Title of report

Table of Contents:

Analysis

Problem Description:

Over the past year, the popularity of chess worldwide has surged with the release of a new TV series (The Queen’s Gambit), and popular online personalities streaming them playing the game to thousands of people. This has shown many why chess is a fun, interesting yet complex game that can be easy to understand with basic knowledge, but extremely difficult to master. This is the issue I am looking to help solve. This project will look to encourage chess players of a lower skill level improve and understand chess on a more intricate level by having them play against an AI (a player that is controlled by the computer) that will tell them what moves are the best to make in a position, while also playing the best moves itself. These lower skilled chess players will be my stakeholders as they can give critical feedback on what they want to be improved in the program to help them learn chess more effectively. I will be using Python to create my program as it can be used for procedural and object-oriented programming while also being an interpreted language meaning it can be easily run on any computer. Also, I am most familiar with Python and its libraries so this will help me make the program as optimised as possible.

Stakeholders:

My stakeholders will be low-level chess players who are looking to improve their play as they will be able to provide accurate and useful feedback on what would help them improve faster and in a more effective manner. I will also be asking for feedback from a higher rated chess player that can help inform the development of the AI. For example, if the higher-level player finds that the AI is making strange decisions in the end game that cause it to give away a strong position for no reason, that feedback can be used to adjust the evaluation function and fix the issue that was occurring. I can also ask the high-level player how they think the lower-level players are improving and what they think would help improve the amateur players at a faster rate.

The lower-level chess players will be using the program to improve their chess skills without the need to pay for online courses, have a constant internet connection or always have someone to play against. My program will be perfect for this as it will be free for the users and easily understood by anyone that wants to use it. The program will also have the option for the user to play against an AI or against another person to give them flexibility between practicing by themselves or showing their skills to another person e.g., the higher-level stakeholder so any progress made by the lower-level player can be tracked more easily.

If the program reaches a point where the higher-level players are struggling to beat the AI, the program can now reach a bigger userbase to help them improve their abilities as well. This improvement in the AI will allow more people to use the program and as a result, give more feedback that will help in the development of the AI: this cycle of development could go indefinitely as higher and higher rated chess players give their input for what they would like the algorithm to do. This will also help give more distinct levels to how well the AI plays (beginner, intermediate, etc.) and allow for more incremental jumps between levels to help users improve faster.

Why a Computational Solution:

Board games are a great example of weighing up logic with risk. While a high-level chess player can calculate many moves in advance while looking at lots of permutations of the position, only a computer can really look for all the possible moves and calculate the probability of those moves resulting in a win, draw or loss. Looking at all the possible moves is the only way to be certain the opposing player has a smaller chance of winning than you do, this is where computers are much more effective than humans. While humans can intuitively discount many moves that they know will produce a bad position, a computer cannot do this. However, it can compute a larger number of moves than a human can. This could lead to the computer finding moves that the human missed as they discount moves that seem to make the position worse, but in the long term, it gives the computer an advantage.

Throughout this project, I will have to use techniques such as abstraction and iteration. Abstraction will be used by the AI to ignore any illegal moves that cannot be made during the game. This will allow for faster processing of the optimal move and, as a result, improve the experience for the user. Iteration will be a constant while the program is running. The program will constantly be looking for a move to be played by the user and waiting for the correct time to execute the next piece of code. The check for a move will have to happen extremely often as using time unnecessarily will be a very bed experience for the user and could result in the user looking the game (if they are low on time at the end).

Research:

Programming Language:

Python – High level language that can be used both procedurally and in an object-oriented manor. It contains many easily accessible libraries to make coding the program easier. The code is run in an interpreted manor.

Java – High level language that can only be used in an object-oriented manor. Also contains lots of libraries that can be used. The code is run both with interpretation and compilation.

C family of languages – High level languages that can be used in both a procedural and object-oriented manor. The code is compiled before it can run

I will use Python to develop my program as I will be able to use both its features of procedural and object-oriented programming to complete my program while making use of the many libraries that can be accessed. The interpretation of Python allows more frequent testing of the program and any minor changes that need to be made can be done so quickly and tested immediately without waiting for the program to compile. I am also most familiar with Python and its techniques meaning I will be able to make the program more efficient.

Development Cycle:

Waterfall – Each stage is implemented one by one and the output of one stage is the unput into the next stage. The previous stages can be re-visited but the steps must be followed in order.

Agile – Non-linear model where each stage can be visited at different points depending on what is needed in the program. Allows for various aspects of the program to be developed at different rates.

Extreme – Same as Agile but the time between each release of the program is much shorter to allow frequent feedback from the user.

I will use the Agile methodology of development as this will allow me to create each section of the program as they are needed in the overall solution and revisit any areas that become a problem later in the development of the program.

Originally, I was planning to use Supervised Machine Learning to help the program choose the best move possible. This however started to look less and less feasible as I could not find a suitable database of PGNs online that I could use as training data. After this, I considered using the OpenAi Chess-Gym library that has been developed for Reinforcement Learning. This was a better idea as it meant I did not have to find any training data, but it would require me to learn the workings of this library and teach the AI the rules of chess before moving forward. Therefore, I have concluded that not using machine learning would be beneficial as I already know how to make “ordinary” programs in Python more efficient and if I were to use the new library, the program would become messy and less maintainable. As a result, the MiniMax algorithm would be the best and most straight forward way to implement an AI for the user to play against.

I found that many popular chess AIs use a mix of MiniMax (with alpha-beta pruning), a transposition table (to store any frequently visited nodes that are the same) and an extremely well optimized evaluation function to find the best moves in the shortest amount of time. They also have hundreds of volunteers that are willing to play games on their machines to help hone the evaluation function to make the engine even more precise in the moves that it makes.

The problem with these popular chess AIs is that they are often online and to access their full function, the user must pay to use it. My program will be able to be used offline and will not require the user to pay anything to use the full function of the program. This will increase the appeal of my program as it can be used anywhere by anyone at any time. Other chess programs that are free to use can only be used by the user through a command line interface. This is not great as it can be difficult for some people to use properly and could turn people away from learning chess if that is one of their first experiences trying to improve.

Features and limitations:

My program will allow the user to decide which colour piece they play the game with, to ensure they improve playing chess with both pieces; it can also add an additional challenge for the user. They will also only be able to move one piece per move by clicking on a piece and dragging it to the desired square. This will be done to ensure there is no way the program could be cheated and possibly overload the minimax algorithm as it tries to compute the best move. The user will not be able to make an illegal move as the program will block out these squares. These features will be the basis for the program and will allow other features, such as multiplayer capabilities, to be added in later versions.

The program will have a built-in evaluation function that will compute the best move for the AI to play and display to the user what it perceives the evaluation function is at, e.g., black has an advantage of 2.3. This will keep the user informed to which moves were good and which might have lost them the game. This is useful as my program is being made to help the user improve and allowing them to know which moves cause them to be in a less favorable position will aid in this endeavor.

This program will not give the best possible moves every time as chess is an extremely complex game with billions upon billions of possible games, it is almost impossible to “solve” chess so that you can find the best winning move in every position possible, as this is believed to take about 1090 years to do so on a conventional computer. The closest we can get to this, however, is by using machine learning and using games that have already been played to make the best estimate for the winning move. This requires a large database of games that have been played at a high level in a format that the computer can easily read and learn from. Even then, the computer will only become as good as the best humans in the world which, while very good, they don’t always find the best move to make in a position.

Requirements:

My program will be required to be interacted with through a GUI (Graphical User Interface) that is easily understood by the user and have a straightforward way to change the difficulty of the AI before the game starts to allow the user to play a game at their current skill level. The pieces should move round the board smoothly and there should be no lag noticed by the user when moving pieces around the board or when the computer is calculating its next move.

These are the minimum requirements to run Python:

x86 64-bit CPU (Intel / AMD architecture)

4 GB RAM

5 GB free disk space

Windows 7 or 10

Mac OS X 10.11 or higher, 64-bit

Linux: RHEL 6/7, 64-bit (almost all libraries also work in Ubuntu)

Success Criteria:

Each version of my program will implement a different aspect of the final solution. This will be done to allow me time to fully focus on one aspect of the program and have it working as well as possible before the next iteration is implemented. This development idea, along with the agile development structure, will help in the long term as I am less likely to need to go back and fix parts of the program later.

Version 1 overview: Version 1 will be used to create the basis of the program that everything else can be built on. I will implement a board on which the user can move both black and white pieces to any position on the board. There will be no rules or turns in this version as its purpose is to prepare for the main functionality to be added later when needed.

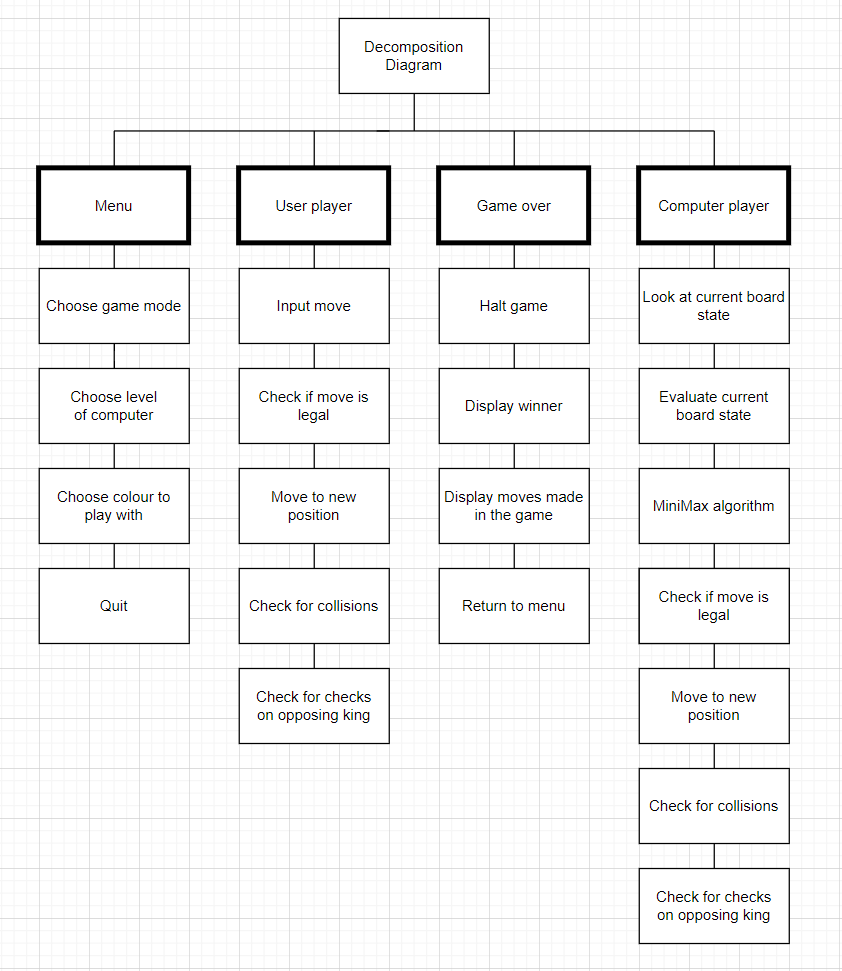
Version 2 overview: The rules of chess (such as castling and checkmate), as well as making the players take turns will be added at this stage. The goal for the end of version 2 is to have a fully functioning 2 player experience that can have the AI added easily in a later version. By incorporating these steps into the same version, I am making the development follow a logical path, building up both the complexity and functionality of the program, to the project’s conclusion.

Version 3 overview: This version will be where the bulk of the computational thinking will be added. The MiniMax, with alpha-beta pruning, will be implemented in conjunction with one another here to allow for quick testing as they are added to the program. The quick testing will help decrease the number of unexpected errors that occur as I move through the rest of the program adding more functionality.

Version 4 overview: Here, I will add more user-friendly features. This will include a start menu to decide how you would like to play the game (single player against the computer or multiplayer with another person). If the user decides to play against the computer, the difficulty will also be made very clear so the user can choose the level the computer plays at. They will be shown both the skill rating and an associated level (beginner, intermediate, etc.). This is so that they can continue to use the program even as they improve at chess and need a more difficult opponent to challenge them.

Design

Problem Decomposition:



The diagram above shows all the main components that will need to be added to the program. This has been done to simplify the development of the program and will allow me to check the functionality of each node and branch as they are completed. I can also show the stakeholders how far along the development process I am easily by using this diagram. I have split the problem into these components as they are the very basics of the program, and it will allow me to more easily see where I can reuse sections of code. For example, the program will have to move a piece to a new position, check for collisions and check for checks on the opposing king both for the human player and the computer player meaning I can use this section of code twice, making my program more efficient. I can reuse the code by making use of functions and passing through the parameters that the player wants. This will also have the effect of making the final solution more maintainable and future-proof as it will be easier to find sections of code that might need to be changed and adapted to meet any new criteria of the project.

The program will make use of classes to allow for easier maintainability and development. These classes will include things such as the current game state. This will allow for multiple instances to be opened very quickly, if necessary. I will use the external library pygame to create the GUI that the user will interact with. The other possible library to create the GUI with is the tkinter library. I am not that familiar with either pygame or tkinter, but I decided to use pygame as it is the one that I have a bit more experience with and will make the development slightly faster. Pygame is also built to create games within Python which is my current use case.

Graphical user interface

Description automatically generatedInterface:

As this program has a large focus on computer-human interaction, a GUI will be the main way the user will interact with the program, so it must be clean and easy to understand. One thing that will help this is making the buttons large enough for them to be easily pressed by the user. The drop-down menus will also ensure no invalid inputs are given and must be filtered out. This will simplify the code as less traps will have to be implemented for input validation. This main menu screen will have to appear every time the user opens the program or wants to quit their current game to start a new one. For this reason, it will need to be given its own function when implementing the code, so it is easily called again.

A picture containing text

Description automatically generatedIn game, the interface will look like a “classic” chess board. The pieces will be laid out correctly, ready for the game to start. The game will start, and the clock will begin to count down when the first piece is moved by the white player. By giving the user a GUI to interact with, it makes the game more intuitive to play as it feels more natural than trying to remember the chess abreviations and typing them into a static text box. Also, by making the timers appear above and below the board for the respective players, it becomes very clear to them how long they have left before they run out of time and loose the game. To make the experience better for the user, when they select a piece to move, the program will highlight the squares where it can legally move to. This will help the user visualise moves that can be made and how they could affect the position.

On the other hand, it could be benificial to my program’s users not to have a clock that is counting down as it will put more emphasis on choosing the correct move and not pressuring them with a timer will help in this aspect.

Key classes and variables:

***Insert variable/class/data structure table here w/ justification***

***Insert paragraph about validation (Valid moves, keyboard is not used w/ explanation)***

Diagram

Description automatically generatedAlgorithms:

MiniMax:

To the side is a flow chart showing the process of the MiniMax algorithm that I will use in my program. This will decide which move will be made by the computer player. It looks ahead to a certain depth and will return the best heuristic value for the computer and what move should be played next in order to reach that point.

Below is the pseudocode of the MiniMax with alpha-beta pruning that gives a more in-depth view onto the algorithm that I will use in my final solution

**function** alphabeta (node, depth, α, β, maximizingPlayer)

**if** depth = 0 **or** node is a terminal node **then**

**return** the heuristic value of node

**if** maximizingPlayer **then**

value := −∞

**for each** child of node **do**

value := max(value, alphabeta(child, depth − 1, α, β, FALSE))

**if** value ≥ β **then**

**break** *(\* β cutoff \*)*

α := max(α, value)

**return** value

**else**

value := +∞

**for each** child of node **do**

value := min(value, alphabeta(child, depth − 1, α, β, TRUE))

**if** value ≤ α **then**

**break** *(\* α cutoff \*)*

β := min(β, value)

**return** value

Starting the game:

Diagram

Description automatically generatedThis will be the first function to be called when the game starts. The use of this will be to initiate the board and any variables that will be needed to be displayed to the user.

Moving a piece:

This function will be running almost constantly after the game is initiated to accommodate for any quick moves that the players might want to make. For this reason, it must be very robust and reliable as if it fails at any point, it will break the core of the program and the players won’t be able to make their moves.

Diagram

Description automatically generated

Checking for takes, checks or checkmates:

This section will obtain the section of code that will allow the user to properly play and finish a game of chess. After the move has been made, the program will run this section of code to look for any collisions between pieces (removing the correct piece if necessary) and looking to see if the game is in a state where it can be ended (with a checkmate or stalemate).

Diagram

Description automatically generated

Pawn Promotion:

This will only have to run when one of the players gets a pawn to the furthest opposite rank. The algorithm will check that there is a pawn there. It will then ask the user what they want to promote the pawn to. The program will then promote the pawn to this piece and remove the pawn from the board.

Diagram

Description automatically generated

Castling:

As one of the most used moves in chess, this algorithm will be used in almost every game and as a result will have to be reliable. However, it can only be run once per game per player. This means that it doesn’t have to be as robust as the algorithm for moving a piece, for example. While the user can only make one castle move per game, the algorithm will have to consistently check to see if the king or either rook has moved from their original state as this will change the user’s ability to castle. If this is not tracked accurately then it could result in unfair games that contain illegal moves.

Diagram

Description automatically generated

Test Plan:

I am going to use a couple of test tables to evaluate my final solution. They will focus on different aspects of the final solution that need to be met, the functionality of the solution and the useability. The First table will focus on program functionality and will contain the success criteria for the user to have a fully functioning program while the second table will focus on the useability of the program for the user (e.g., visual aspects that aren’t essential for the final solution). The tables will be filled out when the project is completed and allow me to see if I met the initial criteria for the project.

I will also be testing periodically throughout the development process, mostly when I complete a new section of code. This will allow me to fix any issues that arise quickly and efficiently, before they cause more problems later in the development cycle and I spend hours upon hours going down a debugging rabbit hole that could have been avoided if I had tested each section. This can be done easily for the start of the program as each iteration builds on the previous version so new features can be tested quickly. When it comes to testing the implementation of alpha-beta pruning, that will be a bit more difficult as there is no visible change to the program that can be measured qualitatively. So, for this, I will have to use a timer to find the difference in time taken (for minimax with and without alpha-beta pruning) for the computer to find a move good move and make a new move. This can be further tested by playing the exact same game between the two to see if there is a noticeable difference in the time taken to complete the game.

|  |  |  |
| --- | --- | --- |
| Functionality Criteria | Pass/Fail | Reasoning/Explanation |
| The user can pick which colour they play as |  |  |
| The user can decide how they would like to play the game (single-player or multi-player) |  |  |
| The user can pick the difficulty of the AI player |  |  |
| The game starts when the user clicks “Play” in the main menu |  |  |
| The timer for white starts when the game starts |  |  |
| The player can select a piece before moving it |  |  |
| The user can change which piece they select |  |  |
| Any moved piece is moved instantly |  |  |
| After a move has been made the clock for that player stops instantly |  |  |
| Any piece that is taken is removed from the board instantly |  |  |
| Any checks force the player to move the king or sacrifice a piece for the king |  |  |
| The game ends when a checkmate, stalemate, draw or timeout is reached |  |  |
| En passent and castling are accurate and kept track of |  |  |
| The MiniMax algorithm is efficient and robust |  |  |
| Pawn promotions remove the pawn and replace it with a piece of the player’s choosing |  |  |
| The pieces cannot be moved off the board by the player |  |  |
| Button to allow the player to play again or return to the main menu |  |  |
| Only one piece can be moved per move (except for castling moves) |  |  |

|  |  |  |
| --- | --- | --- |
| Useability Criteria | Pass/Fail | Reasoning/Explanation |
| The player clock ticks down smoothly and consistently |  |  |
| When a player selects a piece to be moved, the legal moves are highlighted |  |  |
| The moves made are saved to an external text file that can be accessed by the user (PGN) |  |  |
| Button to allow the player to resign and/or restart |  |  |
| The edges of the board are drawn correctly |  |  |
| The window is sized appropriately |  |  |
| The size of the pieces is appropriate |  |  |
| The pieces are centered in their square |  |  |
| The visuals of the game are appealing |  |  |
| No unexpected outputs are given |  |  |
| The game shows a summary of the game once it has ended |  |  |
| There is a button that takes the user back to the main menu at the end of a game |  |  |

Development and Testing

Version 0.1.1

Text

Description automatically generatedThe board is being represented using a 2D array in order to have all the pieces of the square visible to the program and to make any troubleshooting with the GUI later in development easier as the GUI can be checked with the 2D array to find any discrepancies and fix them. The main array is be made of 8 smaller arrays, each with 8 elements. This is to mimic the shape of the chess board. Each of these elements is made of a 2-letter code that represents each piece for both white and black (this will also help for when I add the images for the pieces as they will have the same 2-letter code). I could have used a bit-board arrangement however, this would have taken a lot more memory resources (such as memory) to run efficiently as one board is needed for each piece and colour: a minimum of 32, 64-bit arrays would be needed. I am also not sure what the best way to implement this would be.

In a separate file, I am importing the Python library pygame as it will be an essential part of the program as I believe it is the best tool for easily creating the GUI that the user will see. I am also bringing over the file that contains my representation of the board (above). I will be using multiple, separate files that will interlink with each other as this will increase the maintainability of the code and allow for more organized code as any errors can be found and traced back more easily.

Text

Description automatically generatedNext, I have set some global constants and a library that can be used throughout the code. This has been done to reduce the possibility of systematic errors come runtime, and so that I can change any values, such as the size of the GUI window quickly and easily.

Text

Description automatically generatedThe function loadImages() is used to get the .png pictures for the chess pieces. I have decided to run this function once, at the start of the program, as it can be very intensive for Python to load images constantly. By loading the images once, and at the start, it ensures the program won’t be left struggling and slowing down later while it is trying to run other tasks.

Text

Description automatically generatedOriginally here, I had written out the load image line for every piece. After realising this was not the best way to code this section, I had a rethink and came up with the method above. The reason the code to the left is bad is because it is inefficient and will be difficult to troubleshoot any problems should they arise. For example, if there is a problem with the images not loading properly, I would have to find that issue and rectify it in all the lines of code which is a waste of time.

Shape

Description automatically generatedHere, I will be creating the board and drawing it onto the screen for the user to see. This is the main use of pygame and required the most research on my part. This section took longer than expected to develop as I am not familiar with the syntax and techniques used in pygame. I had to research how to show the window continuously (until the user closes the window), change the background colour and set the size of the window.

Text

Description automatically generated

The above only sets up the blank window that the game will be played in. I now need to get the board itself onto the screen. I decided to call this function and another, to draw the pieces on the board, from inside another function to help keep the code organized.

The drawing of the squares on the board uses some of the variables set out at the start of the program. By using 2 count-controlled loops, with one inside the other, I can colour the squares correctly (with r standing for row and c for column). When looking at the board from either player’s perspective, the top left square is always white. From this, we can use the MOD function to colour the squares properly. If the sum of the square’s row and column is even (answer of 0 after MOD 2), then the square is coloured white. If the sum is odd (answer of 1 after MOD 2), the square is coloured black.

Text

Description automatically generated

Text

Description automatically generatedA black and white checkered surface

Description automatically generated with medium confidence

The next logical step is to have the pieces drawn onto the board into the correct position. This function looks at each square on the board from the 2D array and then puts the correct piece on that square. It uses the IMAGES library I initialised at the start of the program to load the pictures quickly and efficiently. By looping through the elements of the array, I am making the program more efficient and maintainable as the same line of core isn’t repeated multiple times.

Text

Description automatically generated

Square

Description automatically generatedA picture containing text, crossword puzzle

Description automatically generated

Text

Description automatically generatedI had to change the colour of the squares from black to dark grey as the black pieces couldn’t be seen on the black background (as you can see above). While this does change the look of the board slightly, it remains familiar to the user and keeps the game playable. This change was as easy as changing the colour in the drawBoard() function.

Stakeholder feedback:

After a brief conversation with the stakeholder, they said they liked the look of the board and pieces. They referred to the “classic” look of the board I produced and that it appealed to them, saying they liked me “keeping it simple and not trying to re-invent the wheel”. They suggested an option that would allow the user to change the colour of the squares in order to make the experience a bit more personalised. After talking to them about this addition, while both of us really liked the idea, it was decided that due to the scale of the rest of the project, this addition would be put on the backburner and only added at the end of development, if there is time. This was agreed as the addition would only be cosmetic and not make a substantial enough change to the program to justify spending large amounts of valuable development time on it.

Version 0.1.2

The next iteration of the program will contain the ability to move the pieces on the board. To start, this will mean that any piece will be able to move to any square. To do this, I will create a class in the ChessEngine file called Move to control all the moves that will be made. This will be done after I have setup some key lines of code in my main file.

Text

Description automatically generatedFirst, I found the event list for pygame. This was made earlier when making the window appear as this is needed to keep the window running and allow me to close it. In the event list, I created a new if statement that uses a built-in pygame function called MOUSEBUTTONDOWN. This checks to see if the user has pressed their mouse button. This is essential as it allows the user to directly interact with the GUI. Within this statement, I have used another pygame function called mouse.get\_pos(). This finds the co-ordinates ( x , y ) of the mouse pointer when the click is made. This function creates a tuple, allowing us to have both the x and y ordinates within the same variable. Beneath this, I have found the correct square for the program by using the floor division function. This returns the integer value that is left after the division. This is useful as, by doing this, I am certain the value returned will correlate to a square on the board.

Text

Description automatically generatedOne issue that could have arisen if I chose to stay with this implementation is that I would have two variables both trying to hold 2 values, and that could get complicated. To overcome this, I decided to create two more variables. They will be located outside of the loop in order to easily re-use them multiple times. These variables are called sqSelected, which will keep track of the latest click made by the user, and playerClicks, which is a list containing two tuples that contain the two ( x , y ) co-ordinates of the selected squares.

Text

Description automatically generatedText

Description automatically generatedThe next problem arose after thinking about what would happen if the user selected the same square twice in a row. Usually, this would imply wanting the user wanting to deselect that piece so they can move a different one. In my program currently, the program would move the piece on that square to the same square, essentially taking a turn when no piece has been moved. Luckily, this was easy to solve as it just needed a simple if statement that would set the variables sqSelected and playerClicks to empty states, effectively restarting that move for the player. This allows the player to select and deselect a piece to move, meaning they aren’t locked into moving the first piece they click on.

Now I needed to decide how, after the user’s click on a second square, I wanted to move the piece on the board. There are multiple ways I could have done this: tuples, board updates or a class. Using tuples would have allowed me to keep a move log and move the pieces using the same variable but this could get complicated further down the line as I am hoping to add a way to undo moves. Taking snapshots of the board is a different way of moving the pieces, but this would require multiple 2d lists that would take up more space than is necessary. SO the way I settled on in the end was using a class called Moves. By using a class, I am making the code more maintainable and I am also making it easier for myself to add very basic chess move notation.

both clicks on squares are now in playerClicks

* Was this user’s second click? If yes then we want to move piece  
  Decisions to be made here: tuples? Update board? Create move class – more clear, can then do chess notation more easily
* Move to ChessEngine file: makes more sense as it is tied to game state, there will be list of moves in this file pass through board as we want information about pieces capturing and being captured to recall later abstraction to visualize more easily what is going on (self.\_\_\_\_) This allows up to have all information about a move in one place (add en passent and castling later)
* By using a class, we can make a chess notation move log of the game, helps with the player and debugging later. Using libraries, we can convert the list index values into rank and file for chess notation
* Make function to create the chess notation, keeps code organized. This could be more complex, but that would require lots of logic and wouldn’t improve the game massively as this won’t be shown to user so just rank-file for now
* Now making move is a function within GameState() don’t have to worry about the piece being overwritten as it is stored in pieceCaptured don’t have to worry about validation for now as we will make a list that contains valid moves and only lets the user make valid moves, therefore assume move is valid adds move to move log and changes the player’s turns
* Go back to main and call the class with its functions and pass in the parameters. Fairly “cheap algorithm” – setting variables and looking at values in a small list, no for loops
* To run the code and check it is working, we set it up like shown and the moves made will show up

Text

Description automatically generated

Text

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

A screenshot of a game

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated with medium confidenceGraphical user interface

Description automatically generated with low confidence

Evaluation

Bibliography

Appendices (if relevant)