemate and checkmate. | Working as expected. |
| Program quits/terminates as expected. | Working as expected. |
| Any of online game functions working. | None. |
| Be able to make a successful chat application which transmits messages to other users provided it encrypts it. Only for the online mode. | The program sends messages as expected but doesn’t encrypt any of it. |
| Integrate the chat application with the main python file which is engine.py. | Not able to finish it due to time constraints. |
| A subroutine which prints out whose turn it is in both multiplayer and singleplayer mode. | Works as expected. |

## Potential for Future Developments

Currently, I have not been able to finish many of the features of the chess game that I had meant to finish in the objectives. Therefore, the only modes in the game that are currently working is the single player/ai and multiplayer. This was due to the challenges bought by the pandemic which made it difficult to balance out revision with the coursework and due to the uncertain environment of grades awarding. Therefore, if I had the time to finish my coursework, I would integrate all the different objectives that I was meant to finish in one single application which didn’t any other execution files.

Another development which could be implemented is in the inclusion of different mode support i.e. Online. Due to the complex nature of the minimax algorithm with alpha-beta pruning, I was watching videos to build an AI which was sufficient to challenge the users on their skills. Overall, the AI is pretty competitive with move generation that is quite challenging.

## Analysis of user Feedback

Again, I will shed light that due to time constraints I wasn’t able to make a questionnaire to my main clients. However, I have used my brother to give a objective assessment of how he rated the system rating stuff such as the UI, whether if he liked the way the AI played with him, and other important design aspects. I have summarised his thoughts below.

|  |  |  |
| --- | --- | --- |
| **Client name: Joshua Dias** | |  |
| **Features that are desirable** | **Things that need changes** | |
| * Like the colour amalgamation of the chess board. * The animation of the piece was smooth. | * Would have been better if more features were available such as game customisation e.g. timer. * Having a better user interface and the buttons being animated or showing what that mode is about. * Having some element of sound when the pieces were being move similar to the one in the game Chess Lvl. 100 from the windows store. | |

|  |
| --- |
| **Overall, for the game. My brother asked me to rate the game out of 10, of which he rated the game 6/10.**  **However, this data is biased as it lacks population validity and more research was required to gain insights into my client population.** |