

NEA

Toby Mok

<b>1 Analysis</b>	<b>2</b>
1.1 Background . . . . .	2
1.1.1 Game Description . . . . .	2
1.1.2 Current Solutions . . . . .	3
1.1.3 Client Interview . . . . .	4
1.2 Objectives . . . . .	5
1.2.1 Client Objectives . . . . .	5
1.2.2 Other User Considerations . . . . .	7
1.3 Research . . . . .	7
1.3.1 Board Representation . . . . .	7
1.3.2 CPU techniques . . . . .	8
1.3.3 GUI framework . . . . .	9
1.4 Proposed Solution . . . . .	9
1.4.1 Language . . . . .	9
1.4.2 Development Environment . . . . .	10
1.4.3 Source Control . . . . .	11
1.4.4 Techniques . . . . .	11
1.5 Limitations . . . . .	12
1.6 Critical Path Design . . . . .	12
<b>2 Design</b>	<b>14</b>
2.1 System Architecture . . . . .	14
2.1.1 Main Menu . . . . .	15
2.1.2 Settings . . . . .	15
2.1.3 Past Games Browser . . . . .	17
2.1.4 Config . . . . .	18
2.1.5 Game . . . . .	19
2.1.6 Board Editor . . . . .	20
2.2 Algorithms and Techniques . . . . .	21
2.2.1 Minimax . . . . .	21
2.2.2 Minimax improvements . . . . .	22
2.2.3 Board Representation . . . . .	26
2.2.4 Evaluation Function . . . . .	30
2.2.5 Shadow Mapping . . . . .	33
2.2.6 Multithreading . . . . .	36
2.3 Data Structures . . . . .	36
2.3.1 Database . . . . .	36
2.3.2 Linked Lists . . . . .	38
2.3.3 Stack . . . . .	39
2.4 Classes . . . . .	40
2.4.1 Class Diagram . . . . .	44
<b>3 Technical Solution</b>	<b>45</b>
3.1 File Tree Diagram . . . . .	46
3.2 Summary of Complexity . . . . .	47
3.3 Overview . . . . .	47
3.3.1 Main . . . . .	47
3.3.2 Loading Screen . . . . .	48
3.3.3 Helper functions . . . . .	50

3.3.4	Theme . . . . .	58
3.4	GUI . . . . .	59
3.4.1	Laser . . . . .	59
3.4.2	Particles . . . . .	62
3.4.3	Widget Bases . . . . .	65
3.4.4	Widgets . . . . .	74
3.5	Game . . . . .	86
3.5.1	Model . . . . .	86
3.5.2	View . . . . .	90
3.5.3	Controller . . . . .	96
3.5.4	Board . . . . .	101
3.5.5	Bitboards . . . . .	106
3.6	CPU . . . . .	112
3.6.1	Minimax . . . . .	112
3.6.2	Alpha-beta Pruning . . . . .	114
3.6.3	Transposition Table . . . . .	115
3.6.4	Evaluator . . . . .	117
3.6.5	Multithreading . . . . .	119
3.6.6	Zobrist Hashing . . . . .	120
3.6.7	Cache . . . . .	121
3.7	States . . . . .	123
3.7.1	Review . . . . .	123
3.8	Database . . . . .	128
3.8.1	DDL . . . . .	128
3.8.2	DML . . . . .	130
3.9	Shaders . . . . .	133
3.9.1	Shader Manager . . . . .	133
3.9.2	Bloom . . . . .	138
3.9.3	Rays . . . . .	141
3.9.4	Stack . . . . .	145
<b>4</b>	<b>Testing</b> . . . . .	<b>146</b>
4.1	Iterative Testing . . . . .	146
4.1.1	Minimax . . . . .	146
4.1.2	Migrations . . . . .	147
4.2	Unit Tests . . . . .	147
4.2.1	Board Evaluator . . . . .	147
4.2.2	CPU . . . . .	147
4.2.3	Shadow Mapping . . . . .	148
4.3	Final Tests . . . . .	148
4.3.1	Objective 1 . . . . .	148
4.3.2	Objective 2 . . . . .	148
4.3.3	Objective 3 . . . . .	148
4.4	Videos . . . . .	148

# Chapter 1

## Analysis

### 1.1 Background

Mr Myslov is a teacher at Tonbridge School, and currently runs the school chess club. Seldomly, a field day event will be held, in which the club convenes together, playing a chess, or another variant, tournament. This year, Mr Myslov has decided to instead, hold a tournament around another board game, namely laser chess, providing a deviation yet retaining the same familiarity of chess. However, multiple physical sets of laser chess have to be purchased for the entire club to play simultaneously, which is difficult due to it no longer being manufactured. Thus, I have proposed a solution by creating a digital version of the game.

#### 1.1.1 Game Description

Laser Chess is an abstract strategy game played between two opponents. The game differs from regular chess, involving a 10x8 playing board arranged in a predefined condition. The aim of the game is to position your pieces such that your laser beam strikes the opponents Pharoah (the equivalent of a king). Pieces include:

1. Pharoah
  - Equivalent to the king in chess
2. Scarab
  - 2 for each colour
  - Contains dual-sided mirrors, capable of reflecting a laser from any direction
  - Can move into an occupied adjacent square, by swapping positions with the piece on it (even with an opponent's piece)
3. Pyramid
  - 7 for each colour
  - Contains a diagonal mirror used to direct the laser
  - The other 3 out of 4 sides are vulnerable from being hit
4. Anubis

- 2 for each colour
- Large pillar with one mirrored side, vulnerable from the other sides

### 5. Sphinx

- 1 for each colour
- Piece from which the laser is shot from
- Cannot be moved

On each turn, a player may move a piece one square in any direction (similar to the king in regular chess), or rotate a piece clockwise or anticlockwise by 90 degrees. After their move, the laser will automatically be fired. It should be noted that a player's own pieces can also be hit by their own laser. As in chess, a three-fold repetition results in a draw. Players may also choose to forfeit or offer a draw.

#### 1.1.2 Current Solutions

Current free implementations of laser chess that are playable online are limited, seemingly only available on <https://laser-chess.com/>.

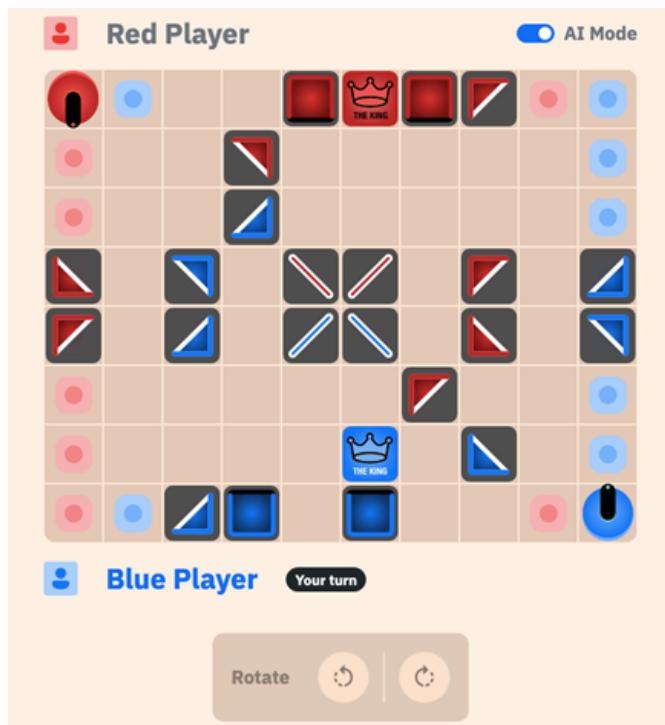


Figure 1.1: Online implementation on laser-chess.com

The game is hosted online and is responsive and visually appealing, with pieces easy to differentiate and displaying their functionality clearly. It also contains a two-player mode for playing between friends, or an option to play against a functional CPU bot. However, the game lacks the following basic functionalities that makes it unsuitable for my client's requests:

- No replay options (going through past moves)
  - A feature to look through previous moves is common in digital board game implementations
  - My client requires this feature as it is an essential tool for learning from past games and to aid in analysing past games
- No option to save and load previous games
  - This QOL feature allows games to be continued on if they cannot be finished in one sitting, and to keep an archive of past games
- Internet connection required
  - My client has specifically requested an offline version as the game will predominantly be played in settings where a connection might not be available (i.e. on a plane or the maths dungeons)
- Unable to change board configuration
  - Most versions of laser chess (i.e. Khet) contain different starting board configurations, each offering a different style of play

Our design will aim to append the missing feature from this website while learning from their commendable UI design.

### 1.1.3 Client Interview

**Q:** Why have you chosen Laser Chess as your request?

**A:** Everyone is familiar with chess, so choosing a game that feels similar, and requires the same thinking process and calculations was important to me. Laser chess fit the requirements, but also provides a different experience in that the new way pieces behave have to be learnt and adapted to. It hopefully will be more fun and a better fit for the boys than other variants such as Othello, as the laser aspects and visuals will keep it stimulating.

*Objectives 1 & 7*

Implementing laser chess in a style similar to normal chess will be important. The client also requests for it to be stimulating, requiring both proper gameplay and custom visuals.

**Q:** Have you explored any alternatives?

**A:** I remember Laser Chess was pretty popular years ago, but now it's harder to find a good implementation I can use, since I don't plan on buying multiple physical copies or paid online copies for every student. I have seen a few free websites offering a decent option, but I'm worried that with the terrible connection in the basement will prove unreliable if everybody tries to connect at once. However, I did find the ease-of-use and simple visuals of some websites pleasing, and something that I wish for in the final product as well.

*Objective 6*

The client's limitations call for a digital implementation that plays offline. Taking inspiration from alternatives, a user-friendly GUI is also expected.

**Q:** What features are you looking for in the final product?

**A:** I'm looking for most features chess websites like Chess.com or Lichess offer, a smooth playing experience with no noticeable bugs. I'm also expecting other features such as having a functional timer, being able to draw and resign, as these are important considerations in our everyday chess games too. Since this will be a digital game, I think having handy features such as indicators for moves and audio cues will also make it more user-friendly and enjoyable. If not for myself, having the option to play against a computer bot will be appreciated as well, since I'll be able to play during lesson time, or in the case of odd numbers in the tournament. All in all, I'd be happy with a final product that plays Laser Chess, but emulates the playing experience of any chess website well.

*Objectives 1 & 3 & 5*

Gameplay similar to that of popular chess websites is important to our client, introducing the requirement of subtle features such as move highlighting. A CPU bot is also important to our client, who enjoys thinking deeply and analysing chess games, and so will prove important both as a learning tool and as an opponent.

---

**Q:** Are there any additional features that might be helpful for your tournament use-cases?

**A:** Being able to configure the gameplay will be useful for setting custom time-controls for everybody. I also would like to archive games and share everybody's matches with the team, so having the functionality to save games, and to go through previous ones, will be highly requested too. Being able to quickly setup board starting positions or share them will also be useful, as this will allow more variety into the tournament and give the stronger players some more interesting options.

*Objectives 2 & 4*

Saving games and customising them is a big logistical priority for a tournament, as this will provide the means to record games and for opponents to all agree on the starting conditions, depending on the circumstances of the tournament.

## 1.2 Objectives

### 1.2.1 Client Objectives

The following objectives should be met to satisfy my clients' requirements:

1. All laser chess game logic should be properly implemented
  - All pieces should display correct behaviour (e.g. reflecting the laser in the correct direction)
  - Option to rotate laser chess pieces should be implemented
  - Pieces should be automatically detected and eliminated when hit by the laser
  - Game should allocate alternating player's turns
  - Players should be able to move to an available square when it is their turn
  - Game should automatically detect when a player has lost or won
  - Three-fold repetition should be automatically detected
  - Travel path of laser should be correctly implemented
2. Save or load game options should be implemented

- Games will be encoded into FEN string format
- Games can be saved locally into the program files
- NOT IMPLEMENTED Players can load positions of previous games and continue playing
- Players should be able to scroll through previous moves

3. Other board game requirements should be implemented

- Timer displaying time left for each player should be displayed
- Time logic should be implemented, pausing when it is the opponent's turn, forfeiting players who run out of time
- Forfeiting should be made available
- Draws should be made available

4. Game settings and config should be customisable

- Piece and board colour should be customisable
- Option to play CPU or another player should be implemented
- Starting player turn and board layout should be customisable
- Timer and duration should be customisable

5. CPU player

- CPU player should be functional and display an adequate level of play
- CPU should be within an adequate timeframe (e.g. 5 seconds)
- CPU should be functional regardless of starting board position

6. Game UI should improve player experience

- Selected pieces should be clearly marked with an indicator
- Indicator showing available squares to move to when clicking on a piece
- Destroying a piece should display a visual and audio cue
- Captured pieces should be displayed for each player
- Status message should display current status of the game (whose turn it is, move a piece, game won etc.)

7. GUI design should be functional and display concise information

- GUI should always remain responsive throughout the running of the program
- Application should be divided into separate sections with their own menus and functionality (e.g. title page, settings)
- Navigation buttons (e.g. return to menu) should concisely display their functionality
- UI should be designed for clarity in mind and visually pleasing
- Application should be responsive, draggable and resizable

### 1.2.2 Other User Considerations

Although my current primary client is Mr Myslov, I aim to make my program shareable and accessible, so other parties who would like to try laser chess can access a comprehensive implementation of the game, which currently is not readily available online. Additionally, the code should be concise and well commented, complemented by proper documentation, so other parties can edit and implement additional features such as multiplayer to their own liking.

## 1.3 Research

Before proceeding with the actual implementation of the game, I will have to conduct research to plan out the fundamental architecture of the game. Reading on available information online, prior research will prevent me from committing unnecessary time to potentially inadequate ideas or code. I will consider the following areas: board representation, CPU techniques and GUI framework.

### 1.3.1 Board Representation

Board representation is the use of a data structure to represent the state of all pieces on the chessboard, and the state of the game itself, at any moment. It is the foundation on which other aspects such as move generation and the evaluation function are built upon, with different methods of implementation having their own advantages and disadvantages on simplicity, execution efficiency and memory footprint. Every board representation can be classified into two categories: piece-centric or square-centric. Piece-centric representations involve keeping track of all pieces on the board and their associated position. Conversely, square-centric representations track every available square, and if it is empty or occupied by a piece. The following are descriptions of various board representations with their respective pros and cons.

#### Square list

Square list, a square-centric representation, involves the encoding of each square residing in a separately addressable memory element, usually in the form of an 8x8 two-dimensional array. Each array element would identify which piece, if any, occupies the given square. A common piece encoding could involve using the integers 1 for a pawn, 2 for knight, 3 for bishop, and + and - for white and black respectively (e.g. a white knight would be +2). This representation is easy to understand and implement, and has easy support for multiple chess variants with different board sizes. However, it is computationally inefficient as nested loop commands must be used in frequently called functions, such as finding a piece location. Move generation is also problematic, as each move must be checked to ensure that it does not wrap around the edge of the board.

#### 0x88

0x88, another square-centric representation, is an 128-byte one-dimensional array, equal to the size of two adjacent chessboards. Each square is represented by an integer, with two nibbles used to represent the rank and file of the respective square. For example, the 8-integer 0x42 (0100 0010) would represent the square (4, 2) in zero-based numbering. The advantage of 0x88 is that faster bitwise operations are used for computing piece transformations. For example, add 16 to the current square number to move to the square on the row above, or add 1 to move to the next column. Moreover, 0x88 allows for efficient off-the-board detection. Every valid square number is under 0x88 in hex (0111 0111), and by performing a bitwise AND operation between

the square number and 0x88 (1000 1000), the destination square can be shown to be invalid if the result is non-zero (i.e. contains 1 on 4th or 8th bit).

## Bitboards

Bitboards, a piece-centric representation, are finite sets of 64 elements, one bit per square. To represent the game, one bitboard is needed for each piece-type and colour, stored as an array of bitboards as part of a position object. For example, a player could have a bitboard for white pawns, where a positive bit indicates the presence of the pawn. Bitboards are fast to incrementally update, such as flipping bits at the source and destination positions for a moved piece. Moreover, bitmaps representing static information, such as spaces attacked by each piece type, can be pre-calculated, and retrieved with a single memory fetch at a later time. Additionally, bitboards can operate on all squares in parallel using bitwise operations, notably, a 64-bit CPU can perform all operations on a 64-bit bitboard in one cycle. Bitboards are therefore far more execution efficient than other board representations. However, bitboards are memory-intensive and may be sparse, sometimes only containing a single bit in 64. They require more source code, and are problematic for devices with a limited number of process registers or processor instruction cache.

### 1.3.2 CPU techniques

#### Minimax

Minimax is a backtracking algorithm that evaluates the best move given a certain depth, assuming optimal play by both players. A game tree of possible moves is formulated, until the leaf node reaches a specified depth. Using a heuristic evaluation function, minimax recursively assigns each node an evaluation based on the following rules:

- If the node represents a white move, the node's evaluation is the *maximum* of the evaluation of its children
- If the node represents a black move, the node's evaluation is the *minimum* of the evaluation of its children

Thus, the algorithm *minimizes* the loss involved when the opponent chooses the move that gives *maximum* loss.

Several additional techniques can be implemented to improve upon minimax. For example, transposition tables are large hash tables storing information about previously reached positions and their evaluation. If the same position is reached via a different sequence of moves, the cached evaluation can be retrieved from the table instead of evaluating each child node, greatly reducing the search space of the game tree. Another, such as alpha-beta pruning can be stacked and applied, which eliminates the need to search large portions of the game tree, thereby significantly reducing the computational time.

#### Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) involves playouts, where games are played to its end by selecting random moves. The result of each playout is then backpropagated up the game tree, updating the weight of nodes visited during the playout, meaning the algorithm successively improves at accurately estimating the values of the most promising moves. MCTS periodically evaluates alternatives to the currently perceived optimal move, and could thereby discover a

better, otherwise overlooked, path. Another benefit is that it does not require an explicit evaluation function, as it relies on statistical sampling as opposed to developed theory on the game state. Additionally, MCTS is scalable and may be parallelized, making it suitable for distributed computing or multi-core architectures. However, the rate of tree growth is exponential, requiring huge amounts of memory. In addition, MCTS requires many iterations to be able to reliably decide the most efficient path.

### 1.3.3 GUI framework

#### Pygame

Pygame is an open-source Python module geared for game development. It offers abundant yet simple APIs for drawing sprites and game objects on a screen-canvas, managing user input, audio et cetera. It also has good documentation, an extensive community, and receives regular updates through its community edition. Although it has greater customizability in drawing custom bitmap graphics and control over the mainloop, it lacks built-in support for UI elements such as buttons and sliders, requiring custom implementation. Moreover, it is less efficient, using 2D pixel arrays and the RAM instead of utilising the GPU for batch rendering, being single-threaded, and running on an interpreted language.

#### PyQt

PyQt is the Python binding for Qt, a cross-platform C++ GUI framework. PyQt contains an extensive set of documentation online, complemented by the documentation and forums for its C++ counterpart. Unlike Pygame, PyQt contains many widgets for common UI elements, and support for concurrency within the framework. Another advantage in using PyQt is its development ecosystem, with peripheral applications such as Qt Designer for layouts, QML for user interfaces, and QSS for styling. Although it is not open-source, containing a commercial licensing plan, I have no plans to commercialize the program, and can therefore utilise the open-source license.

#### OpenGL

Python contains multiple bindings for OpenGL, such as PyOpenGL and ModernGL. Being a widely used standard, OpenGL has the best documentation and support. It also boasts the highest efficiency, designed to be implemented using hardware acceleration through the GPU. However, its main disadvantage is the required complexity compared to the previous frameworks, being primarily a graphical API and not for developing full programs.

## 1.4 Proposed Solution

### 1.4.1 Language

The two main options regarding programming language choice, and their pros and cons, are as listed:

Python	
Pros	Cons

- Versatile and intuitive, uses simple syntax and dynamic typing
  - Supports both object-oriented and procedural programming
  - Rich ecosystem of third-party modules and libraries
  - Interpreted language, good for portability and easy debugging
  - Slow at runtime
  - High memory consumption
- 

Javascript	
Pros	Cons
<ul style="list-style-type: none"> <li>• Generally faster runtime than Python</li> <li>• Simple, dynamically typed and automatic memory management</li> <li>• Versatile, easy integration with both server-side and front-end</li> <li>• Extensive third-party modules</li> <li>• Also supports object-oriented programming</li> </ul>	<ul style="list-style-type: none"> <li>• Mainly focused for web development</li> <li>• Comprehensive knowledge of external frameworks (i.e. Electron) needed for developing desktop applications</li> </ul>

---

I have chosen Python as the programming language for this project. This is mainly due to its extensive third-party modules and libraries available. Python also provides many different GUI frameworks for desktop applications, whereas options are limited for JavaScript due to its focus on web applications. Moreover, the amount of resources and documentation online will prove invaluable for the development process.

Although Python generally has worse performance than JavaScript, speed and memory efficiency are not primary objectives in my project, and should not affect the final program. Therefore, I have prioritised Python's simpler syntax over JavaScript's speed. Being familiar with Python will also allow me to divert more time for development instead of researching new concepts or fixing unfamiliar bugs.

#### 1.4.2 Development Environment

A good development environment improves developer experiences, with features such as auto-indentation and auto-bracket completion for quicker coding. The main development environments under consideration are: Visual Studio Code (VS Code), PyCharm and Sublime Text. I have decided to use VS Code due to its greater library of extensions over Sublime, and its more

user-friendly implementation of features such as version control and GitHub integration. Moreover, VS Code contains many handy features that will speed up the development process, such as its built-in debugging features. Although PyCharm is an extensive IDE, its default features can be supplemented by VS Code extensions. Additionally, VS Code is more lightweight and customizable, and contains vast documentation online.

### 1.4.3 Source Control

A Source Control Tool automates the process of tracking and managing changes in source code. A good source control tool will be essential for my project. It provides the benefits of: protecting the code from human errors (i.e. accidental deletion), enabling easy code experimentation on a clone created through branching from the main project, and by tracking changes through the code history, enabling easier debugging and rollbacks. For my project, I have chosen Git as my version control tool, as it is open-source and provides a more user-friendly interface and documentation over alternatives such as Azure DevOps, and contains sufficient functionality for a small project like mine.

### 1.4.4 Techniques

I have decided on employing the following techniques, based on the pros and cons outlined in the research section above.

#### Board representation

I have chosen to use a bitboard representation for my game. The main consideration was computational efficiency, as a smooth playing experience should be ensured regardless of device used. Bitboards allow for parallel bitwise operations, especially as most modern devices nowadays run on 64-bit architecture CPUs. With bitboards being the mainstream implementation, documentation should also be plentiful.

#### CPU techniques

I have chosen minimax as my searching algorithm. This is due to its relatively simplistic implementation and evaluation accuracy. Additionally, Monte-Carlo Tree Search is computationally intensive, with a high memory requirement and time needed to run with a sufficient number of simulations, which I do not have.

#### GUI framework

I have chosen Pygame as my main GUI framework. This is due to its increased flexibility, in creating custom art and widgets compared to PyQt's defined toolset, which is tailored towards building commercial office applications. Although Pygame contains more overhead and boilerplate code to create standard functionality, I believe that the increased control is worth it for a custom game such as laser chess, which requires dynamic rendering of elements such as the laser beam.

I will also integrate Pygame together with ModernGL, using the convenient APIs in for handling user input and sprite drawing, together with the speed of OpenGL to draw shaders and any other effect overlays.

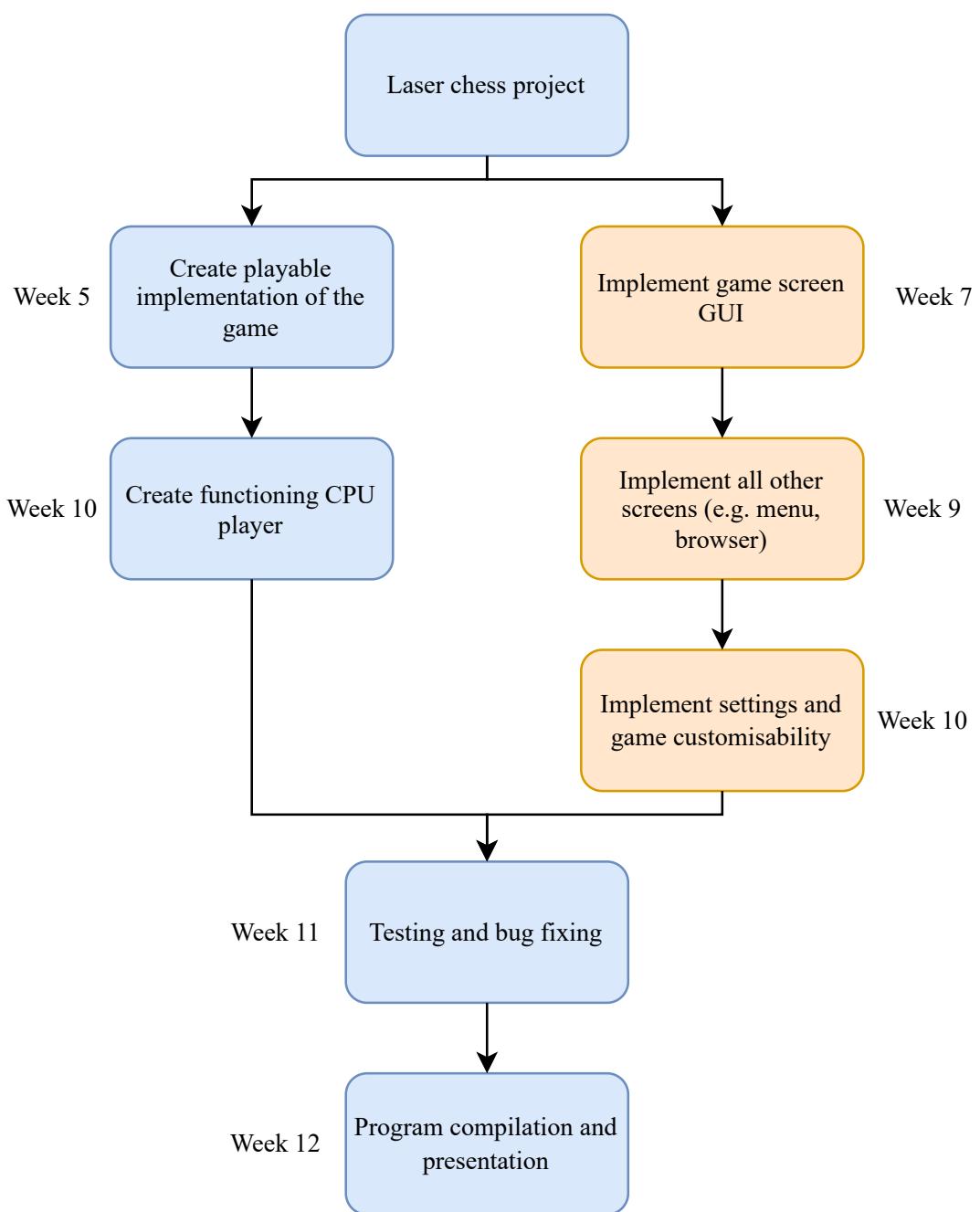
## 1.5 Limitations

I have agreed with my client that due to the multiple versions of Laser Chess that exist online, together with a lack of regulation, an implementation that adheres to the general rules of Laser Chess, and not strictly to a specific version, is acceptable.

Moreover, due to the time constraints on both my schedules for exams and for the date of the tournament, the game only has to be presented in a functional state, and not polished for release, with extra work such as porting to a wide range of OS systems.

## 1.6 Critical Path Design

In order to meet my client's requirement of releasing the game before the next field day, I have given myself a time limit of 12 weeks to develop my game, and have created the following critical path diagram to help me adhere to completing every milestone within the time limit.



# Chapter 2

## Design

### 2.1 System Architecture

In this section, I will lay out the overall logic, and an overview of the steps involved in running my program. By decomposing the program into individual abstracted stages, I can focus on the workings and functionality of each section individually, which makes documenting and coding each section easier. I have also included a flowchart to illustrate the logic of each screen of the program.

I will also create an abstracted GUI prototype in order to showcase the general functionality of the user experience, while acting as a reference for further stages of graphical development. It will consist of individually drawn screens for each stage of the program, as shown in the top-level overview. The elements and layout of each screen are also documented below.

The following is a top-level overview of the logic of the program:

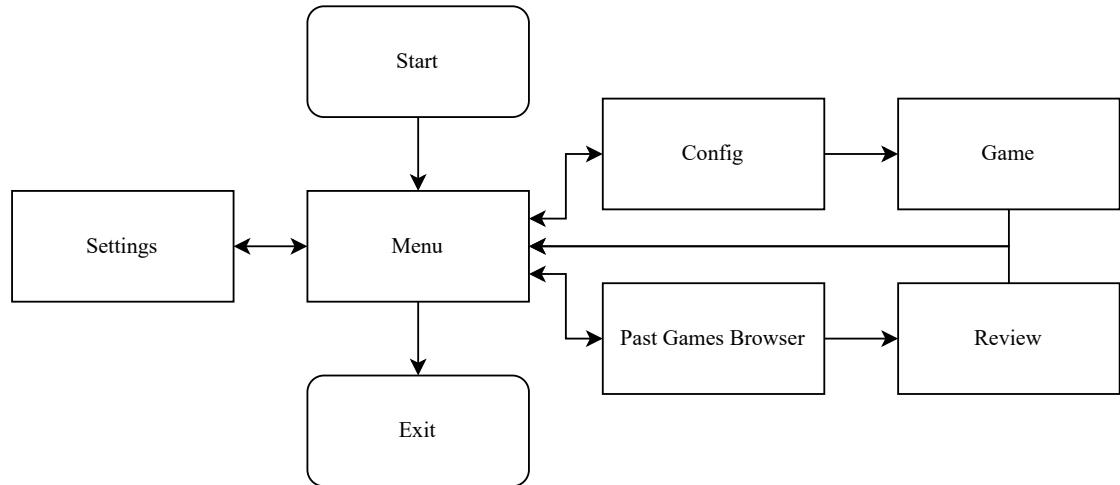


Figure 2.1: Flowchart for Program Overview

### 2.1.1 Main Menu



Figure 2.2: Main Menu screen prototype

The main menu will be the first screen to be displayed, providing access to different stages of the game. The GUI should be simple yet effective, containing clearly-labelled buttons for the user to navigate to different parts of the game.

### 2.1.2 Settings

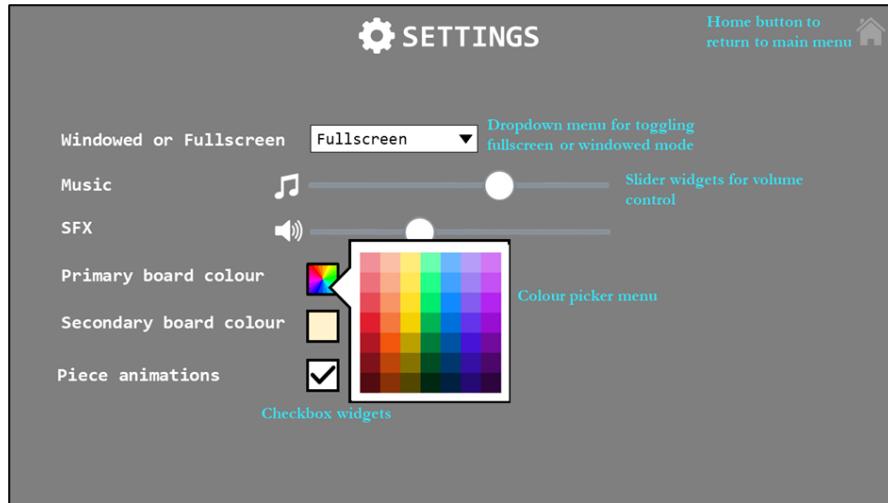


Figure 2.3: Settings screen prototype

The settings menu allows for the user to customise settings related to the program as a whole. The settings will be changed via GUI elements such as buttons and sliders, offering the ability

to customize display mode, volume, board colour etc. Changes to settings will be stored in an intermediate code class, then stored externally into a JSON file. Game settings will instead be changed in the Config screen.

The setting screen should provide a user-friendly interface for changing the program settings intuitively; I have therefore selected appropriate GUI widgets for each setting:

- Windowed or Fullscreen - Drop-down list for selecting between pre-defined options
- Music and SFX - Slider for selecting audio volume, a continuous value
- Board colour - Colour grid for the provision of multiple pre-selected colours
- Piece animation - Checkbox for toggling between on or off

Additionally, each screen is provided with a home button icon on the top right (except the main menu), as a shortcut to return to the main menu.

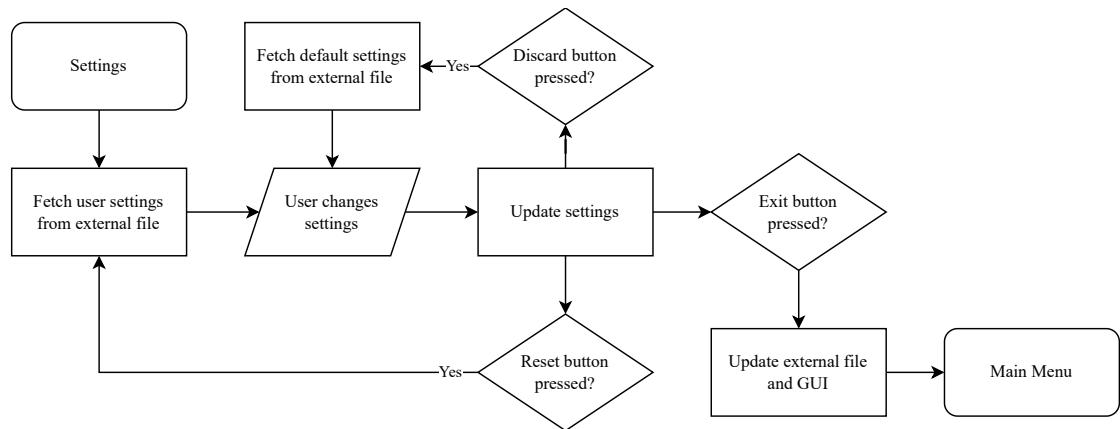


Figure 2.4: Flowchart for Settings

### 2.1.3 Past Games Browser

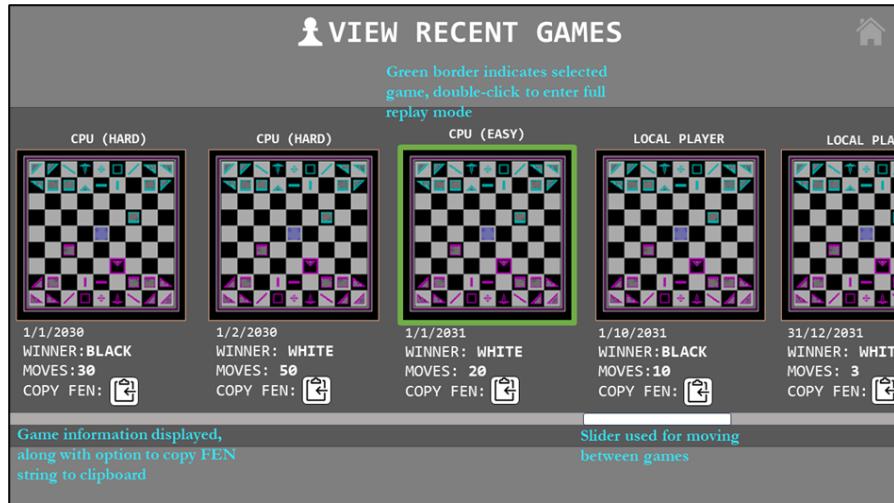


Figure 2.5: Browser screen prototype

The Past Games Browser menu displays a list of previously played games to be replayed. When selecting a game, the replay will render out the saved FEN string into a board position identical to the one played previously, except the user is limited to replaying back and forth between recorded moves. The menu also offers the functionality of sorting games in terms of time, game length etc.

For the GUI, previous games will be displayed on a strip, scrolled through by a horizontal slider. Information about the game will be displayed for each instance, along with the option to copy the FEN string to be stored locally or to be entered into the Review screen. When choosing a past game, a green border will appear to show the current selection, and double clicking enters the user into the full replay mode. While replaying the game, the GUI will appear identical to an actual game. However, the user will be limited to scrolling throughout the moves via the left and right arrow keys.

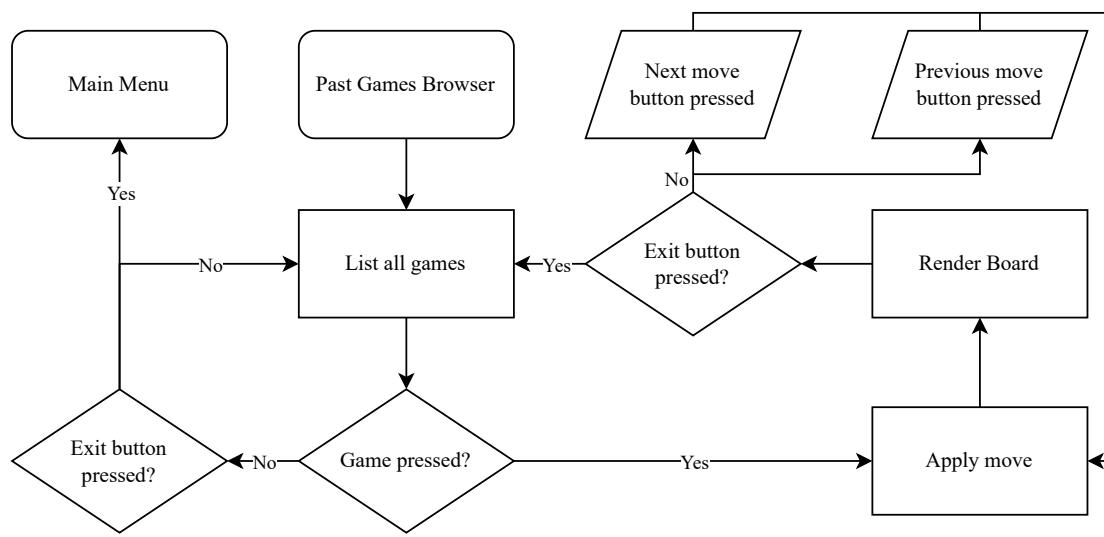


Figure 2.6: Flowchart for Browser

#### 2.1.4 Config

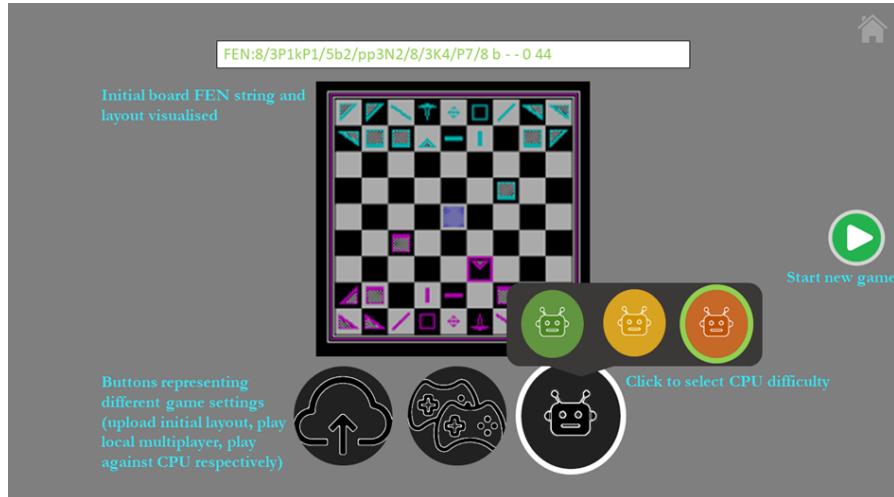


Figure 2.7: Config screen prototype

The config screen comes prior to the actual gameplay screen. Here, the player will be able to change game settings such as toggling the CPU player, time duration, playing as white or black etc.

The config menu is loaded with the default starting position. However, players may enter their own FEN string as an initial position, with the central board updating responsively to give a visual representation of the layout. Players are presented with the additional options to play against a friend, or against a CPU, which displays a drop-down list when pressed to select the CPU difficulty.

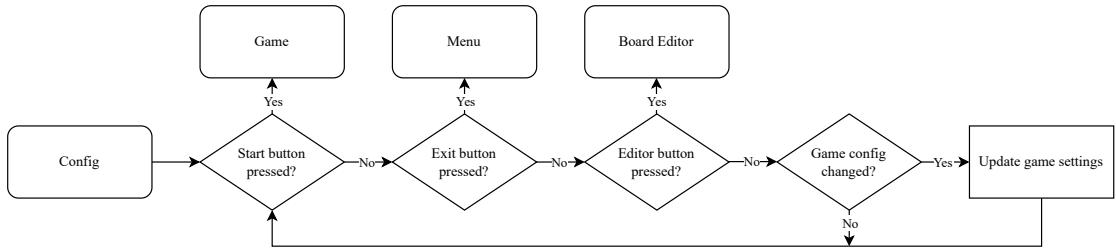


Figure 2.8: Flowchart for Config

### 2.1.5 Game



Figure 2.9: Game screen prototype

During the game, handling of the game logic, such as calculating player turn, calculating CPU moves or laser trajectory, will be computed by the program internally, rendering the updated GUI accordingly in a responsive manner to provide a seamless user experience.

In the game screen, the board is positioned centrally on the screen, surrounded by accompanying widgets displaying information on the current state of the game. The main elements include:

- Status text - displays information on the game state and prompts for each player move
- Rotation buttons - allows each player to rotate the selected piece by 90° for their move
- Timer - displays available time left for each player
- Draw and forfeit buttons - for the named functionalities, confirmed by pressing twice
- Piece display - displays material captured from the opponent for each player

Additionally, the current selected piece will be highlighted, and the available squares to move to will also contain a circular visual cue. Pieces will either be moved by clicking the

target square, or via a drag-and-drop mechanism, accompanied by responsive audio cues. These implementations aim to improve user-friendliness and intuitiveness of the program.

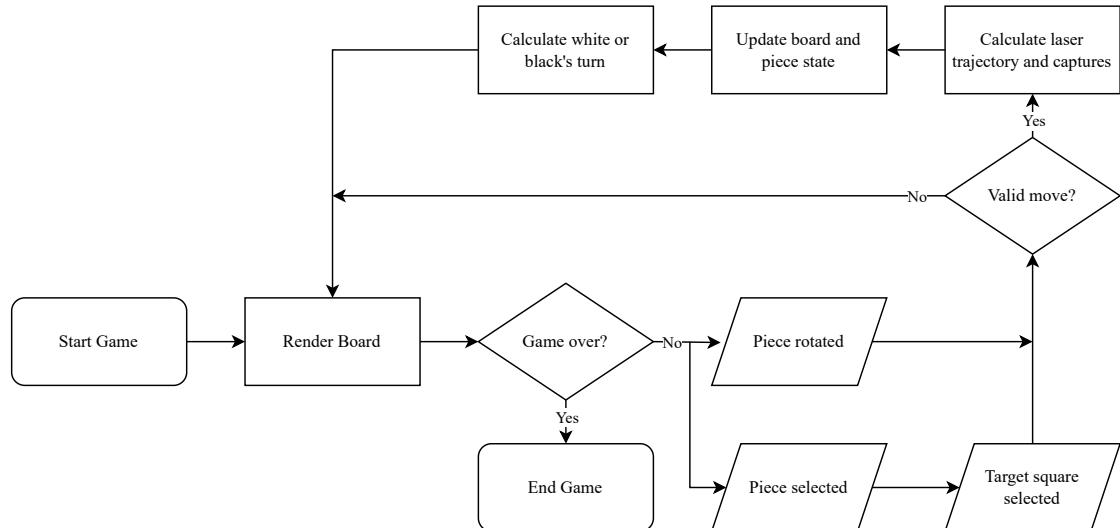


Figure 2.10: Flowchart for Game

### 2.1.6 Board Editor

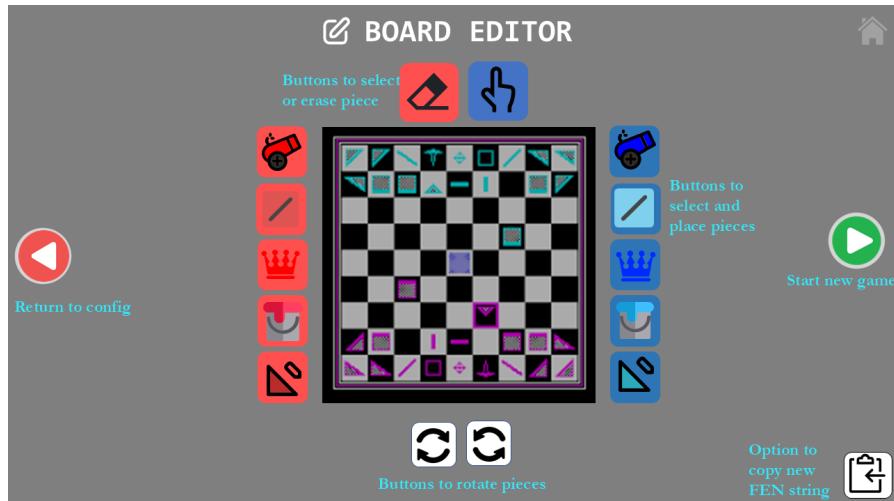


Figure 2.11: Editor screen prototype

The editor screen is used to configure the starting position of the board. Controls should include the ability to place all piece types of either colour, to erase pieces, and easy board manipulation shortcuts such as dragging pieces or emptying the board.

For the GUI, the buttons should clearly represent their functionality, through the use of icons and appropriate colouring (e.g. red for delete).

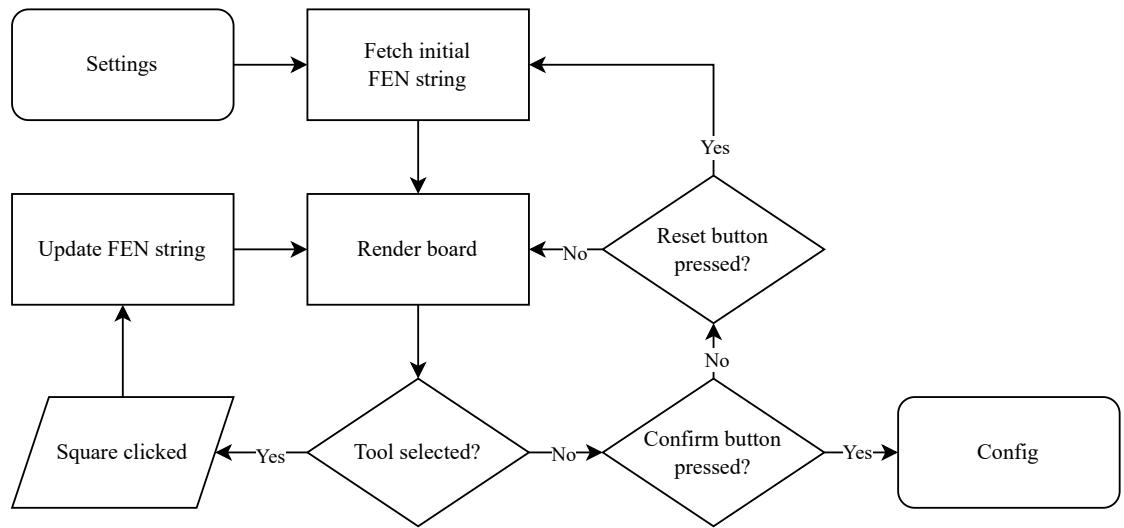


Figure 2.12: Flowchart for board editor

## 2.2 Algorithms and Techniques

### 2.2.1 Minimax

Minimax is a backtracking algorithm commonly used in zero-sum games used to determine the score according to an evaluation function, after a certain number of perfect moves. Minimax aims to minimize the maximum advantage possible for the opponent, thereby minimizing a player's possible loss in a worst-case scenario. It is implemented using a recursive depth-first search, alternating between minimizing and maximizing the player's advantage in each recursive call.

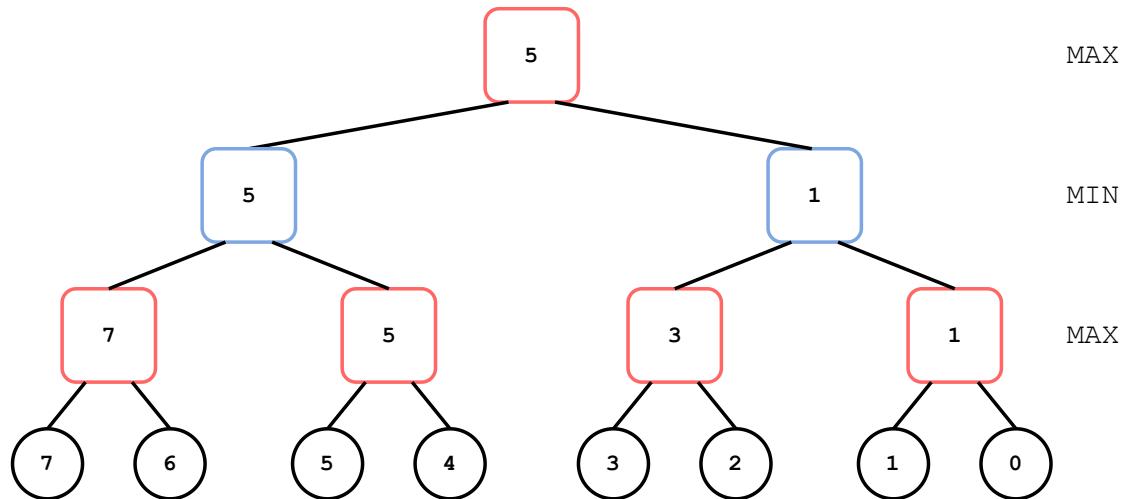


Figure 2.13: Example minimax tree

For the example minimax tree show in Figure 2.13, starting from the bottom leaf node

evaluations, the maximising player would choose the highest values (7, 5, 3, 1). From those values, the minimizing player would choose the lowest values (5, 1). The final value chosen by the maximum player would therefore be the highest of the two, 5.

Implementation in the form of pseudocode is shown below:

---

**Algorithm 1** Minimax pseudocode

---

```

function MINIMAX(node, depth, maximisingPlayer)
    if depth = 0 OR node equals game over then
        return EVALUATE
    end if

    if maximisingPlayer then
        value  $\leftarrow -\infty$ 
        for child of node do
            value  $\leftarrow \text{MAX}(value, \text{MINIMAX}(child, depth - 1, false))$ 
        end for
        return value
    else
        value  $\leftarrow +\infty$ 
        for child of node do
            value  $\leftarrow \text{MIN}(value, \text{MINIMAX}(child, depth - 1, true))$ 
        end for
        return value
    end if
end function

```

---

## 2.2.2 Minimax improvements

### Alpha-beta pruning

Alpha-beta pruning is a search algorithm that aims to decrease the number of nodes evaluated by the minimax algorithm. Alpha-beta pruning stops evaluating a move in the game tree when one refutation is found in its child nodes, proving the node to be worse than previously-examined alternatives. It does this without any potential of pruning away a better move. The algorithm maintains two values: alpha and beta. Alpha ( $\alpha$ ), the upper bound, is the highest value that the maximising player is guaranteed of; Beta ( $\beta$ ), the lower bound, is the lowest value that the minimizing player is guaranteed of. If the condition  $\alpha \geq \beta$  for a node being evaluated, the evaluation process halts and its remaining children nodes are ‘pruned’.

This is shown in the following maximising example:

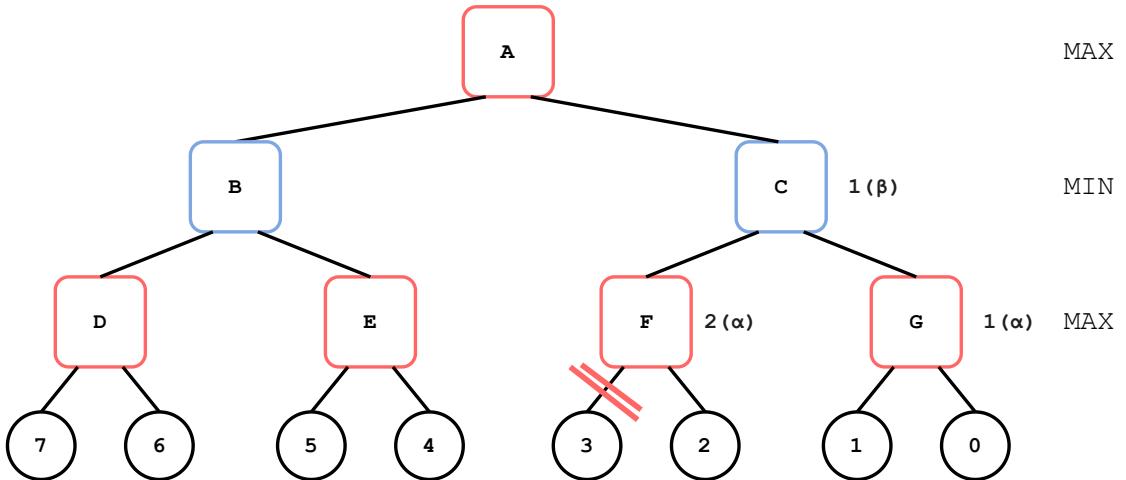


Figure 2.14: Example minimax tree with alpha-beta pruning

Since minimax is a depth-first search algorithm, nodes  $C$  and  $G$  and their  $\alpha$  and  $\beta$  have already been searched. Next, at node  $F$ , the current  $\alpha$  and  $\beta$  are  $-\infty$  and 1 respectively, since the  $\beta$  is passed down from node  $C$ . Searching the first leaf node, the  $\alpha$  subsequently becomes  $\alpha = \max(-\infty, 2)$ . This means that the maximising player at this depth is already guaranteed an evaluation of 2 or greater. Since we know that the minimising player at the depth above is guaranteed a value of 1, there is no point in continuing to search node  $F$ , a node that returns a value of 2 or greater. Hence at node  $F$ , where  $\alpha \geq \beta$ , the branches are pruned.

Alpha-beta pruning therefore prunes insignificant nodes by maintain an upper bound  $\alpha$  and lower bound  $\beta$ . This is an essential optimization as a simple minimax tree increases exponentially in size with each depth ( $O(b^d)$ , with branching factor  $b$  and  $d$  ply depth), and alpha-beta reduces this and the associated computational time considerably.

The pseudocode implementation is shown below:

---

**Algorithm 2** Minimax with alpha-beta pruning pseudocode

---

```
function MINIMAX(node, depth,  $\alpha$ ,  $\beta$ , maximisingPlayer)
    if depth = 0 OR node equals game over then
        return EVALUATE
    end if

    if maximisingPlayer then
        value  $\leftarrow -\infty$ 
        for child of node do
            value  $\leftarrow \text{MAX}(value, \text{MINIMAX}(child, depth - 1, \alpha, \beta, false))$ 
            if value >  $\beta$  then break
        end if
         $\alpha \leftarrow \text{MAX}(\alpha, value)$ 
    end for
    return value
else
    value  $\leftarrow +\infty$ 
    for child of node do
        value  $\leftarrow \text{MIN}(value, \text{MINIMAX}(child, depth - 1, \alpha, \beta, true))$ 
        if value <  $\alpha$  then break
    end if
     $\beta \leftarrow \text{MIN}(\beta, value)$ 
end for
return value
end if
end function
```

---

### Transposition Tables & Zobrist Hashing

Transition tables, a memoisation technique, again greatly reduces the number of moves searched. During a brute-force minimax search with a depth greater than 1, the same positions may be searched multiple times, as the same position can be reached from different sequences of moves. A transposition table caches these same positions (transpositions), along with its associated evaluations, meaning commonly reached positions are not unnecessarily re-searched.

Flags and depth are also stored alongside the evaluation. Depth is required as if the current search comes across a cached position with an evaluation calculated at a lower depth than the current search, the evaluation may be inaccurate. Flags are required for dealing with the uncertainty involved with alpha-beta pruning, and can be any of the following three.

**Exact** flag is used when a node is fully searched without pruning, and the stored and fetched evaluation is accurate.

**Lower** flag is stored when a node receives an evaluation greater than the  $\beta$ , and is subsequently pruned, meaning that the true evaluation could be higher than the value stored. We are thus storing the  $\alpha$  and not an exact value. Thus, when we fetch the cached value, we have to recheck if this value is greater than  $\beta$ . If so, we return the value and this branch is pruned (fail high); If not, nothing is returned, and the exact evaluation is calculated.

**Upper** flag is stored when a node receives an evaluation smaller than the  $\alpha$ , and is subsequently pruned, meaning that the true evaluation could be lower than the value stored. Similarly, when we fetch the cached value, we have to recheck if this value is lower than  $\alpha$ . Again, the current branch is pruned if so (fail low), and an exact evaluation is calculated if not.

The pseudocode implementation for transposition tables is shown below:

---

**Algorithm 3** Minimax with transposition table pseudocode

---

```

function MINIMAX(node, depth,  $\alpha$ ,  $\beta$ , maximisingPlayer)
    hash_key  $\leftarrow$  HASH(node)
    entry  $\leftarrow$  GETENTRY(hash_key)

    if entry.hash_key = hash_key AND entry.hash_key  $\geq$  depth then
        if entry.hash_key = EXACT then
            return entry.value
        else if entry.hash_key = LOWER then
             $\alpha \leftarrow \text{MAX}(\alpha, \text{entry.value})$ 
        else if entry.hash_key = UPPER then
             $\beta \leftarrow \text{MIN}(\beta, \text{entry.value})$ 
        end if
        if  $\alpha \geq \beta$  then
            return entry.value
        end if
    end if

    ...normal minimax...

    entry.value  $\leftarrow$  value
    entry.depth  $\leftarrow$  depth
    if value  $\leq \alpha$  then
        entry.flag  $\leftarrow$  UPPER
    else if value  $\geq \beta$  then
        entry.flag  $\leftarrow$  LOWER
    else
        entry.flag  $\leftarrow$  EXACT
    end if

    return value
end function

```

---

The current board position will be used as the index for a transposition table entry. To convert our board state and bitboards into a valid index, Zobrist hashing may be used. For every square on the chessboard, a random integer is assigned to every piece type (12 in our case, 6 piece type, times 2 for both colours). To initialise a hash, the random integer associated with the piece on a specific square undergoes a XOR operation with the existing hash. The hash is incrementally update with XOR operations every move, instead of being recalculated from scratch improving computational efficiency. Using XOR operations also allows moves to be reversed, proving useful for the functionality to scroll through previous moves. A Zobrist hash is also a better candidate than FEN strings in checking for threefold-repetition, as they are less

intensive to calculate for every move.

The pseudocode implementation for Zobrist hashing is shown below:

---

**Algorithm 4** Zobrist hashing pseudocode

---

*RANDOMINTS represents a pre-initialised array of random integers for each piece type for each square*

```
function HASH _ BOARD(board)
    hash ← 0
    for each square on board do
        if square is not empty then
            hash ⊕ RANDOMINTS[square][piece on square]
        end if
    end for
    return hash
end function

function UPDATEHASH(hash, move)
    hash ⊕ RANDOMINTS[source square][piece]
    hash ⊕ RANDOMINTS[destination square][piece]
    if red to move then
        hash ⊕ hash for red to move ▷ Hash needed for move colour, as two identical positions
        are different if the colour to move is different
    end if
    return hash
end function
```

---

### 2.2.3 Board Representation

#### FEN string

Forsyth-Edwards Notation (FEN) notation provides all information on a particular position in a chess game. I intend to implement methods parsing and generating FEN strings in my program, in order to load desired starting positions and save games for later play. Deviating from the classic 6-part format, a custom FEN string format will be required for our laser chess game, accommodating its different rules from normal chess.

Our custom format implementation is show by the example below:

sc3ncfancpb2/2pc7/3Pd7/pa1Pc1rbra1pb1Pd/pb1Pd1RaRb1pa1Pc/6pb3/7Pa2/2PdNaFaNa3Sa  
r

Our FEN string format contains two parts, denoted by the space between them:

- Part 1: Describes the location of each piece. The construction of this part is defined by the following rules:
  - The board is read from top-left to bottom-right, row by row
  - A number represents the number of empty squares before the next piece
  - A capital letter represents a blue piece, and a lowercase letter represents a red piece

- The letters  $F$ ,  $R$ ,  $P$ ,  $N$ ,  $S$  stand for the pieces Pharaoh, Scarab, Pyramid, Anubis and Sphinx respectively
- Each piece letter is followed by the lowercase letters  $a$ ,  $b$ ,  $c$  or  $d$ , representing a  $0^\circ$ ,  $90^\circ$ ,  $180^\circ$  and  $270^\circ$  degree rotation respectively
- Part 2: States the active colour,  $b$  means blue to move,  $r$  means red to move

Having inputted the desired FEN string board configuration in the config menu, the bitboards for each piece will be initialised with the following functions:

---

#### Algorithm 5 FEN string pseudocode

---

```

function PARSE_FEN_STRING(fen_string, board)
    part_1, part_2  $\leftarrow$  SPLIT(fen_string)
    rank  $\leftarrow$  8
    file  $\leftarrow$  0

    for character in part_1 do
        square  $\leftarrow$  rank  $\times$  8 + file
        if character is alphabetic then
            if character is lower then
                board.bitboards[red][character]  $\mid\mid$  1 << character
            else
                board.bitboards[blue][character]  $\mid\mid$  1 << character
            end if
        else if character is numeric then
            file  $\leftarrow$  file + character
        else if character is / then
            rank  $\leftarrow$  rank - 1
            file  $\leftarrow$  file + 1
        else
            file  $\leftarrow$  file + 1
        end if

        if part_2 is b then
            board.active_colour  $\leftarrow$  b
        else
            board.active_colour  $\leftarrow$  r
        end if
    end for
end function

```

---

The function first processes every piece and corresponding square in the FEN string, modifying each piece bitboard using a bitwise OR operator, with a 1 shifted over to the correctly occupied square using a Left-Shift operator. For the second part, the active colour property of the board class is initialised to the correct player.

## Bitboards

Bitboards are an array of bits representing a position or state of a board game. Multiple bitboards are used with each representing a different property of the game (e.g. scarab position and

scarab rotation), and can be masked together or transformed to answer queries about positions. Bitboards offer an efficient board representation, its performance primarily arising from the speed of parallel bitwise operations used to transform bitboards. To map each board square to a bit in each number, we will assign each square from left to right, with the least significant bit (LSB) assigned to the bottom-left square (A1), and the most significant bit (MSB) to the top-right square (J8).

<b>8</b>	70	71	72	73	74	75	76	77	78	79
<b>7</b>	60	61	62	63	64	65	66	67	68	69
<b>6</b>	50	51	52	53	54	55	56	57	58	59
<b>5</b>	40	41	42	43	44	45	46	47	48	49
<b>4</b>	30	31	32	33	34	35	36	37	38	39
<b>3</b>	20	21	22	23	24	25	26	27	28	29
<b>2</b>	10	11	12	13	14	15	16	17	18	19
<b>1</b>	0	1	2	3	4	5	6	7	8	9
	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	<b>e</b>	<b>f</b>	<b>g</b>	<b>h</b>	<b>j</b>	<b>k</b>

Figure 2.15: Square to bit position mapping

Firstly, we need to initialise each bitboard and place 1s in the correct squares occupied by pieces. This is achieved whilst parsing the FEN-string, as shown in Algorithm 5. Secondly, we should implement an approach to calculate possible moves using our computed bitboards. We can begin by producing a bitboard containing the locations of all pieces, achieved through combining every piece bitboard with bitwise OR operations:

```
all_pieces_bitboard = white_pharaoh_bitboard | black_pharaoh_bitboard |
                     white_scarab_bitboard ...
```

Now, we can utilize this aggregated bitboard to calculate possible positional moves for each piece. For each piece, we can shift the entire bitboard to an adjacent target square (since every piece can only move one adjacent square per turn), and perform a bitwise AND operator with the bitboard containing all pieces, to determine if the target square is already occupied by an existing piece. For example, if we want to compute if the square to the left of our selected piece is available to move to, we will first shift every bit right (as the lowest square index is the LSB on the right, see diagram above), as demonstrated in the following 5x5 example:

	1	0		

Figure 2.16: `shifted_bitboard = piece_bitboard >> 1`

Where green represents the target square shifted into, and orange where the piece used to be. We can then perform a bitwise AND operation with the complement of the all pieces bitboard, where a square with a result of 1 represents an available target square to move to.

$$\text{available\_squares\_right} = (\text{piece\_bitboard} >> 1) \& \sim \text{all\_pieces\_bitboard}$$

However, if the piece is on the leftmost A file, and is shifted to the right, it will be teleported onto the J file on the rank below, which is not a valid move. To prevent these erroneous moves for pieces on the edge of the board, we can utilise an A file mask to mask away any valid moves, as demonstrated below:

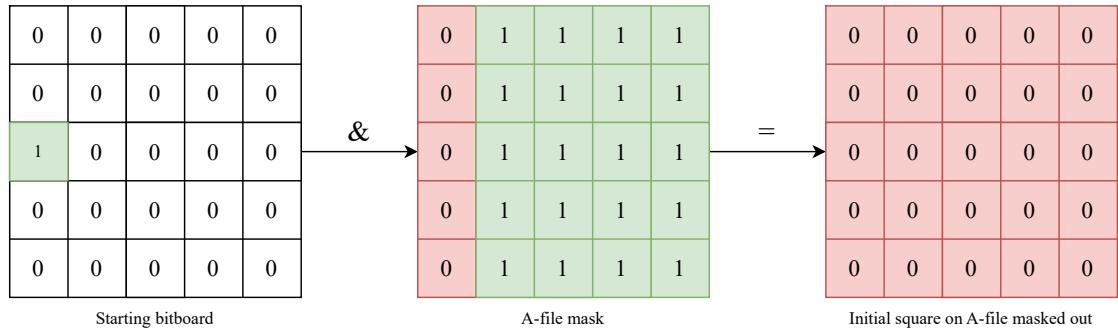


Figure 2.17: A-file mask example

This approach uses the logic that a piece on the A file can never move to a square on the left. Therefore, when calculating if a piece can move to a square on the left, we apply a bitwise AND operator with a mask where every square on the A file is 0; If a piece was on the A file, it will become 0, leaving no possible target squares to move to. The same approach can be mirrored for the far-right J file.

In theory, we do not need to implement the same solution for clipping in regards to ranks, as they are removed automatically by overflow or underflow when shifting bits too far. Our final function to calculate valid moves combines all the logic above: Shifting the selected piece in all

9 adjacent directions by their corresponding bits, masking away pieces trying to move into the edge of the board, combining them with a bitwise OR operator, and finally masking it with the all pieces bitboard to detect which squares are not currently occupied:

---

**Algorithm 6** Finding valid moves pseudocode

---

```

function FIND_VALID_MOVES(selected_square)
    masked_a_square  $\leftarrow$  selected_square & A_FILE_MASK
    masked_j_square  $\leftarrow$  selected_square & J_FILE_MASK

    top_left  $\leftarrow$  masked_a_square << 9
    top_left  $\leftarrow$  masked_a_square << 9
    top_middle  $\leftarrow$  selected_square << 10
    top_right  $\leftarrow$  masked_ << 11
    middle_right  $\leftarrow$  masked_ << 1
    bottom_right  $\leftarrow$  masked_ >> 9
    bottom_middle  $\leftarrow$  selected_square >> 10
    bottom_left  $\leftarrow$  masked_a_square >> 11
    middle_left  $\leftarrow$  masked_a_square >> 1

    possible_moves = top_left | top_middle | top_right | middle_right | bottom_right |
    bottom_middle | bottom_left | middle_left
    valid_moves = possible_moves & ~ ALL_PIECES_BITBOARD

    return valid_moves
end function

```

---

#### 2.2.4 Evaluation Function

The evaluation function is a heuristic algorithm to determine the relative value of a position. It outputs a real number corresponding to the advantage given to a player if reaching the analysed position, usually at a leaf node in the minimax tree. The evaluation function therefore provides the values on which minimax works on to compute an optimal move.

In the majority of evaluation functions, the most significant factor determining the evaluation is the material balance, or summation of values of the pieces. The hand-crafted evaluation function is then optimised by tuning various other positional weighted terms, such as board control and king safety.

#### Material Value

Since laser chess is not widely documented, I have assigned relative strength values to each piece according to my experience playing the game:

- Pharaoh -  $\infty$
- Scarab - 200
- Anubis - 110
- Pyramid - 100

To find the number of pieces, we can iterate through the piece bitboard with the following popcount function:

---

**Algorithm 7** Popcount pseudocode

---

```

function POPCOUNT(bitboard)
    count ← 0
    while bitboard do
        count ← count + 1
        bitboard ← bitboard&(bitboard - 1)
    end while
    return count
end function

```

---

Algorithm 7 continually resets the left-most 1 bit, incrementing a counter for each loop. Once the number of pieces has been established, we multiply this number by the piece value. Repeating this for every piece type, we can thus obtain a value for the total piece value on the board.

### Piece-Square Tables

A piece in normal chess can differ in strength based on what square it is occupying. For example, a knight near the center of the board, controlling many squares, is stronger than a knight on the rim. Similarly, we can implement positional value for Laser Chess through Piece-Square Tables. PSQTs are one-dimensional arrays, with each item representing a value for a piece type on that specific square, encoding both material value and positional simultaneously. Each array will consist of 80 base values representing the piece's material value, with a bonus or penalty added on top for the location of the piece on each square. For example, the following PSQT is for the pharaoh piece type on an example 5x5 board:

0	0	0	0	0
0	0	1	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Piece index

-10	-10	-10	-10	-10
-10	-10	-10	-10	-10
-5	-5	-5	-5	-5
0	0	0	0	0
5	5	5	5	5

Used to reference positional value in PSQT

Figure 2.18: PSQT showing the bonus position value gained for the square occupied by a pharaoh

For asymmetrical PSQTs, we would ideally like to label the board identically from both player's point of views. Although the PSQTs are displayed from the blue player's perspective (blue pharaoh at the bottom of the board), it uses indexes from the red player's perspective, as arrays and lists are defined with index 0 being at the topleft of the board. We would like to flip

the PSQTs to be reused with blue indexes, so that a generic algorithm can be used to sum up and calculate the total positional values for both players.

To utilise a PSQT for blue pieces, a special ‘FLIP’ table can be implemented:

<b>8</b>	70	71	72	73	74	75	76	77	78	79
<b>7</b>	60	61	62	63	64	65	66	67	68	69
<b>6</b>	50	51	52	53	54	55	56	57	58	59
<b>5</b>	40	41	42	43	44	45	46	47	48	49
<b>4</b>	30	31	32	33	34	35	36	37	38	39
<b>3</b>	20	21	22	23	24	25	26	27	28	29
<b>2</b>	10	11	12	13	14	15	16	17	18	19
<b>1</b>	0	1	2	3	4	5	6	7	8	9
	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	<b>e</b>	<b>f</b>	<b>g</b>	<b>h</b>	<b>j</b>	<b>k</b>

Figure 2.19: FLIP table used to map a blue piece index to the red player’s perspective

The FLIP table is just an array of indexes, mapping every blue player’s index onto the corresponding red index. The following expression utilises the FLIP table to retrieve a blue player’s value from the red player’s PSQT:

$$\text{blue\_psqt\_value} = \text{PHAROAH\_PSQT}[\text{FLIP}[\text{square}]]$$

The following function retrieves an array of bitboards representing piece positions from the board class, then sums up all the values of these pieces for both players, referencing the corresponding PSQT:

---

**Algorithm 8** Calculating positional value pseudocode

---

```

function CALCULATE_POSITIONAL_VALUE(bitboards, colour)
    positional_score ← 0
    for all pieces do
        for square in bitboards[piece] do
            if square = 1 then
                if colour is blue then
                    positional_score ← positional_score + PSQT[piece][square]
                else
                    positional_score ← positional_score + PSQT[piece][FLIP[square]]
                end if
            end if
        end for
    end for
    return positional_score
end function

```

---

## Using valid squares

Using Algorithm 6 for finding valid moves, we can implement two more improvements for our evaluation function: Mobility and King Safety.

**Mobility** is the number of legal moves a player has for a given position. This is advantageous in most cases, with a positive correlation between mobility and the strength of a position. To implement this, we simply loop over all pieces of the active colour, and sum up the number of valid moves obtained from the previous algorithm.

**King safety** (Pharaoh safety) describes the level of protection of the pharaoh, being the piece that determines a win or loss. In normal chess, this would be achieved usually by castling, or protection via position or with other pieces. Similarly, since the only way to lose in Laser Chess is via a laser, having pieces surrounding the pharaoh, either to reflect the laser or to be sacrificed, is a sensible tactic and improves king safety. Thus, a value for king safety can be achieved by finding the number of valid moves a pharaoh can make, and subtracting them from the maximum possible of moves (8) to find the number of surrounding pieces.

### 2.2.5 Shadow Mapping

Following the client's requirement for engaging visuals, I have decided to implement shadow mapping for my program, especially as lasers are the main focus of the game. Shadow mapping is a technique used to create graphical hard shadows, with the use of a depth buffer map. I have chosen to implement shadow mapping, instead of alternative lighting techniques such as ray casting and ray marching, as its efficiency is more suitable for real-time usage, and results are visually decent enough for my purposes.

For typical 3D shadow mapping, the standard approach is as follows:

1. Render the scene from the light's point of view
2. Extract a depth buffer texture from the render
3. Compare the distance of a pixel from the light to the value stored in the depth texture
4. If greater, there must be an obstacle in the way reducing the depth map value, therefore that pixel must be in shadow

To implement shadow casting for my 2D game, I have modified some steps and arrived on the final following workflow:

1. Render the scene with only occluding objects shown
2. Crop texture to align the center to the light position
3. To create a 1D depth map, transform Cartesian to polar coordinates, and increase the distance from the origin until a collision with an occluding object
4. Using polar coordinates for the real texture, compare the z-depth to the corresponding value from the depth map
5. Additively blend the light colour if z-depth is less than the depth map value

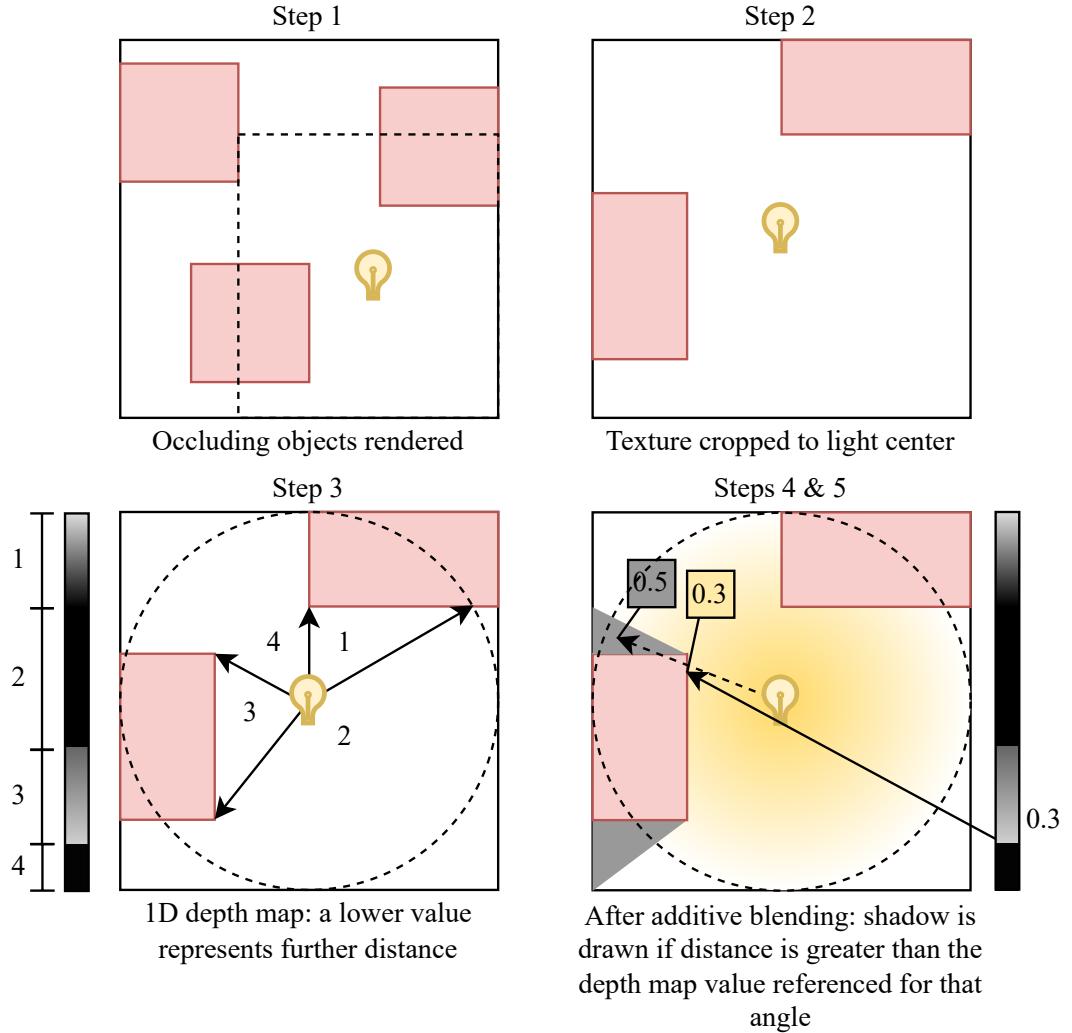


Figure 2.20: Workflow for 2D shadow mapping

Our method requires a coordinate transformation from Cartesian to polar, and vice versa. Polar to Cartesian transformation can be achieved with trigonometry, forming a right-angled triangle in the center and using the following two equations:

$$x = r \cos(\theta)$$

$$y = r \sin(\theta)$$

Cartesian to polar can also similarly be achieved with the right-angled triangle, finding the radius with the Pythagorean theorem, and the angle with arctan. However, since the range of the arctan function is only a half-circle ( $\frac{\pi}{2} < \theta < \frac{3\pi}{2}$ ), we will have to use the atan2 function, which accounts for the negative quadrants, or the following:

$$\theta = 2 \arctan \left( \frac{r - x}{y} \right)$$

There are several disadvantages to shadow mapping. The relevant ones for us are Aliasing and Shadow Acne:

**Aliasing** occurs when the texture size for the depth map is smaller than the light map, causing shadows to be scaled up and rendered with jagged edges.

**Shadow Acne** occurs when the depth from the depth map is so close to the light map value, that precision errors cause unnecessary shadows to be rendered.

These problems can be mitigated by increasing the size of the shadow map size. However, due to memory and hardware constraints, I will have to find a compromised resolution to balance both artifacting and acuity.

### Soft Shadows

The approach above is used only for calculating hard shadows. However, in real-life scenarios, lights are not modelled as a single particle, but instead emitted from a wide light source. This creates an umbra and penumbra, resulting in soft shadows.

To emulate this in our game, we could calculate penumbra values with various methods, however, due to hardware constraints and simplicity again, I have chosen to use the following simpler method:

1. Sample the depth map multiple times, from various differing angles
2. Sum the results using a normal distribution
3. Blur the final result proportional to the length from the center

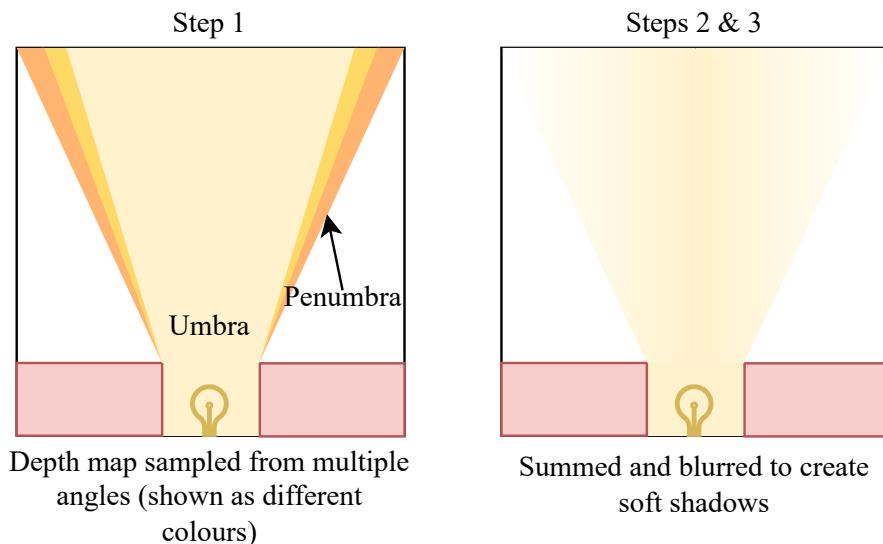


Figure 2.21: Workflow for 2D soft shadows

This method progressively blurs the shadow as the distance from the main shadow (umbra) increases, which results in a convincing estimation while being less computationally intensive.

### 2.2.6 Multithreading

In order to fulfill Objective 7 of a responsive GUI, I will have to employ multi-threading. Since python runs on a single thread natively, code is exected serially, meaning that a time consuming function such as minimax will prevent the running of another GUI-drawing function until it is finished, hence freezing the program. To overcome this, multi-threading can execute both functions in parallel on different threads, meaning the GUI-drawing thread can run while minimax is being computed, and stay responsive. To pass data between threads, since memory is shared between threads, arrays and queues can be used to store results from threads. The following flowchart shows my chosen approach to keep the GUI responsive while minimax is being computed:

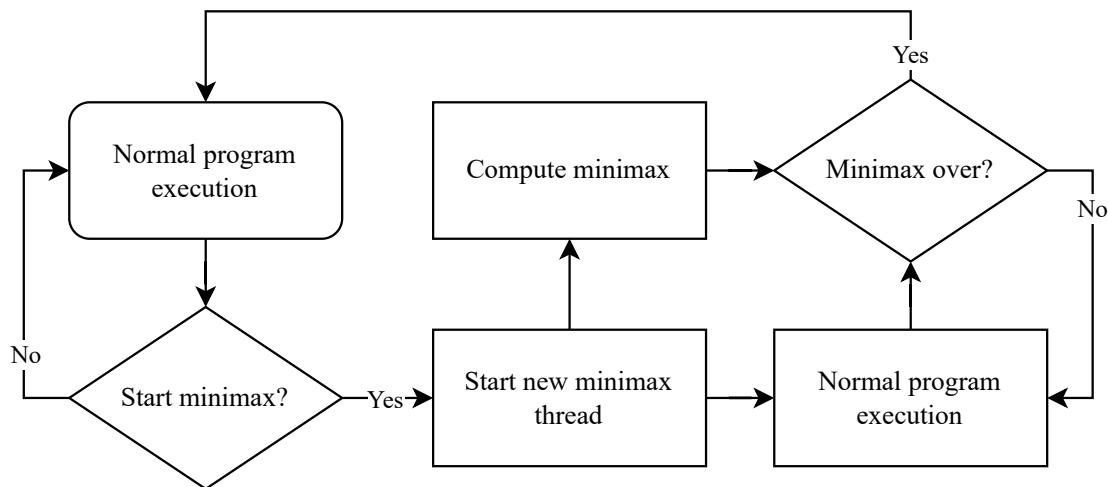


Figure 2.22: Multi-threading for minimax

## 2.3 Data Structures

### 2.3.1 Database

To achieve Objective 2 and stores previous games, I have opted to use a relational database. Choosing between different relational database, I have decided to use SQLite, since it does not require additional server softwards, has good performance with low memory requirements, and adequate for my use cases, with others such as Postgres being overkill.

#### DDL

Only a single entity will be required for my program, a table to store games. The table schema will be defined as follows:

**Table: games**

Field	Key	Data Type	Validation
game_id	Primary	INT	NOT NULL
winner		INT	
cpu_depth		INT	

number_of_moves	INT	NOT NULL
cpu_enabled	BOOL	NOT NULL
moves	TEXT	NOT NULL
initial_board_configuration	TEXT	NOT NULL
time	FLOAT	
created_dt	TIMESTAMP	NOT NULL

Table 2.1: Data table scheme for *games* table

All fields are either generated or retrieved from the board class, with the exception of the moves attribute, which will need to be encoded into a suitable data type such as a string. All attributes are also independent of each other<sup>1</sup>, and so the the table therefore adheres to the third normal form.

To create the entity, a `CREATE` statement like the following can be used:

```

1 CREATE TABLE games(
2     id INTEGER PRIMARY KEY,
3     winner INTEGER,
4     cpu_depth INTEGER,
5     time real NOT NULL,
6     moves TEXT NOT NULL,
7     cpu_enabled INTEGER NOT NULL,
8     created_dt TIMESTAMP NOT NULL,
9     number_of_moves INTEGER NOT NULL,
10    initial_fen_string TEXT NOT NULL,
11 )

```

Removing an entity can also be done in a similar fashion:

```

1 DROP TABLE games

```

Migrations are a version control system to track incremental changes to the schema of a database. Since there is no popular SQL Python-binding libraries that support migrations, I will just be using a manual solution of creating python files that represent a change in my schema, defining functions that make use of SQL `ALTER` statements. This allows me to keep track of any changes, and rollback to a previous schema.

## DML

To insert a new game entry into the table, an `INSERT` statement can be used with the provided array, where the appropiate arguments are binded to the correct attribute via ? placeholders when run.

```

1 INSERT INTO games (
2     cpu_enabled,
3     cpu_depth,
4     winner,
5     time,
6     number_of_moves,
7     moves,
8     initial_fen_string,

```

---

<sup>1</sup>There is a case to be made for *moves* and *number\_of\_moves*, however I have included *number\_of\_moves* to save the computational effort of parsing the moves for every game just to display it on the browser preview section.

```

9     created_dt
10    )
11    VALUES  (?, ?, ?, ?, ?, ?, ?, ?, ?)

```

Moreover, we will need to fetch the number of total game entries in the table to be displayed to the user. To do this, the aggregate function `COUNT` can be used, which is supported by all SQL databases.

```
1   SELECT COUNT(*) FROM games
```

## Pagination

When there are a large number of entries in the table, it would be appropriate to display all the games to the user in a paginated form, where they can scroll between different pages and groups of games. There are multiple methods to paginate data, such as using `LIMIT` and `OFFSET` clauses, or cursor-based pagination, but I have opted to use the `ROW_NUMBER()` function.

`ROW_NUMBER()` is a window function that assigns a sequential integer to a query's result set. If I were to query the entire table, each row would be assigned an integer that could be used to check if the row is in the bounds for the current page, and therefore be displayed. Moreover, the use of an `ORDER BY` clause enables sorting of the output rows, allowing the user to choose what order the games are presented in based on an attribute such as number of moves. A `PARTITION BY` clause will also be used to group the results base on an attribute such as winner prior to sorting, if the user wants to search for games based on multiple criteria with greater ease.

The start row and end row will be passed as parameters to the placeholders in the SQL statement, calculated by multiplying the page number by the number of games per page.

```

1   SELECT * FROM
2     (SELECT ROW_NUMBER() OVER (
3       PARTITION BY attribute1
4       ORDER BY attribute2 ASC
5     ) AS row_num, * FROM games)
6   WHERE row_num >= ? AND row_num <= ?

```

## Security

Security measures such as database file permissions and encryption are common for a SQL database. However, since SQLite is a serverless database, and my program runs without any need for an internet connection, the risk of vulnerabilities is greatly reduced. Additionally, the game data stored on my database is frankly inconsequential, so going to great lengths to protect it wouldn't be to best use of my time. Nevertheless, my SQL Python-binding does support the user of placeholdeers for parameteres, thereby addressing the risk of SQL injection attacks.

### 2.3.2 Linked Lists

Another data structure I intend to implement is linked lists. This will be integrated into widgets such as the carousel or multiple icon button widget, since these will contain a variable number of items, and where  $O(1)$  random access is not a priority. Since moving back and forth between nodes is a must for a carousel widget, the linked list will be doubly-linked, with each node containing to its previous and next node. The list will also need to loop, with the next pointer of the last node pointing back to the first node, making it a circular linked list.

The following pseudocode outlines the basic functionality of the linked list:

---

**Algorithm 9** Circular doubly linked list pseudocode

---

```
function INSERT_AT_FRONT(node)
    if head is none then
        head ← node
        node.next ← node.previous ← head
    else
        node.next ← head
        node.previous ← head.previous
        head.previous.next ← node
        head.previous ← node

        head ← node
    end if
end function
```

**Require:**  $\text{LEN}(list) > 0$

```
function DATA_IN_LIST(data)
    current_node ← head.next
    while current_node ≠ head do
        if current_node.data = data then
            return True
        end if
        current_node ← current_node.next
    end while
    return False
end function
```

**Require:** Data in list

```
function REMOVE(data)
    current_node ← head
    while current_node.data ≠ data do
        current_node ← current_node.next
    end while

    current_node.previous.next ← current_node.next
    current_node.next.previous ← current_node.previous

    delete current_node
end function
```

---

### 2.3.3 Stack

Being a data structure with LIFO ordering, a stack is used for handling moves in the review screen. Starting with full stack of moves, every move undone pops an element off the stack to be processed. This move is then pushed onto a second stack. Therefore, cycling between moves requires pushing and popping between the two stacks, as shown in Figure 2.23. The same functionality can be achieved using a queue, but I have chosen to use two stacks as it is simpler

to implement, as being able to quickly check the number of items in each will come in handy.

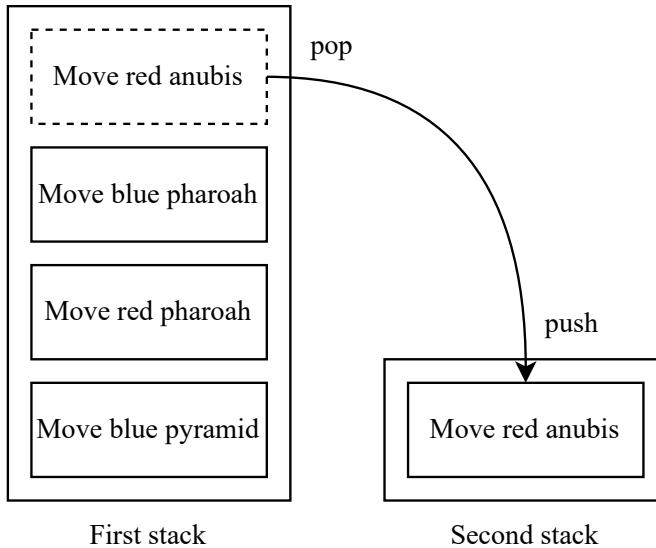


Figure 2.23: *Move red anubis* is undone and pushed onto the second stack

## 2.4 Classes

I will be using an Object-Oriented Programming (OOP) paradigm for my program. OOP reduces repetition of code, as inheritance can be used to abstract repetitive code into a base class, as shown in my widgets implementation. Testing and debugging classes will make my workflow more efficient. This section documents the base classes I am going to implement in my program.

### State

Since there will be multiple screens in my program as demonstrated in Figure 2.1, the State base class will be used to handle the logic for each screen. For each screen, void functions will be inherited and overwritten, each containing their own logic for that specific screen. For example, all screens will call the startup function in Table 2.2 to initialise variables needed for that screen. This polymorphism approach allows me to use another Control class to enable easy switching between screens, without worrying about the internal logic of that screen. Virtual methods also allow methods such as `draw` to be abstracted to the State superclass, reducing code in the inherited subclasses, while allowing them to override the methods and add their own logic.

Method Name	Description
<code>startup</code>	Initialise variables and functions when state set as displayed screen
<code>cleanup</code>	Cleanup any variables and functions when state removed from screen
<code>draw</code>	Draw to display
<code>update</code>	Update any variables for every game tick
<code>handle_resize</code>	Scale GUI when window resized
<code>get_event</code>	Receive pygame events as argument and process them

Table 2.2: Methods for State class

## Widget

I will be implementing my own widget system for creating the game GUI. This allows me to fully customise all graphical elements on the screen, and also create a resizing system that adheres to Objective 7. The default pygame rescaling options also simply resize elements without accounting for aspect ratios or resolution, and I could not find a library that suits my needs. Having a bespoke GUI implementation also justifies my use of Pygame over other Python frameworks.

I will be utilising the Pygame sprite system for my GUI. All GUI widgets will be subclasses inheriting from the base Widget class, which itself is a subclass of the Pygame sprite class. Since Pygame sprites are drawn via a spritegroup class, I will also have to create a custom subclass inheriting that as well. As with the State class, polymorphism will allow the spritegroup class to render all widgets regardless of their functionality. Each widget will override their base methods, especially the draw (set\_image) method, for their own needs. Additionally, I will use getter and setter methods, used with the `@property` decorator in python, to compute attributes mainly used for resizing widgets. This allows me to expose common variables, and to reduce code repetition.

Method Name	Description
set_image	Render widget to internal image attribute for pygame sprite class
set_geometry	Set position and size of image
set_screen_size	Set screen size for resizing purposes
get_event	Receives pygame events and processes them
screen_size*	Returns screen size in pixels
position*	Returns topleft of widget rect
size*	Returns size of widget in pixels
margin*	Returns distance between border and actual widget image
border_width*	Returns border width
border_radius*	Returns border radius for rounded corners
font_size*	Returns font size for text-based widgets

\* represents getter method / property

Table 2.3: Methods for Widget class

I will also employ multiple inheritance to combine different base class functionalities together. For example, I will create a pressable base class, designed to be subclassed along with the widget class. This will provide attributes and methods for widgets that support clicking and dragging. Following Python's Method Resolution Order (MRO), additional base classes should be referenced first, having priority over the base Widget class.

Method Name	Description
get_event	Receives Pygame events and sets current state accordingly
set_state	Sets current Pressable state, called by <code>get_event</code>
set_colours	Set fill colour according to widget Pressable state
current_state*	Returns current Pressable state (e.g. hovered, pressed etc.)

Method Name	Description
* represents getter method / property	

Table 2.4: Methods for example Pressable class

## Game

For my game screen, I will be utilising the Model-View-Controller architectural pattern (MVC). MVC defines three interconnected parts, the model processing information, the view showing the information, and the controlling receiving user inputs and connecting the two. This will allow me to decompose the development process into individual parts for the game logic, graphics and user input, speeding up the development process and making testing easier. It also allows me to implement multiple views, for the pause and win screens as well. For MVC, I will have to implement a game model class, a game controller class, and three classes for each view (game, pause, win). Using aggregation, these will be initially connected and handled by the game state class. For the following methods, I have only showed those pertinent to the MVC pattern:

Method Name	Description
get_event	Receives Pygame events and passes them onto the correct part's event handler
handle_game_event	Receives events and notifies the game model and game view
handle_pause_event	Receives events and notifies the pause view
handle_win_event	Receives events and notifies the win view
...	...

Table 2.5: Methods for Controller class

Method Name	Description
process_model_event	Receives events from the model and calls the relevant method to display that information
convert_mouse_pos	Sends controller class information of widget under mouse
draw	Draw information to display
handle_resize	Scale GUI when window resized
...	...

Table 2.6: Methods for View class

Method Name	Description
register_listener	Subscribes method on view instance to an event type, so that the method receives and processes that event everytime <code>alert_listener</code> is called
alert_listener	Sends event to all subscribed instances
toggle_win	Sends event for win view
toggle_pause	Sends event for pause view
...	...

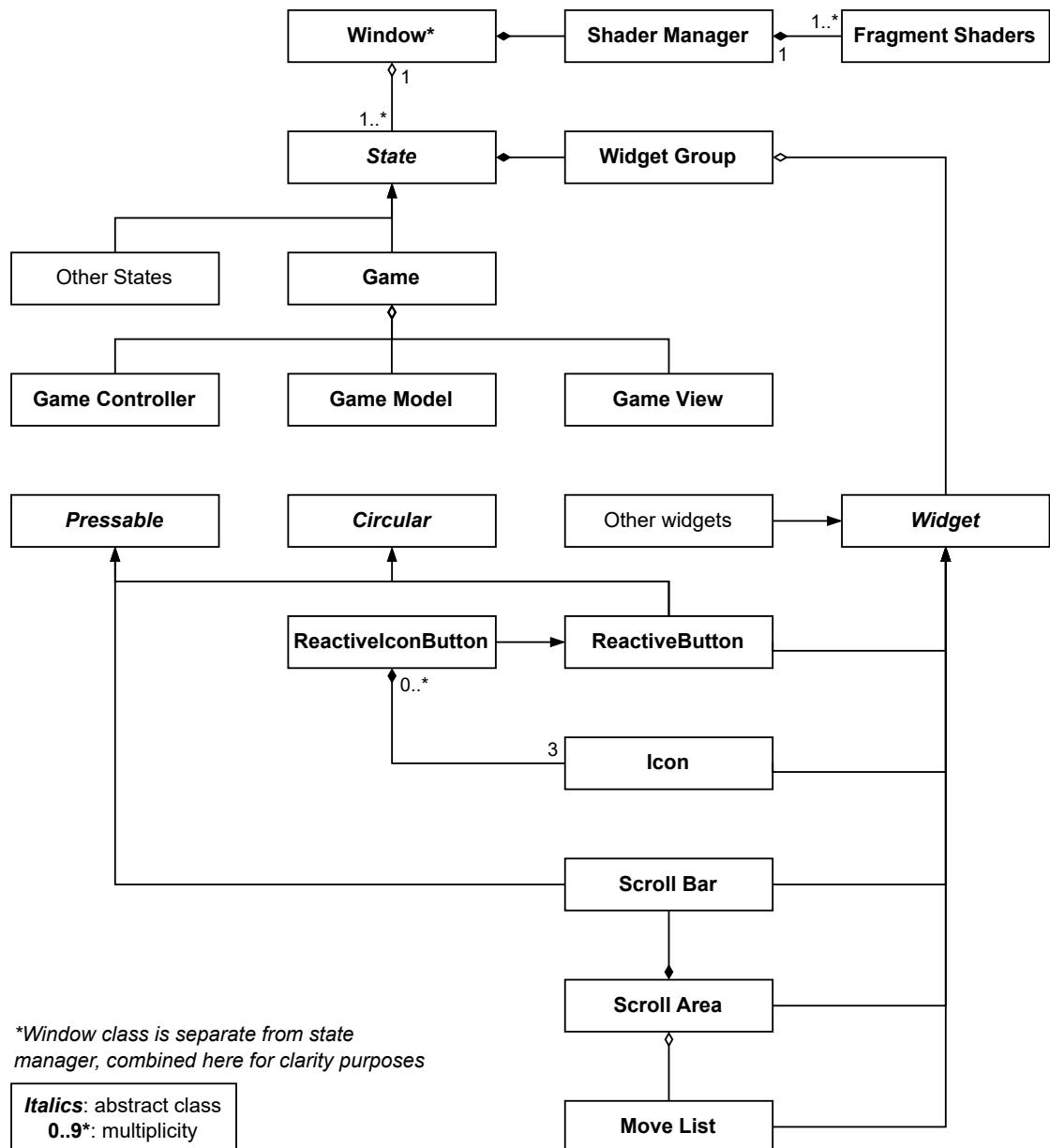
Method Name	Description
-------------	-------------

Table 2.7: Methods for Model class

### Shaders

To use ModernGL with Pygame, I have created classes for each fragment shader, controlled by a main shader manager class. The fragment shader classes will rely on composition: The shader manager creates the fragment shader class; Every fragment shader class takes their shader manager parent instance as an argument, and runs methods on it to produce the final output.

### 2.4.1 Class Diagram



# Chapter 3

## Technical Solution

3.1	File Tree Diagram . . . . .	46
3.2	Summary of Complexity . . . . .	47
3.3	Overview . . . . .	47
3.3.1	Main . . . . .	47
3.3.2	Loading Screen . . . . .	48
3.3.3	Helper functions . . . . .	50
3.3.4	Theme . . . . .	58
3.4	GUI . . . . .	59
3.4.1	Laser . . . . .	59
3.4.2	Particles . . . . .	62
3.4.3	Widget Bases . . . . .	65
3.4.4	Widgets . . . . .	74
3.5	Game . . . . .	86
3.5.1	Model . . . . .	86
3.5.2	View . . . . .	90
3.5.3	Controller . . . . .	96
3.5.4	Board . . . . .	101
3.5.5	Bitboards . . . . .	106
3.6	CPU . . . . .	112
3.6.1	Minimax . . . . .	112
3.6.2	Alpha-beta Pruning . . . . .	114
3.6.3	Transposition Table . . . . .	115
3.6.4	Evaluator . . . . .	117
3.6.5	Multithreading . . . . .	119
3.6.6	Zobrist Hashing . . . . .	120
3.6.7	Cache . . . . .	121
3.7	States . . . . .	123
3.7.1	Review . . . . .	123
3.8	Database . . . . .	128
3.8.1	DDL . . . . .	128
3.8.2	DML . . . . .	130
3.9	Shaders . . . . .	133
3.9.1	Shader Manager . . . . .	133
3.9.2	Bloom . . . . .	138

3.9.3	Rays . . . . .	141
3.9.4	Stack . . . . .	145

### 3.1 File Tree Diagram

To help navigate through the source code, I have included the following directory tree diagram, and put appropriate comments to explain the general purpose of code contained within specific directories and Python files.

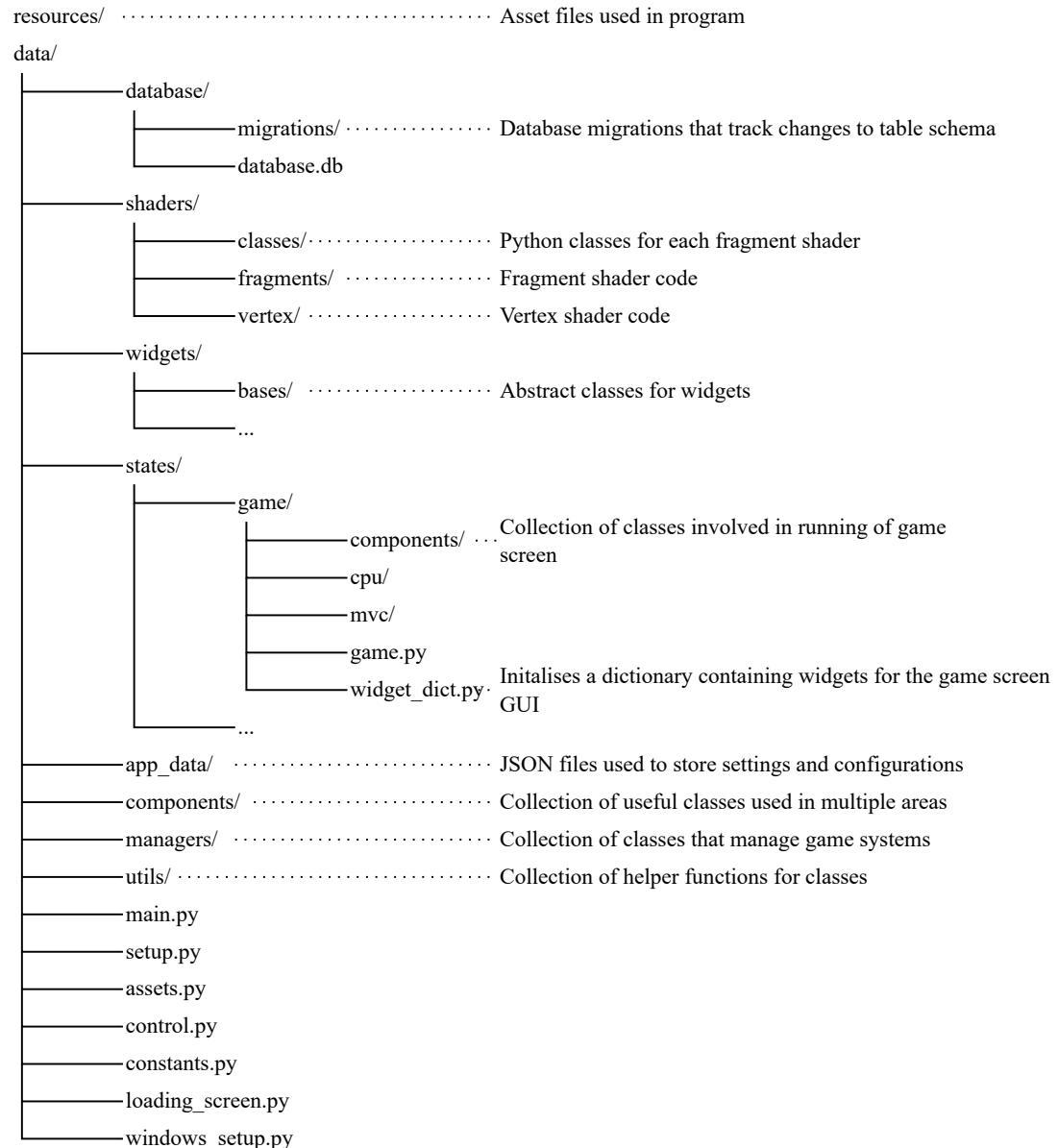


Figure 3.1: File tree diagram

## 3.2 Summary of Complexity

- Alpha-beta pruning and transposition table improvements for Minimax (3.6.2 and 3.6.3)
- Shadow mapping and coordinate transformations (3.9.3)
- Recursive Depth-First Search tree traversal (3.3.4 and 3.6.1)
- Circular doubly-linked list and stack (3.4.3 and 3.9.4)
- Multipass shaders and gaussian blur (3.9.2)
- Aggregate and Window SQL functions (3.8.2)
- OOP techniques (3.4.3 and 3.4.4)
- Multithreading (3.3.2 and 3.6.5)
- Bitboards (3.5.5)
- Zobrist hashing (3.6.6)
- (File handling and JSON parsing) (3.3.3)
- (Dictionary recursion) (3.3.4)
- (Dot product) (3.3.3 and 3.9.2)

## 3.3 Overview

### 3.3.1 Main

The file `main.py` is run by the root file `run.py`. Here resources-intensive classes such as the state and asset files are initialised, while the program displays a loading screen to hide the loading process. The main game loop is then executed.

`main.py`

```
1 from sys import platform
2 # Initialises Pygame
3 import data.setup
4
5 # Windows OS requires some configuration for Pygame to scale GUI continuously
6     # while window is being resized
6 if platform == 'win32':
7     import data.windows_setup as win_setup
8
9 from data.loading_screen import LoadingScreen
10
11 states = [None, None]
12
13 def load_states():
14     """
15         Initialises instances of all screens, executed on another thread with results
16         being stored to the main thread by modifying a mutable such as the states list
17     """
18     from data.control import Control
19     from data.states.game.game import Game
20     from data.states.menu.menu import Menu
```

```

20     from data.states.settings import Settings
21     from data.states.config.config import Config
22     from data.states.browser.browser import Browser
23     from data.states.review.review import Review
24     from data.states.editor.editor import Editor
25
26     state_dict = {
27         'menu': Menu(),
28         'game': Game(),
29         'settings': Settings(),
30         'config': Config(),
31         'browser': Browser(),
32         'review': Review(),
33         'editor': Editor()
34     }
35
36     app = Control()
37
38     states[0] = app
39     states[1] = state_dict
40
41 loading_screen = LoadingScreen(load_states)
42
43 def main():
44     """
45     Executed by run.py, starts main game loop
46     """
47     app, state_dict = states
48
49     if platform == 'win32':
50         win_setup.set_win_resize_func(app.update_window)
51
52     app.setup_states(state_dict, 'menu')
53     app.main_game_loop()

```

### 3.3.2 Loading Screen

Multithreading is used to separate the loading screen GUI from the resources intensive actions in `main.py`, to keep the GUI responsive. The easing function `easeOutBack` is also used to animate the logo.

`loading_screen.py`

```

1 import pygame
2 import threading
3 import sys
4 from pathlib import Path
5 from data.utils.load_helpers import load_gfx, load_sfx
6 from data.managers.window import window
7 from data.managers.audio import audio
8
9 FPS = 30
10 start_ticks = pygame.time.get_ticks()
11 logo_gfx_path = (Path(__file__).parent / '../resources/graphics/gui/icons/logo/
12     logo.png').resolve()
12 sfx_path_1 = (Path(__file__).parent / '../resources/sfx/loading_screen/
13     loading_screen_1.wav').resolve()
13 sfx_path_2 = (Path(__file__).parent / '../resources/sfx/loading_screen/
14     loading_screen_2.wav').resolve()
14
15 def easeOutBack(progress):

```

```

16 """
17 Represents a cubic function for easing the logo position.
18 Starts quickly and has small overshoot, then ends slowly.
19
20 Args:
21     progress (float): x-value for cubic function ranging from 0-1.
22
23 Returns:
24     float:  $2.70x^3 + 1.70x^2 + 0x + 1$ , where x is time elapsed.
25 """
26 c2 = 1.70158
27 c3 = 2.70158
28
29     return c3 * ((progress - 1) ** 3) + c2 * ((progress - 1) ** 2) + 1
30
31 class LoadingScreen:
32     def __init__(self, target_func):
33         """
34             Creates new thread, and sets the load_state() function as its target.
35             Then starts draw loop for the loading screen.
36
37         Args:
38             target_func (Callable): function to be run on thread.
39         """
40         self._clock = pygame.time.Clock()
41         self._thread = threading.Thread(target=target_func)
42         self._thread.start()
43
44         self._logo_surface = load_gfx(logo_gfx_path)
45         self._logo_surface = pygame.transform.scale(self._logo_surface, (96, 96))
46         audio.play_sfx(load_sfx(sfx_path_1))
47         audio.play_sfx(load_sfx(sfx_path_2))
48
49         self.run()
50
51     @property
52     def logo_position(self):
53         duration = 1000
54         displacement = 50
55         elapsed_ticks = pygame.time.get_ticks() - start_ticks
56         progress = min(1, elapsed_ticks / duration)
57         center_pos = ((window.screen.size[0] - self._logo_surface.size[0]) / 2, (
58             window.screen.size[1] - self._logo_surface.size[1]) / 2)
59
60         return (center_pos[0], center_pos[1] + displacement - displacement *
61             easeOutBack(progress))
62
63     @property
64     def logo_opacity(self):
65         return min(255, (pygame.time.get_ticks() - start_ticks) / 5)
66
67     @property
68     def duration_not_over(self):
69         return (pygame.time.get_ticks() - start_ticks) < 1500
70
71     def event_loop(self):
72         """
73             Handles events for the loading screen, no user input is taken except to
74             quit the game.
75         """
76         for event in pygame.event.get():
77             if event.type == pygame.QUIT:

```

```

75             pygame.quit()
76             sys.exit()
77
78     def draw(self):
79         """
80             Draws logo to screen.
81         """
82         window.screen.fill((0, 0, 0))
83
84         self._logo_surface.set_alpha(self.logo_opacity)
85         window.screen.blit(self._logo_surface, self.logo_position)
86
87         window.update()
88
89     def run(self):
90         """
91             Runs while the thread is still setting up our screens, or the minimum
92             loading screen duration is not reached yet.
93         """
94         while self._thread.is_alive() or self.duration_not_over:
95             self.event_loop()
96             self.draw()
97             self._clock.tick(FPS)

```

### 3.3.3 Helper functions

These files provide useful functions for different classes.

`asset_helpers.py` (Functions used for assets and pygame Surfaces)

```

1  import pygame
2  from PIL import Image
3  from functools import cache
4  from random import sample, randint
5  import math
6
7  @cache
8  def scale_and_cache(image, target_size):
9      """
10         Caches image when resized repeatedly.
11
12     Args:
13         image (pygame.Surface): Image surface to be resized.
14         target_size (tuple[float, float]): New image size.
15
16     Returns:
17         pygame.Surface: Resized image surface.
18     """
19     return pygame.transform.scale(image, target_size)
20
21 @cache
22 def smoothscale_and_cache(image, target_size):
23     """
24         Same as scale_and_cache, but with the Pygame smoothscale function.
25
26     Args:
27         image (pygame.Surface): Image surface to be resized.
28         target_size (tuple[float, float]): New image size.
29
30     Returns:
31         pygame.Surface: Resized image surface.
32     """

```

```

33     return pygame.transform.smoothscale(image, target_size)
34
35 def gif_to_frames(path):
36     """
37     Uses the PIL library to break down GIFs into individual frames.
38
39     Args:
40         path (str): Directory path to GIF file.
41
42     Yields:
43         PIL.Image: Single frame.
44     """
45     try:
46         image = Image.open(path)
47
48         first_frame = image.copy().convert('RGBA')
49         yield first_frame
50         image.seek(1)
51
52         while True:
53             current_frame = image.copy()
54             yield current_frame
55             image.seek(image.tell() + 1)
56     except EOFError:
57         pass
58
59 def get_perimeter_sample(image_size, number):
60     """
61     Used for particle drawing class, generates roughly equally distributed points
62     around a rectangular image surface's perimeter.
63
64     Args:
65         image_size (tuple[float, float]): Image surface size.
66         number (int): Number of points to be generated.
67
68     Returns:
69         list[tuple[int, int], ...]: List of random points on perimeter of image
70         surface.
71     """
72     perimeter = 2 * (image_size[0] + image_size[1])
73     # Flatten perimeter to a single number representing the distance from the top-
74     # middle of the surface going clockwise, and create a list of equally spaced
75     # points
76     perimeter_offsets = [(image_size[0] / 2) + (i * perimeter / number) for i in
77     range(0, number)]
78     pos_list = []
79
80     for perimeter_offset in perimeter_offsets:
81         # For every point, add a random offset
82         max_displacement = int(perimeter / (number * 4))
83         perimeter_offset += randint(-max_displacement, max_displacement)
84
85         if perimeter_offset > perimeter:
86             perimeter_offset -= perimeter
87
88         # Convert 1D distance back into 2D points on image surface perimeter
89         if perimeter_offset < image_size[0]:
90             pos_list.append((perimeter_offset, 0))
91         elif perimeter_offset < image_size[0] + image_size[1]:
92             pos_list.append((image_size[0], perimeter_offset - image_size[0]))
93         elif perimeter_offset < image_size[0] + image_size[1] + image_size[0]:
94             pos_list.append((perimeter_offset - image_size[0] - image_size[1],

```

```

        image_size[1]))
90     else:
91         pos_list.append((0, perimeter - perimeter_offset))
92     return pos_list
93
94 def get_angle_between_vectors(u, v, deg=True):
95     """
96     Uses the dot product formula to find the angle between two vectors.
97
98     Args:
99         u (list[int, int]): Vector 1.
100        v (list[int, int]): Vector 2.
101        deg (bool, optional): Return results in degrees. Defaults to True.
102
103    Returns:
104        float: Angle between vectors.
105    """
106    dot_product = sum(i * j for (i, j) in zip(u, v))
107    u_magnitude = math.sqrt(u[0] ** 2 + u[1] ** 2)
108    v_magnitude = math.sqrt(v[0] ** 2 + v[1] ** 2)
109
110    cos_angle = dot_product / (u_magnitude * v_magnitude)
111    radians = math.acos(min(max(cos_angle, -1), 1))
112
113    if deg:
114        return math.degrees(radians)
115    else:
116        return radians
117
118 def get_rotational_angle(u, v, deg=True):
119     """
120     Get bearing angle relative to positive x-axis centered on second vector.
121
122     Args:
123         u (list[int, int]): Vector 1.
124         v (list[int, int]): Vector 2, set as center of axes.
125         deg (bool, optional): Return results in degrees. Defaults to True.
126
127     Returns:
128        float: Bearing angle between vectors.
129    """
130    radians = math.atan2(u[1] - v[1], u[0] - v[0])
131
132    if deg:
133        return math.degrees(radians)
134    else:
135        return radians
136
137 def get_vector(src_vertex, dest_vertex):
138     """
139     Get vector describing translation between two points.
140
141     Args:
142         src_vertex (list[int, int]): Source vertex.
143         dest_vertex (list[int, int]): Destination vertex.
144
145     Returns:
146        tuple[int, int]: Vector between the two points.
147    """
148    return (dest_vertex[0] - src_vertex[0], dest_vertex[1] - src_vertex[1])
149
150 def get_next_corner(vertex, image_size):

```

```

151 """
152 Used in particle drawing system, finds coordinates of the next corner going
153 clockwise, given a point on the perimeter.
154
155 Args:
156     vertex (list[int, int]): Point on perimeter.
157     image_size (list[int, int]): Image size.
158
159 Returns:
160     list[int, int]: Coordinates of corner on perimeter.
161 """
162 corners = [(0, 0), (image_size[0], 0), (image_size[0], image_size[1]), (0,
163 image_size[1])]
164
165 if vertex in corners:
166     return corners[(corners.index(vertex) + 1) % len(corners)]
167
168 if vertex[1] == 0:
169     return (image_size[0], 0)
170 elif vertex[0] == image_size[0]:
171     return image_size
172 elif vertex[1] == image_size[1]:
173     return (0, image_size[1])
174 elif vertex[0] == 0:
175     return (0, 0)
176
177 def pil_image_to_surface(pil_image):
178 """
179 Args:
180     pil_image (PIL.Image): Image to be converted.
181
182 Returns:
183     pygame.Surface: Converted image surface.
184 """
185     return pygame.image.frombytes(pil_image.tobytes(), pil_image.size, pil_image.
186 mode).convert()
187
188 def calculate_frame_index(elapsed_milliseconds, start_index, end_index, fps):
189 """
190 Determine frame of animated GIF to be displayed.
191
192 Args:
193     elapsed_milliseconds (int): Milliseconds since GIF started playing.
194     start_index (int): Start frame of GIF.
195     end_index (int): End frame of GIF.
196     fps (int): Number of frames to be played per second.
197
198 Returns:
199     int: Displayed frame index of GIF.
200 """
201     ms_per_frame = int(1000 / fps)
202     return start_index + ((elapsed_milliseconds // ms_per_frame) % (end_index -
203 start_index))
204
205 def draw_background(screen, background, current_time=0):
206 """
207 Draws background to screen
208
209 Args:
210     screen (pygame.Surface): Screen to be drawn to
211     background (list[pygame.Surface, ...] | pygame.Surface): Background to be
212 drawn, if GIF, list of surfaces indexed to select frame to be drawn

```

```

208     current_time (int, optional): Used to calculate frame index for GIF.
209     Defaults to 0.
210     """
211     if isinstance(background, list):
212         # Animated background passed in as list of surfaces, calculate_frame_index()
213         # used to get index of frame to be drawn
214         frame_index = calculate_frame_index(current_time, 0, len(background), fps
215         =8)
216         scaled_background = scale_and_cache(background[frame_index], screen.size)
217         screen.blit(scaled_background, (0, 0))
218     else:
219         scaled_background = scale_and_cache(background, screen.size)
220         screen.blit(scaled_background, (0, 0))
221
222     def get_highlighted_icon(icon):
223         """
224             Used for pressable icons, draws overlay on icon to show as pressed.
225
226             Args:
227                 icon (pygame.Surface): Icon surface.
228
229             Returns:
230                 pygame.Surface: Icon with overlay drawn on top.
231
232         icon_copy = icon.copy()
233         overlay = pygame.Surface((icon.get_width(), icon.get_height()), pygame.
234             SRCALPHA)
235         overlay.fill((0, 0, 0, 128))
236         icon_copy.blit(overlay, (0, 0))
237         return icon_copy

```

data\_helpers.py (Functions used for file handling and JSON parsing)

```

1 import json
2 from pathlib import Path
3
4 module_path = Path(__file__).parent
5 default_file_path = (module_path / '../app_data/default_settings.json').resolve()
6 user_file_path = (module_path / '../app_data/user_settings.json').resolve()
7 themes_file_path = (module_path / '../app_data/themes.json').resolve()
8
9 def load_json(path):
10     """
11         Args:
12             path (str): Path to JSON file.
13
14         Raises:
15             Exception: Invalid file.
16
17         Returns:
18             dict: Parsed JSON file.
19
20     try:
21         with open(path, 'r') as f:
22             file = json.load(f)
23
24         return file
25     except:
26         raise Exception('Invalid JSON file (data_helpers.py)')
27
28 def get_user_settings():

```

```

29     return load_json(user_file_path)
30
31 def get_default_settings():
32     return load_json(default_file_path)
33
34 def get_themes():
35     return load_json(themes_file_path)
36
37 def update_user_settings(data):
38     """
39     Rewrites JSON file for user settings with new data.
40
41     Args:
42         data (dict): Dictionary storing updated user settings.
43
44     Raises:
45         Exception: Invalid file.
46     """
47     try:
48         with open(user_file_path, 'w') as f:
49             json.dump(data, f, indent=4)
50     except:
51         raise Exception('Invalid JSON file (data_helpers.py)')

```

#### widget\_helpers.py (Files used for creating widgets)

```

1 import pygame
2 from math import sqrt
3
4 def create_slider(size, fill_colour, border_width, border_colour):
5     """
6     Creates surface for sliders.
7
8     Args:
9         size (list[int, int]): Image size.
10        fill_colour (pygame.Color): Fill (inner) colour.
11        border_width (float): Border width.
12        border_colour (pygame.Color): Border colour.
13
14    Returns:
15        pygame.Surface: Slider image surface.
16    """
17    gradient_surface = pygame.Surface(size, pygame.SRCALPHA)
18    border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.height))
19
20    # Draws rectangle with a border radius half of image height, to draw an
21    # rectangle with semicircular cap (obround)
22    pygame.draw.rect(gradient_surface, fill_colour, border_rect, border_radius=int(
23        size[1] / 2))
24    pygame.draw.rect(gradient_surface, border_colour, border_rect, width=int(
25        border_width), border_radius=int(size[1] / 2))
26
27    return gradient_surface
28
29 def create_slider_gradient(size, border_width, border_colour):
30     """
31     Draws surface for colour slider, with a full colour gradient as fill colour.

```

```

32         border_width (float): Border width.
33         border_colour (pygame.Color): Border colour.
34
35     Returns:
36         pygame.Surface: Slider image surface.
37     """
38     gradient_surface = pygame.Surface(size, pygame.SRCALPHA)
39
40     first_round_end = gradient_surface.height / 2
41     second_round_end = gradient_surface.width - first_round_end
42     gradient_y_mid = gradient_surface.height / 2
43
44     # Iterate through length of slider
45     for i in range(gradient_surface.width):
46         draw_height = gradient_surface.height
47
48         if i < first_round_end or i > second_round_end:
49             # Draw semicircular caps if x-distance less than or greater than
50             # radius of cap (half of image height)
51             distance_from_cutoff = min(abs(first_round_end - i), abs(i -
52             second_round_end))
53             draw_height = calculate_gradient_slice_height(distance_from_cutoff,
54             gradient_surface.height / 2)
55
56             # Get colour from distance from left side of slider
57             color = pygame.Color(0)
58             color.hsva = (int(360 * i / gradient_surface.width), 100, 100, 100)
59
60             draw_rect = pygame.FRect((0, 0, 1, draw_height - 2 * border_width))
61             draw_rect.center = (i, gradient_y_mid)
62
63             pygame.draw.rect(gradient_surface, color, draw_rect)
64
65     border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.
66     height))
67     pygame.draw.rect(gradient_surface, border_colour, border_rect, width=int(
68     border_width), border_radius=int(size[1] / 2))
69
70     return gradient_surface
71
72 def calculate_gradient_slice_height(distance, radius):
73     """
74     Calculate height of vertical slice of semicircular slider cap.
75
76     Args:
77         distance (float): x-distance from center of circle.
78         radius (float): Radius of semicircle.
79
80     Returns:
81         float: Height of vertical slice.
82     """
83     return sqrt(radius ** 2 - distance ** 2) * 2 + 2
84
85 def create_slider_thumb(radius, colour, border_colour, border_width):
86     """
87     Creates surface with bordered circle.
88
89     Args:
90         radius (float): Radius of circle.
91         colour (pygame.Color): Fill colour.
92         border_colour (pygame.Color): Border colour.
93         border_width (float): Border width.

```

```

89
90     Returns:
91         pygame.Surface: Circle surface.
92     """
93     thumb_surface = pygame.Surface((radius * 2, radius * 2), pygame.SRCALPHA)
94     pygame.draw.circle(thumb_surface, border_colour, (radius, radius), radius,
95     width=int(border_width))
96     pygame.draw.circle(thumb_surface, colour, (radius, radius), (radius -
97     border_width))
98
99     return thumb_surface
100
101 def create_square_gradient(side_length, colour):
102     """
103     Creates a square gradient for the colour picker widget, gradient transitioning
104     between saturation and value.
105     Uses smoothscale to blend between colour values for individual pixels.
106
107     Args:
108         side_length (float): Length of a square side.
109         colour (pygame.Color): Colour with desired hue value.
110     """
111     square_surface = pygame.Surface((side_length, side_length))
112
113     mix_1 = pygame.Surface((1, 2))
114     mix_1.fill((255, 255, 255))
115     mix_1.set_at((0, 1), (0, 0, 0))
116     mix_1 = pygame.transform.smoothscale(mix_1, (side_length, side_length))
117
118     hue = colour.hsva[0]
119     saturated_rgb = pygame.Color(0)
120     saturated_rgb.hsva = (hue, 100, 100)
121
122     mix_2 = pygame.Surface((2, 1))
123     mix_2.fill((255, 255, 255))
124     mix_2.set_at((1, 0), saturated_rgb)
125     mix_2 = pygame.transform.smoothscale(mix_2, (side_length, side_length))
126
127     mix_1.blit(mix_2, (0, 0), special_flags=pygame.BLEND_MULT)
128
129     square_surface.blit(mix_1, (0, 0))
130
131     return square_surface
132
133 def create_switch(size, colour):
134     """
135     Creates surface for switch toggle widget.
136
137     Args:
138         size (list[int, int]): Image size.
139         colour (pygame.Color): Fill colour.
140
141     Returns:
142         pygame.Surface: Switch surface.
143     """
144     switch_surface = pygame.Surface((size[0], size[1]), pygame.SRCALPHA)
145     pygame.draw.rect(switch_surface, colour, (0, 0, size[0], size[1]),
146     border_radius=int(size[1] / 2))

```

```

147     return switch_surface
148
149 def create_text_box(size, border_width, colours):
150     """
151     Creates bordered textbox with shadow, flat, and highlighted vertical regions.
152
153     Args:
154         size (list[int, int]): Image size.
155         border_width (float): Border width.
156         colours (list[pygame.Color, ...]): List of 4 colours, representing border
157         colour, shadow colour, flat colour and highlighted colour.
158
159     Returns:
160         pygame.Surface: Textbox surface.
161     """
162
163     surface = pygame.Surface(size, pygame.SRCALPHA)
164
165     pygame.draw.rect(surface, colours[0], (0, 0, *size))
166     pygame.draw.rect(surface, colours[2], (border_width, border_width, size[0] - 2
167         * border_width, size[1] - 2 * border_width))
168     pygame.draw.rect(surface, colours[3], (border_width, border_width, size[0] - 2
169         * border_width, border_width))
170     pygame.draw.rect(surface, colours[1], (border_width, size[1] - 2 *
171         border_width, size[0] - 2 * border_width, border_width))
172
173     return surface

```

### 3.3.4 Theme

The theme manager file is responsible for providing an instance where the colour palette and dimensions for the GUI can be accessed.

theme.py

```

1 from data.utils.data_helpers import get_themes, get_user_settings
2
3 themes = get_themes()
4 user_settings = get_user_settings()
5
6 def flatten_dictionary_generator(dictionary, parent_key=None):
6     """
6     Recursive depth-first search to yield all items in a dictionary.
6
6     Args:
6         dictionary (dict): Dictionary to be iterated through.
6         parent_key (str, optional): Prefix added to every key. Defaults to None.
6
6     Yields:
6         dict | tuple[str, str]: Another dictionary or key, value pair.
6     """
6
6     for key, value in dictionary.items():
6         if parent_key:
6             new_key = parent_key + key.capitalize()
6         else:
6             new_key = key
6
6         if isinstance(value, dict):
6             yield from flatten_dictionary(value, new_key).items()
6         else:
6             yield new_key, value

```

```

28 def flatten_dictionary(dictionary, parent_key=''):
29     return dict(flatten_dictionary_generator(dictionary, parent_key))
30
31 class ThemeManager:
32     def __init__(self):
33         self.__dict__.update(flatten_dictionary(themes['colours']))
34         self.__dict__.update(flatten_dictionary(themes['dimensions']))
35
36     def __getitem__(self, arg):
37         """
38             Override default class's __getitem__ dunder method, to make retrieving an
39             instance attribute nicer with [] notation.
40
41             Args:
42                 arg (str): Attribute name.
43
44             Raises:
45                 KeyError: Instance does not have requested attribute.
46
47             Returns:
48                 str | int: Instance attribute.
49
50         item = self.__dict__.get(arg)
51
52         if item is None:
53             raise KeyError('({}.__getitem__) Requested theme item not
54             found:', arg)
55
56     return item
57
58 theme = ThemeManager()

```

## 3.4 GUI

### 3.4.1 Laser

The LaserDraw class draws the laser in both the game and review screens.

`laser_draw.py`

```

1 import pygame
2 from data.utils.board_helpers import coords_to_screen_pos
3 from data.constants import EMPTY_BB, ShaderType, Colour
4 from data.managers.animation import animation
5 from data.managers.window import window
6 from data.managers.audio import audio
7 from data.assets import GRAPHICS, SFX
8 from data.constants import LaserType
9
10 type_to_image = {
11     LaserType.END: ['laser_end_1', 'laser_end_2'],
12     LaserType.STRAIGHT: ['laser_straight_1', 'laser_straight_2'],
13     LaserType.CORNER: ['laser_corner_1', 'laser_corner_2']
14 }
15
16 GLOW_SCALE_FACTOR = 1.5
17
18 class LaserDraw:
19     def __init__(self, board_position, board_size):
20         self._board_position = board_position
21         self._square_size = board_size[0] / 10

```

```

22         self._laser_lists = []
23
24     @property
25     def firing(self):
26         return len(self._laser_lists) > 0
27
28     def add_laser(self, laser_result, laser_colour):
29         """
30             Adds a laser to the board.
31
32             Args:
33                 laser_result (Laser): Laser class instance containing laser trajectory
34                 info.
35                 laser_colour (Colour.RED | Colour.BLUE): Active colour of laser.
36
37             laser_path = laser_result.laser_path.copy()
38             laser_types = [LaserType.END]
39             # List of angles in degree to rotate the laser image surface when drawn
40             laser_rotation = [laser_path[0][1]]
41             laser_lights = []
42
43             # Iterates through every square laser passes through
44             for i in range(1, len(laser_path)):
45                 previous_direction = laser_path[i-1][1]
46                 current_coords, current_direction = laser_path[i]
47
48                 if current_direction == previous_direction:
49                     laser_types.append(LaserType.STRAIGHT)
50                     laser_rotation.append(current_direction)
51                 elif current_direction == previous_direction.get_clockwise():
52                     laser_types.append(LaserType.CORNER)
53                     laser_rotation.append(current_direction)
54                 elif current_direction == previous_direction.get_anticlockwise():
55                     laser_types.append(LaserType.CORNER)
56                     laser_rotation.append(current_direction.get_anticlockwise())
57
58             # Adds a shader ray effect on the first and last square of the laser
59             # trajectory
60             if i in [1, len(laser_path) - 1]:
61                 abs_position = coords_to_screen_pos(current_coords, self.
62                 _board_position, self._square_size)
63                 laser_lights.append([
64                     (abs_position[0] / window.size[0], abs_position[1] / window.
65                     size[1]),
66                     0.35,
67                     (0, 0, 255) if laser_colour == Colour.BLUE else (255, 0, 0),
68                 ])
69
70             # Sets end laser draw type if laser hits a piece
71             if laser_result.hit_square_bitboard != EMPTY_BB:
72                 laser_types[-1] = LaserType.END
73                 laser_path[-1] = (laser_path[-1][0], laser_path[-2][1].get_opposite())
74                 laser_rotation[-1] = laser_path[-2][1].get_opposite()
75
76                 audio.play_sfx(SFX['piece_destroy'])
77
78             laser_path = [(coords, rotation, type) for (coords, dir), rotation, type
79             in zip(laser_path, laser_rotation, laser_types)]
80             self._laser_lists.append((laser_path, laser_colour))
81
82             window.clear_effect(ShaderType.RAYS)
83             window.set_effect(ShaderType.RAYS, lights=laser_lights)

```

```

79         animation.set_timer(1000, self.remove_laser)
80
81     audio.play_sfx(SFX['laser_1'])
82     audio.play_sfx(SFX['laser_2'])
83
84     def remove_laser(self):
85         """
86             Removes a laser from the board.
87         """
88         self._laser_lists.pop(0)
89
90         if len(self._laser_lists) == 0:
91             window.clear_effect(ShaderType.RAYS)
92
93     def draw_laser(self, screen, laser_list, glow=True):
94         """
95             Draws every laser on the screen.
96
97             Args:
98                 screen (pygame.Surface): The screen to draw on.
99                 laser_list (list): The list of laser segments to draw.
100                glow (bool, optional): Whether to draw a glow effect. Defaults to True
101
102            laser_path, laser_colour = laser_list
103            laser_list = []
104            glow_list = []
105
106            for coords, rotation, type in laser_path:
107                square_x, square_y = coords_to_screen_pos(coords, self._board_position
108                , self._square_size)
109
110                image = GRAPHICS[type_to_image[type][laser_colour]]
111                rotated_image = pygame.transform.rotate(image, rotation.to_angle())
112                scaled_image = pygame.transform.scale(rotated_image, (self.
113                _square_size + 1, self._square_size + 1)) # +1 to prevent rounding creating
114                black lines
115                laser_list.append((scaled_image, (square_x, square_y)))
116
117                # Scales up the laser image surface as a glow surface
118                scaled_glow = pygame.transform.scale(rotated_image, (self._square_size
119                * GLOW_SCALE_FACTOR, self._square_size * GLOW_SCALE_FACTOR))
120                offset = self._square_size * ((GLOW_SCALE_FACTOR - 1) / 2)
121                glow_list.append((scaled_glow, (square_x - offset, square_y - offset)))
122
123
124    def draw(self, screen):
125        """
126            Draws all lasers on the screen.
127
128            Args:
129                screen (pygame.Surface): The screen to draw on.
130            """
131            for laser_list in self._laser_lists:
132                self.draw_laser(screen, laser_list)
133
134

```

```

135     def handle_resize(self, board_position, board_size):
136         """
137             Handles resizing of the board.
138
139         Args:
140             board_position (tuple[int, int]): The new position of the board.
141             board_size (tuple[int, int]): The new size of the board.
142         """
143         self._board_position = board_position
144         self._square_size = board_size[0] / 10

```

### 3.4.2 Particles

The `ParticlesDraw` class draws particles in both the game and review screens. The particles are either fragmented pieces when destroyed, or laser particles emitted from the Sphinx. Particles are given custom velocity, rotation, opacity and size parameters.

`particles_draw.py`

```

1 import pygame
2 from random import randint
3 from data.utils.asset_helpers import get_perimeter_sample, get_vector,
4     get_angle_between_vectors, get_next_corner
5 from data.states.game.components.piece_sprite import PieceSprite
6
7 class ParticlesDraw:
8     def __init__(self, gravity=0.2, rotation=180, shrink=0.5, opacity=150):
9         self._particles = []
10        self._glow_particles = []
11
12        self._gravity = gravity
13        self._rotation = rotation
14        self._shrink = shrink
15        self._opacity = opacity
16
17    def fragment_image(self, image, number):
18        image_size = image.get_rect().size
19        """
20            1. Takes an image surface and samples random points on the perimeter.
21            2. Iterates through points, and depending on the nature of two consecutive
22                points, finds a corner between them.
23            3. Draws a polygon with the points as the vertices to mask out the area
24                not in the fragment.
25
26        Args:
27            image (pygame.Surface): Image to fragment.
28            number (int): The number of fragments to create.
29
30        Returns:
31            list[pygame.Surface]: List of image surfaces with fragment of original
32            surface drawn on top.
33        """
34        center = image.get_rect().center
35        points_list = get_perimeter_sample(image_size, number)
36        fragment_list = []
37
38        points_list.append(points_list[0])
39
40        # Iterate through points_list, using the current point and the next one
41        for i in range(len(points_list) - 1):
42            vertex_1 = points_list[i]

```

```

39         vertex_2 = points_list[i + 1]
40         vector_1 = get_vector(center, vertex_1)
41         vector_2 = get_vector(center, vertex_2)
42         angle = get_angle_between_vectors(vector_1, vector_2)
43
44         cropped_image = pygame.Surface(image_size, pygame.SRCALPHA)
45         cropped_image.fill((0, 0, 0, 0))
46         cropped_image.blit(image, (0, 0))
47
48         corners_to_draw = None
49
50         if vertex_1[0] == vertex_2[0] or vertex_1[1] == vertex_2[1]: # Points
51             on the same side
52             corners_to_draw = 4
53
54         elif abs(vertex_1[0] - vertex_2[0]) == image_size[0] or abs(vertex_1
55             [1] - vertex_2[1]) == image_size[1]: # Points on opposite sides
56             corners_to_draw = 2
57
58         elif angle < 180: # Points on adjacent sides
59             corners_to_draw = 3
60
61     else:
62         corners_to_draw = 1
63
64     corners_list = []
65     for j in range(corners_to_draw):
66         if len(corners_list) == 0:
67             corners_list.append(get_next_corner(vertex_2, image_size))
68         else:
69             corners_list.append(get_next_corner(corners_list[-1],
70                 image_size))
71
72     pygame.draw.polygon(cropped_image, (0, 0, 0, 0), (center, vertex_2, *
73     corners_list, vertex_1))
74
75     fragment_list.append(cropped_image)
76
77     return fragment_list
78
79 def add_captured_piece(self, piece, colour, rotation, position, size):
80     """
81     Adds a captured piece to fragment into particles.
82
83     Args:
84         piece (Piece): The piece type.
85         colour (Colour): The active colour of the piece.
86         rotation (int): The rotation of the piece.
87         position (tuple[int, int]): The position where particles originate
88         from.
89         size (tuple[int, int]): The size of the piece.
90     """
91
92     piece_sprite = PieceSprite(piece, colour, rotation)
93     piece_sprite.set_geometry((0, 0), size)
94     piece_sprite.set_image()
95
96     particles = self.fragment_image(piece_sprite.image, 5)
97
98     for particle in particles:
99         self.add_particle(particle, position)
100
101 def add_sparks(self, radius, colour, position):

```

```

96     """
97     Adds laser spark particles.
98
99     Args:
100         radius (int): The radius of the sparks.
101         colour (Colour): The active colour of the sparks.
102         position (tuple[int, int]): The position where particles originate
103         from.
104         """
105         for i in range(randint(10, 15)):
106             velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
107             random_colour = [min(max(val + randint(-20, 20), 0), 255) for val in
108                 colour]
109             self._particles.append([None, [radius, random_colour], [*position],
110             velocity, 0])
111
112     def add_particle(self, image, position):
113         """
114         Adds a particle.
115
116         Args:
117             image (pygame.Surface): The image of the particle.
118             position (tuple): The position of the particle.
119             """
120             velocity = [randint(-15, 15) / 10, randint(-20, 0) / 10]
121
122             # Each particle is stored with its attributes: [surface, copy of surface,
123             position, velocity, lifespan]
124             self._particles.append([image, image.copy(), [*position], velocity, 0])
125
126     def update(self):
127         """
128             Updates each particle and its attributes.
129
130             for i in range(len(self._particles) - 1, -1, -1):
131                 particle = self._particles[i]
132
133                 #update position
134                 particle[2][0] += particle[3][0]
135                 particle[2][1] += particle[3][1]
136
137                 #update lifespan
138                 self._particles[i][4] += 0.01
139
140                 if self._particles[i][4] >= 1:
141                     self._particles.pop(i)
142                     continue
143
144                 if isinstance(particle[1], pygame.Surface): # Particle is a piece
145                     # Update velocity
146                     particle[3][1] += self._gravity
147
148                     # Update size
149                     image_size = particle[1].get_rect().size
150                     end_size = ((1 - self._shrink) * image_size[0], (1 - self._shrink)
151                     * image_size[1])
152                     target_size = (image_size[0] - particle[4] * (image_size[0] -
153                     end_size[0]), image_size[1] - particle[4] * (image_size[1] - end_size[1]))
154
155                     # Update rotation
156                     rotation = (self._rotation if particle[3][0] <= 0 else -self.
157                     _rotation) * particle[4]

```

```

151
152         updated_image = pygame.transform.scale(pygame.transform.rotate(
153             particle[1], rotation), target_size)
154
155     elif isinstance(particle[1], list): # Particle is a spark
156         # Update size
157         end_radius = (1 - self._shrink) * particle[1][0]
158         target_radius = particle[1][0] - particle[4] * (particle[1][0] -
159             end_radius)
160
161         updated_image = pygame.Surface((target_radius * 2, target_radius *
162             2), pygame.SRCALPHA)
163         pygame.draw.circle(updated_image, particle[1][1], (target_radius,
164             target_radius), target_radius)
165
166         # Update opacity
167         alpha = 255 - particle[4] * (255 - self._opacity)
168
169         updated_image.fill((255, 255, 255, alpha), None, pygame.
170             BLEND_RGBA_MULT)
171
172         particle[0] = updated_image
173
174     def draw(self, screen):
175         """
176             Draws the particles, indexing the surface and position attributes for each
177             particle.
178
179         Args:
180             screen (pygame.Surface): The screen to draw on.
181         """
182
183         screen.blit([
184             (particle[0], particle[2]) for particle in self._particles
185         ])

```

### 3.4.3 Widget Bases

Widget bases are the base classes for my widgets system. They contain both attributes and getter methods that provide basic functionality such as size and position, and abstract methods to be overridden. These bases are also designed to be used with multiple inheritance, where multiple bases can be combined to add functionality to the final widget. Encapsulation also allows me to simplify interactions between widgets, as using getter methods instead of protected attributes allows me to add logic while accessing an attribute, such as in `widget.py`, where the logic to fetch the parent surface instead of the windows screen is hidden within the base class.

## Widget

All widgets are a subclass of the `Widget` class.

`widget.py`

```

1 import pygame
2 from data.constants import SCREEN_SIZE
3 from data.managers.theme import theme
4 from data.assets import DEFAULT_FONT
5
6 DEFAULT_SURFACE_SIZE = SCREEN_SIZE
7 REQUIRED_KWARGS = ['relative_position', 'relative_size']
8

```

```

9  class _Widget(pygame.sprite.Sprite):
10     def __init__(self, **kwargs):
11         """
12             Every widget has the following attributes:
13
14             surface (pygame.Surface): The surface the widget is drawn on.
15             raw_surface_size (tuple[int, int]): The initial size of the window screen,
16                 remains constant.
17             parent (_Widget, optional): The parent widget position and size is
18                 relative to.
19
20                 Relative to current surface:
21                 relative_position (tuple[float, float]): The position of the widget
22                     relative to its surface.
23                 relative_size (tuple[float, float]): The scale of the widget relative to
24                     its surface.
25
26                 Remains constant, relative to initial screen size:
27                 relative_font_size (float, optional): The relative font size of the widget
28
29                 relative_margin (float): The relative margin of the widget.
30                 relative_border_width (float): The relative border width of the widget.
31                 relative_border_radius (float): The relative border radius of the widget.
32
33                 anchor_x (str): The horizontal anchor direction ('left', 'right', 'center'
34                    ').
35                 anchor_y (str): The vertical anchor direction ('top', 'bottom', 'center').
36                 fixed_position (tuple[int, int], optional): The fixed position of the
37                     widget in pixels.
38                 border_colour (pygame.Color): The border color of the widget.
39                 text_colour (pygame.Color): The text color of the widget.
40                 fill_colour (pygame.Color): The fill color of the widget.
41                 font (pygame.freetype.Font): The font used for the widget.
42             """
43             super().__init__()
44
45             for required_kwarg in REQUIRED_KWARGS:
46                 if required_kwarg not in kwargs:
47                     raise KeyError(f'({_Widget.__init__}) Required keyword "{
48                         required_kwarg}" not in base kwargs')
49
50             self._surface = None # Set in WidgetGroup, as needs to be reassigned every
51                 frame
52             self._raw_surface_size = DEFAULT_SURFACE_SIZE
53
54             self._parent = kwargs.get('parent')
55
56             self._relative_font_size = None # Set in subclass
57
58             self._relative_position = kwargs.get('relative_position')
59             self._relative_margin = theme['margin'] / self._raw_surface_size[1]

```

```

60         self._anchor_y = kwargs.get('anchor_y') or 'top'
61         self._fixed_position = kwargs.get('fixed_position')
62         scale_mode = kwargs.get('scale_mode') or 'both'
63
64     if kwargs.get('relative_size'):
65         match scale_mode:
66             case 'height':
67                 self._relative_size = kwargs.get('relative_size')
68             case 'width':
69                 self._relative_size = ((kwargs.get('relative_size')[0] * self.
70 surface_size[0]) / self.surface_size[1], (kwargs.get('relative_size')[1] *
71 self.surface_size[0]) / self.surface_size[1])
72             case 'both':
73                 self._relative_size = ((kwargs.get('relative_size')[0] * self.
74 surface_size[0]) / self.surface_size[1], kwargs.get('relative_size')[1])
75             case _:
76                 raise ValueError('_Widget.__init__) Unknown scale mode:', scale_mode)
77         else:
78             self._relative_size = (1, 1)
79
80     if 'margin' in kwargs:
81         self._relative_margin = kwargs.get('margin') / self._raw_surface_size[1]
82
83         if (self._relative_margin * 2) > min(self._relative_size[0], self._relative_size[1]):
84             raise ValueError('_Widget.__init__) Margin larger than specified size!')
85
86     if 'border_width' in kwargs:
87         self._relative_border_width = kwargs.get('border_width') / self._raw_surface_size[1]
88
89     if 'border_radius' in kwargs:
90         self._relative_border_radius = kwargs.get('border_radius') / self._raw_surface_size[1]
91
92     if 'border_colour' in kwargs:
93         self._border_colour = pygame.Color(kwargs.get('border_colour'))
94
95     if 'fill_colour' in kwargs:
96         self._fill_colour = pygame.Color(kwargs.get('fill_colour'))
97
98     if 'text_colour' in kwargs:
99         self._text_colour = pygame.Color(kwargs.get('text_colour'))
100
101     if 'font' in kwargs:
102         self._font = kwargs.get('font')
103
104     @property
105     def surface_size(self):
106         """
107             Gets the size of the surface widget is drawn on.
108             Can be either the window size, or another widget size if assigned to a parent.
109
110             Returns:
111                 tuple[int, int]: The size of the surface.
112         """
113         if self._parent:
114             return self._parent.size

```

```

112         else:
113             return self._raw_surface_size
114
115     @property
116     def position(self):
117         """
118             Gets the position of the widget.
119             Accounts for fixed position attribute, where widget is positioned in
120             pixels regardless of screen size.
121             Accounts for anchor direction, where position attribute is calculated
122             relative to one side of the screen.
123
124             Returns:
125                 tuple[int, int]: The position of the widget.
126             """
127
128         x, y = None, None
129         if self._fixed_position:
130             x, y = self._fixed_position
131         if x is None:
132             x = self._relative_position[0] * self.surface_size[0]
133         if y is None:
134             y = self._relative_position[1] * self.surface_size[1]
135
136         if self._anchor_x == 'left':
137             x = x
138         elif self._anchor_x == 'right':
139             x = self.surface_size[0] - x - self.size[0]
140         elif self._anchor_x == 'center':
141             x = (self.surface_size[0] / 2 - self.size[0] / 2) + x
142
143         if self._anchor_y == 'top':
144             y = y
145         elif self._anchor_y == 'bottom':
146             y = self.surface_size[1] - y - self.size[1]
147         elif self._anchor_y == 'center':
148             y = (self.surface_size[1] / 2 - self.size[1] / 2) + y
149
150         # Position widget relative to parent, if exists.
151         if self._parent:
152             return (x + self._parent.position[0], y + self._parent.position[1])
153         else:
154             return (x, y)
155
156     @property
157     def size(self):
158         return (self._relative_size[0] * self.surface_size[1], self._relative_size
159             [1] * self.surface_size[1])
160
161     @property
162     def margin(self):
163         return self._relative_margin * self._raw_surface_size[1]
164
165     @property
166     def border_width(self):
167         return self._relative_border_width * self._raw_surface_size[1]
168
169     @property
170     def border_radius(self):
171         return self._relative_border_radius * self._raw_surface_size[1]
172
173     @property
174     def font_size(self):

```

```

171         return self._relative_font_size * self.surface_size[1]
172
173     def set_image(self):
174         """
175             Abstract method to draw widget.
176         """
177         raise NotImplementedError
178
179     def set_geometry(self):
180         """
181             Sets the position and size of the widget.
182         """
183         self.rect = self.image.get_rect()
184
185         if self._anchor_x == 'left':
186             if self._anchor_y == 'top':
187                 self.rect.topleft = self.position
188             elif self._anchor_y == 'bottom':
189                 self.rect.topleft = self.position
190             elif self._anchor_y == 'center':
191                 self.rect.topleft = self.position
192         elif self._anchor_x == 'right':
193             if self._anchor_y == 'top':
194                 self.rect.topleft = self.position
195             elif self._anchor_y == 'bottom':
196                 self.rect.topleft = self.position
197             elif self._anchor_y == 'center':
198                 self.rect.topleft = self.position
199         elif self._anchor_x == 'center':
200             if self._anchor_y == 'top':
201                 self.rect.topleft = self.position
202             elif self._anchor_y == 'bottom':
203                 self.rect.topleft = self.position
204             elif self._anchor_y == 'center':
205                 self.rect.topleft = self.position
206
207     def set_surface_size(self, new_surface_size):
208         """
209             Sets the new size of the surface widget is drawn on.
210
211             Args:
212                 new_surface_size (tuple[int, int]): The new size of the surface.
213
214         """
215         self._raw_surface_size = new_surface_size
216
217     def process_event(self, event):
218         """
219             Abstract method to handle events.
220
221             Args:
222                 event (pygame.Event): The event to process.
223
224         """
225         raise NotImplementedError

```

## Circular

The Circular class provides functionality to support widgets which rotate between text/icons.

`circular.py`

```

1 from data.components.circular_linked_list import CircularLinkedList
2

```

```

3  class _Circular:
4      def __init__(self, items_dict, **kwargs):
5          # The key, value pairs are stored within a dictionary, while the keys to
6          # access them are stored within circular linked list.
7          self._items_dict = items_dict
8          self._keys_list = CircularLinkedList(list(items_dict.keys()))
9
10     @property
11     def current_key(self):
12         """
13             Gets the current head node of the linked list, and returns a key stored as
14             the node data.
15             Returns:
16                 Data of linked list head.
17             """
18
19     @property
20     def current_item(self):
21         """
22             Gets the value in self._items_dict with the key being self.current_key.
23             Returns:
24                 Value stored with key being current head of linked list.
25             """
26
27     def set_next_item(self):
28         """
29             Sets the next item in as the current item.
30             """
31
32         self._keys_list.shift_head()
33
34     def set_previous_item(self):
35         """
36             Sets the previous item as the current item.
37             """
38
39         self._keys_list.unshift_head()
40
41     def set_to_key(self, key):
42         """
43             Sets the current item to the specified key.
44
45             Args:
46                 key: The key to set as the current item.
47
48             Raises:
49                 ValueError: If no nodes within the circular linked list contains the
50                 key as its data.
51
52         if self._keys_list.data_in_list(key) is False:
53             raise ValueError('(_Circular.set_to_key) Key not found:', key)
54
55         for _ in range(len(self._items_dict)):
56             if self.current_key == key:
57                 self.set_image()
58                 self.set_geometry()
59                 return
60
61         self.set_next_item()

```

## Circular Linked List

The `CircularLinkedList` class implements a circular doubly-linked list. Used for the internal logic of the `Circular` class.

`circular_linked_list.py`

```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.next = None
5          self.previous = None
6
7  class CircularLinkedList:
8      def __init__(self, list_to_convert=None):
9          """
10          Initialises a CircularLinkedList object.
11
12          Args:
13              list_to_convert (list, optional): Creates a linked list from existing
14              items. Defaults to None.
15          """
16          self._head = None
17
18          if list_to_convert:
19              for item in list_to_convert:
20                  self.insert_at_end(item)
21
22  def __str__(self):
23      """
24      Returns a string representation of the circular linked list.
25
26      Returns:
27          str: Linked list formatted as string.
28      """
29      if self._head is None:
30          return '| empty |'
31
32      characters = '| -> '
33      current_node = self._head
34      while True:
35          characters += str(current_node.data) + ' -> '
36          current_node = current_node.next
37
38          if current_node == self._head:
39              characters += '|'
40
41      def insert_at_beginning(self, data):
42          """
43          Inserts a node at the beginning of the circular linked list.
44
45          Args:
46              data: The data to insert.
47          """
48          new_node = Node(data)
49
50          if self._head is None:
51              self._head = new_node
52              new_node.next = self._head
53              new_node.previous = self._head
54          else:
55              new_node.next = self._head
```

```

56         new_node.previous = self._head.previous
57         self._head.previous.next = new_node
58         self._head.previous = new_node
59
60         self._head = new_node
61
62     def insert_at_end(self, data):
63         """
64             Inserts a node at the end of the circular linked list.
65
66             Args:
67                 data: The data to insert.
68             """
69         new_node = Node(data)
70
71         if self._head is None:
72             self._head = new_node
73             new_node.next = self._head
74             new_node.previous = self._head
75         else:
76             new_node.next = self._head
77             new_node.previous = self._head.previous
78             self._head.previous.next = new_node
79             self._head.previous = new_node
80
81     def insert_at_index(self, data, index):
82         """
83             Inserts a node at a specific index in the circular linked list.
84             The head node is taken as index 0.
85
86             Args:
87                 data: The data to insert.
88                 index (int): The index to insert the data at.
89
90             Raises:
91                 ValueError: Index is out of range.
92             """
93         if index < 0:
94             raise ValueError('Invalid index! (CircularLinkedList.insert_at_index)')
95
96         if index == 0 or self._head is None:
97             self.insert_at_beginning(data)
98         else:
99             new_node = Node(data)
100            current_node = self._head
101            count = 0
102
103            while count < index - 1 and current_node.next != self._head:
104                current_node = current_node.next
105                count += 1
106
107            if count == (index - 1):
108                new_node.next = current_node.next
109                new_node.previous = current_node
110                current_node.next = new_node
111            else:
112                raise ValueError('Index out of range! (CircularLinkedList.
113                insert_at_index)')
114
115     def delete(self, data):
116         """

```

```

116     Deletes a node with the specified data from the circular linked list.
117
118     Args:
119         data: The data to delete.
120
121     Raises:
122         ValueError: No nodes in the list contain the specified data.
123     """
124     if self._head is None:
125         return
126
127     current_node = self._head
128
129     while current_node.data != data:
130         current_node = current_node.next
131
132         if current_node == self._head:
133             raise ValueError('Data not found in circular linked list! (' +
CircularLinkedList.delete)')
134
135         if self._head.next == self._head:
136             self._head = None
137         else:
138             current_node.previous.next = current_node.next
139             current_node.next.previous = current_node.previous
140
141     def data_in_list(self, data):
142         """
143             Checks if the specified data is in the circular linked list.
144
145             Args:
146                 data: The data to check.
147
148             Returns:
149                 bool: True if the data is in the list, False otherwise.
150         """
151         if self._head is None:
152             return False
153
154         current_node = self._head
155         while True:
156             if current_node.data == data:
157                 return True
158
159             current_node = current_node.next
160             if current_node == self._head:
161                 return False
162
163     def shift_head(self):
164         """
165             Shifts the head of the circular linked list to the next node.
166         """
167         self._head = self._head.next
168
169     def unshift_head(self):
170         """
171             Shifts the head of the circular linked list to the previous node.
172         """
173         self._head = self._head.previous
174
175     def get_head(self):
176         """

```

```

177     Gets the head node of the circular linked list.
178
179     Returns:
180         Node: The head node.
181     """
182     return self._head

```

### 3.4.4 Widgets

Each state contains a `WIDGET_DICT` map, which contains and initialises each widget with their own attributes, and provides references to run methods on them in the state code. Each `WIDGET_DICT` is passed into a `widgetGroup` object, which is responsible for drawing, resizing and handling all widgets for the current state. Below is a list of all the widgets I have implemented:

- BoardThumbnailButton
- MultipleIconButton
- ReactiveIconButton
- BoardThumbnail
- ReactiveButton
- VolumeSlider
- ColourPicker
- ColourButton
- BrowserStrip
- PieceDisplay
- BrowserItem
- TextButton
- IconButton
- ScrollArea
- Chessboard
- TextInput
- Rectangle
- MoveList
- Dropdown
- Carousel
- Switch
- Timer
- Text
- Icon
- (`_ColourDisplay`)
- (`_ColourSquare`)
- (`_ColourSlider`)
- (`_SliderThumb`)
- (`_Scrollbar`)

### CustomEvent

The `CustomEvent` class is used to pass data between states and widgets. An event argument is passed into interactive widgets; When a widget wants to pass data back to the state, it returns the event, and adds any attributes that is required. The state then receives and handles these returned events accordingly.

#### `custom_event.py`

```

1 from data.constants import GameEventType, SettingsEventType, ConfigEventType,
2     BrowserEventType, EditorEventType
3
4 required_args = {
5     GameEventType.BOARD_CLICK: ['coords'],
6     GameEventType.ROTATE_PIECE: ['rotation_direction'],
7     GameEventType.SET_LASER: ['laser_result'],
8     GameEventType.UPDATE_PIECES: ['move_notation'],
9     GameEventType.TIMER_END: ['active_colour'],
10    GameEventType.PIECE_DROP: ['coords', 'piece', 'colour', 'rotation',
11        'remove_overlay'],
12    SettingsEventType.COLOUR_SLIDER_SLIDE: ['colour'],
13    SettingsEventType.PRIMARY_COLOUR_PICKER_CLICK: ['colour'],
14    SettingsEventType.SECONDARY_COLOUR_PICKER_CLICK: ['colour'],
15    SettingsEventType.DROPDOWN_CLICK: ['selected_word'],

```

```

14     SettingsEventType.VOLUME_SLIDER_CLICK: ['volume', 'volume_type'],
15     SettingsEventType.SHADER_PICKER_CLICK: ['data'],
16     SettingsEventType.PARTICLES_CLICK: ['toggled'],
17     SettingsEventType.OPENGL_CLICK: ['toggled'],
18     ConfigEventType.TIME_TYPE: ['time'],
19     ConfigEventType.FEN_STRING_TYPE: ['time'],
20     ConfigEventType.CPU_DEPTH_CLICK: ['data'],
21     ConfigEventType.PVC_CLICK: ['data'],
22     ConfigEventType.PRESET_CLICK: ['fen_string'],
23     BrowserEventType.BROWSER_STRIP_CLICK: ['selected_index'],
24     BrowserEventType.PAGE_CLICK: ['data'],
25     EditorEventType.PICK_PIECE_CLICK: ['piece', 'active_colour'],
26     EditorEventType.ROTATE_PIECE_CLICK: ['rotation_direction'],
27 }
28
29 class CustomEvent():
30     def __init__(self, type, **kwargs):
31         self.__dict__.update(kwargs)
32         self.type = type
33
34     @classmethod
35     def create_event(event_cls, event_type, **kwargs):
36         """
37             @classmethod Factory method used to instance CustomEvent object, to check
38             for required keyword arguments
39
40             Args:
41                 event_cls (CustomEvent): Reference to own class.
42                 event_type: The state EventType.
43
44             Raises:
45                 ValueError: If required keyword argument for passed event type not
46                 present.
47                 ValueError: If keyword argument passed is not required for passed
48                 event type.
49
50             Returns:
51                 CustomEvent: Initialised CustomEvent instance.
52             """
53
54         if event_type in required_args:
55
56             for required_arg in required_args[event_type]:
57                 if required_arg not in kwargs:
58                     raise ValueError(f"Argument '{required_arg}' required for {event_type.name} event ({GameEvent.create_event})")
59
60             for kwarg in kwargs:
61                 if kwarg not in required_args[event_type]:
62                     raise ValueError(f"Argument '{kwarg}' not included in
63             required_args dictionary for event '{event_type}'! ({GameEvent.create_event})")
64
65         return event_cls(event_type, **kwargs)
66
67     else:
68         return event_cls(event_type)

```

## ReactiveIconButton

The `ReactiveIconButton` widget is a pressable button that changes the icon displayed when it is hovered or pressed.

```

reactive_icon_button.py

1 from data.widgets.reactive_button import ReactiveButton
2 from data.constants import WidgetState
3 from data.widgets.icon import Icon
4
5 class ReactiveIconButton(ReactiveButton):
6     def __init__(self, base_icon, hover_icon, press_icon, **kwargs):
7         # Composition is used here, to initialise the Icon widgets for each widget
8         state
9             widgets_dict = {
10                 WidgetState.BASE: Icon(
11                     parent=kwargs.get('parent'),
12                     relative_size=kwargs.get('relative_size'),
13                     relative_position=(0, 0),
14                     icon=base_icon,
15                     fill_colour=(0, 0, 0, 0),
16                     border_width=0,
17                     margin=0,
18                     fit_icon=True,
19                 ),
20                 WidgetState.HOVER: Icon(
21                     parent=kwargs.get('parent'),
22                     relative_size=kwargs.get('relative_size'),
23                     relative_position=(0, 0),
24                     icon=hover_icon,
25                     fill_colour=(0, 0, 0, 0),
26                     border_width=0,
27                     margin=0,
28                     fit_icon=True,
29                 ),
30                 WidgetState.PRESS: Icon(
31                     parent=kwargs.get('parent'),
32                     relative_size=kwargs.get('relative_size'),
33                     relative_position=(0, 0),
34                     icon=press_icon,
35                     fill_colour=(0, 0, 0, 0),
36                     border_width=0,
37                     margin=0,
38                     fit_icon=True,
39                 )
40             }
41             super().__init__(
42                 widgets_dict=widgets_dict,
43                 **kwargs
44             )

```

## ReactiveButton

The `ReactiveButton` widget is the parent class for `ReactiveIconButton`. It provides the methods for clicking, rotating between widget states, positioning etc.

`reactive_button.py`

```

1 from data.components.custom_event import CustomEvent
2 from data.widgets.bases.pressable import _Pressable
3 from data.widgets.bases.circular import _Circular
4 from data.widgets.bases.widget import _Widget
5 from data.constants import WidgetState
6
7 class ReactiveButton(_Pressable, _Circular, _Widget):

```

```

8     def __init__(self, widgets_dict, event, center=False, **kwargs):
9         # Multiple inheritance used here, to combine the functionality of multiple
10        super_classes
11            _Pressable.__init__(
12                self,
13                event=event,
14                hover_func=lambda: self.set_to_key(WidgetState.HOVER),
15                down_func=lambda: self.set_to_key(WidgetState.PRESS),
16                up_func=lambda: self.set_to_key(WidgetState.BASE),
17                **kwargs
18            )
19            # Aggregation used to cycle between external widgets
20            _Circular.__init__(self, items_dict=widgets_dict)
21            _Widget.__init__(self, **kwargs)
22
23            self._center = center
24
25            self.initialise_new_colours(self._fill_colour)
26
27    @property
28    def position(self):
29        """
30            Overrides position getter method, to always position icon in the center if
31            self._center is True.
32
33            Returns:
34                list[int, int]: Position of widget.
35        """
36        position = super().position
37
38        if self._center:
39            self._size_diff = (self.size[0] - self.rect.width, self.size[1] - self.
40                .rect.height)
41            return (position[0] + self._size_diff[0] / 2, position[1] + self.
42                _size_diff[1] / 2)
43        else:
44            return position
45
46    def set_image(self):
47        """
48            Sets current icon to image.
49        """
50
51        self.current_item.set_image()
52        self.image = self.current_item.image
53
54    def set_geometry(self):
55        """
56            Sets size and position of widget.
57        """
58        super().set_geometry()
59        self.current_item.set_geometry()
60        self.current_item.rect.topleft = self.rect.topleft
61
62    def set_surface_size(self, new_surface_size):
63        """
64            Overrides base method to resize every widget state icon, not just the
65            current one.
66
67            Args:
68                new_surface_size (list[int, int]): New surface size.
69            """
70        super().set_surface_size(new_surface_size)

```

```

65         for item in self._items_dict.values():
66             item.set_surface_size(new_surface_size)
67
68     def process_event(self, event):
69         """
70         Processes Pygame events.
71
72         Args:
73             event (pygame.Event): Event to process.
74
75         Returns:
76             CustomEvent: CustomEvent of current item, with current key included
77         """
78         widget_event = super().process_event(event)
79         self.current_item.process_event(event)
80
81         if widget_event:
82             return CustomEvent(**vars(widget_event), data=self.current_key)

```

## ColourSlider

The `ColourSlider` widget is instanced in the `ColourPicker` class. It provides a slider for changing between hues for the colour picker, using the functionality of the `SliderThumb` class.

`colour_slider.py`

```

1  import pygame
2  from data.utils.widget_helpers import create_slider_gradient
3  from data.utils.asset_helpers import smoothscale_and_cache
4  from data.widgets.slider_thumb import _SliderThumb
5  from data.widgets.bases.widget import _Widget
6  from data.constants import WidgetState
7
8  class _ColourSlider(_Widget):
9      def __init__(self, relative_width, **kwargs):
10          super().__init__(relative_size=(relative_width, relative_width * 0.2), **
11                           kwargs)
12
13          # Initialise slider thumb.
14          self._thumb = _SliderThumb(radius=self.size[1] / 2, border_colour=self.
15                                      _border_colour)
16
17          self._selected_percent = 0
18          self._last_mouse_x = None
19
20          self._gradient_surface = create_slider_gradient(self.gradient_size, self.
21                                                       border_width, self._border_colour)
22          self._empty_surface = pygame.Surface(self.size, pygame.SRCALPHA)
23
24
25          @property
26          def gradient_size(self):
27              return (self.size[0] - 2 * (self.size[1] / 2), self.size[1] / 2)
28
29
30          @property
31          def gradient_position(self):
32              return (self.size[1] / 2, self.size[1] / 4)
33
34
35          @property
36          def thumb_position(self):
37              return (self.gradient_size[0] * self._selected_percent, 0)
38
39
40          @property

```

```

34     def selected_colour(self):
35         colour = pygame.Color(0)
36         colour.hsva = (int(self._selected_percent * 360), 100, 100, 100)
37         return colour
38
39     def calculate_gradient_percent(self, mouse_pos):
40         """
41             Calculate what percentage slider thumb is at based on change in mouse
42             position.
43
44             Args:
45                 mouse_pos (list[int, int]): Position of mouse on window screen.
46
47             Returns:
48                 float: Slider scroll percentage.
49
50             if self._last_mouse_x is None:
51                 return
52
53             x_change = (mouse_pos[0] - self._last_mouse_x) / (self.gradient_size[0] -
54             2 * self.border_width)
55             return max(0, min(self._selected_percent + x_change, 1))
56
57     def relative_to_global_position(self, position):
58         """
59             Transforms position from being relative to widget rect, to window screen.
60
61             Args:
62                 position (list[int, int]): Position relative to widget rect.
63
64             Returns:
65                 list[int, int]: Position relative to window screen.
66
67             relative_x, relative_y = position
68             return (relative_x + self.position[0], relative_y + self.position[1])
69
70     def set_colour(self, new_colour):
71         """
72             Sets selected_percent based on the new colour's hue.
73
74             Args:
75                 new_colour (pygame.Color): New slider colour.
76
77             colour = pygame.Color(new_colour)
78             hue = colour.hsva[0]
79             self._selected_percent = hue / 360
80             self.set_image()
81
82     def set_image(self):
83         """
84             Draws colour slider to widget image.
85
86             # Scales initialised gradient surface instead of redrawing it everytime
87             set_image is called
88             gradient_scaled = smoothscale_and_cache(self._gradient_surface, self.
89             gradient_size)
90
91             self.image = pygame.transform.scale(self._empty_surface, (self.size))
92             self.image.blit(gradient_scaled, self.gradient_position)
93
94             # Resets thumb colour, image and position, then draws it to the widget
95             image

```

```

91         self._thumb.initialise_new_colours(self.selected_colour)
92         self._thumb.set_surface(radius=self.size[1] / 2, border_width=self.
93         border_width)
94         self._thumb.set_position(self.relative_to_global_position((self.
95         thumb_position[0], self.thumb_position[1])))
96
97     thumb_surface = self._thumb.get_surface()
98     self.image.blit(thumb_surface, self.thumb_position)
99
100    def process_event(self, event):
101        """
102            Processes Pygame events.
103
104            Args:
105                event (pygame.Event): Event to process.
106
107            Returns:
108                pygame.Color: Current colour slider is displaying.
109
110        if event.type not in [pygame.MOUSEMOTION, pygame.MOUSEBUTTONDOWN, pygame.
111        MOUSEBUTTONUP]:
112            return
113
114        # Gets widget state before and after event is processed by slider thumb
115        before_state = self._thumb.state
116        self._thumb.process_event(event)
117        after_state = self._thumb.state
118
119        # If widget state changes (e.g. hovered -> pressed), redraw widget
120        if before_state != after_state:
121            self.set_image()
122
123        if event.type == pygame.MOUSEMOTION:
124            if self._thumb.state == WidgetState.PRESS:
125                # Recalculates slider colour based on mouse position change
126                selected_percent = self.calculate_gradient_percent(event.pos)
127                self._last_mouse_x = event.pos[0]
128
129                if selected_percent is not None:
130                    self._selected_percent = selected_percent
131
132        return self.selected_colour
133
134    if event.type == pygame.MOUSEBUTTONUP:
135        # When user stops scrolling, return new slider colour
136        self._last_mouse_x = None
137        return self.selected_colour
138
139    if event.type == pygame.MOUSEBUTTONDOWN or before_state != after_state:
140        # Redraws widget when slider thumb is hovered or pressed
141        return self.selected_colour

```

## TextInput

The `TextInput` widget is used for inputting fen strings and time controls.  
`text_input.py`

```

1 import pyperclip
2 import pygame
3 from data.constants import WidgetState, CursorMode, INPUT_COLOURS
4 from data.components.custom_event import CustomEvent

```

```

5  from data.widgets.bases.pressable import _Pressable
6  from data.managers.logs import initialise_logger
7  from data.managers.animation import animation
8  from data.widgets.bases.box import _Box
9  from data.managers.cursor import cursor
10 from data.managers.theme import theme
11 from data.widgets.text import Text
12
13 logger = initialise_logger(__name__)
14
15 class TextInput(_Box, _Pressable, Text):
16     def __init__(self, event, blinking_interval=530, validator=(lambda x: True),
17                  default='', placeholder='PLACEHOLDER TEXT', placeholder_colour=(200, 200, 200),
18                  cursor_colour=theme['textSecondary'], **kwargs):
19         self._cursor_index = None
20         # Multiple inheritance used here, adding the functionality of pressing,
21         # and custom box colours, to the text widget
22         _Box.__init__(self, box_colours=INPUT_COLOURS)
23         _Pressable.__init__(
24             self,
25             event=None,
26             hover_func=lambda: self.set_state_colour(WidgetState.HOVER),
27             down_func=lambda: self.set_state_colour(WidgetState.PRESS),
28             up_func=lambda: self.set_state_colour(WidgetState.BASE),
29             sfx=None
30         )
31         Text.__init__(self, text="", center=False, box_colours=INPUT_COLOURS[
32             WidgetState.BASE], **kwargs)
33
34         self.initialise_new_colours(self._fill_colour)
35         self.set_state_colour(WidgetState.BASE)
36
37         pygame.key.set_repeat(500, 50)
38
39         self._blinking_fps = 1000 / blinking_interval
40         self._cursor_colour = cursor_colour
41         self._cursor_colour_copy = cursor_colour
42         self._placeholder_colour = placeholder_colour
43         self._text_colour_copy = self._text_colour
44
45         self._placeholder_text = placeholder
46         self._is_placeholder = None
47         if default:
48             self._text = default
49             self.is_placeholder = False
50         else:
51             self._text = self._placeholder_text
52             self.is_placeholder = True
53
54         self._event = event
55         self._validator = validator
56         self._blinking_cooldown = 0
57
58         self._empty_cursor = pygame.Surface((0, 0), pygame.SRCALPHA)
59
60         @property
61         # Encapsulated getter method
62         def is_placeholder(self):

```

```

63         return self._is_placeholder
64
65     @is_placeholder.setter
66     # Encapsulated setter method, used to replace text colour if placeholder text
67     # is shown
68     def is_placeholder(self, is_true):
69         self._is_placeholder = is_true
70
71         if is_true:
72             self._text_colour = self._placeholder_colour
73         else:
74             self._text_colour = self._text_colour_copy
75
76     @property
77     def cursor_size(self):
78         cursor_height = (self.size[1] - self.border_width * 2) * 0.75
79         return (cursor_height * 0.1, cursor_height)
80
81     @property
82     def cursor_position(self):
83         current_width = (self.margin / 2)
84         for index, metrics in enumerate(self._font.get_metrics(self._text, size=
85             self.font_size)):
86             if index == self._cursor_index:
87                 return (current_width - self.cursor_size[0], (self.size[1] - self.
88                     cursor_size[1]) / 2)
89
90         glyph_width = metrics[4]
91         current_width += glyph_width
92         return (current_width - self.cursor_size[0], (self.size[1] - self.
93             cursor_size[1]) / 2)
94
95     @property
96     def text(self):
97         if self.is_placeholder:
98             return ''
99
100        return self._text
101
102    def relative_x_to_cursor_index(self, relative_x):
103        """
104            Calculates cursor index using mouse position relative to the widget
105            position.
106
107            Args:
108                relative_x (int): Horizontal distance of the mouse from the left side
109                of the widget.
110
111            Returns:
112                int: Cursor index.
113
114            current_width = 0
115
116            for index, metrics in enumerate(self._font.get_metrics(self._text, size=
117                self.font_size)):
118                glyph_width = metrics[4]
119
120                if current_width >= relative_x:
121                    return index
122
123                current_width += glyph_width

```

```

118         return len(self._text)
119
120     def set_cursor_index(self, mouse_pos):
121         """
122             Sets cursor index based on mouse position.
123
124         Args:
125             mouse_pos (list[int, int]): Mouse position relative to window screen.
126
127         if mouse_pos is None:
128             self._cursor_index = mouse_pos
129             return
130
131         relative_x = mouse_pos[0] - (self.margin / 2) - self.rect.left
132         relative_x = max(0, relative_x)
133         self._cursor_index = self.relative_x_to_cursor_index(relative_x)
134
135     def focus_input(self, mouse_pos):
136         """
137             Draws cursor and sets cursor index when user clicks on widget.
138
139         Args:
140             mouse_pos (list[int, int]): Mouse position relative to window screen.
141
142         if self.is_placeholder:
143             self._text = ''
144             self.is_placeholder = False
145
146             self.set_cursor_index(mouse_pos)
147             self.set_image()
148             cursor.set_mode(CursorMode.IBEAM)
149
150     def unfocus_input(self):
151         """
152             Removes cursor when user unselects widget.
153
154         if self._text == '':
155             self._text = self._placeholder_text
156             self.is_placeholder = True
157             self.resize_text()
158
159             self.set_cursor_index(None)
160             self.set_image()
161             cursor.set_mode(CursorMode.ARROW)
162
163     def set_text(self, new_text):
164         """
165             Called by a state object to change the widget text externally.
166
167         Args:
168             new_text (str): New text to display.
169
170         Returns:
171             CustomEvent: Object containing the new text to alert state of a text
172             update.
173
174             super().set_text(new_text)
175             return CustomEvent(**vars(self._event), text=self.text)
176
177     def process_event(self, event):
178         """
179             Processes Pygame events.

```

```

179
180     Args:
181         event (pygame.Event): Event to process.
182
183     Returns:
184         CustomEvent: Object containing the new text to alert state of a text
185         update.
186         """
187
188         previous_state = self.get_widget_state()
189         super().process_event(event)
190         current_state = self.get_widget_state()
191
192         match event.type:
193             case pygame.MOUSEMOTION:
194                 if self._cursor_index is None:
195                     return
196
197                 # If mouse is hovering over widget, turn mouse cursor into an I-
198                 # beam
199                 if self.rect.collidepoint(event.pos):
200                     if cursor.get_mode() != CursorMode.IBEAM:
201                         cursor.set_mode(CursorMode.IBEAM)
202
203                 else:
204                     if cursor.get_mode() == CursorMode.IBEAM:
205                         cursor.set_mode(CursorMode.ARROW)
206
207             return
208
209             case pygame.MOUSEBUTTONDOWN:
210                 # When user selects widget
211                 if previous_state == WidgetState.PRESS:
212                     self.focus_input(event.pos)
213
214                 # When user unselects widget
215                 if current_state == WidgetState.BASE and self._cursor_index is not
216                 None:
217                     self.unfocus_input()
218
219             return CustomEvent(**vars(self._event), text=self.text)
220
221             case pygame.KEYDOWN:
222                 if self._cursor_index is None:
223                     return
224
225
226                 # Handling Ctrl-C and Ctrl-V shortcuts
227                 if event.mod & (pygame.KMOD_CTRL):
228                     if event.key == pygame.K_c:
229                         logger.info('COPIED')
230
231
232                     elif event.key == pygame.K_v:
233                         pasted_text = pyperclip.paste()
234                         pasted_text = ''.join(char for char in pasted_text if 32
235                         <= ord(char) <= 127)
236                         self._text = self._text[:self._cursor_index] + pasted_text
237                         self._text += self._text[self._cursor_index:]
238                         self._cursor_index += len(pasted_text)
239
240
241                     self.resize_text()
242                     self.set_image()
243                     self.set_geometry()
244
245             return
246
247             match event.key:

```

```

236             case pygame.K_BACKSPACE:
237                 if self._cursor_index > 0:
238                     self._text = self._text[:self._cursor_index - 1] +
239                         self._text[self._cursor_index:]
240                     self._cursor_index = max(0, self._cursor_index - 1)
241
242             case pygame.K_RIGHT:
243                 self._cursor_index = min(len(self._text), self.
244 _cursor_index + 1)
245
246             case pygame.K_LEFT:
247                 self._cursor_index = max(0, self._cursor_index - 1)
248
249             case pygame.K_ESCAPE:
250                 self.unfocus_input()
251                 return CustomEvent(**vars(self._event), text=self.text)
252
253             case pygame.K_RETURN:
254                 self.unfocus_input()
255                 return CustomEvent(**vars(self._event), text=self.text)
256
257             case _:
258                 if not event.unicode:
259                     return
260
261                     potential_text = self._text[:self._cursor_index] + event.
262 unicode + self._text[self._cursor_index:]
263
264                     # Validator lambda function used to check if inputted text
265                     # is valid before displaying
266                     # e.g. Time control input has a validator function
267                     checking if text represents a float
268                     if self._validator(potential_text) is False:
269                         return
270
271                     self._text = potential_text
272                     self._cursor_index += 1
273
274                     self._blinking_cooldown += 1
275                     animation.set_timer(500, lambda: self.subtract_blinking_cooldown
276 (1))
277
278                     self.resize_text()
279                     self.set_image()
280                     self.set_geometry()
281
282     def subtract_blinking_cooldown(self, cooldown):
283         """
284             Subtracts blinking cooldown after certain timeframe. When
285             blinking_cooldown is 1, cursor is able to be drawn.
286
287             Args:
288                 cooldown (float): Duration before cursor can no longer be drawn.
289             """
290             self._blinking_cooldown = self._blinking_cooldown - cooldown
291
292     def set_image(self):
293         """
294             Draws text input widget to image.
295             """
296             super().set_image()

```

```

291         if self._cursor_index is not None:
292             scaled_cursor = pygame.transform.scale(self._empty_cursor, self.
293             cursor_size)
294             scaled_cursor.fill(self._cursor_colour)
295             self.image.blit(scaled_cursor, self.cursor_position)
296
297     def update(self):
298         """
299             Overrides based update method, to handle cursor blinking.
300         """
301         super().update()
302         # Calculate if cursor should be shown or not
303         cursor_frame = animation.calculate_frame_index(0, 2, self._blinking_fps)
304         if cursor_frame == 1 and self._blinking_cooldown == 0:
305             self._cursor_colour = (0, 0, 0, 0)
306         else:
307             self._cursor_colour = self._cursor_colour_copy
308         self.set_image()

```

## 3.5 Game

### 3.5.1 Model

game\_model.py

```

1  from data.states.game.components.fen_parser import encode_fen_string
2  from data.constants import Colour, GameEventType, EMPTY_BB
3  from data.states.game.widget_dict import GAME_WIDGETS
4  from data.states.game.cpu.cpu_thread import CPUThread
5  from data.states.game.cpu.engines import ABMinimaxCPU
6  from data.components.custom_event import CustomEvent
7  from data.utils.bitboard_helpers import is_occupied
8  from data.states.game.components.board import Board
9  from data.utils import input_helpers as ip_helpers
10 from data.states.game.components.move import Move
11 from data.managers.logs import initialise_logger
12
13 logger = initialise_logger(__name__)
14
15 class GameModel:
16     def __init__(self, game_config):
17         self._listeners = {
18             'game': [],
19             'win': [],
20             'pause': []
21         }
22         self._board = Board(fen_string=game_config['FEN_STRING'])
23
24         self.states = {
25             'CPU_ENABLED': game_config['CPU_ENABLED'],
26             'CPU_DEPTH': game_config['CPU_DEPTH'],
27             'AWAITING_CPU': False,
28             'WINNER': None,
29             'PAUSED': False,
30             'ACTIVE_COLOUR': game_config['COLOUR'],
31             'TIME_ENABLED': game_config['TIME_ENABLED'],
32             'TIME': game_config['TIME'],
33             'START_FEN_STRING': game_config['FEN_STRING'],
34             'MOVES': [],
35             'ZOBRIST_KEYS': []

```

```

36         }
37
38     self._cpu = ABMinimaxCPU(self.states['CPU_DEPTH'], self.cpu_callback,
39                             verbose=False)
40     self._cpu_thread = CPUThread(self._cpu)
41     self._cpu_thread.start()
42     self._cpu_move = None
43
44     logger.info(f'Initialising CPU depth of {self.states["CPU_DEPTH"]}')
45
46     def register_listener(self, listener, parent_class):
47         """
48             Registers listener method of another MVC class.
49
50             Args:
51                 listener (callable): Listener callback function.
52                 parent_class (str): Class name.
53
54             self._listeners[parent_class].append(listener)
55
56     def alert_listeners(self, event):
57         """
58             Alerts all registered classes of an event by calling their listener
59             function.
60
61             Args:
62                 event (GameEventType): Event to pass as argument.
63
64             Raises:
65                 Exception: If an unrecognised event tries to be passed onto listeners.
66
67         for parent_class, listeners in self._listeners.items():
68             match event.type:
69                 case GameEventType.UPDATE_PIECES:
70                     if parent_class in 'game':
71                         for listener in listeners: listener(event)
72
73                 case GameEventType.SET LASER:
74                     if parent_class == 'game':
75                         for listener in listeners: listener(event)
76
77                 case GameEventType.PAUSE_CLICK:
78                     if parent_class in ['pause', 'game']:
79                         for listener in listeners:
80                             listener(event)
81
82                 case _:
83                     raise Exception('Unhandled event type (GameModel.
84 alert_listeners)')
85
86     def set_winner(self, colour=None):
87         """
88             Sets winner.
89
90             Args:
91                 colour (Colour, optional): Describes winner colour, or draw. Defaults
92                 to None.
93
94             self.states['WINNER'] = colour
95
96     def toggle_paused(self):
97         """

```

```

94     Toggles pause screen, and alerts pause view.
95     """
96     self.states['PAUSED'] = not self.states['PAUSED']
97     game_event = CustomEvent.create_event(GameEventType.PAUSE_CLICK)
98     self.alert_listeners(game_event)
99
100    def get_terminal_move(self):
101        """
102            Debugging method for inputting a move from the terminal.
103
104            Returns:
105                Move: Parsed move.
106            """
107        while True:
108            try:
109                move_type = ip_helpers.parse_move_type(input('Input move type (m/r): '))
110                src_square = ip_helpers.parse_notation(input("From: "))
111                dest_square = ip_helpers.parse_notation(input("To: "))
112                rotation = ip_helpers.parse_rotation(input("Enter rotation (a/b/c/d): "))
113                return Move.instance_from_notation(move_type, src_square,
114                                                    dest_square, rotation)
114            except ValueError as error:
115                logger.warning('Input error (Board.get_move): ' + str(error))
116
117    def make_move(self, move):
118        """
119            Takes a Move object and applies it to the board.
120
121            Args:
122                move (Move): Move to apply.
123            """
124            colour = self._board.bitboards.get_colour_on(move.src)
125            piece = self._board.bitboards.get_piece_on(move.src, colour)
126            # Apply move and get results of laser trajectory
127            laser_result = self._board.apply_move(move, add_hash=True)
128
129            self.alert_listeners(CustomEvent.create_event(GameEventType.SET LASER,
130                                                    laser_result=laser_result))
130
131            # Sets new active colour and checks for a win
132            self.states['ACTIVE_COLOUR'] = self._board.get_active_colour()
133            self.set_winner(self._board.check_win())
134
135            move_notation = move.to_notation(colour, piece, laser_result.
136                                              hit_square_bitboard)
137
138            self.alert_listeners(CustomEvent.create_event(GameEventType.UPDATE_PIECES,
139                                                    move_notation=move_notation))
140
141            # Adds move to move history list for review screen
142            self.states['MOVES'].append({
143                'time': {
144                    Colour.BLUE: GAME_WIDGETS['blue_timer'].get_time(),
145                    Colour.RED: GAME_WIDGETS['red_timer'].get_time()
146                },
147                'move': move_notation,
148                'laserResult': laser_result
149            })
150
151    def make_cpu_move(self):

```

```

150     """
151     Starts CPU calculations on the separate thread.
152     """
153     self.states['AWAITING_CPU'] = True
154     self._cpu_thread.start_cpu(self.get_board())
155
156     def cpu_callback(self, move):
157         """
158             Callback function passed to CPU thread. Called when CPU stops processing.
159
160             Args:
161                 move (Move): Move that CPU found.
162
163             if self.states['WINNER'] is None:
164                 # CPU move passed back to main thread by reassigning variable
165                 self._cpu_move = move
166                 self.states['AWAITING_CPU'] = False
167
168     def check_cpu(self):
169         """
170             Constantly checks if CPU calculations are finished, so that make_move can
171             be run on the main thread.
172
173             if self._cpu_move is not None:
174                 self.make_move(self._cpu_move)
175                 self._cpu_move = None
176
177     def kill_thread(self):
178         """
179             Interrupt and kill CPU thread.
180
181             self._cpu_thread.kill_thread()
182             self.states['AWAITING_CPU'] = False
183
184     def is_selectable(self, bitboard):
185         """
186             Checks if square is occupied by a piece of the current active colour.
187
188             Args:
189                 bitboard (int): Bitboard representing single square.
190
191             Returns:
192                 bool: True if square is occupied by a piece of the current active
193                 colour. False if not.
194
195             return is_occupied(self._board.bitboards.combined_colour_bitboards[self.
196             states['ACTIVE_COLOUR']], bitboard)
197
198     def get_available_moves(self, bitboard):
199         """
200             Gets all surrounding empty squares. Used for drawing overlay.
201
202             Args:
203                 bitboard (int): Bitboard representing single center square.
204
205             Returns:
206                 int: Bitboard representing all empty surrounding squares.
207
208             if (bitboard & self._board.get_all_active_pieces()) != EMPTY_BB:
209                 return self._board.get_valid_squares(bitboard)
210
211             return EMPTY_BB

```

```

209
210     def get_piece_list(self):
211         """
212             Returns:
213                 list[Piece, ...]: Array of all pieces on the board.
214         """
215         return self._board.get_piece_list()
216
217     def get_piece_info(self, bitboard):
218         """
219             Args:
220                 bitboard (int): Square containing piece.
221
222             Returns:
223                 tuple[Colour, Rotation, Piece]: Piece information.
224         """
225         colour = self._board.bitboards.get_colour_on(bitboard)
226         rotation = self._board.bitboards.get_rotation_on(bitboard)
227         piece = self._board.bitboards.get_piece_on(bitboard, colour)
228         return (piece, colour, rotation)
229
230     def get_fen_string(self):
231         return encode_fen_string(self._board.bitboards)
232
233     def get_board(self):
234         return self._board

```

### 3.5.2 View

game\_view.py

```

1  import pygame
2  from data.constants import GameEventType, Colour, StatusText, Miscellaneous,
   ShaderType
3  from data.states.game.components.overlay_draw import OverlayDraw
4  from data.states.game.components.capture_draw import CaptureDraw
5  from data.states.game.components.piece_group import PieceGroup
6  from data.states.game.components.laser_draw import LaserDraw
7  from data.states.game.components.father import DragAndDrop
8  from data.utils.bitboard_helpers import bitboard_to_coords
9  from data.utils.board_helpers import screen_pos_to_coords
10 from data.states.game.widget_dict import GAME_WIDGETS
11 from data.components.custom_event import CustomEvent
12 from data.components.widget_group import WidgetGroup
13 from data.components.cursor import Cursor
14 from data.managers.window import window
15 from data.managers.audio import audio
16 from data.assets import SFX
17
18 class GameView:
19     def __init__(self, model):
20         self._model = model
21         self._hide_pieces = False
22         self._selected_coords = None
23         self._event_to_func_map = {
24             GameEventType.UPDATE_PIECES: self.handle_update_pieces,
25             GameEventType.SET LASER: self.handle_set_laser,
26             GameEventType.PAUSE_CLICK: self.handle_pause,
27         }
28
29         # Register model event handling with process_model_event()

```

```

30         self._model.register_listener(self.process_model_event, 'game')
31
32     # Initialise WidgetGroup with map of widgets
33     self._widget_group = WidgetGroup(GAME_WIDGETS)
34     self._widget_group.handle_resize(window.size)
35     self.initialise_widgets()
36
37     self._cursor = Cursor()
38     self._laser_draw = LaserDraw(self.board_position, self.board_size)
39     self._overlay_draw = OverlayDraw(self.board_position, self.board_size)
40     self._drag_and_drop = DragAndDrop(self.board_position, self.board_size)
41     self._capture_draw = CaptureDraw(self.board_position, self.board_size)
42     self._piece_group = PieceGroup()
43     self.handle_update_pieces()
44
45     self.set_status_text(StatusText.PLAYER_MOVE)
46
47     @property
48     def board_position(self):
49         return GAME_WIDGETS['chessboard'].position
50
51     @property
52     def board_size(self):
53         return GAME_WIDGETS['chessboard'].size
54
55     @property
56     def square_size(self):
57         return self.board_size[0] / 10
58
59     def initialise_widgets(self):
60         """
61             Run methods on widgets stored in GAME_WIDGETS dictionary to reset them.
62         """
63         GAME_WIDGETS['move_list'].reset_move_list()
64         GAME_WIDGETS['move_list'].kill()
65         GAME_WIDGETS['help'].kill()
66         GAME_WIDGETS['tutorial'].kill()
67
68         GAME_WIDGETS['scroll_area'].set_image()
69
70         GAME_WIDGETS['chessboard'].refresh_board()
71
72         GAME_WIDGETS['blue_piece_display'].reset_piece_list()
73         GAME_WIDGETS['red_piece_display'].reset_piece_list()
74
75     def set_status_text(self, status):
76         """
77             Sets text on status text widget.
78
79             Args:
80                 status (StatusText): The game stage for which text should be displayed
81             for.
82         """
83         match status:
84             case StatusText.PLAYER_MOVE:
85                 GAME_WIDGETS['status_text'].set_text(f"{self._model.states['ACTIVE_COLOUR'].name}'s turn to move")
86             case StatusText.CPU_MOVE:
87                 GAME_WIDGETS['status_text'].set_text(f"CPU calculating a crazy
move...")
88             case StatusText.WIN:
89                 if self._model.states['WINNER'] == Miscellaneous.DRAW:

```

```

89             GAME_WIDGETS['status_text'].set_text(f"Game is a draw! Boring
...")
90         else:
91             GAME_WIDGETS['status_text'].set_text(f"{self._model.states['
WINNER'].name} won!")
92         case StatusText.DRAW:
93             GAME_WIDGETS['status_text'].set_text(f"Game is a draw! Boring...")
94
95     def handle_resize(self):
96         """
97             Handles resizing of the window.
98         """
99         self._overlay_draw.handle_resize(self.board_position, self.board_size)
100        self._capture_draw.handle_resize(self.board_position, self.board_size)
101        self._piece_group.handle_resize(self.board_position, self.board_size)
102        self._laser_draw.handle_resize(self.board_position, self.board_size)
103        self._laser_draw.handle_resize(self.board_position, self.board_size)
104        self._widget_group.handle_resize(window.size)
105
106        if self._laser_draw.firing:
107            self.update_laser_mask()
108
109    def handle_update_pieces(self, event=None):
110        """
111            Callback function to update pieces after move.
112
113            Args:
114                event (GameEventType, optional): If updating pieces after player move,
115                    event contains move information. Defaults to None.
116                toggle_timers (bool, optional): Toggle timers on and off for new
117                    active colour. Defaults to True.
118
119            piece_list = self._model.get_piece_list()
120            self._piece_group.initialise_pieces(piece_list, self.board_position, self.
121                board_size)
122
123            if event:
124                GAME_WIDGETS['move_list'].append_to_move_list(event.move_notation)
125                GAME_WIDGETS['scroll_area'].set_image()
126                audio.play_sfx(SFX['piece_move'])
127
128            if self._model.states['ACTIVE_COLOUR'] == Colour.BLUE:
129                self.set_status_text(StatusText.PLAYER_MOVE)
130            elif self._model.states['CPU_ENABLED'] is False:
131                self.set_status_text(StatusText.PLAYER_MOVE)
132            else:
133                self.set_status_text(StatusText.CPU_MOVE)
134
135            if self._model.states['WINNER'] is not None:
136                self.toggle_timer(self._model.states['ACTIVE_COLOUR'], False)
137                self.toggle_timer(self._model.states['ACTIVE_COLOUR'].get_flipped_
138                    colour(), False)
139
140                self.set_status_text(StatusText.WIN)
141
142            def handle_set_laser(self, event):
143                """
144                    Callback function to draw laser after move.

```

```

145
146     Args:
147         event (GameEventType): Contains laser trajectory information.
148         """
149         laser_result = event.laser_result
150
151         # If laser has hit a piece
152         if laser_result.hit_square_bitboard:
153             coords_to_remove = bitboard_to_coords(laser_result.hit_square_bitboard
154         )
155             self._piece_group.remove_piece(coords_to_remove)
156
157             if laser_result.piece_colour == Colour.BLUE:
158                 GAME_WIDGETS['red_piece_display'].add_piece(laser_result.piece_hit
159             )
160             elif laser_result.piece_colour == Colour.RED:
161                 GAME_WIDGETS['blue_piece_display'].add_piece(laser_result.
162                     piece_hit)
163
164             # Draw piece capture GFX
165             self._capture_draw.add_capture(
166                 laser_result.piece_hit,
167                 laser_result.piece_colour,
168                 laser_result.piece_rotation,
169                 coords_to_remove,
170                 laser_result.laser_path[0][0],
171                 self._model.states['ACTIVE_COLOUR']
172             )
173
174     def handle_pause(self, event=None):
175         """
176             Callback function for pausing timer.
177
178         Args:
179             event (None): Event argument not used.
180             """
181             is_active = not(self._model.states['PAUSED'])
182             self.toggle_timer(self._model.states['ACTIVE_COLOUR'], is_active)
183
184     def initialise_timers(self):
185         """
186             Initialises both timers with the correct amount of time and starts the
187             timer for the active colour.
188             """
189             if self._model.states['TIME_ENABLED']:
190                 GAME_WIDGETS['blue_timer'].set_time(self._model.states['TIME'] * 60 *
191                     1000)
192                 GAME_WIDGETS['red_timer'].set_time(self._model.states['TIME'] * 60 *
193                     1000)
194             else:
195                 GAME_WIDGETS['blue_timer'].kill()
196                 GAME_WIDGETS['red_timer'].kill()
197
198             self.toggle_timer(self._model.states['ACTIVE_COLOUR'], True)
199
200     def toggle_timer(self, colour, is_active):
201         """
202             Stops or resumes timer.

```

```

200
201     Args:
202         colour (Colour): Timer to toggle.
203         is_active (bool): Whether to pause or resume timer.
204     """
205     if colour == Colour.BLUE:
206         GAME_WIDGETS['blue_timer'].set_active(is_active)
207     elif colour == Colour.RED:
208         GAME_WIDGETS['red_timer'].set_active(is_active)
209
210     def update_laser_mask(self):
211         """
212             Uses pygame.mask to create a mask for the pieces.
213             Used for occluding the ray shader.
214         """
215         temp_surface = pygame.Surface(window.size, pygame.SRCALPHA)
216         self._piece_group.draw(temp_surface)
217         mask = pygame.mask.from_surface(temp_surface, threshold=127)
218         mask_surface = mask.to_surface(unsetColor=(0, 0, 255), setColor=(255,
219             0, 0, 255))
220
221         window.set_apply_arguments(ShaderType.RAYS, occlusion=mask_surface)
222
223     def draw(self):
224         """
225             Draws GUI and pieces onto the screen.
226         """
227         self._widget_group.update()
228         self._capture_draw.update()
229
230         self._widget_group.draw()
231         self._overlay_draw.draw(window.screen)
232
233         if self._hide_pieces is False:
234             self._piece_group.draw(window.screen)
235
236         self._laser_draw.draw(window.screen)
237         self._drag_and_drop.draw(window.screen)
238         self._capture_draw.draw(window.screen)
239
240     def process_model_event(self, event):
241         """
242             Registered listener function for handling GameModel events.
243             Each event is mapped to a callback function, and the appropriate one is run
244
245         Args:
246             event (GameEventType): Game event to process.
247
248         Raises:
249             KeyError: If an unrecognised event type is passed as the argument.
250         """
251         try:
252             self._event_to_func_map.get(event.type)(event)
253         except:
254             raise KeyError('Event type not recognized in Game View (GameView.
255             process_model_event):', event.type)
256
257     def set_overlay_coords(self, available_coords_list, selected_coord):
258         """
259             Set board coordinates for potential moves overlay.

```

```

259     Args:
260         available_coords_list (list[tuple[int, int]], ...): Array of
261             coordinates
262             selected_coord (list[int, int]): Coordinates of selected piece.
263             """
264             self._selected_coords = selected_coord
265             self._overlay_draw.set_selected_coords(selected_coord)
266             self._overlay_draw.set_available_coords(available_coords_list)
267
268     def get_selected_coords(self):
269         return self._selected_coords
270
271     def set_dragged_piece(self, piece, colour, rotation):
272         """
273             Passes information of the dragged piece to the dragging drawing class.
274
275         Args:
276             piece (Piece): Piece type of dragged piece.
277             colour (Colour): Colour of dragged piece.
278             rotation (Rotation): Rotation of dragged piece.
279             """
280             self._drag_and_drop.set_dragged_piece(piece, colour, rotation)
281
282     def remove_dragged_piece(self):
283         """
284             Stops drawing dragged piece when user lets go of piece.
285             """
286             self._drag_and_drop.remove_dragged_piece()
287
288     def convert_mouse_pos(self, event):
289         """
290             Passes information of what mouse cursor is interacting with to a
291             GameController object.
292
293         Args:
294             event (pygame.Event): Mouse event to process.
295
296         Returns:
297             CustomEvent | None: Contains information what mouse is doing.
298             """
299             clicked_coords = screen_pos_to_coords(event.pos, self.board_position, self
300             .board_size)
301
302             if event.type == pygame.MOUSEBUTTONDOWN:
303                 if clicked_coords:
304                     return CustomEvent.create_event(GameEventType.BOARD_CLICK, coords=
305                         clicked_coords)
306
307             else:
308                 return None
309
310             elif event.type == pygame.MOUSEBUTTONUP:
311                 if self._drag_and_drop.dragged_sprite:
312                     piece, colour, rotation = self._drag_and_drop.get_dragged_info()
313                     piece_dragged = self._drag_and_drop.remove_dragged_piece()
314                     return CustomEvent.create_event(GameEventType.PIECE_DROP, coords=
315                         clicked_coords, piece=piece, colour=colour, rotation=rotation, remove_overlay=
316                         piece_dragged)
317
318     def add_help_screen(self):
319         """
320             Draw help overlay when player clicks on the help button.

```

```

315     """
316     self._widget_group.add(GAME_WIDGETS['help'])
317     self._widget_group.handle_resize(window.size)
318
319     def add_tutorial_screen(self):
320         """
321             Draw tutorial overlay when player clicks on the tutorial button.
322         """
323         self._widget_group.add(GAME_WIDGETS['tutorial'])
324         self._widget_group.handle_resize(window.size)
325         self._hide_pieces = True
326
327     def remove_help_screen(self):
328         GAME_WIDGETS['help'].kill()
329
330     def remove_tutorial_screen(self):
331         GAME_WIDGETS['tutorial'].kill()
332         self._hide_pieces = False
333
334     def process_widget_event(self, event):
335         """
336             Passes Pygame event to WidgetGroup to allow individual widgets to process
337             events.
338
339             Args:
340                 event (pygame.Event): Event to process.
341
342             Returns:
343                 CustomEvent | None: A widget event.
344         """
345
346         return self._widget_group.process_event(event)

```

### 3.5.3 Controller

game\_controller.py

```

1 import pygame
2 from data.constants import GameEventType, MoveType, StatusText, Miscellaneous
3 from data.utils import bitboard_helpers as bb_helpers
4 from data.states.game.components.move import Move
5 from data.managers.logs import initialise_logger
6
7 logger = initialise_logger(__name__)
8
9 class GameController:
10     def __init__(self, model, view, win_view, pause_view, to_menu, to_new_game):
11         self._model = model
12         self._view = view
13         self._win_view = win_view
14         self._pause_view = pause_view
15
16         self._to_menu = to_menu
17         self._to_new_game = to_new_game
18
19         self._view.initialise_timers()
20
21     def cleanup(self, next):
22         """
23             Handles game quit, either leaving to main menu or restarting a new game.
24
25             Args:

```

```

26         next (str): New state to switch to.
27         """
28         self._model.kill_thread()
29
30         if next == 'menu':
31             self._to_menu()
32         elif next == 'game':
33             self._to_new_game()
34
35     def make_move(self, move):
36         """
37         Handles player move.
38
39         Args:
40             move (Move): Move to make.
41         """
42         self._model.make_move(move)
43         self._view.set_overlay_coords([], None)
44
45         if self._model.states['CPU_ENABLED']:
46             self._model.make_cpu_move()
47
48     def handle_pause_event(self, event):
49         """
50         Processes events when game is paused.
51
52         Args:
53             event (GameEventType): Event to process.
54
55         Raises:
56             Exception: If event type is unrecognised.
57         """
58         game_event = self._pause_view.convert_mouse_pos(event)
59
60         if game_event is None:
61             return
62
63         match game_event.type:
64             case GameEventType.PAUSE_CLICK:
65                 self._model.toggle_paused()
66
67             case GameEventType.MENU_CLICK:
68                 self.cleanup('menu')
69
70             case _:
71                 raise Exception('Unhandled event type (GameController.handle_event)')
72
73     def handle_winner_event(self, event):
74         """
75         Processes events when game is over.
76
77         Args:
78             event (GameEventType): Event to process.
79
80         Raises:
81             Exception: If event type is unrecognised.
82         """
83         game_event = self._win_view.convert_mouse_pos(event)
84
85         if game_event is None:
86             return

```

```

87
88     match game_event.type:
89         case GameEventType.MENU_CLICK:
90             self.cleanup('menu')
91             return
92
93         case GameEventType.GAME_CLICK:
94             self.cleanup('game')
95             return
96
97         case _:
98             raise Exception('Unhandled event type (GameController.handle_event')
99
100    def handle_game_widget_event(self, event):
101        """
102            Processes events for game GUI widgets.
103
104        Args:
105            event (GameEventType): Event to process.
106
107        Raises:
108            Exception: If event type is unrecognised.
109
110        Returns:
111            CustomEvent | None: A widget event.
112        """
113        widget_event = self._view.process_widget_event(event)
114
115        if widget_event is None:
116            return None
117
118        match widget_event.type:
119            case GameEventType.ROTATE_PIECE:
120                src_coords = self._view.get_selected_coords()
121
122                if src_coords is None:
123                    logger.info('None square selected')
124                    return
125
126                move = Move.instance_from_coords(MoveType.ROTATE, src_coords,
127                                                src_coords, rotation_direction=widget_event.rotation_direction)
128                self.make_move(move)
129
130            case GameEventType.RESIGN_CLICK:
131                self._model.set_winner(self._model.states['ACTIVE_COLOUR'].get_flipped_colour())
132                self._view.set_status_text(StatusText.WIN)
133
134            case GameEventType.DRAW_CLICK:
135                self._model.set_winner(Miscellaneous.DRAW)
136                self._view.set_status_text(StatusText.DRAW)
137
138            case GameEventType.TIMER_END:
139                if self._model.states['TIME_ENABLED']:
140                    self._model.set_winner(widget_event.active_colour.get_flipped_colour())
141
142            case GameEventType.MENU_CLICK:
143                self.cleanup('menu')
144
145            case GameEventType.HELP_CLICK:

```

```

145             self._view.add_help_screen()
146
147         case GameEventType.TUTORIAL_CLICK:
148             self._view.add_tutorial_screen()
149
150     case _:
151         raise Exception('Unhandled event type (GameController.handle_event')
152
153     return widget_event.type
154
155 def check_cpu(self):
156     """
157     Checks if CPU calculations are finished every frame.
158     """
159     if self._model.states['CPU_ENABLED'] and self._model.states['AWAITING_CPU']:
160         self._model.check_cpu()
161
162 def handle_game_event(self, event):
163     """
164     Processes Pygame events for main game.
165
166     Args:
167         event (pygame.Event): If event type is unrecognised.
168
169     Raises:
170         Exception: If event type is unrecognised.
171     """
172     # Pass event for widgets to process
173     widget_event = self.handle_game_widget_event(event)
174
175     if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.
176 KEYDOWN]:
177         if event.type != pygame.KEYDOWN:
178             game_event = self._view.convert_mouse_pos(event)
179         else:
180             game_event = None
181
182         if game_event is None:
183             if widget_event is None:
184                 if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
185                     # If user releases mouse click not on a widget
186                     self._view.remove_help_screen()
187                     self._view.remove_tutorial_screen()
188                 if event.type == pygame.MOUSEBUTTONUP:
189                     # If user releases mouse click on neither a widget or
190                     # board
191                     self._view.set_overlay_coords(None, None)
192
193     return
194
195     match game_event.type:
196         case GameEventType.BOARD_CLICK:
197             if self._model.states['AWAITING_CPU']:
198                 return
199
200             clicked_coords = game_event.coords
201             clicked_bitboard = bb_helpers.coords_to_bitboard(
202             clicked_coords)
203             selected_coords = self._view.get_selected_coords()

```

```

202         if selected_coords:
203             if clicked_coords == selected_coords:
204                 # If clicking on an already selected square, start
205                 # dragging piece on that square
206                 self._view.set_dragged_piece(*self._model.
207                 get_piece_info(clicked_bitboard))
208                 return
209
210             selected_bitboard = bb_helpers.coords_to_bitboard(
211                 selected_coords)
212             available_bitboard = self._model.get_available_moves(
213                 selected_bitboard)
214
215             if bb_helpers.is_occupied(clicked_bitboard,
216                 available_bitboard):
217                 # If the newly clicked square is not the same as the
218                 # old one, and is an empty surrounding square, make a move
219                 move = Move.instance_from_coords(MoveType.MOVE,
220                     selected_coords, clicked_coords)
221                 self.make_move(move)
222             else:
223                 # If the newly clicked square is not the same as the
224                 # old one, but is an invalid square, unselect the currently selected square
225                 self._view.set_overlay_coords(None, None)
226
227             # Select hovered square if it is same as active colour
228             elif self._model.is_selectable(clicked_bitboard):
229                 available_bitboard = self._model.get_available_moves(
230                     clicked_bitboard)
231                 self._view.set_overlay_coords(bb_helpers.
232                     bitboard_to_coords_list(available_bitboard), clicked_coords)
233                 self._view.set_dragged_piece(*self._model.get_piece_info(
234                     clicked_bitboard))
235
236         case GameEventType.PIECE_DROP:
237             hovered_coords = game_event.coords
238
239             # if piece is dropped onto the board
240             if hovered_coords:
241                 hovered_bitboard = bb_helpers.coords_to_bitboard(
242                     hovered_coords)
243                 selected_coords = self._view.get_selected_coords()
244                 selected_bitboard = bb_helpers.coords_to_bitboard(
245                     selected_coords)
246                 available_bitboard = self._model.get_available_moves(
247                     selected_bitboard)
248
249                 if bb_helpers.is_occupied(hovered_bitboard,
250                     available_bitboard):
251                     # Make a move if mouse is hovered over an empty
252                     # surrounding square
253                     move = Move.instance_from_coords(MoveType.MOVE,
254                         selected_coords, hovered_coords)
255                     self.make_move(move)
256
257             if game_event.remove_overlay:
258                 self._view.set_overlay_coords(None, None)
259
260             self._view.remove_dragged_piece()
261
262         case _:

```

```

246             raise Exception('Unhandled event type (GameController.
247 handle_event)', game_event.type)
248
249     def handle_event(self, event):
250         """
251         Pass a Pygame event to the correct handling function according to the
252         game state.
253
254         Args:
255             event (pygame.Event): Event to process.
256         """
257         if event.type in [pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.
258 MOUSEMOTION, pygame.KEYDOWN]:
259             if self._model.states['PAUSED']:
260                 self.handle_pause_event(event)
261             elif self._model.states['WINNER'] is not None:
262                 self.handle_winner_event(event)
263             else:
264                 self.handle_game_event(event)
265
266         if event.type == pygame.KEYDOWN:
267             if event.key == pygame.K_ESCAPE:
268                 self._model.toggle_paused()
269             elif event.key == pygame.K_l:
270                 logger.info('\nSTOPPING CPU')
271                 self._model._cpu_thread.stop_cpu() #temp
272
273

```

### 3.5.4 Board

The `Board` class implements the Laser Chess board, and is responsible for handling moves, captures, and win conditions.

`board.py`

```

1  from data.states.game.components.move import Move
2  from data.states.game.components.laser import Laser
3
4  from data.constants import Colour, Piece, Rank, File, MoveType, RotationDirection,
5      Miscellaneous, A_FILE_MASK, J_FILE_MASK, ONE_RANK_MASK, EIGHT_RANK_MASK,
6      EMPTY_BB
7  from data.states.game.components.bitboard_collection import BitboardCollection
8  from data.utils import bitboard_helpers as bb_helpers
9  from collections import defaultdict
10
11 class Board:
12     def __init__(self, fen_string="sc3ncfcncpb2/2pc7/3Pd6/pa1Pc1rbra1pb1Pd/
13     pb1Pd1RaRb1pa1Pc/6pb3/7Pa2/2PdNaFaNa3Sa b"):
14         self.bitboards = BitboardCollection(fen_string)
15         self.hash_list = [self.bitboards.get_hash()]
16
17     def __str__(self):
18         """
19             Returns a string representation of the board.
20
21             Returns:
22                 str: Board formatted as string.
23         """
24         characters = ''
25         pieces = defaultdict(int)
26
27         for rank in reversed(Rank):

```

```

25         for file in File:
26             mask = 1 << (rank * 10 + file)
27             blue_piece = self.bitboards.get_piece_on(mask, Colour.BLUE)
28             red_piece = self.bitboards.get_piece_on(mask, Colour.RED)
29
30             if blue_piece:
31                 pieces[blue_piece.value.upper()] += 1
32                 characters += f'{blue_piece.upper()} '
33             elif red_piece:
34                 pieces[red_piece.value] += 1
35                 characters += f'{red_piece} '
36             else:
37                 characters += '. '
38
39             characters += '\n\n'
40
41         characters += str(dict(pieces))
42         characters += f'\nCURRENT PLAYER TO MOVE: {self.bitboards.active_colour.
name}\n'
43         return characters
44
45     def get_piece_list(self):
46         """
47             Converts the board bitboards to a list of pieces.
48
49             Returns:
50                 list: List of Pieces.
51             """
52         return self.bitboards.convert_to_piece_list()
53
54     def get_active_colour(self):
55         """
56             Gets the active colour.
57
58             Returns:
59                 Colour: The active colour.
60             """
61         return self.bitboards.active_colour
62
63     def to_hash(self):
64         """
65             Gets the hash of the current board state.
66
67             Returns:
68                 int: A Zobrist hash.
69             """
70         return self.bitboards.get_hash()
71
72     def check_win(self):
73         """
74             Checks for a Pharaoh capture or threefold-repetition.
75
76             Returns:
77                 Colour | Miscellaneous: The winning colour, or Miscellaneous.DRAW.
78             """
79             for colour in Colour:
80                 if self.bitboards.get_piece_bitboard(Piece.PHARAOH, colour) ==
81                 EMPTY_BB:
82                     return colour.get_flipped_colour()
83
84             if self.hash_list.count(self.hash_list[-1]) >= 3:
85                 return Miscellaneous.DRAW

```

```

85
86         return None
87
88     def apply_move(self, move, fire_laser=True, add_hash=False):
89         """
90             Applies a move to the board.
91
92         Args:
93             move (Move): The move to apply.
94             fire_laser (bool): Whether to fire the laser after the move.
95             add_hash (bool): Whether to add the board state hash to the hash list.
96
97         Returns:
98             Laser: The laser trajectory result.
99             """
100
101        piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.
102            active_colour)
103
104        if piece_symbol is None:
105            raise ValueError('Invalid move - no piece found on source square')
106        elif piece_symbol == Piece.SPHINX:
107            raise ValueError('Invalid move - sphinx piece is immovable')
108
109        if move.move_type == MoveType.MOVE:
110            possible_moves = self.get_valid_squares(move.src)
111            if bb_helpers.is_occupied(move.dest, possible_moves) is False:
112                raise ValueError('Invalid move - destination square is occupied')
113
114            piece_rotation = self.bitboards.get_rotation_on(move.src)
115
116            self.bitboards.update_move(move.src, move.dest)
117            self.bitboards.update_rotation(move.src, move.dest, piece_rotation)
118
119        elif move.move_type == MoveType.ROTATE:
120            piece_symbol = self.bitboards.get_piece_on(move.src, self.bitboards.
121            active_colour)
122            piece_rotation = self.bitboards.get_rotation_on(move.src)
123
124            if move.rotation_direction == RotationDirection.CLOCKWISE:
125                new_rotation = piece_rotation.get_clockwise()
126            elif move.rotation_direction == RotationDirection.ANTICLOCKWISE:
127                new_rotation = piece_rotation.get_anticlockwise()
128
129            self.bitboards.update_rotation(move.src, move.src, new_rotation)
130
131        laser = None
132        if fire_laser:
133            laser = self.fire_laser(add_hash)
134
135        if add_hash:
136            self.hash_list.append(self.bitboards.get_hash())
137
138        self.bitboards.flip_colour()
139
140        return laser
141
142    def undo_move(self, move, laser_result):
143        """
144            Undoes a move on the board.
145
146        Args:
147            move (Move): The move to undo.

```

```

145         laser_result (Laser): The laser trajectory result.
146     """
147     self.bitboards.flip_colour()
148
149     if laser_result.hit_square_bitboard:
150         # Get info of destroyed piece, and add it to the board again
151         src = laser_result.hit_square_bitboard
152         piece = laser_result.piece_hit
153         colour = laser_result.piece_colour
154         rotation = laser_result.piece_rotation
155
156         self.bitboards.set_square(src, piece, colour)
157         self.bitboards.clear_rotation(src)
158         self.bitboards.set_rotation(src, rotation)
159
160     # Create new Move object that is the inverse of the passed move
161     if move.move_type == MoveType.MOVE:
162         reversed_move = Move.instance_from_bitboards(MoveType.MOVE, move.dest,
163             move.src)
164     elif move.move_type == MoveType.ROTATE:
165         reversed_move = Move.instance_from_bitboards(MoveType.ROTATE, move.src
166             , move.src, move.rotation_direction.get_opposite())
167
168     self.apply_move(reversed_move, fire_laser=False)
169     self.bitboards.flip_colour()
170
171     def remove_piece(self, square_bitboard):
172         """
173             Removes a piece from a given square.
174
175             Args:
176                 square_bitboard (int): The bitboard representation of the square.
177
178             self.bitboards.clear_square(square_bitboard, Colour.BLUE)
179             self.bitboards.clear_square(square_bitboard, Colour.RED)
180             self.bitboards.clear_rotation(square_bitboard)
181
182     def get_valid_squares(self, src_bitboard, colour=None):
183         """
184             Gets valid squares for a piece to move to.
185
186             Args:
187                 src_bitboard (int): The bitboard representation of the source square.
188                 colour (Colour, optional): The active colour of the piece.
189
190             Returns:
191                 int: The bitboard representation of valid squares.
192
193             target_top_left = (src_bitboard & A_FILE_MASK & EIGHT_RANK_MASK) << 9
194             target_top_middle = (src_bitboard & EIGHT_RANK_MASK) << 10
195             target_top_right = (src_bitboard & J_FILE_MASK & EIGHT_RANK_MASK) << 11
196             target_middle_right = (src_bitboard & J_FILE_MASK) << 1
197
198             target_bottom_right = (src_bitboard & J_FILE_MASK & ONE_RANK_MASK) >> 9
199             target_bottom_middle = (src_bitboard & ONE_RANK_MASK) >> 10
200             target_bottom_left = (src_bitboard & A_FILE_MASK & ONE_RANK_MASK)>> 11
201             target_middle_left = (src_bitboard & A_FILE_MASK) >> 1
202
203             possible_moves = target_top_left | target_top_middle | target_top_right |
204             target_middle_right | target_bottom_right | target_bottom_middle |
205             target_bottom_left | target_middle_left

```

```

203         if colour is not None:
204             valid_possible_moves = possible_moves & ~self.bitboards.
205             combined_colour_bitboards[colour]
206         else:
207             valid_possible_moves = possible_moves & ~self.bitboards.
208             combined_all_bitboard
209
210     return valid_possible_moves
211
212     def get_mobility(self, colour):
213         """
214             Gets all valid squares for a given colour.
215
216             Args:
217                 colour (Colour): The colour of the pieces.
218
219             Returns:
220                 int: The bitboard representation of all valid squares.
221             """
222
223     active_pieces = self.get_all_active_pieces(colour)
224     possible_moves = 0
225
226
227     for square in bb_helpers.occupied_squares(active_pieces):
228         possible_moves += bb_helpers.pop_count(self.get_valid_squares(square))
229
230
231     return possible_moves
232
233     def get_all_active_pieces(self, colour):
234         """
235             Gets all active pieces for the current player.
236
237             Args:
238                 colour (Colour): Active colour of pieces to retrieve. Defaults to None
239
240
241             Returns:
242                 int: The bitboard representation of all active pieces.
243             """
244
245     if colour is None:
246         colour = self.bitboards.active_colour
247
248     active_pieces = self.bitboards.combined_colour_bitboards[colour]
249     sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINK, colour)
250     return active_pieces ^ sphinx_bitboard
251
252     def fire_laser(self, remove_hash):
253         """
254             Fires the laser and removes hit pieces.
255
256             Args:
257                 remove_hash (bool): Whether to clear the hash list if a piece is hit.
258
259             Returns:
260                 Laser: The result of firing the laser.
261             """
262
263     laser = Laser(self.bitboards)
264
265
266     if laser.hit_square_bitboard:
267         self.remove_piece(laser.hit_square_bitboard)
268
269
270     if remove_hash:
271         self.hash_list = [] # Remove all hashes for threefold repetition,

```

```

    as the position is impossible to be repeated after a piece is removed
262     return laser
263
264     def generate_square_moves(self, src):
265         """
266             Generates all valid moves for a piece on a given square.
267
268         Args:
269             src (int): The bitboard representation of the source square.
270
271         Yields:
272             Move: A valid move for the piece.
273         """
274         for dest in bb_helpers.occupied_squares(self.get_valid_squares(src)):
275             yield Move(MoveType.MOVE, src, dest)
276
277     def generate_all_moves(self, colour):
278         """
279             Generates all valid moves for a given colour.
280
281         Args:
282             colour (Colour): The colour of the pieces.
283
284         Yields:
285             Move: A valid move for the active colour.
286         """
287         sphinx_bitboard = self.bitboards.get_piece_bitboard(Piece.SPHINX, colour)
288         # Remove source squares for Sphinx pieces, as they cannot be moved
289         sphinx_masked_bitboard = self.bitboards.combined_colour_bitboards[colour]
290         ~ sphinx_bitboard
291
292         for square in bb_helpers.occupied_squares(sphinx_masked_bitboard):
293             # Generate movement moves
294             yield from self.generate_square_moves(square)
295
296             # Generate rotational moves
297             for rotation_direction in RotationDirection:
298                 yield Move(MoveType.ROTATE, square, rotation_direction=
299                           rotation_direction)

```

### 3.5.5 Bitboards

The `BitboardCollection` class uses helper functions found in `bitboard_helpers.py` such as `pop_count`, to initialise and manage bitboard transformations.

`bitboard_collection.py`

```

1  from data.constants import Rank, File, Piece, Colour, Rotation, RotationIndex,
2      EMPTY_BB
3  from data.states.game.components.fen_parser import parse_fen_string
4  from data.states.game.cpu.zobrist_hasher import ZobristHasher
5  from data.utils import bitboard_helpers as bb_helpers
6  from data.managers.logs import initialise_logger
7
8  logger = initialise_logger(__name__)
9
9  class BitboardCollection:
10     def __init__(self, fen_string):
11         self.piece_bitboards = [{char: EMPTY_BB for char in Piece}, {char:
12             EMPTY_BB for char in Piece}]
12         self.combined_colour_bitboards = [EMPTY_BB, EMPTY_BB]

```

```
13     self.combined_all_bitboard = EMPTY_BB
14     self.rotation_bitboards = [EMPTY_BB, EMPTY_BB]
15     self.active_colour = Colour.BLUE
16     self._hasher = ZobristHasher()
17
18     try:
19         if fen_string:
20             self.piece_bitboards, self.combined_colour_bitboards, self.
21             combined_all_bitboard, self.rotation_bitboards, self.active_colour =
22             parse_fen_string(fen_string)
23                 self.initialise_hash()
24     except ValueError as error:
25         logger.error('Please input a valid FEN string:', error)
26         raise error
27
28     def __str__(self):
29         """
30             Returns a string representation of the bitboards.
31
32             Returns:
33                 str: Bitboards formatted with piece type and colour shown.
34         """
35         characters = ''
36         for rank in reversed(Rank):
37             for file in File:
38                 bitboard = 1 << (rank * 10 + file)
39
40                 colour = self.get_colour_on(bitboard)
41                 piece = self.get_piece_on(bitboard, Colour.BLUE) or self.
42                 get_piece_on(bitboard, Colour.RED)
43
44                 if piece is not None:
45                     characters += f'{piece.upper() if colour == Colour.BLUE
46 else piece} '
47                 else:
48                     characters += '. '
49
50         characters += '\n\n'
51
52     return characters
53
54     def get_rotation_string(self):
55         """
56             Returns a string representation of the board rotations.
57
58             Returns:
59                 str: Board formatted with only rotations shown.
60         """
61         characters = ''
62         for rank in reversed(Rank):
63
64             for file in File:
65                 mask = 1 << (rank * 10 + file)
66                 rotation = self.get_rotation_on(mask)
67                 has_piece = bb_helpers.is_occupied(self.combined_all_bitboard,
68                                         mask)
69
70                 if has_piece:
71                     characters += f'{rotation.upper()} '
72                 else:
73                     characters += '. '
```

```

70         characters += '\n\n'
71
72     return characters
73
74     def initialise_hash(self):
75         """
76             Initialises the Zobrist hash for the current board state.
77         """
78         for piece in Piece:
79             for colour in Colour:
80                 piece_bitboard = self.get_piece_bitboard(piece, colour)
81
82                 for occupied_bitboard in bb_helpers.occupied_squares(
83                     piece_bitboard):
84                     self._hasher.apply_piece_hash(occupied_bitboard, piece, colour
85
86                 for bitboard in bb_helpers.loop_all_squares():
87                     rotation = self.get_rotation_on(bitboard)
88                     self._hasher.apply_rotation_hash(bitboard, rotation)
89
90             if self.active_colour == Colour.RED:
91                 self._hasher.apply_red_move_hash()
92
93     def flip_colour(self):
94         """
95             Flips the active colour and updates the Zobrist hash.
96         """
97         self.active_colour = self.active_colour.get_flipped_colour()
98
99         if self.active_colour == Colour.RED:
100            self._hasher.apply_red_move_hash()
101
102     def update_move(self, src, dest):
103         """
104             Updates the bitboards for a move.
105
106             Args:
107                 src (int): The bitboard representation of the source square.
108                 dest (int): The bitboard representation of the destination square.
109
110             piece = self.get_piece_on(src, self.active_colour)
111
112             self.clear_square(src, Colour.BLUE)
113             self.clear_square(dest, Colour.BLUE)
114             self.clear_square(src, Colour.RED)
115             self.clear_square(dest, Colour.RED)
116
117             self.set_square(dest, piece, self.active_colour)
118
119     def update_rotation(self, src, dest, new_rotation):
120         """
121             Updates the rotation bitboards for a move.
122
123             Args:
124                 src (int): The bitboard representation of the source square.
125                 dest (int): The bitboard representation of the destination square.
126                 new_rotation (Rotation): The new rotation.
127
128             self.clear_rotation(src)
129             self.set_rotation(dest, new_rotation)

```

```

130     def clear_rotation(self, bitboard):
131         """
132             Clears the rotation for a given square.
133
134         Args:
135             bitboard (int): The bitboard representation of the square.
136         """
137         old_rotation = self.get_rotation_on(bitboard)
138         rotation_1, rotation_2 = self.rotation_bitboards
139         self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.clear_square(
140             rotation_1, bitboard)
141         self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.clear_square(
142             rotation_2, bitboard)
143
144         self._hasher.apply_rotation_hash(bitboard, old_rotation)
145
146     def clear_square(self, bitboard, colour):
147         """
148             Clears a square piece and rotation for a given colour.
149
150         Args:
151             bitboard (int): The bitboard representation of the square.
152             colour (Colour): The colour to clear.
153         """
154         piece = self.get_piece_on(bitboard, colour)
155
156         if piece is None:
157             return
158
159         piece_bitboard = self.get_piece_bitboard(piece, colour)
160         colour_bitboard = self.combined_colour_bitboards[colour]
161         all_bitboard = self.combined_all_bitboard
162
163         self.piece_bitboards[colour][piece] = bb_helpers.clear_square(
164             piece_bitboard, bitboard)
165         self.combined_colour_bitboards[colour] = bb_helpers.clear_square(
166             colour_bitboard, bitboard)
167         self.combined_all_bitboard = bb_helpers.clear_square(all_bitboard,
168             bitboard)
169
170         self._hasher.apply_piece_hash(bitboard, piece, colour)
171
172     def set_rotation(self, bitboard, rotation):
173         """
174             Sets the rotation for a given square.
175
176         Args:
177             bitboard (int): The bitboard representation of the square.
178             rotation (Rotation): The rotation to set.
179         """
180
181         rotation_1, rotation_2 = self.rotation_bitboards
182         self._hasher.apply_rotation_hash(bitboard, rotation)
183
184         match rotation:
185             case Rotation.UP:
186                 return
187             case Rotation.RIGHT:
188                 self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.
189                 set_square(rotation_1, bitboard)
190                 return
191             case Rotation.DOWN:
192                 self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.

```

```

    set_square(rotation_2, bitboard)
186        return
187        case Rotation.LEFT:
188            self.rotation_bitboards[RotationIndex.FIRSTBIT] = bb_helpers.
189    set_square(rotation_1, bitboard)
190            self.rotation_bitboards[RotationIndex.SECONDBIT] = bb_helpers.
191    set_square(rotation_2, bitboard)
192        return
193        case _:
194            raise ValueError('Invalid rotation input (bitboard.py):', rotation
195        )
196
197    def set_square(self, bitboard, piece, colour):
198        """
199        Sets a piece on a given square.
200
201        Args:
202            bitboard (int): The bitboard representation of the square.
203            piece (Piece): The piece to set.
204            colour (Colour): The colour of the piece.
205        """
206
207        piece_bitboard = self.get_piece_bitboard(piece, colour)
208        colour_bitboard = self.combined_colour_bitboards[colour]
209        all_bitboard = self.combined_all_bitboard
210
211        self.piece_bitboards[colour][piece] = bb_helpers.set_square(piece_bitboard
212            , bitboard)
213        self.combined_colour_bitboards[colour] = bb_helpers.set_square(
214            colour_bitboard, bitboard)
215        self.combined_all_bitboard = bb_helpers.set_square(all_bitboard, bitboard)
216
217        self._hasher.apply_piece_hash(bitboard, piece, colour)
218
219    def get_piece_bitboard(self, piece, colour):
220        """
221        Gets the bitboard for a piece type for a given colour.
222
223        Args:
224            piece (Piece): The piece bitboard to get.
225            colour (Colour): The colour of the piece.
226
227        Returns:
228            int: The bitboard representation for all squares occupied by that
229            piece type.
230        """
231        return self.piece_bitboards[colour][piece]
232
233    def get_piece_on(self, target_bitboard, colour):
234        """
235        Gets the piece on a given square for a given colour.
236
237        Args:
238            target_bitboard (int): The bitboard representation of the square.
239            colour (Colour): The colour of the piece.
240
241        Returns:
242            Piece: The piece on the square, or None if square is empty.
243        """
244        if not (bb_helpers.is_occupied(self.combined_colour_bitboards[colour],
245            target_bitboard)):
246            return None
247
```

```

240         return next(
241             (piece for piece in Piece if
242                 bb_helpers.is_occupied(self.get_piece_bitboard(piece, colour),
243                 target_bitboard)),
244             None)
245
246     def get_rotation_on(self, target_bitboard):
247         """
248             Gets the rotation on a given square.
249
250             Args:
251                 target_bitboard (int): The bitboard representation of the square.
252
253             Returns:
254                 Rotation: The rotation on the square.
255             """
256             rotationBits = [bb_helpers.is_occupied(self.rotation_bitboards[
257                 RotationIndex.SECONDBIT], target_bitboard), bb_helpers.is_occupied(self.
258                 rotation_bitboards[RotationIndex.FIRSTBIT], target_bitboard)]
259
260             match rotationBits:
261                 case [False, False]:
262                     return Rotation.UP
263                 case [False, True]:
264                     return Rotation.RIGHT
265                 case [True, False]:
266                     return Rotation.DOWN
267                 case [True, True]:
268                     return Rotation.LEFT
269
270     def get_colour_on(self, target_bitboard):
271         """
272             Gets the colour of the piece on a given square.
273
274             Args:
275                 target_bitboard (int): The bitboard representation of the square.
276
277             Returns:
278                 Colour: The colour of the piece on the square.
279             """
280             for piece in Piece:
281                 if self.get_piece_bitboard(piece, Colour.BLUE) & target_bitboard != EMPTY_BB:
282                     return Colour.BLUE
283                 elif self.get_piece_bitboard(piece, Colour.RED) & target_bitboard != EMPTY_BB:
284                     return Colour.RED
285
286     def get_piece_count(self, piece, colour):
287         """
288             Gets the count of a given piece type and colour.
289
290             Args:
291                 piece (Piece): The piece to count.
292                 colour (Colour): The colour of the piece.
293
294             Returns:
295                 int: The number of that piece of that colour on the board.
296             """
297             return bb_helpers.pop_count(self.get_piece_bitboard(piece, colour))
298
299     def get_hash(self):

```

```

297     """
298     Gets the Zobrist hash of the current board state.
299
300     Returns:
301         int: The Zobrist hash.
302     """
303     return self._hasher.hash
304
305     def convert_to_piece_list(self):
306         """
307             Converts all bitboards to a list of pieces.
308
309             Returns:
310                 list: Board represented as a 2D list of Piece and Rotation objects.
311             """
312     piece_list = []
313
314     for i in range(80):
315         if x := self.get_piece_on(1 << i, Colour.BLUE):
316             rotation = self.get_rotation_on(1 << i)
317             piece_list.append((x.upper(), rotation))
318         elif y := self.get_piece_on(1 << i, Colour.RED):
319             rotation = self.get_rotation_on(1 << i)
320             piece_list.append((y, rotation))
321         else:
322             piece_list.append(None)
323
324     return piece_list

```

## 3.6 CPU

This section includes my implementation for the CPU engine run on minimax, including its various improvements and accessory classes.

Every CPU engine class is a subclass of a `BaseCPU` abstract class, and therefore contains the same attribute and method names. This means polymorphism can be used again to easily test and vary the difficulty by switching out which CPU engine is used.

The method `find_move` is called by the CPU thread. `search` is then called recursively to traverse the minimax tree, and find an optimal move. The move is then returned to `find_move` and passed and run with the callback function.

### 3.6.1 Minimax

`minimax.py`

```

1  from data.states.game.cpu.base import BaseCPU
2  from data.constants import Score, Colour
3  from random import choice
4
5  class MinimaxCPU(BaseCPU):
6      def __init__(self, max_depth, callback, verbose=False):
7          super().__init__(callback, verbose)
8          self._max_depth = max_depth
9
10     def find_move(self, board, stop_event):
11         """
12             Finds the best move for the current board state.
13

```

```

14     Args:
15         board (Board): The current board state.
16         stop_event (threading.Event): Event used to kill search from an
17             external class.
18         """
19         self.initialise_stats()
20         best_score, best_move = self.search(board, self._max_depth, stop_event)
21
22         if self._verbose:
23             self.print_stats(best_score, best_move)
24
25         self._callback(best_move)
26
27     def search(self, board, depth, stop_event):
28         """
29         Recursively DFS through minimax tree with evaluation score.
30
31     Args:
32         board (Board): The current board state.
33         depth (int): The current search depth.
34         stop_event (threading.Event): Event used to kill search from an
35             external class.
36
37     Returns:
38         tuple[int, Move]: The best score and the best move found.
39         """
40
41         if (base_case := super().search(board, depth, stop_event)):
42             return base_case
43
44         best_move = None
45
46         # Blue is the maximising player
47         if board.get_active_colour() == Colour.BLUE:
48             max_score = -Score.INFINITE
49
50             for move in board.generate_all_moves(Colour.BLUE):
51                 laser_result = board.apply_move(move)
52
53                 new_score = self.search(board, depth - 1, stop_event)[0]
54
55                 if new_score > max_score:
56                     max_score = new_score
57                     best_move = move
58
59             board.undo_move(move, laser_result)
60
61         return max_score, best_move
62
63     else:
64         min_score = Score.INFINITE
65
66         for move in board.generate_all_moves(Colour.RED):
67             laser_result = board.apply_move(move)
68             new_score = self.search(board, depth - 1, stop_event)[0]
69
70             if new_score < min_score:
71                 min_score = new_score
72                 best_move = move
73
74             elif new_score == min_score:
75                 choice([best_move, move])

```

```

74         board.undo_move(move, laser_result)
75
76     return min_score, best_move
77

```

### 3.6.2 Alpha-beta Pruning

`alpha_beta.py`

```

1  from data.constants import Score, Colour
2  from data.states.game.cpu.base import BaseCPU
3  from random import choice
4
5  class ABMinimaxCPU(BaseCPU):
6      def __init__(self, max_depth, callback, verbose=True):
7          super().__init__(callback, verbose)
8          self._max_depth = max_depth
9
10     def initialise_stats(self):
11         """
12             Initialises the number of prunes to the statistics dictionary to be logged
13
14         """
15         super().initialise_stats()
16         self._stats['beta_prunes'] = 0
17         self._stats['alpha_prunes'] = 0
18
19     def find_move(self, board, stop_event):
20         """
21             Finds the best move for the current board state.
22
23             Args:
24                 board (Board): The current board state.
25                 stop_event (threading.Event): Event used to kill search from an
26                 external class.
27
28         """
29         self.initialise_stats()
30         best_score, best_move = self.search(board, self._max_depth, -Score.INFINITE, Score.INFINITE, stop_event)
31
32         if self._verbose:
33             self.print_stats(best_score, best_move)
34
35         self._callback(best_move)
36
37     def search(self, board, depth, alpha, beta, stop_event):
38         """
39             Recursively DFS through minimax tree while pruning branches using the
40             alpha and beta bounds.
41
42             Args:
43                 board (Board): The current board state.
44                 depth (int): The current search depth.
45                 alpha (int): The upper bound value.
46                 beta (int): The lower bound value.
47                 stop_event (threading.Event): Event used to kill search from an
48                 external class.
49
50             Returns:
51                 tuple[int, Move]: The best score and the best move found.
52
53

```

```

48     if (base_case := super().search(board, depth, stop_event)):
49         return base_case
50
51     best_move = None
52
53     # Blue is the maximising player
54     if board.get_active_colour() == Colour.BLUE:
55         max_score = -Score.INFINITE
56
57     for move in board.generate_all_moves(Colour.BLUE):
58         laser_result = board.apply_move(move)
59         new_score = self.search(board, depth - 1, alpha, beta, stop_event)
60         [0]
61
62         if new_score > max_score:
63             max_score = new_score
64             best_move = move
65
66         board.undo_move(move, laser_result)
67
68         alpha = max(alpha, max_score)
69
70         if beta <= alpha:
71             self._stats['alpha_prunes'] += 1
72             break
73
74     return max_score, best_move
75
76 else:
77     min_score = Score.INFINITE
78
79     for move in board.generate_all_moves(Colour.RED):
80         laser_result = board.apply_move(move)
81         new_score = self.search(board, depth - 1, alpha, beta, stop_event)
82         [0]
83
84         if new_score < min_score:
85             min_score = new_score
86             best_move = move
87
88         board.undo_move(move, laser_result)
89
90         beta = min(beta, min_score)
91         if beta <= alpha:
92             self._stats['beta_prunes'] += 1
93             break
94
95     return min_score, best_move

```

### 3.6.3 Transposition Table

For adding transposition table functionality to my other engine classes, I have decided to use a mixin design architecture. This allows me to reuse code by adding mixins to many different classes, and inject additional transposition table methods and functionality into other engines.

```

1 from data.states.game.cpu.engines.alpha_beta import ABMinimaxCPU, ABNegamaxCPU
2 from data.states.game.cpu.transposition_table import TranspositionTable
3
4 class TranspositionTableMixin:

```

```

5     def __init__(self, *args, **kwargs):
6         super().__init__(*args, **kwargs)
7         self._table = TranspositionTable()
8
9     def search(self, board, depth, alpha, beta, stop_event):
10        """
11            Searches transposition table for a cached move before running a full
12            search if necessary.
13            Caches the searched result.
14
15            Args:
16                board (Board): The current board state.
17                depth (int): The current search depth.
18                alpha (int): The upper bound value.
19                beta (int): The lower bound value.
20                stop_event (threading.Event): Event used to kill search from an
21                external class.
22
23            Returns:
24                tuple[int, Move]: The best score and the best move found.
25        """
26        hash = board.to_hash()
27        score, move = self._table.get_entry(hash, depth, alpha, beta)
28
29        if score is not None:
30            self._stats['cache_hits'] += 1
31            self._stats['nodes'] += 1
32
33        return score, move
34    else:
35        # If board hash entry not found in cache, run a full search
36        score, move = super().search(board, depth, alpha, beta, stop_event)
37        self._table.insert_entry(score, move, hash, depth, alpha, beta)
38
39    return score, move
40
41 class TTMinimaxCPU(TranspositionTableMixin, ABMinimaxCPU):
42     def initialise_stats(self):
43         """
44             Initialises cache statistics to be logged.
45         """
46         super().initialise_stats()
47         self._stats['cache_hits'] = 0
48
49     def print_stats(self, score, move):
50         """
51             Logs the statistics for the search.
52
53             Args:
54                 score (int): The best score found.
55                 move (Move): The best move found.
56
57             # Calculate number of cached entries retrieved as a percentage of all
58             # nodes
59             self._stats['cache_hits_percentage'] = round(self._stats['cache_hits'] /
60             self._stats['nodes'], 3)
61             self._stats['cache_entries'] = len(self._table)
62             super().print_stats(score, move)

```

### 3.6.4 Evaluator

```
evaluator.py

1 from data.utils.bitboard_helpers import pop_count, occupied_squares,
2     bitboard_to_index
3 from data.states.game.components.psqt import PSQT, FLIP
4 from data.managers.logs import initialise_logger
5 from data.constants import Colour, Piece, Score
6
7 logger = initialise_logger(__name__)
8
9 class Evaluator:
10     def __init__(self, verbose=True):
11         self._verbose = verbose
12
13     def evaluate(self, board, absolute=False):
14         """
15             Evaluates and returns a numerical score for the board state.
16
17             Args:
18                 board (Board): The current board state.
19                 absolute (bool): Whether to always return the absolute score from the
20                     active colour's perspective (for NegaMax).
21
22             Returns:
23                 int: Score representing advantage/disadvantage for the player.
24
25         blue_score = (
26             self.evaluate_material(board, Colour.BLUE) +
27             self.evaluate_position(board, Colour.BLUE) +
28             self.evaluate_mobility(board, Colour.BLUE) +
29             self.evaluate_pharaoh_safety(board, Colour.BLUE)
30         )
31
32         red_score = (
33             self.evaluate_material(board, Colour.RED) +
34             self.evaluate_position(board, Colour.RED) +
35             self.evaluate_mobility(board, Colour.RED) +
36             self.evaluate_pharaoh_safety(board, Colour.RED)
37
38         if self._verbose:
39             logger.info(f'Material: {self.evaluate_material(board, Colour.BLUE)} |'
40             f'{self.evaluate_material(board, Colour.RED)}')
41             logger.info(f'Position: {self.evaluate_position(board, Colour.BLUE)} |'
42             f'{self.evaluate_position(board, Colour.RED)}')
43             logger.info(f'Mobility: {self.evaluate_mobility(board, Colour.BLUE)} |'
44             f'{self.evaluate_mobility(board, Colour.RED)}')
45             logger.info(f'Safety: {self.evaluate_pharaoh_safety(board, Colour.BLUE)} |'
46             f'{self.evaluate_pharaoh_safety(board, Colour.RED)})')
47             logger.info(f'Overall score: {blue_score - red_score}')
48
49         if absolute and board.get_active_colour() == Colour.RED:
50             return red_score - blue_score
51         else:
52             return blue_score - red_score
53
54     def evaluate_material(self, board, colour):
55         """
56             Evaluates the material score for a given colour.
57
58
59
59
```

```

53     Args:
54         board (Board): The current board state.
55         colour (Colour): The colour to evaluate.
56
57     Returns:
58         int: Sum of all piece scores.
59     """
60     return (
61         Score.SPHINX * board.bitboards.get_piece_count(Piece.SPHINX, colour) +
62         Score.PYRAMID * board.bitboards.get_piece_count(Piece.PYRAMID, colour)
63         +
64         Score.ANUBIS * board.bitboards.get_piece_count(Piece.ANUBIS, colour) +
65         Score.SCARAB * board.bitboards.get_piece_count(Piece.SCARAB, colour)
66     )
67
68     def evaluate_position(self, board, colour):
69     """
70         Evaluates the positional score for a given colour.
71
72     Args:
73         board (Board): The current board state.
74         colour (Colour): The colour to evaluate.
75
76     Returns:
77         int: Score representing positional advantage/disadvantage.
78     """
79     score = 0
80
81     for piece in Piece:
82         if piece == Piece.SPHINX:
83             continue
84
85         piece_bitboard = board.bitboards.get_piece_bitboard(piece, colour)
86
87         for bitboard in occupied_squares(piece_bitboard):
88             index = bitboard_to_index(bitboard)
89             # Flip PSQT if using from blue player's perspective
90             index = FLIP[index] if colour == Colour.BLUE else index
91
92             score += PSQT[piece][index] * Score.POSITION
93
94     return score
95
96     def evaluate_mobility(self, board, colour):
97     """
98         Evaluates the mobility score for a given colour.
99
100    Args:
101        board (Board): The current board state.
102        colour (Colour): The colour to evaluate.
103
104    Returns:
105        int: Score on numerical representation of mobility.
106
107    number_of_moves = board.get_mobility(colour)
108    return number_of_moves * Score.MOVE
109
110    def evaluate_pharaoh_safety(self, board, colour):
111    """
112        Evaluates the safety of the Pharaoh for a given colour.
113
114    Args:

```

```

114         board (Board): The current board state.
115         colour (Colour): The colour to evaluate.
116
117     Returns:
118         int: Score representing mobility of the Pharaoh.
119         """
120
121     pharoah_bitboard = board.bitboards.get_piece_bitboard(Piece.PHAROAH,
122     colour)
123
124     if pharoah_bitboard:
125         pharoah_available_moves = pop_count(board.get_valid_squares(
126             pharoah_bitboard, colour))
127         return (8 - pharoah_available_moves) * Score.PHAROAH_SAFETY
128     else:
129         return 0

```

### 3.6.5 Multithreading

A CPUThread is initialised with a CPU engine at the start of the game state, and run whenever it is the CPU's turn to move.

cpu\_thread.py

```

1 import threading
2 import time
3 from data.managers.logs import initialise_logger
4
5 logger = initialise_logger(__name__)
6
7 class CPUThread(threading.Thread):
8     def __init__(self, cpu, verbose=False):
9         super().__init__()
10        self._stop_event = threading.Event()
11        self._running = True
12        self._verbose = verbose
13        self.daemon = True
14
15        self._board = None
16        self._cpu = cpu
17
18    def kill_thread(self):
19        """
20            Kills the CPU and terminates the thread by stopping the run loop.
21        """
22        self.stop_cpu()
23        self._running = False
24
25    def stop_cpu(self):
26        """
27            Kills the CPU's move search.
28        """
29        self._stop_event.set()
30        self._board = None
31
32    def start_cpu(self, board):
33        """
34            Starts the CPU's move search.
35        """
36        Args:
37            board (Board): The current board state.
38        """

```

```

39         self._stop_event.clear()
40         self._board = board
41
42     def run(self):
43         """
44             Periodically checks if the board variable is set.
45             If it is, then starts CPU search.
46         """
47         while self._running:
48             if self._board and self._cpu:
49                 self._cpu.find_move(self._board, self._stop_event)
50                 self.stop_cpu()
51             else:
52                 time.sleep(1)
53                 if self._verbose:
54                     logger.debug(f'(CPUThread.run) Thread {threading.get_native_id} idling...')


```

### 3.6.6 Zobrist Hashing

`zobrist_hasher.py`

```

1  from random import randint
2  from data.utils.bitboard_helpers import bitboard_to_index
3  from data.constants import Piece, Colour, Rotation
4
5  # Initialise random values for each piece type on every square
6  # (5 x 2 colours) pieces + 4 rotations, for 80 squares
7  zobrist_table = [[randint(0, 2 ** 64) for i in range(14)] for j in range(80)]
8  # Hash for when the red player's move
9  red_move_hash = randint(0, 2 ** 64)
10
11 # Maps piece to the correct random value
12 piece_lookup = {
13     Colour.BLUE: [
14         piece: i for i, piece in enumerate(Piece)
15     ],
16     Colour.RED: [
17         piece: i + 5 for i, piece in enumerate(Piece)
18     ],
19 }
20
21 # Maps rotation to the correct random value
22 rotation_lookup = [
23     rotation: i + 10 for i, rotation in enumerate(Rotation)
24 ]
25
26 class ZobristHasher:
27     def __init__(self):
28         self.hash = 0
29
30     def get_piece_hash(self, index, piece, colour):
31         """
32             Gets the random value for the piece type on the given square.
33
34             Args:
35                 index (int): The index of the square.
36                 piece (Piece): The piece on the square.
37                 colour (Colour): The colour of the piece.
38
39             Returns:

```

```

40             int: A 64-bit value.
41         """
42         piece_index = piece_lookup[colour][piece]
43         return zobrist_table[index][piece_index]
44
45     def get_rotation_hash(self, index, rotation):
46         """
47             Gets the random value for theon the given square.
48
49         Args:
50             index (int): The index of the square.
51             rotation (Rotation): The rotation on the square.
52             colour (Colour): The colour of the piece.
53
54         Returns:
55             int: A 64-bit value.
56         """
57         rotation_index = rotation_lookup[rotation]
58         return zobrist_table[index][rotation_index]
59
60     def apply_piece_hash(self, bitboard, piece, colour):
61         """
62             Updates the Zobrist hash with a new piece.
63
64         Args:
65             bitboard (int): The bitboard representation of the square.
66             piece (Piece): The piece on the square.
67             colour (Colour): The colour of the piece.
68
69             index = bitboard_to_index(bitboard)
70             piece_hash = self.get_piece_hash(index, piece, colour)
71             self.hash ^= piece_hash
72
73     def apply_rotation_hash(self, bitboard, rotation):
74         """
75             Updates the Zobrist hash with a new rotation.
76
77         Args:
78             bitboard (int): The bitboard representation of the square.
79             rotation (Rotation): The rotation on the square.
80
81             index = bitboard_to_index(bitboard)
82             rotation_hash = self.get_rotation_hash(index, rotation)
83             self.hash ^= rotation_hash
84
85     def apply_red_move_hash(self):
86         """
87             Applies the Zobrist hash for the red player's move.
88             self.hash ^= red_move_hash

```

### 3.6.7 Cache

`transposition_table.py`

```

1  from data.constants import TranspositionFlag
2
3  class TranspositionEntry:
4      def __init__(self, score, move, flag, hash_key, depth):
5          self.score = score
6          self.move = move
7          self.flag = flag

```

```

8         self.hash_key = hash_key
9         self.depth = depth
10
11    class TranspositionTable:
12        def __init__(self, max_entries=50000):
13            self._max_entries = max_entries
14            self._table = dict()
15
16        def calculate_entry_index(self, hash_key):
17            """
18                Gets the dictionary key for a given Zobrist hash.
19
20            Args:
21                hash_key (int): A Zobrist hash.
22
23            Returns:
24                str: Key for the given hash.
25            """
26            # return hash_key % self._max_entries
27            return str(hash_key)
28
29        def insert_entry(self, score, move, hash_key, depth, alpha, beta):
30            """
31                Inserts an entry into the transposition table.
32
33            Args:
34                score (int): The evaluation score.
35                move (Move): The best move found.
36                hash_key (int): The Zobrist hash key.
37                depth (int): The depth of the search.
38                alpha (int): The upper bound value.
39                beta (int): The lower bound value.
40
41            Raises:
42                Exception: Invalid depth or score.
43            """
44            if depth == 0 or alpha < score < beta:
45                flag = TranspositionFlag.EXACT
46                score = score
47            elif score <= alpha:
48                flag = TranspositionFlag.UPPER
49                score = alpha
50            elif score >= beta:
51                flag = TranspositionFlag.LOWER
52                score = beta
53            else:
54                raise Exception('TranspositionTable.insert_entry')
55
56            self._table[self.calculate_entry_index(hash_key)] = TranspositionEntry(
57                score, move, flag, hash_key, depth)
58
59            if len(self._table) > self._max_entries:
60                # Removes the longest-existing entry to free up space for more up-to-
61                # date entries
62                # Expression to remove leftmost item taken from https://docs.python.
63                # org/3/library/collections.html#ordereddict-objects
64                (k := next(iter(self._table))), self._table.pop(k))
65
66        def get_entry(self, hash_key, depth, alpha, beta):
67            """
68                Gets an entry from the transposition table.

```

```

67     Args:
68         hash_key (int): The Zobrist hash key.
69         depth (int): The depth of the search.
70         alpha (int): The alpha value for pruning.
71         beta (int): The beta value for pruning.
72
73     Returns:
74         tuple[int, Move] | tuple[None, None]: The evaluation score and the
75         best move found, if entry exists.
76         """
77
78         index = self.calculate_entry_index(hash_key)
79
80         if index not in self._table:
81             return None, None
82
83         entry = self._table[index]
84
85         if entry.hash_key == hash_key and entry.depth >= depth:
86             if entry.flag == TranspositionFlag.EXACT:
87                 return entry.score, entry.move
88
89             if entry.flag == TranspositionFlag.LOWER and entry.score >= beta:
90                 return entry.score, entry.move
91
92             if entry.flag == TranspositionFlag.UPPER and entry.score <= alpha:
93                 return entry.score, entry.move
94
95         return None, None

```

## 3.7 States

Every state class calls their `startup` method when switched to, and `cleanup` when exited. Within the `startup` function, the state widgets dictionary is passed into a `WidgetGroup` object. The `process_event` method is called on the `WidgetGroup` every frame to process user input, and handle the returned events accordingly. The `WidgetGroup` object can therefore be thought of as a controller, and the state as the model, and the widgets as the view.

### 3.7.1 Review

The `Review` state uses this logic to allow users to scroll through moves in their past games.  
`review.py`

```

1 import pygame
2 from collections import deque
3 from data.states.game.components.capture_draw import CaptureDraw
4 from data.states.game.components.piece_group import PieceGroup
5 from data.constants import ReviewEventType, Colour, ShaderType
6 from data.states.game.components.laser_draw import LaserDraw
7 from data.utils.bitboard_helpers import bitboard_to_coords
8 from data.states.review.widget_dict import REVIEW_WIDGETS
9 from data.utils.browser_helpers import get_winner_string
10 from data.states.game.components.board import Board
11 from data.components.game_entry import GameEntry
12 from data.managers.logs import initialise_logger
13 from data.managers.window import window
14 from data.control import _State
15 from data.assets import MUSIC
16

```

```

17 logger = initialise_logger(__name__)
18
19 class Review(_State):
20     def __init__(self):
21         super().__init__()
22
23         self._moves = deque()
24         self._popped_moves = deque()
25         self._game_info = {}
26
27         self._board = None
28         self._piece_group = None
29         self._laser_draw = None
30         self._capture_draw = None
31
32     def cleanup(self):
33         """
34             Cleanup function. Clears shader effects.
35         """
36         super().cleanup()
37
38         window.clear_apply_arguments(ShaderType.BLOOM)
39         window.clear_effect(ShaderType.RAYS)
40
41         return None
42
43     def startup(self, persist):
44         """
45             Startup function. Initialises all objects, widgets and game data.
46
47             Args:
48                 persist (dict): Dict containing game entry data.
49
50         """
51         super().startup(REVIEW_WIDGETS, MUSIC['review'])
52
53         window.set_apply_arguments(ShaderType.BASE, background_type=ShaderType.
54 BACKGROUND_WAVES)
55         window.set_apply_arguments(ShaderType.BLOOM, highlight_colours=[(pygame.
56 Color('0x95e0cc')).rgb, pygame.Color('0xf14e52').rgb], colour_intensity=0.8)
57         REVIEW_WIDGETS['help'].kill()
58
59         self._moves = deque(GameEntry.parse_moves(persist.pop('moves', '')))
60         self._popped_moves = deque()
61         self._game_info = persist
62
63         self._board = Board(self._game_info['start_fen_string'])
64         self._piece_group = PieceGroup()
65         self._laser_draw = LaserDraw(self.board_position, self.board_size)
66         self._capture_draw = CaptureDraw(self.board_position, self.board_size)
67
68         self.initialise_widgets()
69         self.simulate_all_moves()
70         self.refresh_pieces()
71         self.refresh_widgets()
72
73         self.draw()
74
75     @property
76     def board_position(self):
77         return REVIEW_WIDGETS['chessboard'].position

```

```

77     def board_size(self):
78         return REVIEW_WIDGETS['chessboard'].size
79
80     @property
81     def square_size(self):
82         return self.board_size[0] / 10
83
84     def initialise_widgets(self):
85         """
86             Initializes the widgets for a new game.
87         """
88         REVIEW_WIDGETS['move_list'].reset_move_list()
89         REVIEW_WIDGETS['move_list'].kill()
90         REVIEW_WIDGETS['scroll_area'].set_image()
91
92         REVIEW_WIDGETS['winner_text'].set_text(f'WINNER: {get_winner_string(self._game_info["winner"])}')
93         REVIEW_WIDGETS['blue_piece_display'].reset_piece_list()
94         REVIEW_WIDGETS['red_piece_display'].reset_piece_list()
95
96         if self._game_info['time_enabled']:
97             REVIEW_WIDGETS['timer_disabled_text'].kill()
98         else:
99             REVIEW_WIDGETS['blue_timer'].kill()
100            REVIEW_WIDGETS['red_timer'].kill()
101
102    def refresh_widgets(self):
103        """
104            Refreshes the widgets after every move.
105        """
106        REVIEW_WIDGETS['move_number_text'].set_text(f'MOVE NO: {(len(self._moves) / 2:.1f) / ((len(self._moves) + len(self._popped_moves)) / 2:.1f)}')
107        REVIEW_WIDGETS['move_colour_text'].set_text(f'{self.calculate_colour().name} TO MOVE')
108
109        if self._game_info['time_enabled']:
110            if len(self._moves) == 0:
111                REVIEW_WIDGETS['blue_timer'].set_time(float(self._game_info['time']) * 60 * 1000)
112                REVIEW_WIDGETS['red_timer'].set_time(float(self._game_info['time']) * 60 * 1000)
113            else:
114                REVIEW_WIDGETS['blue_timer'].set_time(float(self._moves[-1]['blue_time']) * 60 * 1000)
115                REVIEW_WIDGETS['red_timer'].set_time(float(self._moves[-1]['red_time']) * 60 * 1000)
116
117        REVIEW_WIDGETS['scroll_area'].set_image()
118
119    def refresh_pieces(self):
120        """
121            Refreshes the pieces on the board.
122        """
123        self._piece_group.initialise_pieces(self._board.get_piece_list(), self._board_position, self._board_size)
124
125    def simulate_all_moves(self):
126        """
127            Simulates all moves at the start of every game to obtain laser results and
128            fill up piece display and move list widgets.
129        """
130        for index, move_dict in enumerate(self._moves):

```

```

130         laser_result = self._board.apply_move(move_dict['move'], fire_laser=True)
131         self._moves[index]['laser_result'] = laser_result
132
133         if laser_result.hit_square_bitboard:
134             if laser_result.piece_colour == Colour.BLUE:
135                 REVIEW_WIDGETS['red_piece_display'].add_piece(laser_result,
136 piece_hit)
137             elif laser_result.piece_colour == Colour.RED:
138                 REVIEW_WIDGETS['blue_piece_display'].add_piece(laser_result,
139 piece_hit)
140
141     REVIEW_WIDGETS['move_list'].append_to_move_list(move_dict['unparsed_move'])
142
143     def calculate_colour(self):
144         """
145             Calculates the current active colour to move.
146
147             Returns:
148                 Colour: The current colour to move.
149
150             if self._game_info['start_fen_string'][-1].lower() == 'b':
151                 initial_colour = Colour.BLUE
152             elif self._game_info['start_fen_string'][-1].lower() == 'r':
153                 initial_colour = Colour.RED
154
155             if len(self._moves) % 2 == 0:
156                 return initial_colour
157             else:
158                 return initial_colour.get_flipped_colour()
159
160     def handle_move(self, move, add_piece=True):
161         """
162             Handles applying or undoing a move.
163
164             Args:
165                 move (dict): The move to handle.
166                 add_piece (bool): Whether to add the captured piece to the display.
167             Defaults to True.
168
169             laser_result = move['laser_result']
170             active_colour = self.calculate_colour()
171             self._laser_draw.add_laser(laser_result, laser_colour=active_colour)
172
173             if laser_result.hit_square_bitboard:
174                 if laser_result.piece_colour == Colour.BLUE:
175                     if add_piece:
176                         REVIEW_WIDGETS['red_piece_display'].add_piece(laser_result,
177 piece_hit)
178                     else:
179                         REVIEW_WIDGETS['red_piece_display'].remove_piece(laser_result,
180 piece_hit)
181                     elif laser_result.piece_colour == Colour.RED:
182                         if add_piece:
183                             REVIEW_WIDGETS['blue_piece_display'].add_piece(laser_result,
184 piece_hit)
185                         else:
186                             REVIEW_WIDGETS['blue_piece_display'].remove_piece(laser_result,
187 piece_hit)
188
189             self._capture_draw.add_capture(

```

```

183         laser_result.piece_hit,
184         laser_result.piece_colour,
185         laser_result.piece_rotation,
186         bitboard_to_coords(laser_result.hit_square_bitboard),
187         laser_result.laser_path[0][0],
188         active_colour,
189         shake=False
190     )
191
192     def update_laser_mask(self):
193         """
194             Updates the laser mask for the light rays effect.
195         """
196         temp_surface = pygame.Surface(window.size, pygame.SRCALPHA)
197         self._piece_group.draw(temp_surface)
198         mask = pygame.mask.from_surface(temp_surface, threshold=127)
199         mask_surface = mask.to_surface(unsetcolor=(0, 0, 255), setcolor=(255,
200         0, 0, 255))
201
202         window.set_apply_arguments(ShaderType.RAYS, occlusion=mask_surface)
203
204     def get_event(self, event):
205         """
206             Processes Pygame events.
207
208             Args:
209                 event (pygame.event.Event): The event to handle.
210
211             if event.type in [pygame.MOUSEBUTTONUP, pygame.KEYDOWN]:
212                 REVIEW_WIDGETS['help'].kill()
213
214             widget_event = self._widget_group.process_event(event)
215
216             if widget_event is None:
217                 return
218
219             match widget_event.type:
220                 case None:
221                     return
222
223                 case ReviewEventType.MENU_CLICK:
224                     self.next = 'menu'
225                     self.done = True
226
227                 case ReviewEventType.PREVIOUS_CLICK:
228                     if len(self._moves) == 0:
229                         return
230
231                     # Pop last applied move off first stack
232                     move = self._moves.pop()
233                     # Pushed onto second stack
234                     self._popped_moves.append(move)
235
236                     # Undo last applied move
237                     self._board.undo_move(move['move'], laser_result=move['
laser_result'])
238                     self.handle_move(move, add_piece=False)
239                     REVIEW_WIDGETS['move_list'].pop_from_move_list()
240
241                     self.refresh_pieces()
242                     self.refresh_widgets()
243                     self.update_laser_mask()

```

```

243
244     case ReviewEventType.NEXT_CLICK:
245         if len(self._popped_moves) == 0:
246             return
247
248         # Peek at second stack to get last undone move
249         move = self._popped_moves[-1]
250
251         # Reapply last undone move
252         self._board.apply_move(move['move'])
253         self.handle_move(move, add_piece=True)
254         REVIEW_WIDGETS['move_list'].append_to_move_list(move['unparsed_move'])
255
256         # Pop last undone move from second stack
257         self._popped_moves.pop()
258         # Push onto first stack
259         self._moves.append(move)
260
261         self.refresh_pieces()
262         self.refresh_widgets()
263         self.update_laser_mask()
264
265     case ReviewEventType.HELP_CLICK:
266         self._widget_group.add(REVIEW_WIDGETS['help'])
267         self._widget_group.handle_resize(window.size)
268
269     def handle_resize(self):
270         """
271             Handles resizing of the window.
272         """
273         super().handle_resize()
274         self._piece_group.handle_resize(self.board_position, self.board_size)
275         self._laser_draw.handle_resize(self.board_position, self.board_size)
276         self._capture_draw.handle_resize(self.board_position, self.board_size)
277
278         if self._laser_draw.firing:
279             self.update_laser_mask()
280
281     def draw(self):
282         """
283             Draws all components onto the window screen.
284         """
285         self._capture_draw.update()
286         self._widget_group.draw()
287         self._piece_group.draw(window.screen)
288         self._laser_draw.draw(window.screen)
289         self._capture_draw.draw(window.screen)

```

## 3.8 Database

This section outlines my database implementation using Python sqlite3.

### 3.8.1 DDL

As mentioned in Section 2.3.1, the `migrations` directory contains a collection of Python scripts that edit the game table schema. The files are named with their changes and datetime labelled for organisational purposes.

```

create_games_table_19112024.py

1 import sqlite3
2 from pathlib import Path
3
4 database_path = (Path(__file__).parent / '../database.db').resolve()
5
6 def upgrade():
7     """
8     Upgrade function to create games table.
9     """
10    connection = sqlite3.connect(database_path)
11    cursor = connection.cursor()
12
13    cursor.execute('''
14        CREATE TABLE games(
15            id INTEGER PRIMARY KEY,
16            cpu_enabled INTEGER NOT NULL,
17            cpu_depth INTEGER,
18            winner INTEGER,
19            time_enabled INTEGER NOT NULL,
20            time REAL,
21            number_of_ply INTEGER NOT NULL,
22            moves TEXT NOT NULL
23        )
24    ''')
25
26    connection.commit()
27    connection.close()
28
29 def downgrade():
30     """
31     Downgrade function to revert table creation.
32     """
33    connection = sqlite3.connect(database_path)
34    cursor = connection.cursor()
35
36    cursor.execute('''
37        DROP TABLE games
38    ''')
39
40    connection.commit()
41    connection.close()
42
43 upgrade()
44 # downgrade()

```

Using the `ALTER` command allows me to rename table columns.

```

change_fen_string_column_name_23122024.py

1 import sqlite3
2 from pathlib import Path
3
4 database_path = (Path(__file__).parent / '../database.db').resolve()
5
6 def upgrade():
7     """
8     Upgrade function to rename fen_string column.
9     """
10    connection = sqlite3.connect(database_path)
11    cursor = connection.cursor()

```

```

12
13     cursor.execute('''
14         ALTER TABLE games RENAME COLUMN fen_string TO final_fen_string
15     ''')
16
17     connection.commit()
18     connection.close()
19
20 def downgrade():
21     """
22     Downgrade function to revert fen_string column renaming.
23     """
24     connection = sqlite3.connect(database_path)
25     cursor = connection.cursor()
26
27     cursor.execute('''
28         ALTER TABLE games RENAME COLUMN final_fen_string TO fen_string
29     ''')
30
31     connection.commit()
32     connection.close()
33
34 upgrade()
35 # downgrade()

```

### 3.8.2 DML

`database_helpers.py`

```

1 import sqlite3
2 from pathlib import Path
3 from datetime import datetime
4
5 database_path = (Path(__file__).parent / '../database/database.db').resolve()
6
7 def insert_into_games(game_entry):
8     """
9     Inserts a new row into games table.
10
11     Args:
12         game_entry (GameEntry): GameEntry object containing game information.
13     """
14     connection = sqlite3.connect(database_path, detect_types=sqlite3.
15         PARSE_DECLTYPES)
15     cursor = connection.cursor()
16
17     # Datetime added for created_dt column
18     game_entry = (*game_entry, datetime.now())
19
20     cursor.execute('''
21         INSERT INTO games (cpu_enabled, cpu_depth, winner, time_enabled, time,
22         number_of_ply, moves, start_fen_string, final_fen_string, created_dt)
23         VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
24     ''', game_entry)
25
26     connection.commit()
27     connection.close()
28
29 def get_all_games():
30     """
31     Get all rows in games table.

```

```

31
32     Returns:
33         list[dict]: List of game entries represented as dictionaries.
34
35     connection = sqlite3.connect(database_path, detect_types=sqlite3.
36 PARSE_DECLTYPES)
37     connection.row_factory = sqlite3.Row
38     cursor = connection.cursor()
39
40     cursor.execute('''
41         SELECT * FROM games
42     ''')
43     games = cursor.fetchall()
44
45     connection.close()
46
47     return [dict(game) for game in games]
48
48 def delete_all_games():
49     """
50     Delete all rows in games table.
51     """
52     connection = sqlite3.connect(database_path)
53     cursor = connection.cursor()
54
55     cursor.execute('''
56         DELETE FROM games
57     ''')
58
59     connection.commit()
60     connection.close()
61
62 def delete_game(id):
63     """
64     Deletes specific row in games table using id attribute.
65
66     Args:
67         id (int): Primary key for row.
68
69     connection = sqlite3.connect(database_path)
70     cursor = connection.cursor()
71
72     cursor.execute('''
73         DELETE FROM games WHERE id = ?
74     ''', (id,))
75
76     connection.commit()
77     connection.close()
78
79 def get_ordered_games(column, ascend=True, start_row=1, end_row=10):
80     """
81     Get specific number of rows from games table ordered by a specific column(s).
82
83     Args:
84         column (_type_): Column to sort by.
85         ascend (bool, optional): Sort ascending or descending. Defaults to True.
86         start_row (int, optional): First row returned. Defaults to 1.
87         end_row (int, optional): Last row returned. Defaults to 10.
88
89     Raises:
90         ValueError: If ascend argument or column argument are invalid types.
91

```

```

92     Returns:
93         list[dict]: List of ordered game entries represented as dictionaries.
94     """
95     if not isinstance(ascend, bool) or not isinstance(column, str):
96         raise ValueError('({}) Invalid input arguments!'.format(database_helpers.get_ordered_games))
97
98     connection = sqlite3.connect(database_path, detect_types=sqlite3.PARSE_DECLTYPES)
99     connection.row_factory = sqlite3.Row
100    cursor = connection.cursor()
101
102    # Match ascend bool to correct SQL keyword
103    if ascend:
104        ascend_arg = 'ASC'
105    else:
106        ascend_arg = 'DESC'
107
108    # Partition by winner, then order by time and number_of_ply
109    if column == 'winner':
110        cursor.execute("""
111             SELECT * FROM
112                 (SELECT ROW_NUMBER() OVER (
113                     PARTITION BY winner
114                     ORDER BY time {ascend_arg}, number_of_ply {ascend_arg}
115                 ) AS row_num, * FROM games)
116             WHERE row_num >= ? AND row_num <= ?
117             """ , (start_row, end_row))
118    else:
119        # Order by time or number_of_ply only
120        cursor.execute("""
121             SELECT * FROM
122                 (SELECT ROW_NUMBER() OVER (
123                     ORDER BY {column} {ascend_arg}
124                 ) AS row_num, * FROM games)
125             WHERE row_num >= ? AND row_num <= ?
126             """ , (start_row, end_row))
127
128    games = cursor.fetchall()
129
130    connection.close()
131
132    return [dict(game) for game in games]
133
134 def get_number_of_games():
135     """
136     Returns:
137         int: Number of rows in the games.
138     """
139     connection = sqlite3.connect(database_path)
140     cursor = connection.cursor()
141
142     cursor.execute("""
143         SELECT COUNT(ROWID) FROM games
144     """)
145
146     result = cursor.fetchall()[0][0]
147
148     connection.close()
149
150     return result
151

```

```
152 # delete_all_games()
```

## 3.9 Shaders

### 3.9.1 Shader Manager

The `ShaderManager` class is responsible for handling all shader passes, handling the Pygame display, and combining both and drawing the result to the window screen. The class also inherits from the `SMPProtocol` class, an interface class containing all required `ShaderManager` methods and attributes to aid with syntax highlighting in the fragment shader classes.

Fragment shaders such as `Bloom` are applied by default, and others such as `Ray` are applied during runtime through calling methods on `ShaderManager`, and adding the appropriate fragment shader class to the internal shader pass list.

`shader.py`

```
1 from pathlib import Path
2 from array import array
3 import moderngl
4 from data.shaders.classes import shader_pass_lookup
5 from data.shaders.protocol import SMPProtocol
6 from data.constants import ShaderType
7
8 shader_path = (Path(__file__).parent / '../shaders/').resolve()
9
10 SHADER_PRIORITY = [
11     ShaderType.CRT,
12     ShaderType.SHAKE,
13     ShaderType.BLOOM,
14     ShaderType.CHROMATIC_ABBREVIATION,
15     ShaderType.RAYS,
16     ShaderType.GRAYSCALE,
17     ShaderType.BASE,
18 ]
19
20 pygame_quad_array = array('f', [
21     -1.0, 1.0, 0.0, 0.0,
22     1.0, 1.0, 0.0, 0.0,
23     -1.0, -1.0, 0.0, 1.0,
24     1.0, -1.0, 1.0, 1.0,
25 ])
26
27 opengl_quad_array = array('f', [
28     -1.0, -1.0, 0.0, 0.0,
29     1.0, -1.0, 1.0, 0.0,
30     -1.0, 1.0, 0.0, 1.0,
31     1.0, 1.0, 1.0, 1.0,
32 ])
33
34 class ShaderManager(SMPProtocol):
35     def __init__(self, ctx: moderngl.Context, screen_size):
36         self._ctx = ctx
37         self._ctx.gc_mode = 'auto'
38
39         self._screen_size = screen_size
40         self._opengl_buffer = self._ctx.buffer(data=opengl_quad_array)
41         self._pygame_buffer = self._ctx.buffer(data=pygame_quad_array)
42         self._shader_list = [ShaderType.BASE]
43
```

```

44         self._vert_shaders = {}
45         self._frag_shaders = {}
46         self._programs = {}
47         self._vaos = {}
48         self._textures = {}
49         self._shader_passes = {}
50         self.framebuffers = {}
51
52     self.load_shader(ShaderType.BASE)
53     self.load_shader(ShaderType._CALIBRATE)
54     self.create_framebuffer(ShaderType._CALIBRATE)
55
56     def load_shader(self, shader_type, **kwargs):
57         """
58             Loads a given shader by creating a VAO reading the corresponding .frag
59             file.
60
61             Args:
62                 shader_type (ShaderType): The type of shader to load.
63                 **kwargs: Additional arguments passed when initialising the fragment
64             shader class.
65
66             self._shader_passes[shader_type] = shader_pass_lookup[shader_type](self,
67             **kwargs)
68             self.create_vao(shader_type)
69
70     def clear_shaders(self):
71         """
72             Clears the shader list, leaving only the base shader.
73
74             self._shader_list = [ShaderType.BASE]
75
76     def create_vao(self, shader_type):
77         """
78             Creates a vertex array object (VAO) for the given shader type.
79
80             Args:
81                 shader_type (ShaderType): The type of shader.
82
83             frag_name = shader_type[1:] if shader_type[0] == '_' else shader_type
84             vert_path = Path(shader_path / 'vertex/base.vert').resolve()
85             frag_path = Path(shader_path / f'fragments/{frag_name}.frag').resolve()
86
87             self._vert_shaders[shader_type] = vert_path.read_text()
88             self._frag_shaders[shader_type] = frag_path.read_text()
89
90             program = self._ctx.program(vertex_shader=self._vert_shaders[shader_type],
91             fragment_shader=self._frag_shaders[shader_type])
92             self._programs[shader_type] = program
93
94             if shader_type == ShaderType._CALIBRATE:
95                 self._vaos[shader_type] = self._ctx.vertex_array(self._programs[shader_type], [(self._pygame_buffer, '2f 2f', 'vert', 'texCoords')])
96             else:
97                 self._vaos[shader_type] = self._ctx.vertex_array(self._programs[shader_type], [(self._opengl_buffer, '2f 2f', 'vert', 'texCoords')]))
98
99     def create_framebuffer(self, shader_type, size=None, filter=moderngl.NEAREST):
100
101         Creates a framebuffer for the given shader type.
102
103         Args:

```

```

100         shader_type (ShaderType): The type of shader.
101         size (tuple[int, int], optional): The size of the framebuffer.
102             Defaults to screen size.
103             filter (moderngl.Filter, optional): The texture filter. Defaults to
104             NEAREST.
105             """
106             texture_size = size or self._screen_size
107             texture = self._ctx.texture(size=texture_size, components=4)
108             texture.filter = (filter, filter)
109
110             self._textures[shader_type] = texture
111             self.framebuffers[shader_type] = self._ctx.framebuffer(color_attachments=[self._textures[shader_type]])
112
113     def render_to_fbo(self, shader_type, texture, output_fbo=None, program_type=None,
114                      use_image=True, **kwargs):
115         """
116             Applies the shaders and renders the resultant texture to a framebuffer
117             object (FBO).
118
119             Args:
120                 shader_type (ShaderType): The type of shader.
121                 texture (moderngl.Texture): The texture to render.
122                 output_fbo (moderngl.Framebuffer, optional): The output framebuffer.
123                     Defaults to None.
124                 program_type (ShaderType, optional): The program type. Defaults to
125                     None.
126                 use_image (bool, optional): Whether to use the image uniform. Defaults
127                     to True.
128                 **kwargs: Additional uniforms for the fragment shader.
129             """
130             fbo = output_fbo or self.framebuffers[shader_type]
131             program = self._programs[program_type] if program_type else self._programs
132             [shader_type]
133             vao = self._vaos[program_type] if program_type else self._vaos[shader_type]
134
135             fbo.use()
136             texture.use(0)
137
138             if use_image:
139                 program['image'] = 0
140                 for uniform, value in kwargs.items():
141                     program[uniform] = value
142
143             vao.render(mode=moderngl.TRIANGLE_STRIP)
144
145     def apply_shader(self, shader_type, **kwargs):
146         """
147             Applies a shader of the given type and adds it to the list.
148
149             Args:
150                 shader_type (ShaderType): The type of shader to apply.
151
152             Raises:
153                 ValueError: If the shader is already being applied.
154             """
155             if shader_type in self._shader_list:
156                 return
157
158             self.load_shader(shader_type, **kwargs)
159             self._shader_list.append(shader_type)

```

```

153     # Sort shader list based on the order in SHADER_PRIORITY, so that more
154     # important shaders are applied first
155     self._shader_list.sort(key=lambda shader: -SHADER_PRIORITY.index(shader))
156
157     def remove_shader(self, shader_type):
158         """
159         Removes a shader of the given type from the list.
160
161         Args:
162             shader_type (ShaderType): The type of shader to remove.
163
164         if shader_type in self._shader_list:
165             self._shader_list.remove(shader_type)
166
167     def render_output(self):
168         """
169         Renders the final output to the screen.
170
171         # Render to the screen framebuffer
172         self._ctx.screen.use()
173
174         # Take the texture of the last framebuffer to be rendered to, and render
175         # that to the screen framebuffer
176         output_shader_type = self._shader_list[-1]
177         self.get_fbo_texture(output_shader_type).use(0)
178         self._programs[output_shader_type]['image'] = 0
179
180         self._vaos[output_shader_type].render(mode=moderngl.TRIANGLE_STRIP)
181
182     def get_fbo_texture(self, shader_type):
183         """
184         Gets the texture from the specified shader type's FBO.
185
186         Args:
187             shader_type (ShaderType): The type of shader.
188
189         Returns:
190             moderngl.Texture: The texture from the FBO.
191
192         return self.framebuffers[shader_type].color_attachments[0]
193
194     def calibrate_pygame_surface(self, pygame_surface):
195         """
196         Converts the Pygame window surface into an OpenGL texture.
197
198         Args:
199             pygame_surface (pygame.Surface): The finished Pygame surface.
200
201         Returns:
202             moderngl.Texture: The calibrated texture.
203
204         texture = self._ctx.texture(pygame_surface.size, 4)
205         texture.filter = (moderngl.NEAREST, moderngl.NEAREST)
206         texture.swizzle = 'BGRA'
207         # Take the Pygame surface's pixel array and draw it to the new texture
208         texture.write(pygame_surface.get_view('1'))
209
210         # ShaderType._CALIBRATE has a VAO containing the pygame_quad_array
211         # coordinates, as Pygame uses different texture coordinates than ModernGL
212         # textures
213         self.render_to_fbo(ShaderType._CALIBRATE, texture)
214
215         return self.get_fbo_texture(ShaderType._CALIBRATE)

```

```

211
212     def draw(self, surface, arguments):
213         """
214             Draws the Pygame surface with shaders applied to the screen.
215
216         Args:
217             surface (pygame.Surface): The final Pygame surface.
218             arguments (dict): A dict of { ShaderType: Args } items, containing
219                 keyword arguments for every fragment shader.
220             """
221         self._ctx.viewport = (0, 0, *self._screen_size)
222         texture = self.calibrate_pygame_surface(surface)
223
224         for shader_type in self._shader_list:
225             self._shader_passes[shader_type].apply(texture, **arguments.get(
226                 shader_type, {}))
227             texture = self.get_fbo_texture(shader_type)
228
229         self.render_output()
230
231     def __del__(self):
232         """
233             Cleans up ModernGL resources when the ShaderManager object is deleted.
234
235         self.cleanup()
236
237     def cleanup(self):
238         """
239             Cleans up resources used by the ModernGL.
240             Probably unnecessary as the 'auto' garbage collection mode is used.
241
242             self._pygame_buffer.release()
243             self._opengl_buffer.release()
244             for program in self._programs:
245                 self._programs[program].release()
246             for texture in self._textures:
247                 self._textures[texture].release()
248             for vao in self._vaos:
249                 self._vaos[vao].release()
250             for framebuffer in self.framebuffers:
251                 self.framebuffers[framebuffer].release()
252
253     def handle_resize(self, new_screen_size):
254         """
255             Handles resizing of the screen.
256
257         Args:
258             new_screen_size (tuple[int, int]): The new screen size.
259
260         self._screen_size = new_screen_size
261
262         # Recreate all framebuffers to prevent scaling issues
263         for shader_type in self.framebuffers:
264             filter = self._textures[shader_type].filter[0]
265             self.create_framebuffer(shader_type, size=self._screen_size, filter=
266                 filter)

```

### 3.9.2 Bloom

The `Bloom` shader effect is a common shader effect giving the illusion of a bright light. It consists of blurred fringes of light extending from the borders of bright areas. This effect can be achieved through obtaining all bright areas of the image, applying a Gaussian blur, and blending the blur additively onto the original image.

My `ShaderManager` class works with this multi-pass shader approach by reading the texture from the last shader's framebuffer for each pass.

#### Extracting bright colours

The `highlight_brightness` fragment shader extracts all colours that are bright enough to exert the bloom effect.

`highlight_brightness.frag`

```
1 # version 330 core
2
3 in vec2 uvs;
4 out vec4 f_colour;
5
6 uniform sampler2D image;
7 uniform float threshold;
8 uniform float intensity;
9
10 void main() {
11     vec4 pixel = texture(image, uvs);
12     // Dot product used to calculate brightness of a pixel from its RGB values
13     // Values taken from https://en.wikipedia.org/wiki/Relative_luminance
14     float brightness = dot(pixel.rgb, vec3(0.2126, 0.7152, 0.0722));
15     float isBright = step(threshold, brightness);
16
17     f_colour = vec4(vec3(pixel.rgb * intensity) * isBright, 1.0);
18 }
```

#### Blur

The `Blur` class implements a two-pass Gaussian blur. This is preferably over a one-pass blur, as the complexity is  $O(2n)$ , sampling  $n$  pixels twice, as opposed to  $O(n^2)$ . I have implemented this using the ping-pong technique, with the first pass for blurring the image horizontally, and the second pass for blurring vertically, and the resultant textures being passed repeatedly between two framebuffers.

`blur.py`

```
1 from data.shaders.protocol import SMProtocol
2 from data.constants import ShaderType
3
4 BLUR_ITERATIONS = 4
5
6 class _Blur:
7     def __init__(self, shader_manager: SMProtocol):
8         self._shader_manager = shader_manager
9
10     shader_manager.create_framebuffer(ShaderType._BLUR)
11
12     shader_manager.create_framebuffer("blurPing")
13     shader_manager.create_framebuffer("blurPong")
14
15     def apply(self, texture):
```

```

16     """
17     Applies Gaussian blur to a given texture.
18
19     Args:
20         texture (moderngl.Texture): Texture to blur.
21     """
22     self._shader_manager.get_fbo_texture("blurPong").write(texture.read())
23
24     for _ in range(BLUR_ITERATIONS):
25         # Apply horizontal blur
26         self._shader_manager.render_to_fbo(
27             ShaderType._BLUR,
28             texture=self._shader_manager.get_fbo_texture("blurPong"),
29             output_fbo=self._shader_manager.framebuffers["blurPing"],
30             passes=5,
31             horizontal=True
32         )
33         # Apply vertical blur
34         self._shader_manager.render_to_fbo(
35             ShaderType._BLUR,
36             texture=self._shader_manager.get_fbo_texture("blurPing"), # Use
37             horizontal blur result as input texture
38             output_fbo=self._shader_manager.framebuffers["blurPong"],
39             passes=5,
40             horizontal=False
41
42     self._shader_manager.render_to_fbo(ShaderType._BLUR, self._shader_manager.
43     get_fbo_texture("blurPong"))

blur.frag
1 // Modified from https://learnopengl.com/Advanced-Lighting/Bloom
2 #version 330 core
3
4 in vec2 uvs;
5 out vec4 f_colour;
6
7 uniform sampler2D image;
8 uniform bool horizontal;
9 uniform int passes;
10 uniform float weight[5] = float[] (0.227027, 0.1945946, 0.1216216, 0.054054,
11                                     0.016216);
12 void main() {
13     vec2 offset = 1.0 / textureSize(image, 0);
14     vec3 result = texture(image, uvs).rgb * weight[0];
15
16     if (horizontal) {
17         for (int i = 1 ; i < passes ; ++i) {
18             result += texture(image, uvs + vec2(offset.x * i, 0.0)).rgb * weight[i];
19             result += texture(image, uvs - vec2(offset.x * i, 0.0)).rgb * weight[i];
20         }
21     }
22     else {
23         for (int i = 1 ; i < passes ; ++i) {
24             result += texture(image, uvs + vec2(0.0, offset.y * i)).rgb * weight[i];
25             result += texture(image, uvs - vec2(0.0, offset.y * i)).rgb * weight[i];
26         }
27     }
28 }
```

```

26         }
27     }
28     f_colour = vec4(result, 1.0);
30 }
```

## Combining

The `Bloom` class combines the two operations, taking the highlighted areas, blurs them, and adds the RGB values for the final result onto the original texture to simulate bloom.

`bloom.py`

```

1  from data.shaders.classes.highlight_brightness import _HighlightBrightness
2  from data.shaders.classes.highlight_colour import _HighlightColour
3  from data.shaders.protocol import SMProtocol
4  from data.shaders.classes.blur import _Blur
5  from data.constants import ShaderType
6
7  BLOOM_INTENSITY = 0.6
8
9  class Bloom:
10     def __init__(self, shader_manager: SMProtocol):
11         self._shader_manager = shader_manager
12
13         shader_manager.load_shader(ShaderType._BLUR)
14         shader_manager.load_shader(ShaderType._HIGHLIGHT_BRIGHTNESS)
15         shader_manager.load_shader(ShaderType._HIGHLIGHT_COLOUR)
16
17         shader_manager.create_framebuffer(ShaderType.BLOOM)
18         shader_manager.create_framebuffer(ShaderType._BLUR)
19         shader_manager.create_framebuffer(ShaderType._HIGHLIGHT_BRIGHTNESS)
20         shader_manager.create_framebuffer(ShaderType._HIGHLIGHT_COLOUR)
21
22     def apply(self, texture, highlight_surface=None, highlight_colours=[], surface_intensity=BLOOM_INTENSITY, brightness_intensity=BLOOM_INTENSITY, colour_intensity=BLOOM_INTENSITY):
23         """
24             Applies a bloom effect to a given texture.
25
26         Args:
27             texture (moderngl.Texture): Texture to apply bloom to.
28             highlight_surface (pygame.Surface, optional): Surface to use as the highlights. Defaults to None.
29             highlight_colours (list[list[int, int, int], ...], optional): Colours to use as the highlights. Defaults to [].
30             surface_intensity (_type_, optional): Intensity of bloom applied to the highlight surface. Defaults to BLOOM_INTENSITY.
31             brightness_intensity (_type_, optional): Intensity of bloom applied to the highlight brightness. Defaults to BLOOM_INTENSITY.
32             colour_intensity (_type_, optional): Intensity of bloom applied to the highlight colours. Defaults to BLOOM_INTENSITY.
33         """
34
35         if highlight_surface:
36             # Calibrate Pygame surface and apply blur
37             glare_texture = self._shader_manager.calibrate_pygame_surface(
38                 highlight_surface)
39             _Blur(self._shader_manager).apply(glare_texture)
40
41             self._shader_manager.get_fbo_texture(ShaderType._BLUR).use(1)
42             self._shader_manager.render_to_fbo(ShaderType.BLOOM, texture,
43                 blurredImage=1, intensity=surface_intensity)
```

```

41         # Set bloom-applied texture as the base texture
42         texture = self._shader_manager.get_fbo_texture(ShaderType.BLOOM)
43
44         # Extract bright colours (highlights) from the texture
45         _HighlightBrightness(self._shader_manager).apply(texture, intensity=
46             brightness_intensity)
47         highlight_texture = self._shader_manager.get_fbo_texture(ShaderType.
48             _HIGHLIGHT_BRIGHTNESS)
49
50         # Use colour as highlights
51         for colour in highlight_colours:
52             _HighlightColour(self._shader_manager).apply(texture, old_highlight=
53                 highlight_texture, colour=colour, intensity=colour_intensity)
54             highlight_texture = self._shader_manager.get_fbo_texture(ShaderType.
55                 _HIGHLIGHT_COLOUR)
56
57         # Apply Gaussian blur to highlights
58         _Blur(self._shader_manager).apply(highlight_texture)
59
60         # Add the pixel values for the highlights onto the base texture
61         self._shader_manager.get_fbo_texture(ShaderType._BLUR).use(1)
62         self._shader_manager.render_to_fbo(ShaderType.BLOOM, texture, blurredImage=
63             1, intensity=BLOOM_INTENSITY)

```

### 3.9.3 Rays

The `Ray` shader is applied whenever the sphinx shoots a laser. It simulates a 2D light source, providing pixel perfect shadows, through the shadow mapping technique outlined in Section 2.2.5. The laser demo seen on the main menu screen is also achieved using the Ray shader, by clamping the angle at which it emits light to a narrower range.

#### Occlusion

The occlusion fragment shader processes all pixels with a given colour value as being occluding. `occlusion.frag`

```

1 # version 330 core
2
3 in vec2 uvs;
4 out vec4 f_colour;
5
6 uniform sampler2D image;
7 uniform vec3 checkColour;
8
9 void main() {
10     vec4 pixel = texture(image, uvs);
11
12     // If pixel is occluding colour, set pixel to white
13     if (pixel.rgb == checkColour) {
14         f_colour = vec4(1.0, 1.0, 1.0, 1.0);
15     // Else, set pixel to black
16     } else {
17         f_colour = vec4(vec3(0.0), 1.0);
18     }
19 }

```

## Shadowmap

The shadowmap fragment shader takes the occluding texture and creates a 1D shadow map.  
`shadowmap.frag`

```
1 # version 330 core
2
3 #define PI 3.1415926536;
4
5 in vec2 uvs;
6 out vec4 f_colour;
7
8 uniform sampler2D image;
9 uniform float resolution;
10 uniform float THRESHOLD=0.99;
11
12 void main() {
13     float maxDistance = 1.0;
14
15     for (float y = 0.0 ; y < resolution ; y += 1.0) {
16         //rectangular to polar filter
17         float currDistance = y / resolution;
18
19         vec2 norm = vec2(uvs.x, currDistance) * 2.0 - 1.0; // Range from [0, 1] ->
19 [-1, 1]
20         float angle = (1.5 - norm.x) * PI; // Range from [-1, 1] -> [0.5PI, 2.5PI]
21         float radius = (1.0 + norm.y) * 0.5; // Range from [-1, 1] -> [0, 1]
22
23         //coord which we will sample from occlude map
24         vec2 coords = vec2(radius * -sin(angle), radius * -cos(angle)) / 2.0 +
24 0.5;
25
26         // Sample occlusion map
27         vec4 occluding = texture(image, coords);
28
29         // If pixel is not occluding (Red channel value below threshold), set
29 maxDistance to current distance
30         // If pixel is occluding, don't change distance
31         // maxDistance therefore is the distance from the center to the nearest
31 occluding pixel
32         maxDistance = max(maxDistance * step(occluding.r, THRESHOLD), min(
32 maxDistance, currDistance));
33     }
34
35     f_colour = vec4(vec3(maxDistance), 1.0);
36 }
```

## Lightmap

The lightmap shader checks if a pixel is in shadow, blurs the result, and applies the radial light source.

`lightmap.frag`

```
1 # version 330 core
2
3 #define PI 3.14159265
4
5 in vec2 uvs;
6 out vec4 f_colour;
7
8 uniform float softShadow;
```

```

9 uniform float resolution;
10 uniform float falloff;
11 uniform vec3 lightColour;
12 uniform vec2 angleClamp;
13 uniform sampler2D occlusionMap;
14 uniform sampler2D image;
15
16 vec3 normLightColour = lightColour / 255;
17 vec2 radiansClamp = angleClamp * (PI / 180);
18
19 float sample(vec2 coord, float r) {
20     /*
21     Sample from the 1D distance map.
22
23     Returns:
24     float: 1.0 if sampled radius is greater than the passed radius, 0.0 if not.
25     */
26     return step(r, texture(image, coord).r);
27 }
28
29 void main() {
30     // Cartesian to polar transformation
31     // Range from [0, 1] -> [-1, 1]
32     vec2 norm = uvs.xy * 2.0 - 1.0;
33     float angle = atan(norm.y, norm.x);
34     float r = length(norm);
35
36     // The texture coordinates to sample our 1D lookup texture
37     // Always 0.0 on y-axis, as the texture is 1D
38     float x = (angle + PI) / (2.0 * PI); // Normalise angle to [0, 1]
39     vec2 tc = vec2(x, 0.0);
40
41     // Sample the 1D lookup texture to check if pixel is in light or in shadow
42     // Gives us hard shadows
43     // 1.0 -> in light, 0.0, -> in shadow
44     float inLight = sample(tc, r);
45     // Clamp angle so that only pixels within the range are in light
46     inLight = inLight * step(angle, radiansClamp.y) * step(radiansClamp.x, angle);
47
48     // Multiply the blur amount by the distance from the center
49     // So that the blurring increases as distance increases
50     float blur = (1.0 / resolution) * smoothstep(0.0, 0.1, r);
51
52     // Use gaussian blur to apply blur effecy
53     float sum = 0.0;
54
55     sum += sample(vec2(tc.x - blur * 4.0, tc.y), r) * 0.05;
56     sum += sample(vec2(tc.x - blur * 3.0, tc.y), r) * 0.09;
57     sum += sample(vec2(tc.x - blur * 2.0, tc.y), r) * 0.12;
58     sum += sample(vec2(tc.x - blur * 1.0, tc.y), r) * 0.15;
59
60     sum += inLight * 0.16;
61
62     sum += sample(vec2(tc.x + blur * 1.0, tc.y), r) * 0.15;
63     sum += sample(vec2(tc.x + blur * 2.0, tc.y), r) * 0.12;
64     sum += sample(vec2(tc.x + blur * 3.0, tc.y), r) * 0.09;
65     sum += sample(vec2(tc.x + blur * 4.0, tc.y), r) * 0.05;
66
67     // Mix with the softShadow uniform to toggle degree of softShadows
68     float finalLight = mix(inLight, sum, softShadow);
69
70     // Multiply the final light value with the distance, to give a radial falloff

```

```

71     // Use as the alpha value, with the light colour being the RGB values
72     f_colour = vec4(normLightColour, finalLight * smoothstep(1.0, falloff, r));
73 }

```

## Class

The `Rays` class takes in a texture and array of light information, applies the aforementioned shaders, and blends the final result with the original texture.

`rays.py`

```

1  from data.shaders.classes.lightmap import _Lightmap
2  from data.shaders.classes.blend import _Blend
3  from data.shaders.protocol import SMProtocol
4  from data.shaders.classes.crop import _Crop
5  from data.constants import ShaderType
6
7  class Rays:
8      def __init__(self, shader_manager: SMProtocol, lights):
9          self._shader_manager = shader_manager
10         self._lights = lights
11
12         # Load all necessary shaders
13         shader_manager.load_shader(ShaderType._LIGHTMAP)
14         shader_manager.load_shader(ShaderType._BLEND)
15         shader_manager.load_shader(ShaderType._CROP)
16         shader_manager.create_framebuffer(ShaderType.RAYS)
17
18     def apply(self, texture, occlusion=None, softShadow=0.3):
19         """
20             Applies the light rays effect to a given texture.
21
22             Args:
23                 texture (moderngl.Texture): The texture to apply the effect to.
24                 occlusion (pygame.Surface, optional): A Pygame mask surface to use as
25                 the occlusion texture. Defaults to None.
26         """
27
28         final_texture = texture
29
30         # Iterate through array containing light information
31         for pos, radius, colour, *args in self._lights:
32             # Topleft of light source square
33             light_topleft = (pos[0] - (radius * texture.size[1] / texture.size[0]),
34             pos[1] - radius)
35             # Relative size of light compared to texture
36             relative_size = (radius * 2 * texture.size[1] / texture.size[0],
37             radius * 2)
38
39             # Crop texture to light source diameter, and to position light source
40             # at the center
41             _Crop(self._shader_manager).apply(texture, relative_pos=light_topleft,
42             relative_size=relative_size)
43             cropped_texture = self._shader_manager.get_fbo_texture(ShaderType.
44             _CROP)
45
46             if occlusion:
47                 # Calibrate Pygame mask surface and crop it
48                 occlusion_texture = self._shader_manager.calibrate_pygame_surface(
49                 occlusion)
50
51                 _Crop(self._shader_manager).apply(occlusion_texture, relative_pos=
52                 light_topleft, relative_size=relative_size)

```

```

43         occlusion_texture = self._shader_manager.get_fbo_texture(
44             ShaderType._CROP)
45         else:
46             occlusion_texture = None
47
48         # Apply lightmap shader, shadowmap and occlusion are included within
49         # the _Lightmap class
50         _Lightmap(self._shader_manager).apply(cropped_texture, colour,
51             softShadow, occlusion_texture, *args)
52         light_map = self._shader_manager.get_fbo_texture(ShaderType._LIGHTMAP)
53
54         # Blend the final result with the original texture
55         _Blend(self._shader_manager).apply(final_texture, light_map,
56         light_topleft)
57         final_texture = self._shader_manager.get_fbo_texture(ShaderType._BLEND
58     )
59
60     self._shader_manager.render_to_fbo(ShaderType.RAYS, final_texture)

```

### 3.9.4 Stack

# Chapter 4

## Testing

### 4.1 Iterative Testing

I have been playtesting the program throughout the development process to find any bugs and fix them accordingly. However, a few issues have required additional measures.

#### 4.1.1 Minimax

Since minimax is recursive algorithm, debugging it has proven a challenge. I have therefore configured the Python `logging` library to help collect information on the minimax tree for every function call.

`base.py`

```
1  def print_stats(self, score, move):
2      """
3          Prints statistics after traversing tree.
4
5          Args:
6              score (int): Final score obtained after traversal.
7              move (Move): Best move obtained after traversal.
8
9          If self._verbose is False:
10              return
11
12          self._stats['time_taken'] = round(1000 * (time.time() - self._stats['
13              time_taken']), 3)
14          self._stats['ms_per_node'] = round(self._stats['time_taken'] / self._stats
15              ['nodes'], 3)
16
17          # Prints stats across multiple lines
18          if self._verbose is True:
19              logger.info(f'{self.__str__()} Search Results:')
20              logger.info(printer.pformat(self._stats))
21              logger.info(f'Best score: {score}    Best move: {move}')
22
23          # Prints stats in a compacted format
24          elif self._verbose.lower() == 'compact':
25              logger.info(self._stats)
26              logger.info(f'Best score: {score}    Best move: {move}')
```

Listing 4.1: BaseCPU Method for logging minimax statistics

### 4.1.2 Migrations

To correct errors made to the `games` table, since recreating it would mean deleting all existing games, I have opted to use migrations to fix bugs by editing the table schema, as shown in Section 3.8.1.

## 4.2 Unit Tests

### 4.2.1 Board Evaluator

To test every aspect of the evaluation function, I have set up some unit tests with custom positions using my editor screen. These positions are designed to test every aspect of the evaluation, along with some obviously imbalanced positions to test the overall accuracy of the evaluation function. All positions are set up to give an advantage to the blue player.

Evaluating	FEN string	Score	Passed
Material	<code>sc9/10/10/4paPa4/5Pa4/10/10/9Sa b</code>	124	✓
Position	<code>sc9/4nanana3/10/10/10/4NaNaNa3/10/9Sa b</code>	66	✓
Mobility	See footnote <sup>1</sup>	196	✓
King Safety	<code>sc4fa3pa/10/10/10/10/10/10/5FaPa2Sa b</code>	3	✓
Combined	See footnote <sup>2</sup>	437	✓

Table 4.1: Board evaluator test results

### 4.2.2 CPU

Similarly, to evaluate the strength of my CPU, I have setup some custom positions that I already know the best continuation of, and run each CPU engine on them to test if they can solve it.

`sc9/pafa8/Fa9/10/10/10/10/9Sa b`

I have also personally played against the CPU engines to gauge its strength in a realistic setting. The results are shown below:

Me	CPU
2	5

Table 4.2: Score of me vs CPU (I am very bad)

Mr Myslov	CPU
1	6

Table 4.3: Score of Mr Myslov vs CPU (Mr Myslov is even worse)

<sup>1</sup>`scpapa7/papapa7/papapa1Pa1Pa1Pa1/10/4Pa1Pa1Pa1/10/4Pa1Pa3/9Sa b`

<sup>2</sup>`scnclfcncpbpb3/pa9/pb1pc1rbpa3Pd/1Pc2Pd4Pc/2Pd1RaRb4/10/7Pa2/2PdNaFaNa3Sa b`

### 4.2.3 Shadow Mapping

To test the shadow mapping algorithm, I have set up some occluding objects together with a light source. Since visuals are subjective, me and my client have deemed the following results to be adequate.

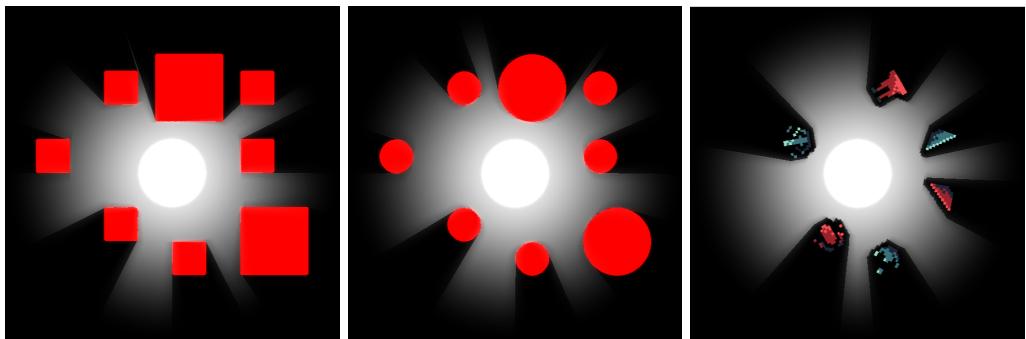


Figure 4.1: Shadow mapping algorithm tests (softShadow=0.5, radius=0.5)

## 4.3 Final Tests

### 4.3.1 Objective 1

### 4.3.2 Objective 2

### 4.3.3 Objective 3

## 4.4 Videos