OCR GCE A

COMPUTER SCIENCE PROJECT

H446-03

Name : Alex Dyer

Candidate Number: 8572

Tuxford Academy: 28148

Title of Project: KnightOwl Chess Bot

H446-03 – Project CONTENTS

Table of Contents

[A. Analysis 5](#_Toc190868578)

[Project Identification 5](#_Toc190868579)

[Stakeholders 5](#_Toc190868580)

[Why is this Suited to a Computational Solution? 5](#_Toc190868581)

[Stakeholder Statistics 7](#_Toc190868582)

[My Microsoft Forms Questions 7](#_Toc190868583)

[Final Stakeholder Responses 8](#_Toc190868584)

[Research 11](#_Toc190868585)

[Existing Similar Solutions 11](#_Toc190868586)

[Features of My Proposed Solution 15](#_Toc190868587)

[Requirements 16](#_Toc190868588)

[Software and Hardware Requirements 16](#_Toc190868589)

[Stakeholder Requirements 17](#_Toc190868590)

[Success Criteria 18](#_Toc190868591)

[B. Design 19](#_Toc190868592)

[Problem Decomposition 19](#_Toc190868593)

[Decomposition Overview 19](#_Toc190868594)

[Detailed Decomposition 19](#_Toc190868595)

[Systems Diagrams 23](#_Toc190868596)

[All Algorithms for Main.py 23](#_Toc190868597)

[All Algorithms for Moves.py 24](#_Toc190868598)

[All Algorithms for Graphics.py 32](#_Toc190868599)

[Usability Features 37](#_Toc190868600)

[Navigation 37](#_Toc190868601)

[Consistency 37](#_Toc190868602)

[User Feedback 37](#_Toc190868603)

[Visual Clarity 38](#_Toc190868604)

[Error Prevention 38](#_Toc190868605)

[Variables and Classes 41](#_Toc190868606)

[Iterative Development Test Data 43](#_Toc190868607)

[Post-Development Testing 49](#_Toc190868608)

[C: Developing the coded solution 50](#_Toc190868609)

[Development Process 50](#_Toc190868610)

[Setting Up the Screen – **V1.1** 50](#_Toc190868611)

[Adding the Board Array 50](#_Toc190868612)

[Adding the Piece Files and Drawing the Board 51](#_Toc190868613)

[Drawing the Pieces 53](#_Toc190868614)

[Moving the Pieces (No Move Generation Yet) 55](#_Toc190868615)

[Setting up the File for Move Generation 57](#_Toc190868616)

[Finding all the Possible Pawn Moves 58](#_Toc190868617)

[Adding the Moves for All Other Pieces 59](#_Toc190868618)

[Tracking the King’s Location 61](#_Toc190868619)

[Adding Checks from the Knight AND Pawn 62](#_Toc190868620)

[Rook/Queen (Horizontal and Vertical Only) Checks and Pins 64](#_Toc190868621)

[Adding Bishop/Queen (Diagonal Only) CHecks 66](#_Toc190868622)

[Making Escaping Check Mandatory 68](#_Toc190868623)

[Making Pins Effective 71](#_Toc190868624)

[Adding Move Highlighting 72](#_Toc190868625)

[King Move Debug 72](#_Toc190868626)

[Castling 73](#_Toc190868627)

[Pawn Promotion 74](#_Toc190868628)

[En Passant 76](#_Toc190868629)

[Pawn Promotion Graphics Glitch – **V1.2** 77](#_Toc190868630)

[Castling Out of Check Debug 79](#_Toc190868631)

[Castling Through/Into Check Bug Fix 79](#_Toc190868632)

[Menu Screen – **V2.1** 80](#_Toc190868633)

[Telling People Whose Turn it is 82](#_Toc190868634)

[How To Play Screen 82](#_Toc190868635)

[Draw by Insufficient Material – **V2.2** 83](#_Toc190868636)

[Code Modules 85](#_Toc190868637)

[Main 85](#_Toc190868638)

[Moves 88](#_Toc190868639)

[Intro Screen 93](#_Toc190868640)

[Graphics 95](#_Toc190868641)

[Engine 95](#_Toc190868642)

[Vars 96](#_Toc190868643)

[Colours 96](#_Toc190868644)

[Variables/Structure Names 96](#_Toc190868645)

[D. Evaluation 96](#_Toc190868646)

[Project Appendixes 98](#_Toc190868647)

[References 98](#_Toc190868648)

# A. Analysis

## Project Identification

There have been many different games made over the millennia. Some are older than others, some are newer, and some can be played on a screen with a controller or a keyboard. But one game that has withstood the test of time is Chess. Chess is a strategy game where each player controls sixteen pieces – a king, a queen, two rooks, two knights, two bishops, and eight pawns. They must manoeuvre these pieces around an 8x8 chequered board to checkmate their opponent’s King. This game has been played and studied obsessively for over 1,500 years.

More recently, AI has advanced so that many bots have been programmed to play chess to a higher level than any human. Working off the Elo scale, the greatest player of all time, Magnus Carlsen, has an approximate Elo of 2,900. This is the highest human chess Elo in history. However, chess computers such as Stockfish, Torch, and AlphaZero have been made to beat Magnus Carlsen and the rest of the world’s approximately 1,800 Grandmasters (GMs). But one thing that is a lot rarer is a chess bot aimed at being beatable by people of around 1000-1300 Elo. People always go on about these absurdly good chess players and chess bots. How about one that plays similarly to a human at an intermediate to advanced level of chess? This is the absence that I would like to remedy with my project.

I will use Python and one of Python’s graphics libraries Pygame to create KnightOwl. The features of a computer that would be required to use this would be the use of a keyboard and mouse as Python does not work with touchscreens.

### Stakeholders

The clients and general demographics of this bot are chess players from complete beginners to about 1,600 Elo. Because over 70% of chess players fall within this range, the stakeholders will be a representative sample, ranging from people who have just begun playing chess to people who are very comfortable with the game and its various strategies.

Stakeholders for the chess bot represent casual players who play chess occasionally to take a break from work/school to experienced people who want a new challenge or an easy game.

The game, for the most part, will be playable with just a mouse, but there will be a function that takes chess notation as an input too. It will come with an analysis feature that breaks a game down into single moves and explains the pros and cons of each move.

### Why is this Suited to a Computational Solution?

This project lends itself to a computational solution in many ways. The solution will be a chess computer that isn’t so hard to beat that it demoralises new players and angers gifted players, but one that anyone can play against and have a decent chance at winning against.

#### Problem Recognition

The main problem with this is creating a chess bot that plays like a human, as most chess bots have a completely different play style from any human because they’re programmed to make a good move or a worse move randomly. The chances of a good move compared to a bad move differ based on the not’s supposed Elo. My problem is that to make this work, I have to program a specific style into the computer and get it to stick to it. Most human chess players play with their unique style and preferences, and that’s why no two chess games are the same. The challenge is to create a bot that works in the same way.

#### Decomposition

This issue can be broken down into 5 smaller steps:

1. Work out the graphics for the program (e.g. the board, pieces, movement animation, etc)
2. Code the game itself
3. Build a basic bot
4. Improve the bot
5. Work out the finer details, such as a playstyle and aggressive/defensive tactical/positional preferences

When these steps are complete, the program doesn’t lag, and the bot can beat me (13-1400 Elo) about half the time, it’s done. With machine learning and reinforcement learning, I should be able to get the bot to play against itself a sufficient amount of times so that it can train itself to get to a high enough level, putting its findings into an SQL database. I can hard-code any chess theory needed into the database, to make it seem more like a human.

#### Divide and Conquer

Although these steps seem challenging, they are perfectly feasible. Solving each of these independently and combining them into a complete, modular program uses the divide-and-conquer way of problem-solving.

## Stakeholder Statistics

### My Microsoft Forms Questions

I will summarise the main questions that I have asked my stakeholders about their general chess-playing, and whether they’d be interested in my proposed project.

#### Current Chess Info for each Stakeholder

Questions 1-3 summarise how often the stakeholder plays chess and how good they are at it

1. Do they play chess?
2. Current chess Elo
3. How often do they play chess?

#### Views on Current Chess Bots

Questions 4-6 are on the stakeholder’s typical opponents and their views on the existing bots

1. How often do they play against bots?
2. Would they consider playing a new chess bot?

#### Would they Play my Chess Bot?

Questions 7-9 are views regarding my chess bot

1. Would they be interested in seeing a chess bot that plays like a human would up to an Elo of 1600?
2. What name would be suggested for the chess bot?

### Final Stakeholder Responses

A white background with black text

Description automatically generated

A graph with different colored bars

Description automatically generated

A white background with black text

Description automatically generated

The first three questions show what quantity of stakeholders play chess and their strengths/regularity with chess. The statistics show that most of the stakeholders play chess and, of the people who do play chess, the majority are between 400 and 1100 Elo. This would mean that a 1600 Elo bot would be a difficult, but reachable challenge for them. Also, a lot of them either play rarely or weekly, meaning that, hopefully, my bot would get the people who play rarely to play more often and therefore increase the popularity of chess.

A white background with black text

Description automatically generated

A white background with black text

Description automatically generated

A screenshot of a computer

Description automatically generated

According to the statistics from the survey, most people usually or only play other people, however, they would like to see and, for the most part, would definitely or possibly consider playing a bot at around 1600 Elo. As shown above, all stakeholders asked would like to see a bot that plays like a human to an Elo of 1600. Therefore, I think that the chess bot will be popular among chess players of all skill levels.

A screenshot of a phone

Description automatically generated

I will decide on a name nearer the time

## Research

### Existing Similar Solutions

#### chess.com

##### Overview

A screenshot of a game

Description automatically generated

Chess.com contains over fifty chess bots of various skill levels, ranging from absolute beginner levels to being better than any Grandmaster. This provides a wide range of fun bots who each go for their own various opening selection, tactics, fortes, and catchphrases. Complete with an easy-to-use user interface, social/friends database to play chess with friends, online capabilities, puzzles, and lessons, this is a user-friendly and fun way to play chess against anyone, regardless of whether or not you know them.

The user interface is immensely simple and easy to follow as the menu is always displayed and there are large buttons on-screen labelled with anything that can be done on the site. The home screen contains the menu, a suggestion of a player on your friends list to play against, puzzles and a game review suggestion, as shown below:

A screenshot of a game

Description automatically generated

A screenshot of a game

Description automatically generated

The ‘Game Review’ function allows a player to get the chess engine to analyse one game a day on the free version, or unlimited depending on what subscription plan they have. It contains game accuracy, the number of different types of moves made, the best moves for each position, why they were the best moves and an Elo prediction.

A screenshot of a game

Description automatically generated

A screenshot of a game

Description automatically generated

Higher ratings mean better gameplay, as do higher accuracies. The Game Review feature is very easy to understand and very informative of good and bad moves and sums them up into two numerical values (Elo and accuracy)

##### What can I apply To My Solution?

In my solution, I would like to build an easy-to-follow user interface, similar to chess.com’s, as well as an analysis/game review function to show players where they can improve. The main difference between the bots on chess.com and the bot I plan to code is that the bots on chess.com play completely differently from humans, in so far as they are designed to have a randomised chance to play a move, changed based on the supposed rating of each bot. Mine is supposed to play more similarly to a human and play more human moves that, although they may not be the best moves of an engine, are more likely to feel natural to a beginner/intermediate chess player.

#### Lichess.org

A screenshot of a computer

Description automatically generated

This is another chess site that is one of chess.com’s main competitors. Although it isn’t as good as chess.com, it has a massive opening database, showing the most common and the best human moves for hundreds of assorted chess openings. It also has a FEN analysis function, meaning you can get the engine to analyse any given position. Similar to chess.com, it also has a game analysis function, however, this one isn’t as good as it only shows the bad moves and doesn’t rate the good moves made by the player. As shown in the chess.com diagram above, chess.com shows all moves ranging through brilliant, great, best…mistake, miss, blunder. Lichess only focuses on inaccuracies, mistakes, and blunders. It is a very good, albeit immensely critical chess engine.

Another feature that it has in common with chess.com is the fact that the player can also play the computer. However, the computers here aren’t as immersive as the ones on chess.com, as they don’t possess the same skill range, catchphrases, or range of bots.

##### What Can I Apply to My Solution?

The only feature that I would probably use for my chess bot would be the opening database because it would help me differentiate ‘Book’ moves and theory from any middlegame/endgame moves. It would also vastly help see what rating the user plays like, based on how common the opening is and whether they do something classic for many people of a low Elo, which is to attempt to invent and utilise their openings, instead of learning any theory.

### Features of My Proposed Solution

#### My Initial Solution Proposition

My solution will be a chess bot that plays like a 1,600 Elo human because, although both lichess and chess.com both contain their own bots, neither of them contains bots that play like a human. In this case, I think that a combination of the lichess opening database, put into my own SQL database, and a similar range to the chess.com bots, in that, I could attempt to do different difficulties, ie: 400, 700, 1000, 1300, 1600 Elos. These bots will have a visual move-log and game review/self-analysis functions similar to the ones offered by chess.com, however, unlike chess.com, there is unlimited access to these features for free. There will also be a function to save played games so that they can be looked at again later down the line for improvements and future analysis.

#### Potential Limitations

The main limitation of my solution is that, although it incorporates a lot of functions based around the bot, it won’t include any online functionality, puzzles, or lessons, as the main point of this solution is to code the bot(s).

Another is that, because I’m using Pygame instead of a more advanced GUI, the UI is likely to be fairly straightforward. It will include some buttons for which bot you want to play against, and a list of recent saved games, with analysis if analysis is requested for the game in question.

## Requirements

### Software and Hardware Requirements

#### Hardware Requirements

**A computer capable of running an up-to-date version of Pygame** –The chess bot will need to be run on a computer capable of running Python 3.10, MySQL and Pygame. A laptop with a processor capable of running at 30+ fps is recommended, but not required

**A mouse and keyboard** – Pygame isn’t compatible with touchscreens, so mobile devices/iPads won’t be compatible with the software

**6GB RAM** – 4GB is the minimum RAM requirement to run Pygame, and 2GB is my current estimate of how much RAM will be required to access the MySQL database. Anything below this could potentially cause the chess bot to crash

**3GB HDD/SSD space** – Pygame will need 3GB of HDD or SSD space to run, MySQL needs another 2GB and the image files need about 100KB, so at least 7-8GB will be needed to record

#### Software Requirements

**Windows, Mac, or Linux OS** – These are the Operating Systems that support Python and SQL

**480p** **Resolution** – This will be the minimum monitor resolution required to play the chess game as it needs to be a resolution where the user does not need to strain their eyes to make out what pieces are what, so they don’t damage their eyes

**Python Interpreter** – The code will be written in Python, with MySQL and Pygame libraries

**MySQL for Python** – The opening database will require an SQL database, and therefore a Python plugin for SQL

**Pygame for Python** – The GUI will require a Python GUI library to run the chess bot graphics

### Stakeholder Requirements

#### Design

|  |  |
| --- | --- |
| Requirement | Explanation |
| Basic user interface with the ability to change the difficulties of the bots | So the user can adjust and access the features of the program easily |
| Easy-to-understand, informative analysis programs for any game played | Allows the user to rate their performance on each game based on the engine’s analysis of their game, and their analysis |
| An instructions page | Shows the user any key binds associated with the program, how they are used and how to navigate the program if they don’t understand it or aren’t good with technology in general |

#### Functionality

|  |  |
| --- | --- |
| Requirement | Explanation |
| Use of mouse to control moves | The mouse’s position is recorded by Pygame and is used to control which piece was clicked and where it was moved to |
| Use of keyboard to undo moves and input chess notation | The backspace key is going to be the keybind for undoing moves, and the main keyboard will be used to input chess notation as another way to control piece movement |
| Instructions on how to edit the theme of the board and the style of the pieces | If the user dislikes the look of the chess board and pieces, there will be instructions on how to edit the themes within the game |

## Success Criteria

|  |  |
| --- | --- |
| Criterion | How to show it’s been met |
| The main window showing a working chess engine | Screenshot of working graphics/video of a working engine |
| The AI makes non-randomised moves | Run a game through Stockfish to show a decent amount of logic |
| Simple, navigable design | Screenshot of the GUI, with a menu, chess engine and instructions |
| A working Exit button | The Exit button (namely the big X in the top right) is clicked and shows that it closes the window |
| Instructions | A screenshot of a page that’s accessible through the main menu, walking the user through the keybinds and any other features that they need to know and that points them to a site where they can learn to play chess if they don’t already know |
| The chess AI plays at the desired level (1,600 Elo) | The AI can beat me over 50% of the time that it plays me, as shown through the PGN files of the games |

# B. Design

## Problem Decomposition

### Decomposition Overview

There are many smaller steps that my chess engine can be broken down into. These are as follows:

* Board Representation and Generation
* Turn Tracking
* Piece Movement (Excluding Special Moves)
* Castling
* Pawn Promotion
* En Passant
* Move Validation and Illegal Move Prevention
* Detecting Checkmate and Stalemate
* Event Handling
* Board Colour

### Detailed Decomposition

#### Board Representation and Generation

This is solved specifically in my graphics.py file and is a problem that can be solved on its own, but it’s vital that it be solved first before anything else so that testing is easier. These functions (namely generating the board and pieces) will be called upon for each iteration of the main loop and therefore, to a degree, these are closely linked to the main.py file. Although there are existing functions that could do this for me, I felt that it would suit my code better if I coded these algorithms myself, to ensure that they work exactly how I want and need them to, especially if some of the code required is closed source. Although the main functions of this are to generate the board and the pieces, it also highlights squares that pieces can move to and the square that gives the user the ability to select a piece to promote a pawn to (I will go over these in more detail later in the decomposition).

#### Turn Tracking

Although I would consider this a problem in its own right, and a vital component to the overall solution, this can be implemented easily, solely through the use of a Boolean flag, which is true if its white’s turn to move and false if its black’s turn, or vice versa. This would be used within the main.py and moves.py files, as it would be essential to the main workings of the program, as well as finding all the legal moves for one side or the other in a position so that it knows which side it should be finding moves for. This is used as I coded the board array so each square containing a piece or a pawn also holds its colour (i.e. ‘bQ’ would mean Black Queen or ‘wp’ would mean White Pawn – each square has two characters, with ‘--’ meaning empty)

#### Basic PIece Movement (Discounting Checking if a Move is Legal)

Although basic move generation and finding pins and checks are both to do with finding and narrowing down the possible moves, I would count them as separate problems as they are each very large problems and, sequentially, checks and pins needs to have move generation as a prerequisite. I implemented move generation through moves.py and several helper functions. I will go through the helper functions shortly. Similarly to the graphics, there are pre-made algorithms for this online, but I wanted to ensure that the algorithms fitted correctly with how I was coding the engine, so I made them myself and put them in a separate file and many functions and procedures to optimise the readability and therefore the maintainability. These, for the most part, are each standalone functions, with only the ‘getValidMoves’ function linking back to main.py. This will be done by using ‘getAllPossibleMoves,’ which will loop through the board array and, when it finds a piece of the colour whose turn it is to move, runs the respective helper function to find all the possible moves in the position for the piece in question.

##### getting All the Possible Moves

This will be performed by the aptly named ‘getAllPossibleMoves’ function. As stated above, it loops through each piece and runs a helper function based on what piece it is. It doesn’t worry about legal and illegal moves, as this is done by another function later in the decomposition

##### Piece Movement Helper Functions

These are all called ‘get[Insert Piece Name Here]Moves’ and each run an algorithm to find any and all possible moves for a piece, using the ends of the board and the locations of enemy pieces and ally pieces to break loops for the bishop and rook (technically the queen too, though for that, I just call the rook and bishop functions to improve the efficiency of the translation). These employ some of the more basic logic within the program to find every potential move. They take the self.moves array and whose turn it is to move from the getAllPossibleMoves function

#### castling

This is another of the functions that is within the moves.py file. It is considered to be one of the three special moves in chess, and I needed to code it separately to other moves as it functions differently to them. This will be called by the getValidMoves function and, if the following conditions are met, the engine will add some or none of the possible castling moves to the moves array:

##### Conditions:

* There must not be any pieces between the King and Rook
* The King must not be in check
* The King must not move through check to castle
* Neither the King nor Rook can have moved before castling

If all these conditions are met, then the respective castling moves are added to the moves array

#### Pawn Promotion

As the second of the three ‘special’ moves, this will also have to be coded separately. As a move, it is added to the potential moves list within ‘getPawnMoves.’ The main difference is within the Main class. If pawn promotion takes place, the rest of the game stops until the player chooses a piece to promote to. Although the most common choice is, understandably, the Queen, the Rook, the Bishop, and the Knight are also valid options to promote to. This choice will be managed in the Graphics and Main classes. The Graphics class will create the square for selecting the piece to promote to and the Main class will halt the rest of the game until a piece is selected using another Boolean flag.

#### En Passant

This is the third special move, and this will be coded by editing the ‘getPawnMoves’ function. This will be done by adding a variable to hold the previous piece moved and where to. If it was a pawn and it moved two spaces, then en passant is possible. Then, the code checks to see if a friendly pawn is adjacent to the hostile one and, if so, en passant is added to the moves array. In the main file, if en passant is selected, it alters the board differently and removes the hostile pawn from the board, even though the friendly pawn didn’t actually move to its square.

#### Move Validation and Illegal Move Prevention

Within ‘moves.py,’ I will add a function that has the purpose of looking for checks and pins. It does this by using an algorithm similar to ‘getAllPossibleMoves,’ in so far as it will add to check and pin arrays by running a helper function dedicated to each piece in turn. Although the knight and pawn algorithms will be fairly simple as a total of 10 squares have to be checked for both of them combined, rook, bishop and queen are going to be a lot more complex to implement. This will be solvable by going to the ends of the board or as far as is needed in every direction from the King to find either pinned pieces or checks. This is a problem in its own right as very few of the other components of the program will affect it in any way and will involve difficult logic. Once the checks and pins have been found, I can use the index of the pinned piece’s location or that of the checked King’s on the board to ensure that any illegal moves can be removed. This will be done by cross-referencing the location index with the first two values in each move in the moves array. If they match, then the move refers to the pinned or checked piece. From there, each move for that piece can be checked to see if the King can be taken next move. If it can, then that move is an illegal move and can be removed from the array.

#### Detecting Checkmate and Stalemate

This can be tracked through the ‘main.py’ file as, once one side has made a move, the opposite side’s legal moves are calculated. If the moves array is empty, there are no legal moves and therefore the game is over. From there, whether or not the King is in check can be calculated and if so, it’s checkmate and he checkmated side loses. However, if the King is not in check, it’s a draw by stalemate.

Another way to have a draw in a game is by insufficient material. This can be implemented by counting how many pieces are on the board at any given time. If it’s 6 or less, the material can be checked and if it’s a certain combination of pieces, neither side can checkmate, therefore it’s a draw as neither side can win.

#### Event Handling

For the event handling, I utilised some features of the Pygame graphics engine and told it to look for the pygame.QUIT event when the X in the top right corner is clicked, the pygame.MOUSEBUTTONDOWN event if the mouse is clicked and for pygame.KEYDOWN if a key is pressed. Within pygame.MOUSEBUTTONDOWN, it is checked whether a piece or a move selected is legal and whether the user wants to deselect the piece or move it.

#### Board Colour

This is an additional feature that isn’t a specific part of chess but makes the game more interactive for the user. When a key is pressed (most likely the left CTRL button), the dark squares change colour. Within colours.py is a list of 12 RGB codes that are cycled through when clicked. This is managed within main.py and is a simple feature that gives the user some ability to personalise the board.

## Systems Diagrams

### All Algorithms for Main.py

Pseudocode can be found in the Pseudocode Folder within write-up

#### Main Loop

[main.py flow chart view link](https://miro.com/welcomeonboard/YlFpaEFWZ1lrY3oxM3dtc2cwTEVRMHJ1VnJ1c1BnYmJmd2JMREdCUWhRSUY5N2tkd25EOFVtSFB0RlF0bUdHd3wzNDU4NzY0NTk3NzA1ODM5NDE2fDI=?share_link_id=241004072459)

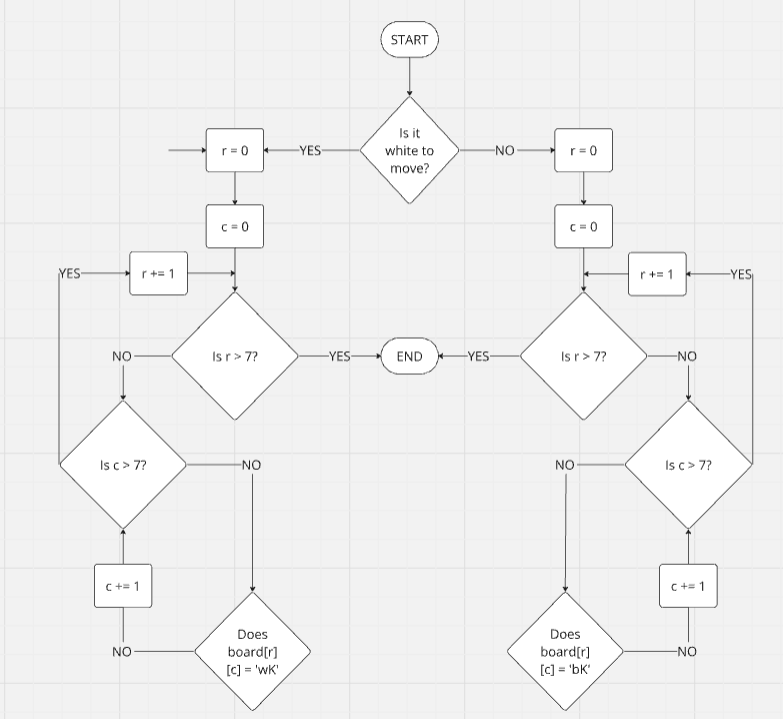
A diagram of a flowchart

Description automatically generated

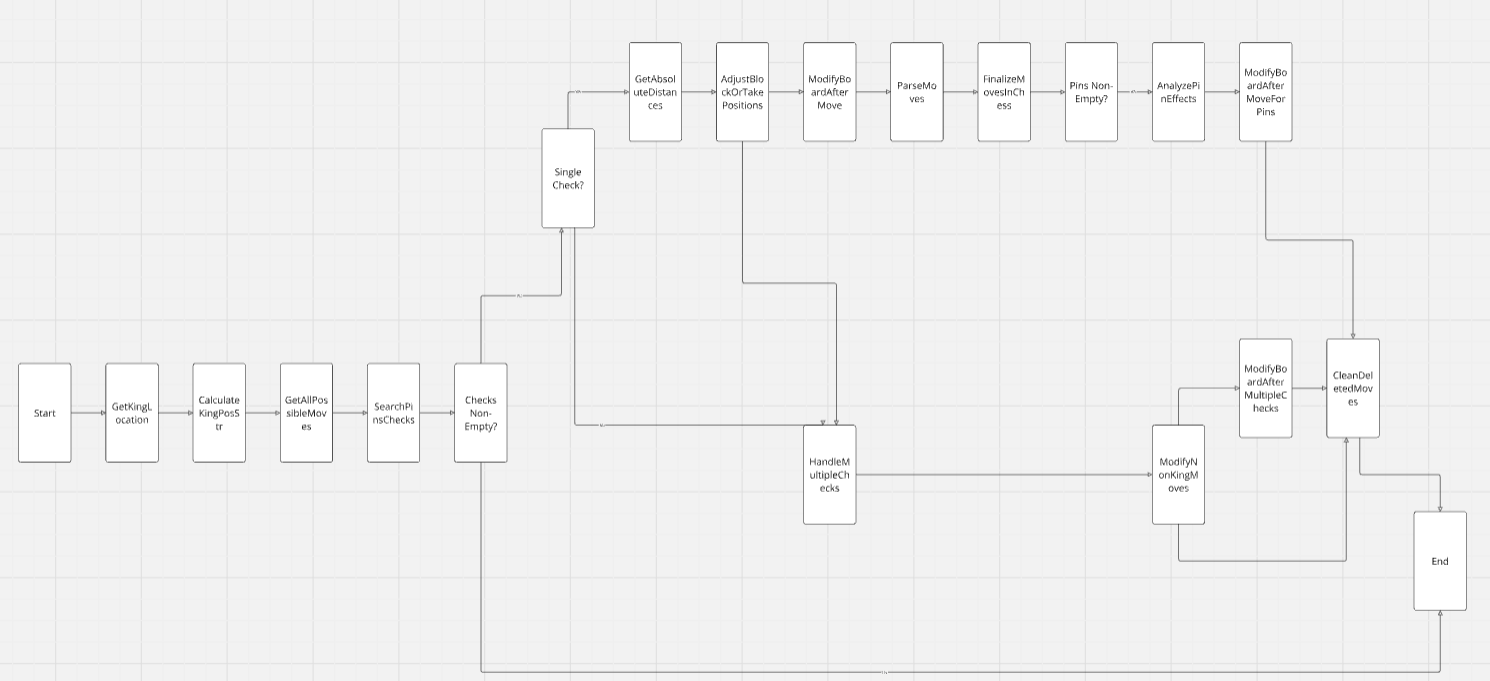
### All Algorithms for Moves.py

Pseudocode can be found in the Pseudocode Folder within write-up

#### Finding the King’s Location



#### Getting All Valid Moves



#### Getting All Possible Moves (Excluding Checks and Pins)

A diagram of a flowchart

Description automatically generated

#### Checking If Castling Is Possible

A diagram of a black king

Description automatically generated

#### All Algorithms for PIece Movement

<https://miro.com/app/board/uXjVKi6uJQE=/>

#### Algorithm for Searching for Pins and Checks

A diagram of a check

Description automatically generated

#### Algorithm for Knight CHecks

A diagram of a algorithm

Description automatically generated

#### Algorithm for Pawn Checks

A screenshot of a computer screen

Description automatically generated

#### Checking for Rook/Queen Checks

Queen checks in this flow chart are horizontal or vertical only

<https://miro.com/app/board/uXjVKidCQyE=/>

A diagram of a flowchart

Description automatically generated

#### Algorithm for Bishop/Queen Checks

Queen checks in this algorithm are diagonal only

A white square with black text

Description automatically generated

### All Algorithms for Graphics.py­­­­­­­­

Pseudocode can be found in the Pseudocode Folder within write-up

#### Drawing the 8x8 Board

A diagram of a flowchart

Description automatically generated

#### Move HIghlighting

A diagram of a flowchart

Description automatically generated

#### Creating the Pawn Promotion Options Square

A diagram of a flowchart

Description automatically generated

#### Drawing the Pieces

A diagram of a flowchart

Description automatically generated

Every part of my solution, as can be seen in the flowcharts above, performs its own unique task and is used alongside several other functions to create the chess engine.

main.py: This is used to hold all the functions and procedures together and make them run in the correct order. It also checks if a selected move is valid and if it is, it adjusts the board array to show the new position.

moves.py: This is a huge file that holds all the functionality needed to find all the legal moves in a position for either side including the removal of any moves that would endanger the King and returns all the moves in an array that can be understood by main.py. I will go over the functions within this later.

graphics.py: This file contains the methods to draw the pieces and the board to the screen. It also highlights squares that each piece can move to when they are selected and creates the square when a pawn promotes

moves.py in detail:

I will need a lot of functions to determine which moves are legal and which put the King in danger. I will go through each necessary function with an overview of what it is used for and why

getKingLocation: This will iterate through each square on the board searching for the King of the player whose move is next

getValidMoves: This will be the main function in this class. It is the one that is called from main.py and will call all the others in the file in the necessary order. In addition, it takes all the feasible moves in the position (by calling getAllPossibleMoves) and any checks and pins (by calling searchForChecksAndPins) and removes any ‘Illegal’ moves (moves that would endanger the player’s own King

getAllPossibleMoves: This iterates through every square on the board. For every piece on the current player’s side, it will run a function corresponding to what it is and stores the potential moves in an array to be altered by getValidMoves

castling: This checks whether or not the current player is able to castle. It does this by checking there are no pieces between the King and Rook, that neither has moved, no hostile piece attacks a square the King would have to move over, and the King isn’t in check

get[PIECE]Moves: There are six of these functions. Each one finds the moves for one type of piece and adds them to the possible move array. Pawn moves and King moves have the additional features that they check for En Passant and removing squares attacked by the opponent, respectively.

searchForPinsAndChecks: This runs 4 functions to find checks from any piece except the King. Pawns and Knights have their own. Rook/Queen does horizontal and vertical. Bishop/Queen does diagonals.

find[PIECE]Checks: These search for checks from their specific type of piece. If any are found, they are returned in an array.

## Usability Features

### Navigation

I will go for a basic system navigation system to make it as easy as possible for the users to navigate. To achieve this, I will make a menu screen (see below) with ‘Play’ and ‘How to Play’ buttons to take them to the game and to instructions on using the system, respectively. How to play also includes an interactive, clickable link to a website that teaches the user how to play chess.

### Consistency

All data used in the game is kept consistent. Being an engine, no personal data is stored, so I don’t need to worry about the Data Protection Act (1998) or GDPR (2018). The only data required is the data needed to get the current game to run properly. This includes the current position on the board, the current colour of the dark squares on the board (customisable by clicking the left control buton) and the piece that has currently been selected.

### User Feedback

I will take user feedback into great consideration when I create the engine. I will ensure that, for every prototype, I get feedback, so I know what needs to be improved and/or added.

### Visual Clarity

Everything in the program will be very obvious and easy to understand. When a piece is clicked, legal moves for that piece will be highlighted and the pieces will be easy to distinguish from one another.

### Error Prevention

I can prevent errors by ensuring that any checks of arrays remains within the index limits of the array. For example, if I’m looking for checks on the board, I could make sure that the board co-ordinate (7, 7) is never exceeded.

#### Main Menu

A green sign with black text and a horse head

Description automatically generated

#### Instructions Screen

A green and black text on a green background

Description automatically generated

## Variables and Classes

Here is a list of the key variables and classes and what they are used for:

#### Variables and Data Structures:

width – Holds the width of the screen

height – Holds the height of the screen

row/colSize – A constant to hold how many rows or columns there are (stays as 8 for the whole program)

squareSize – width // rowSize – Holds the length/width of each square

Board – 2D array holding the current position on the board. I used a 2D array here because it allows me to use the first dimension for the columns and the second for rows

moves – Holds all the valid moves for that turn in the same format as in squares to make it easy to see if they match

squares – Holds the coordinates of the squares clicked. If the entire array doesn’t match up to a value in moves, it gets emptied

clicks – Works closely with the squares variable. This holds the number of clicks the player has done. If it reaches two, that’s how the computer knows to check the squares variable

w/bKingLocation – Holds the location of each King on the board to help search for checks and pins

w/bKingHasMoved – Boolean flags that determine whether each King has moved and therefore whether castling is a possibility

inCheck – Boolean flag that shows whether or not a player is in check

checks – Array that holds the current check (or two checks if it is a double check)

pins – Array that holds any current pins

validMoves – A list of every legal move in the position

font – Holds the font for any text required

r/c – Shorthand for any row or column variables

moveLog – This holds all the previous moves (in coordinate notation – [1, 3, 2, 1]

RFMoveLog – RankFileMoveLog holds the previous moves in Rank/File Notation (i.e. Nxc7+). (Post-development note: This would only have been used if I had done an AI. Because of time constraints, I had to stick to a two-player chess engine)

#### Classes:

Main: This holds all the code to run the software, ensure that a move selected is valid and to ensure that the rest of the code within other classes run in the correct order and at the correct times. The reasoning behind this was so that I had a file in which a structure for the sequence of the code was both visible and obvious.

Moves: This class has a large number of different functions and procedures to allow for a sequential process of finding all the possible moves in a given position and narrowing them down to ensure that checks and pins are taken into consideration. This file was kept separate from the main body of the code so that the code was more easily readable and therefore more maintainable. The only reference to it inside the Main class was to call the ‘getValidMoves’ function, which links to every other in the file.

Screens: This contains the code that creates the Main Menu screen and How To Play which appear on startup of the program. This was implemented within my final changes because it came to my attention after some initial client feedback that some people were struggling to understand how to use the software on their own. As a result of this, I added the Menu and How to Play, including an interactive link to a webpage with a very good tutorial on how to play chess. In addition, I added a small README file in the hopes that some users would read that as well, but in the understanding that that was unlikely, hence the need for the How to Play screen.

Graphics: This has 4 helper methods to, once again, make the code more readable and therefore more maintainable. These methods are to draw the board and pieces, to highlight squares that a selected piece can move to and to create a square so the user can select a piece to promote a pawn to.

Engine: I was planning to have more in this class, but it just holds the initial board position and the move logs. It also contains a get method for the board, so that, if needed, I could have multiple instances of the board that don’t affect each other without having to create a separate object.

## Iterative Development Test Data

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test No. | What is being tested | Expected Result | Actual Result | How I will test this | Actions Needed |
| 1 | Board Rendering | The board shows on the screen | The board showed correctly on the screen | I will run the program and see if the board appears | Running the program |
| 2 | Piece Rendering | The pieces show on the screen in their correct positions | The pieces showed on the screen in their correct positions | Run the program to check whether the pieces are on the board and in the correct places | Running the program |
| 3 | Colour Customisation | When I press the ‘Ctrl’ button on the left of the keyboard, the dark squares should change colour | The colour changed correctly when I pressed ‘Ctrl’ | Press the ‘Ctrl’ button to see if the colour of the dark squares changes to the correct colour | Press the ‘Ctrl’ button |
| 4 | Movement (Limitless) | I can select a piece and a destination with the mouse, and it should move the selected piece there, removing any piece that was already there if needed | I selected a piece and a destination, and it moved there without any trouble | Move each piece around to random squares. If they move properly, this is working | Move pieces to random squares |
| 5 | Pawn Movement | I should be able to select a pawn, and it should be able to move to a square that complies with the rules of chess | I was able to select a pawn and move it to a square that complied with the rules of chess  Try to move each piece, in turn, to squares they can move to, and to squares they can’t move to | Run the program and select a pawn. Try to move it somewhere it can go, then somewhere it can’t | Selection of a pawn and a square it can move to, then one it can’t move to |
| 6 | Added Piece Movement (ignoring check and pins) | All the pieces need to move in the correct way and to the correct squares | All the pieces moved to the right places (pins and check were ignored) |  | Selection of each piece and squares they can and can’t move to, respectively. Testing both valid and erroneous data |
| 7 | Finding the King’s Location | The program should return the location (coordinates) of the King, respective of whose move it is | The program correctly returned the coordinates | Print the returned coordinates from the function. If the coordinates match the Kings’ locations ((0, 4) for black and (7, 4) for white), it works | Add a print statement to return the coordinates, run the program to see if the starting coordinates are correct, then move the Kings to see if the new coordinates are also correct |
| 8 | Pawn Checks | If the King is in check from a pawn, remove all moves that leave the King in check and leave any that don’t | Checks from a Pawn were correctly recognised | See if the game correctly recognises a pawn check | Alter the board array so the King is easy to put in check from a pawn, then run the game and see if it is recognised. Test non-check pawn moves to see if they are flagged too |
| 9 | Knight Checks | If the King is in check from a Knight, remove all moves that leave the King in check and leave any that don’t | Checks from a Knight were correctly recognised | See if the game correctly recognises a knight check | Alter the board array so the King is easy to put in check from a Knight, then run the game and see if it is recognised. Test non-check knight moves to see if they are flagged too |
| 10 | Rook and Queen Checks | If the King is in check from a Rook or a Queen, remove all moves that leave the King in check and leave any that don’t | Checks from a Rook or Queen were correctly recognised | See if the game correctly recognises a rook check or a horizontal/vertical check from a queen | Alter the board array so the King is easy to put in check from a rook or queen, then run the game and see if it is recognised. Test non-check moves to see if they are flagged too |
| 11 | Bishop and Queen Checks | If the King is in check from a Bishop or a Queen, remove all moves that leave the King in check and leave any that don’t | Checks from a Bishop or Queen were correctly recognised | See if the game correctly recognises a bishop check or a diagonal check from a queen | Alter the board array so the King is easy to put in check from a bishop or queen, then run the game and see if it is recognised. Test non-check moves to see if they are flagged too |
| 12 | Making Escaping Check Mandatory | Now that the program understands when the King is in check, it needs to be able to narrow down the moves to the ones which allow it to escape check | All illegal moves were removed for each move and all legal moves were kept | Put the King in check from a variety of pieces and check the moves array to see if the only available moves are ones that get the King out of check | I will add a print statement to output the array for moves, and I can check it to ensure that there are no illegal moves in it |
| 13 | Adding Checkmate and Stalemate | The game will recognise either a side winning or a draw by stalemate | Both checkmate and stalemate were recognised and playable | Play through game to get a variety of checkmates and stalemates. If they are all correctly recognised, it works | Play through several games to get to checkmate and to get to stalemate. The relevant message will appear if it is recognised |
| 14 | RankFile Notation Conversion | The latest move in the movelog variable will be converted to chess notation and stored in a separate array | The conversion was done successfully. Castling and Promotion will be added when they are coded in | I’ll check the move log, and the chess notation move log and see if they match up | Print both move logs and ensure that they match |
| 15 | Move Highlighting | When a piece is clicked, its possible moves will be shown on screen | Upon being clicked, the potential moves were highlighted with a red outline | When a piece is selected, the squares it can move to should be highlighted | Run the program and select each piece in turn. Then make some moves and make sure that the correct squares are still highlighted |
| 16 | Stopping the King Moving into Check by Taking a Piece | The piece movement functions had to be edited so that removing illegal moves for the King included being unable to take a piece that is defended | The King can no longer take defended enemy pieces | I added this step during the development, when I realised that, because I was reusing the same algorithms, the King’s moves weren’t being amended to prevent it taking a defended enemy piece. I altered them so that, if the Kings moves are being checked, taking a piece, and moving into check is removed | Put the King in check from a defended piece that is one square away from the King. I can then select the King to see if taking the piece is highlighted as an option. I can then do the same for an undefended piece |
| 17 | Castling | The King should be able to castle following the general rules of the move | Castling was added successfully, and the moves were correctly added to the move log | Get to a position in which castling is possible and try to castle. Also, get to positions in which castling would be possible, but one condition isn’t met to test that it all works | Play the game so that it gets to any necessary position to ensure that castling is only permitted where it should be |
| 18 | Pawn Promotion | Add the functionality so that a pawn at the end of the board can become a piece of the player’s choosing | The pawn promotion algorithm worked and allowed the pawn to change into any given piece (within the limits needed) | Get to a position where a pawn can promote and try changing it, in turn, to each piece (Bishop, Knight, Rook, Queen) | Alter the board array so that a pawn can promote next move. Then I can try promoting it to each piece |
| 19 | En Passant | Allow for the En Passant rule when a pawn moves two squares | En Passant got added to the moves array and is playable | Ensure that en passant is only playable when an enemy pawn has just moved two squares and is horizontally adjacent to a friendly pawn, but is only playable for that pawn | Play through so that en passant is playable, then test if it works |
| 20 | Menu Screen | I added a menu screen so that a ‘How to Play’ screen can be added later | The menu screen worked, the play button took the player to the game and the How to Play button will be coded in due course | Check that the Menu Screen is the first thing to appear upon running the program, and that the buttons work | Run the program to check it runs as expected. Check each button in turn to make sure that they do what is needed |
| 21 | How to Play Screen | The ‘How to Play’ button was coded to take the user to an instructions screen | The button and screen worked correctly and included an interactive link to a site where the user can learn chess | Make sure that the Menu and How-to-Play screens can go back and forth smoothly, and that the link on ‘How to Play’ works correctly | Run the program and press the ‘How to Play’ button, then ESC to go back to the Main Menu. After that, I can go back to the Instructions screen and click the link to check that that works too |
|  |  |  |  |  |  |

## Post-Development Testing

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Num | Test Subject | How to test | Good/Bad Boundary | Expected Outcome | Actual Outcome | Actions Needed |
| 1 | Game Enjoyability | Survey random people who have used my chess engine | >70% Positive Response | Around the 70-75% mark | 85% gave positive feedback | None |
| 2 | Controls | Watching testers (some with and some without prior chess knowledge) play | Any questions asked about how to play aren’t repeated and the user feels comfortable playing after <10 minutes | An average of 8-12 minutes.  If it’s above 10, I’ll edit the ‘How to Play’ to make it more understandable based on user feedback | The average time to learn the controls was about 8-9 minutes, so I don’t need to alter the Instructions | None |
| 3 | Beginner-Friendly | Get people who know how to play chess but aren’t particularly good to use it and give feedback | >90% of testers pick up how to play within 10 minutes and find it easy to use | People will find it easy to use, and the move highlights should make it easier to spot moves | Beginners understood the square highlighting system and found it a lot easier to use than most engines | None |

# C: Developing the coded solution

## Development Process

### Setting Up the Screen – **V1.1**

A green rectangular object with a white border

Description automatically generated

This was my first step. I created the Main class to hold the main sequencing for the program and to create the order for the events to happen in. I decided to take an Object-Oriented approach to the program so that there is a definite structure throughout, I can access my various modules quickly, and I can access variables in one file from another easily.

### Adding the Board Array

A green and white square with dots

Description automatically generated

The next step was to start creating some of the main variables throughout the program. I called this file ‘Vars,’ though it would later be changed to ‘chess\_engine.’ I made the self.board variable as a 2D list (I messed up slightly here and made it 1 dimensional by accident) so that it can be altered after each move, and so that it has a row/column structure to it to make it easy to access each square on the board. self.whiteToMove determines whose move it is. If it is true, it is white’s turn to move, otherwise it is black’s.

### Adding the Piece Files and Drawing the Board

main.py

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

vars.py



I made the new vars file, after moving the previous contents to chess\_engine.py. Although you can’t make a constant in Python, I moved the general constants to a separate file where I can manually change them.

I added the draw\_squares function in order to give myself the basis to graphically test my project iteratively so that it is easier for me to spot where I need to improve. It is also the foundations for my GUI, so the user will know where the pieces are, it makes it easier for them to calculate and it provides an interactive platform for the user to play the game.

### Drawing the Pieces

main.py

A screenshot of a computer program

Description automatically generated

chess\_engine.py (formerly vars.py)

A screenshot of a computer program

Description automatically generated

I had a bug with the board array (as seen above) where I made it a 1-dimensional list by accident, so I changed that to 2 dimensional here.

I also coded the algorithm to draw the pieces on the screen (they are not currently moveable). This was achieved by iterating through the board and drawing the pieces in their respective squares based on the pseudonyms given to them in the pieces array in main.py. The board’s 8x8 structure gives me easy access to the pieces’ positions, as well as accessing any square I need on the board. This means that rendering the pieces is made much easier as a result. Testing means that I can ensure that everything is in its correct place at the start.

### Moving the Pieces (No Move Generation Yet)

main.py

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

I added piece movement by assigning the mouse position tuple to a variable and using the coordinates to work out what, if anything, was clicked. Later, I’ll extend this to check if the desired move is a legal move (involving castling, en passant and pawn promotion). This is the basic structure to alter the board array as required and therefore move the positions of the pieces on screen and will be built on in the future. This is the graphical foundation for the movement of the pieces on screen and therefore is the foundation for the frontend of the program.

### Setting up the File for Move Generation

moves.py

A computer screen shot of a program code

Description automatically generated

A screenshot of a computer program

Description automatically generated

My next step was to begin to put together the function structure for move generation. The code above shows the ‘getAllPossibleMoves’ function, which iterates through the board array and, upon finding a piece, will run the function associated with it. The point of this is to generate all the potentially possible moves before check and pins are considered. It just gets any moves that could be made by the pieces following the basic movement rules of chess.

### Finding all the Possible Pawn Moves

moves.py

A computer screen shot of a program

AI-generated content may be incorrect.

main.py

A screenshot of a computer program

AI-generated content may be incorrect.

The next thing I did was to begin to generate the moves (still without checks and pins). This began with the pawns, where I checked the two squares in front of it. For efficiency, I only checked two squares in front when the square directly in front was clear and the pawn was on its starting rank. I also checked the squares one forward and to each side to see if a piece can be captured that turn. This is the beginning of the true move generation, so that now the pawns can move to the squares they should be able to move to. Although I considered implementing pawn promotion and en passant at this stage, I decided it was probably too early to add them and would mean I’d be testing too much at once, so I’ll add those later. I count this as a problem in its own right, separate to the moves of the other pieces, because of the unique mechanics of the pawn’s movement. Therefore, it requires a different approach to that of the other pieces.

### Adding the Moves for All Other Pieces

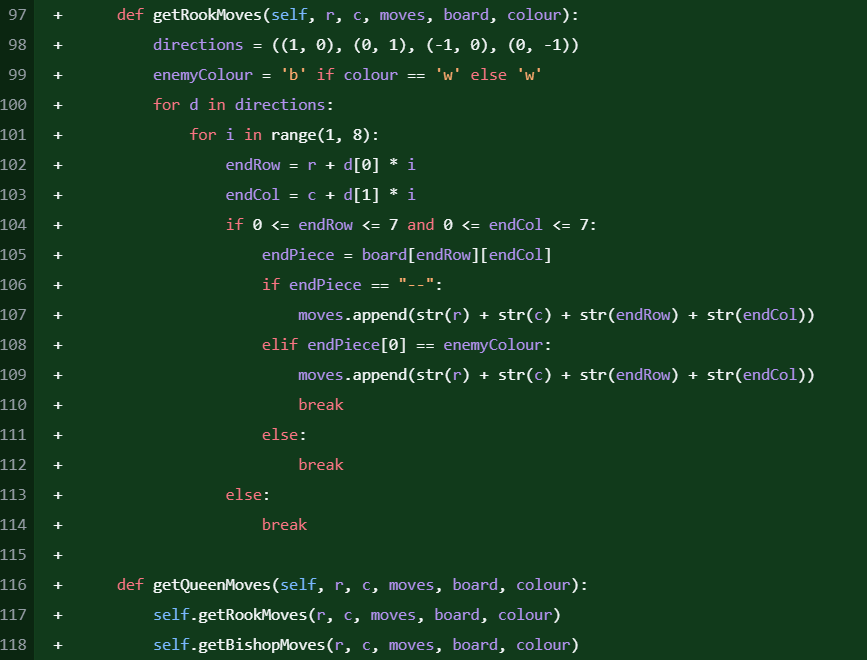
moves.py

A computer screen with text on it

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.



A computer screen shot of a program code

AI-generated content may be incorrect.

I added the possible moves for all the other pieces to the game so that now random moves can’t be made, and the user must adhere to the rules of chess.

Each helper function ensured that only legal moves (discounting checks and pins) were generated by checking positions on the board and avoiding friendly pieces. The implementation of these functions provided the foundation for refined move validation in the following stages.

### Tracking the King’s Location

moves.py

A computer screen shot of a program

AI-generated content may be incorrect.

For a chess engine, knowing the whereabouts of each King on the board is paramount. Therefore, I added this helper function to the Moves class to assist with legal move generation and with checks and pins in the future. The function is optimised for efficiency as it stops searching once the required King has been found.

### Adding Checks from the Knight AND Pawn

moves.py

A computer screen shot of a program

AI-generated content may be incorrect. A computer code on a green background

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.

A computer code on a green background

AI-generated content may be incorrect.

At this stage, the program now filtered the moves to only include possible ones following the basic movement rules of chess. Therefore, the next natural step in the process was to filter out the illegal moves from the legal moves. In relation to finding checks from other pieces, the Knights and Pawns were fairly straightforward. For the knight, I used another directions array and looped through it, checking each relevant square as it goes by using the King’s location and going out by the relevant squares (as long as the checked square doesn’t go off the board). For the pawns, I checked the two squares diagonal from the King, nearer to the enemy side. If there is a pawn on either of them, the King is in check. This resulted in the moves array beginning to be filtered to only include legal moves. However, knights and pawns can’t pin pieces, so the others are going to be much more difficult to implement.

### Rook/Queen (Horizontal and Vertical Only) Checks and Pins

 A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

Next, I added the Rook and the horizontal and vertical Queen checks. To do this, I put the rank and file that the King is stood on into different arrays and iterated outwards. If an enemy Rook or Queen is found on any of them, the checks array is updated. If an ally piece is found, and then a hostile Rook or Queen is found, the pins array is updated.

### Adding Bishop/Queen (Diagonal Only) CHecks

moves.py

A computer screen shot of a program code

AI-generated content may be incorrect.A screen shot of a computer code

AI-generated content may be incorrect.



I added the Bishop/Queen checks in a near-identical way to the way in which I added the Rook/Queen checks. The general algorithm is the same because the program needs to do the same thing, just different directions, hence why the values in the directions array are different.

### Making Escaping Check Mandatory

A screen shot of a computer program

AI-generated content may be incorrect.A computer screen shot of a program

AI-generated content may be incorrect.

A computer screen shot of text

AI-generated content may be incorrect.A computer screen shot of a program code

AI-generated content may be incorrect.

A green square with white lines

AI-generated content may be incorrect.

To ensure that the king can legally escape check, the program must analyse all the possible moves and evaluate which ones successfully move the King out of check. The algorithm first calls getAllPossibleMoves to generate the potential moves (legal and illegal). These are then reduced by the searchForPinsAndChecks algorithm to purely the legal options.

For a single check, these valid moves include moving the king to a safe square, capturing the attacking piece, or blocking the check with another piece. The algorithm constructs a list of all squares between the checking piece and the king, allowing moves that place a piece on these squares. Additionally, any legal king moves to squares not controlled by the opponent remain valid.

For a double check, the only feasible moves that remove the check from both pieces are King moves to squares that aren’t controlled by any hostile pieces, since blocking the attack or removing an attacker from the board aren’t feasible options as there would still be one attacker left. All other moves are removed from the list.

These prevent the player from making any move that does not completely remove the King from the line of attack of all hostile pieces.

### Making Pins Effective

A computer screen shot of a program code

AI-generated content may be incorrect.A green square with a black border

AI-generated content may be incorrect.

After having made check effective, my next step is to do the same for pins. Because I already had the array containing all the pins in the position for whichever side was moving, I just had to work out which piece was pinned from each value in the list and where they could and couldn’t move to. For this, I created a new instance of the board and ran ‘getAllPossibleMoves’ for each opponent move. Although this is an inefficient way of doing this, the game broke if I tried it any other way. If any opponent piece could take the King after a move, that move is classed as illegal and is removed from the list. This is only done for a move by a supposedly pinned piece to increase efficiency as much as possible.

### Adding Move Highlighting

A green screen with white text

AI-generated content may be incorrect.

To improve accessibility for newer players (the main purpose behind the chess engine), a move highlighting feature was implemented. When a player selects a piece, all of its legal moves are shown on the board by highlighting the squares that a selected piece can move to. This helps players quickly understand possible move options, significantly reducing blunders. The highlighting system updates whenever pieces are selected or deselected or when a move is made. By integrating this feature, the chess engine enhances usability for the target audience, so they can learn valid piece movements.

### King Move Debug

A computer screen shot of text

AI-generated content may be incorrect.

After testing my code thus far, I noticed that the King could take a defended piece to get out of check. I realised that this was because, when I ran getAllPossibleMoves, it was coded to stop if a friendly piece was reached, which was what was happening when I registered all the opponent’s moves. As a result, I added the above code to getKingMoves. It tries every potential King move and then runs ‘getAllPossibleMoves’ to see if it would endanger the King’s life. This can be carried over to castling in a later step.

### Castling

main.py

A screen shot of a computer program

AI-generated content may be incorrect.A green screen with white text

AI-generated content may be incorrect.

moves.py

A computer screen shot of a program

AI-generated content may be incorrect.

Castling is the first of the three ‘Special moves’ in chess. This was solvable on its own because, although this would need to be linked to the rest of the Moves class, it doesn’t need to have had any more code written beforehand. It is quite difficult to code in comparison to other moves because it has so many requirements, such as not having moved the King or the Rook that is being included in the castle. After the inclusion of this, pawn promotion and en passant, the first prototype will be completed

### Pawn Promotion

main.py

A computer screen shot of a program code

AI-generated content may be incorrect.A computer screen with white text

AI-generated content may be incorrect.A computer code on a green background

AI-generated content may be incorrect.

graphics.py

A screenshot of a computer program

AI-generated content may be incorrect.

The second of the three special moves is pawn promotion. If a pawn makes it to the end of the board, it can become a rook, bishop, knight, or queen. In this case, the graphics.py file draws a square on the board where the player can pick a piece to promote to. Then, main.py checks where was clicked (respective of which square the pawn promoted on), and the board array gets altered accordingly.

### En Passant

main.py

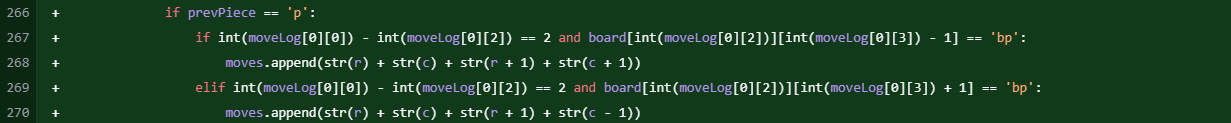
A computer screen with white text

AI-generated content may be incorrect.

moves.py

A screenshot of a computer program

AI-generated content may be incorrect.



En Passant was implemented by recording the last move and checking it against certain parameters. These being that the previous move was that a pawn had moved two squares forward on the previous turn and that a friendly pawn was adjacent to it. This rule ensures adherence to standard chess gameplay by preventing pawns from bypassing potential captures.

This is the completion of **V1.1** (Prototype 1)

### Pawn Promotion Graphics Glitch – **V1.2**

chess\_main

A screenshot of a computer program

AI-generated content may be incorrect.

graphics

A screenshot of a computer program

AI-generated content may be incorrect.

My first improvement to Prototype 1 was when I noticed that, when a pawn was promoted, the pawn was still visible behind where the user chose what piece to promote to. Although this wasn’t a major issue with gameplay, I decided it would look better if the pawn was covered, and the promotion choice stood out better against it.

### Castling Out of Check Debug

moves

A screenshot of a computer program

AI-generated content may be incorrect.

When I sent the first prototype to some users to test, they also noticed that I had neglected to tell the game that the King couldn’t castle if in check. This was remedied here by passing the Boolean flag of whether the King is in check into the castling function and telling the engine to keep the castling moves out of the array if the King is in check.

### Castling Through/Into Check Bug Fix

moves

A screenshot of a computer program

AI-generated content may be incorrect.

Another issue found by the client was that the King was able to castle into check or castle through an attacked square. This was sorted by passing the all the enemy’s possible moves into the castling method. These were iterated through and, if an enemy piece could move to one of the squares required for castling, castling was made unavailable for that move.

This marked the completion of **V1.2**

### Menu Screen – **V2.1**

chess\_main

A green screen with white text

AI-generated content may be incorrect.

intro\_screen

A screen shot of a computer program

AI-generated content may be incorrect.A computer screen shot of a program code

AI-generated content may be incorrect.

After sending V1.2 off for testing, the main request I had was to add a menu and a how to play screen. As a result, I created a new file for these mew screens and have so far added the main menu. It’s a simple layout which has the game’s logo at the top and ‘Play’ and ‘How to Play’ buttons.

### Telling People Whose Turn it is

chess\_main

A screen shot of a computer screen

AI-generated content may be incorrect.

Because this chess engine is designed around being beginner friendly, I added a feature showing whose move it is at the top of the screen during the game. This should make it easier for new players to know whose turn it is, since it now tells them on the screen

### How To Play Screen

intro\_screen

A computer screen shot of a program code

AI-generated content may be incorrect.

The How to Play Screen is made so that, on screen, it explains how to use the chess engine, and at the top, there is a link to a website explaining how to play chess. This is so that people who aren’t as familiar with chess engines or the game itself can get a good understanding of both the game and the software.

This marks the completion of **V2.1**

### Draw by Insufficient Material – **V2.2**

chess\_main

A computer screen shot of a program code

AI-generated content may be incorrect.A computer screen with text and numbers

AI-generated content may be incorrect.

The last thing I needed to implement was the ‘Draw by Insufficient Material’ condition. This is necessary because, if one side has a King and two Knights or one of a few other combinations, they can’t checkmate the opponents. If both side has this issue, the game can’t be won, so it is a draw. Therefore, I added code to count how many pieces were left and work out which pieces to see if they fitted one of a few specific conditions, as shown above.

## Code Modules

### Main

A screen shot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.

### Moves

A screen shot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.A screen shot of a computer

AI-generated content may be incorrect.

### Intro Screen

A screen shot of a computer program

AI-generated content may be incorrect.

### Graphics

A computer code on a black background

AI-generated content may be incorrect.

### Engine

(Misnomer as it just holds the initial board and move-logs)

A screen shot of a computer program

AI-generated content may be incorrect.

### Vars

(File for Constants)

A black and brown background with red and white text

AI-generated content may be incorrect.

### Colours

A screenshot of a computer

AI-generated content may be incorrect.

## Variables/Structure Names

I used a constant style throughout my code with the camelCase naming convention for all variables, with their names showing the use for each one to ensure maintainability. This was the case for all variables except any counter variables used in loops, such as ‘r’ and ‘c,’ which were abbreviations of ‘row’ and ‘column’ respectively, and ‘d’ which is a direction variable (i.e. (0, 1)).

## Validation

The main area for input validation within my code is Pygame Event Handling. This includes Pygame checking for a mouse click or keyboard input mostly, as well as it checking for the Quit button.

# D. Testing

## Developmental Testing

For the actual testing, please see the videos in the ‘Testing’ folder under ‘Write Up’

|  |  |  |
| --- | --- | --- |
| Video Num | Description | Outcome (S/F) |
| 001 | I tried to get the board to appear on-screen by using modular arithmetic to determine if each square should be light or dark so that the users would have a graphical interface on which to play the game. This failed because it couldn’t locate the PNG file for the White Rook. Although this doesn’t directly hinder the board’s design, I wanted to get the images sorted early so I didn’t have to load them in later in the program. | F |
| 002 | After altering the file structure of the software, the PNG had now been found. The modular arithmetic was the wrong way round initially (not shown in the video), but I corrected it, so the squares appeared in the correct places. | S |
| 003 | I got the pieces to appear on the screen in their correct positions by looping through a 2-Dimensional array and using the constants in my vars file to work out the on-screen coordinates for each piece’s position. | S |
| 004 | Using Pygame’s KEYDOWN event, I told the game to change the colour of the dark squares when the left control button is pressed by looping through an array of denary colour codes. There are 12 colours available. | S |
| 005 | By adding variables to register the number of clicks made and the squares selected, the pieces are able to move around the board. Although there is currently no noticeable difference between legal and illegal moves, this is the first step towards move generation, which is the basis of the entire game. | S |
| 006 | Using a function called ‘getAllPossibleMoves’, I collected all the potential moves for any piece anywhere on the board. I then wrote the function for the pawn moves and attempted to add the pawn’s moves to an array. The moves couldn’t be played because the data type that the moves were being added to the array in differed from that of the comparisons when moves were being checked, so it didn’t think any of the moves on the board were legal. | F |
| 007 | The data type within the if statement has been altered to match the data type used within the ‘squares’ array (holds the coordinates of the selected squares), meaning that the values held in the validMoves array now work and the pawns can move normally (checks and pins are currently discounted)  However, during this test I noticed that using the undo function kept the board as it was and added a piece to the board were the previously moved piece was. This was due to a bug with the board array. The undo function is purely for testing, instead of a function within the game as someone who realises that they made a mistake may use the function to their advantage | S |
| 008 | I added a bit of code to the Undo function to update the board properly and then it worked as it should. As stated above, this function is purely for testing purposes as it could cause unfairness in games if someone runs a game back if they’re losing | S |
| 009 | Using the same framework as for pawns (getAllPossibleMoves + Helper Methods), I added the functions defining the parameters of the movement of the other pieces on the board. This involved using a directions array and looping through the array and, for some pieces, checking each square in a direction until another piece is found or the edge of the board is reached | S |
| 010 | The main prerequisite for finding checks and pins is to find the King’s location at any given time. Therefore, within the Moves class, I added a function that determines the row and column that the King is currently stood on to give a baseline for the future checks and pins | S |
| 011 | The easiest piece to look for checks from are, by far, pawns. Therefore I started with them. The checks use a similar framework to finding the possible moves. ‘searchForPinsAndChecks’ compiles a list of the checks and a list of the pins against one side in any given position | S |
| 012 | After having done the pawns, the next task was to find the Knight’s checks. This was accomplished through | S |
| 013 |  | F |
| 014 |  | S |
| 015 |  | S |
| 016 |  | S |
| 017 |  | S |
| 018 |  | F |
| 019 |  | F |
| 020 |  | S |
| 021 |  | S |
| 022 |  | S |
| 023 |  | S |
| 024 |  | F |
| 025 |  | S |
| 026 |  | S |
| 027 |  | S |
| 028 |  | S |

## Evaluative Testing

# E. Evaluation

<See H446-03 Project Advice Booklet for help and guidance of what must go here.>

# Project Appendixes

Insert as many project appendixes as you need for your project.

These might include, but are not limited to:

* Complete Code Listing (ESSENTIAL)
* Interview Transcripts
* Meeting notes
* Observation notes or questionnaires

# References