NEA

Toby Mok

Contents

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/`.

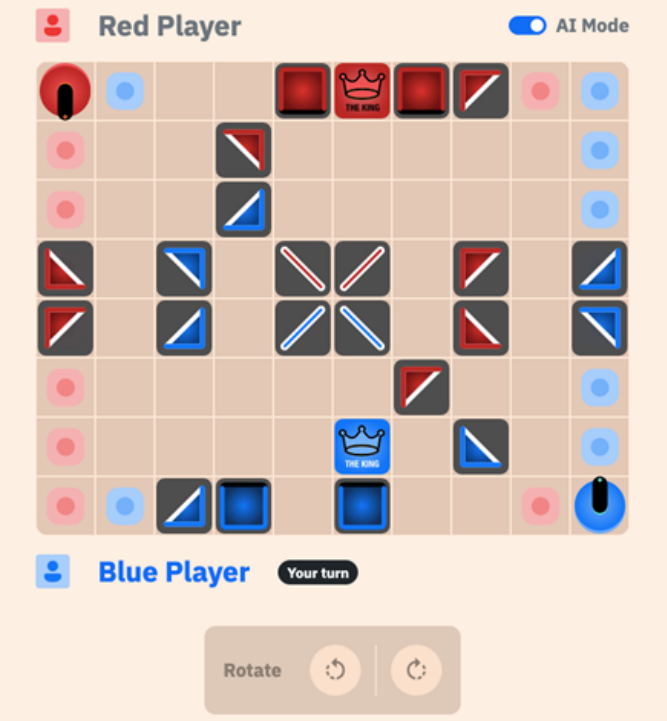


Figure 1: Online implementation on laser-chess.com

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 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 unreliably 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 a 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 be 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 significant 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. Unliked 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

- Generally faster runtime than Python
- Simple, dynamically typed and automatic memory management
- Versatile, easy integration with both server-side and front-end
- Extensive third-party modules
- Also supports object-oriented programming

- Mainly focused for web development
- Comprehensive knowledge of external frameworks (i.e. Electron) needed for developing desktop applications

---

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 mines.

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.

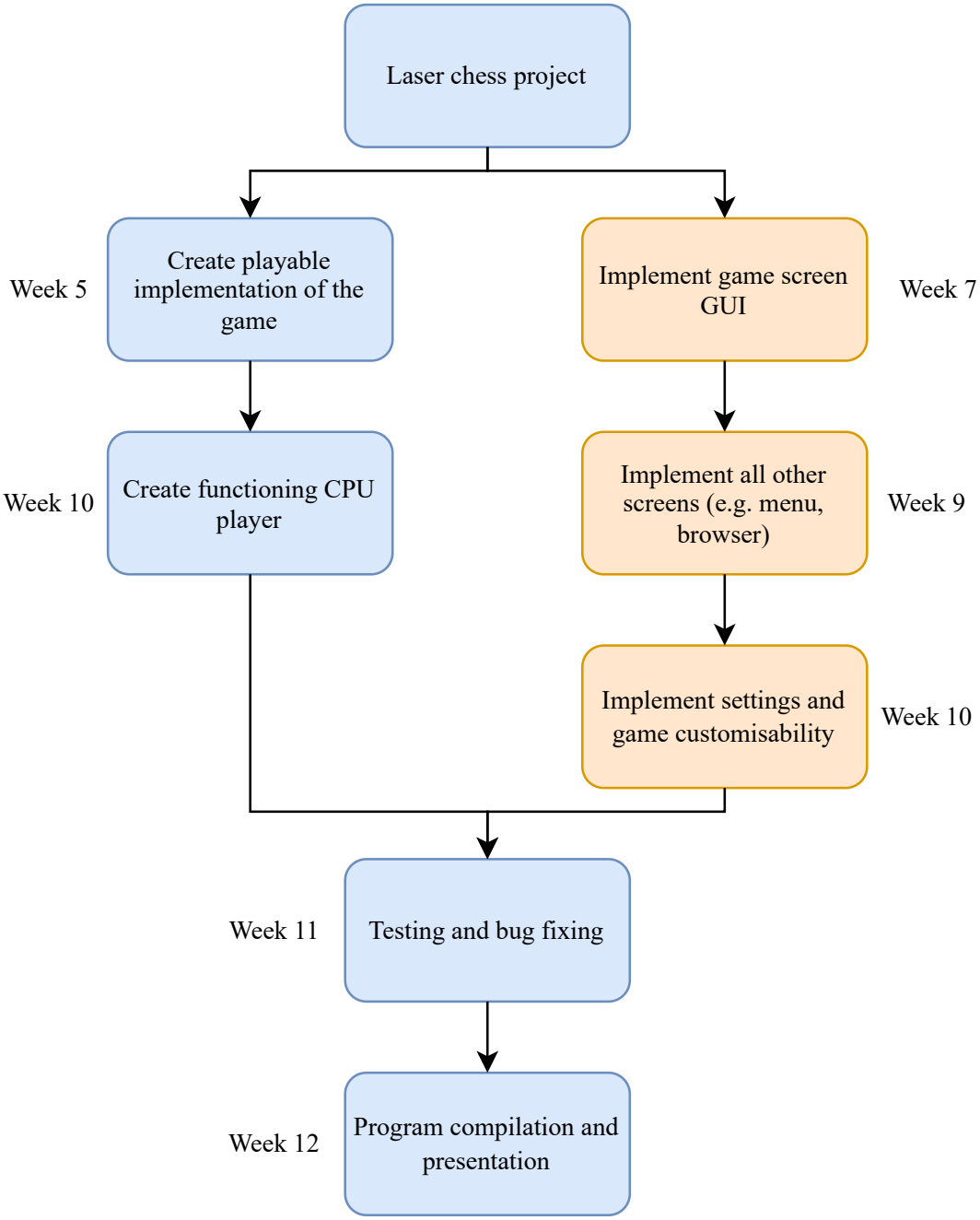
1.5 Limitations

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.


Laser chess project
Week 5
Create playable implementation of the game
Implement game screen GUI
Week 7
Week 10
Create functioning CPU player
Implement all other screens (e.g. menu, browser)
Week 9
Implement settings and game customisability
Week 10
Week 11
Testing and bug fixing
Week 12
Program compilation and presentation

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:

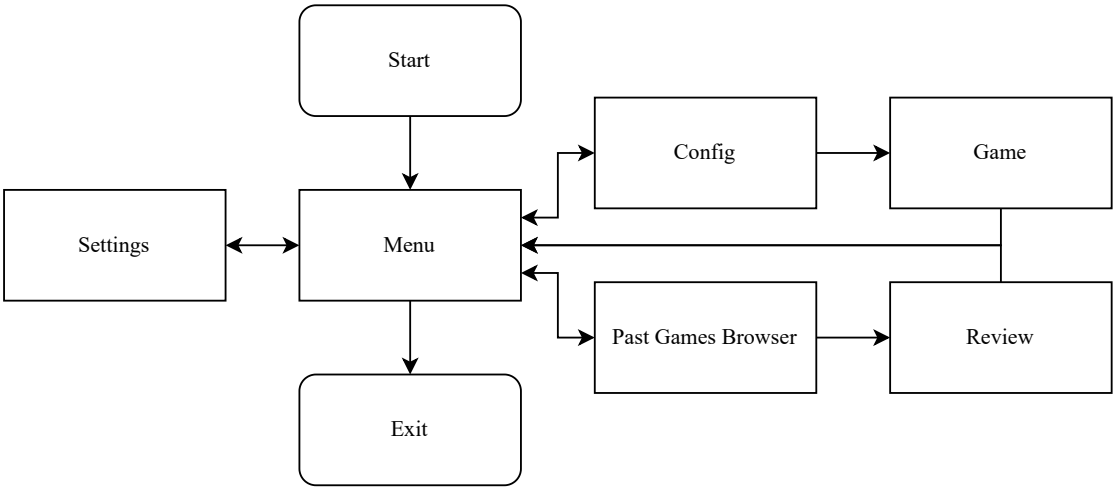


Figure 2: Flowchart for Program Overview

2.1.1 Main Menu

Figure 3: 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

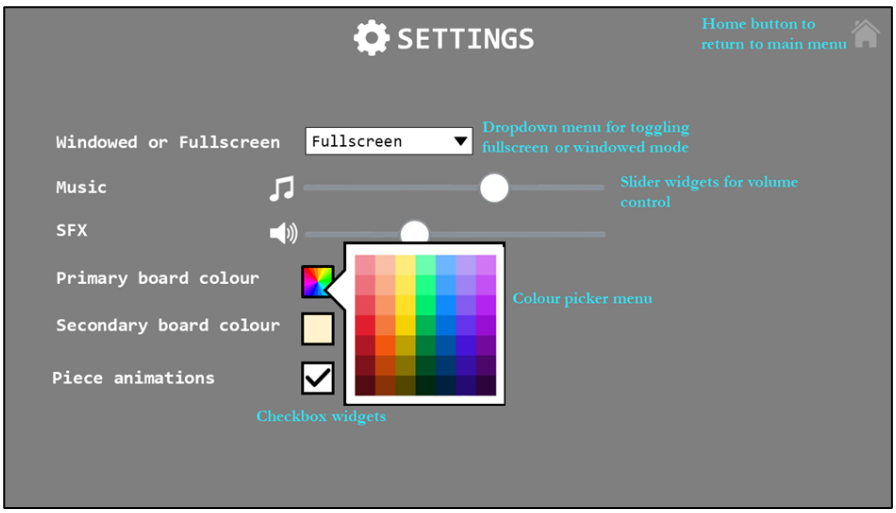


Figure 4: Settings screen prototype

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.

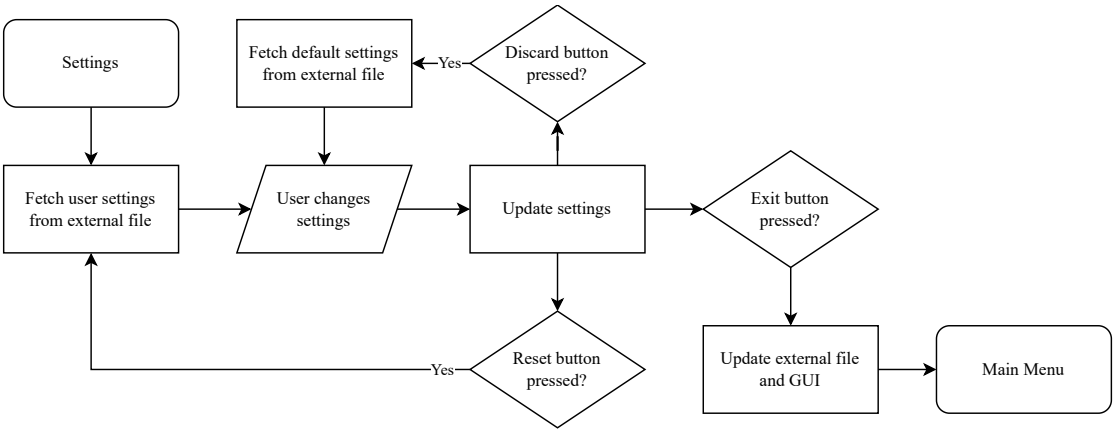


Figure 5: Flowchart for Settings

2.1.3 Past Games Browser

Figure 6: 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.

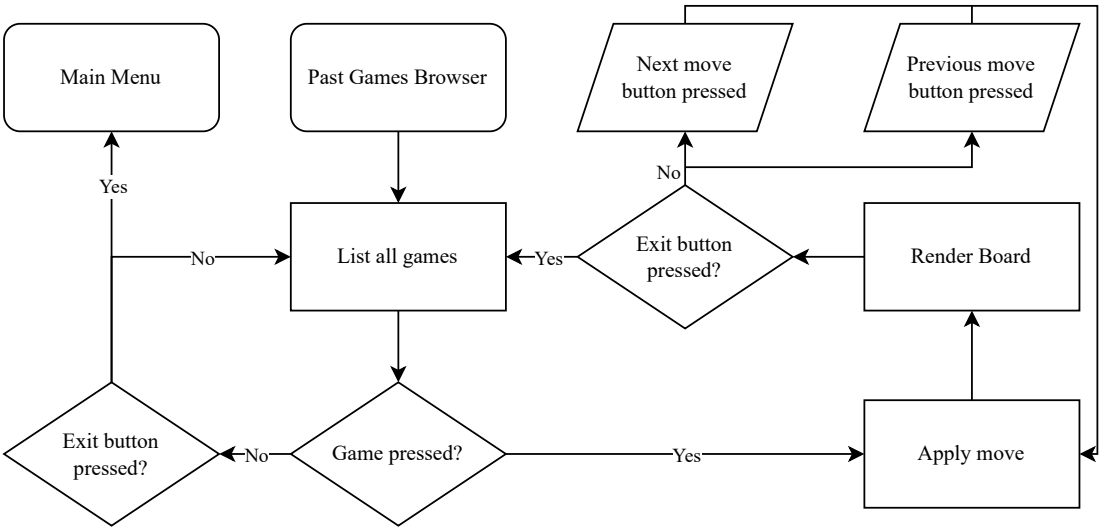


Figure 7: Flowchart for Browser

2.1.4 Config

Figure 8: 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.

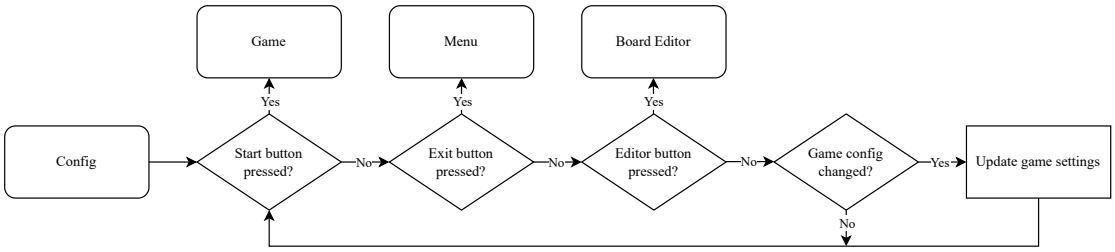


Figure 9: Flowchart for Config

2.1.5 Game

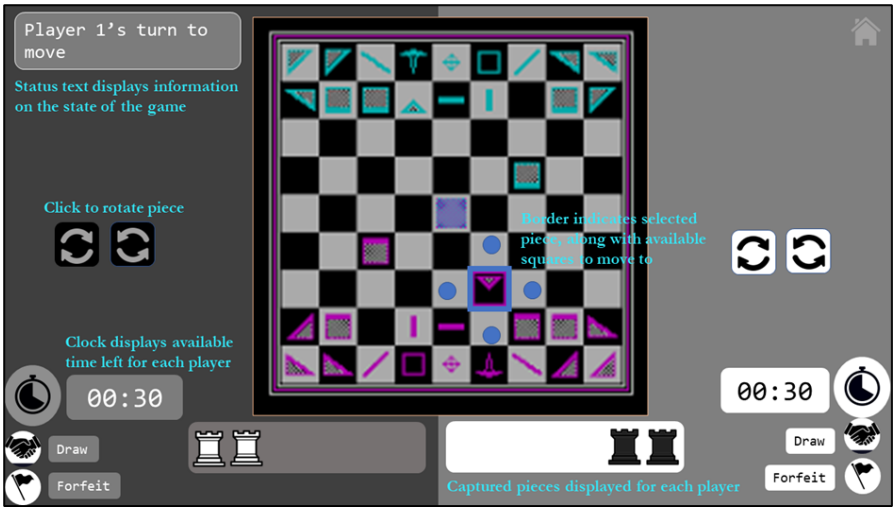


Figure 10: Game screen prototype

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, surrounding 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.

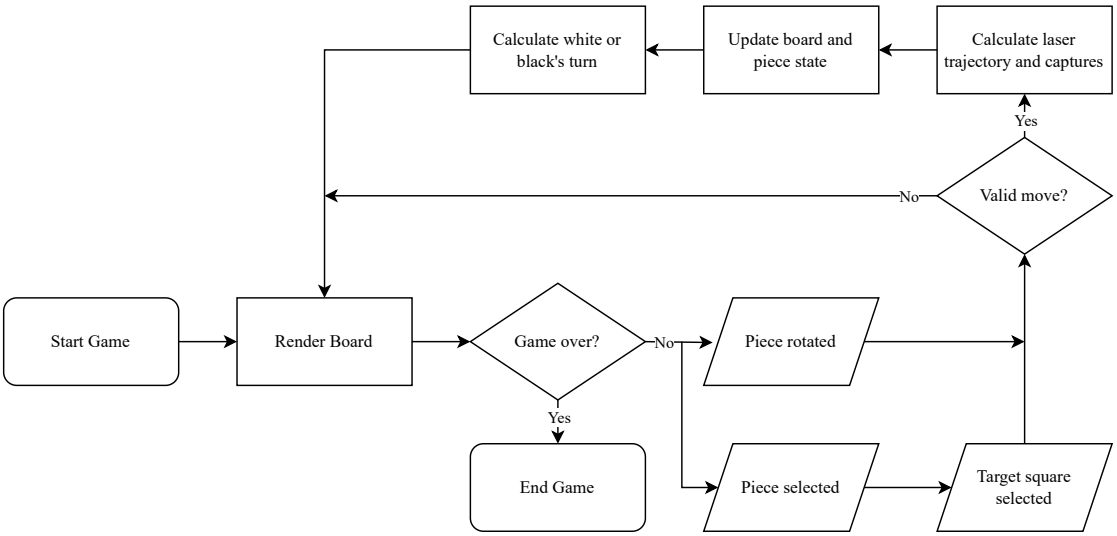


Figure 11: Flowchart for Game

2.1.6 Board Editor

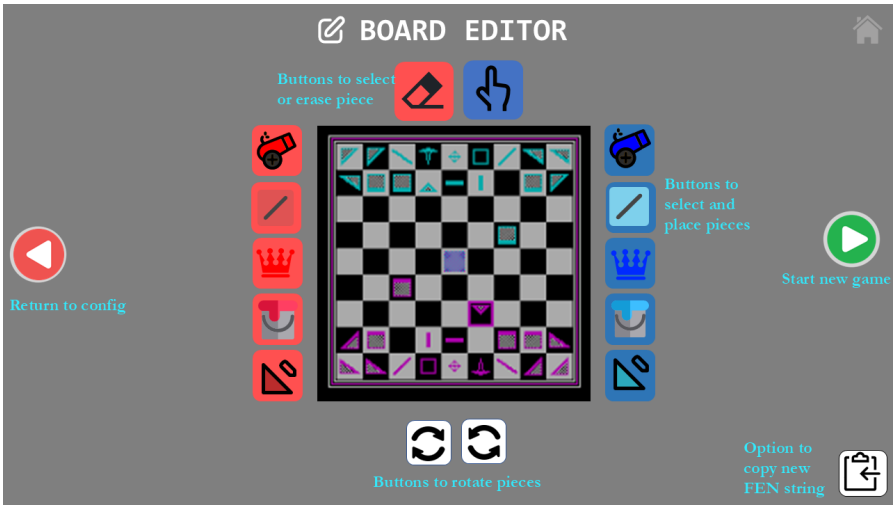


Figure 12: Editor screen prototype

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 appropiate colouring (e.g. red for delete).

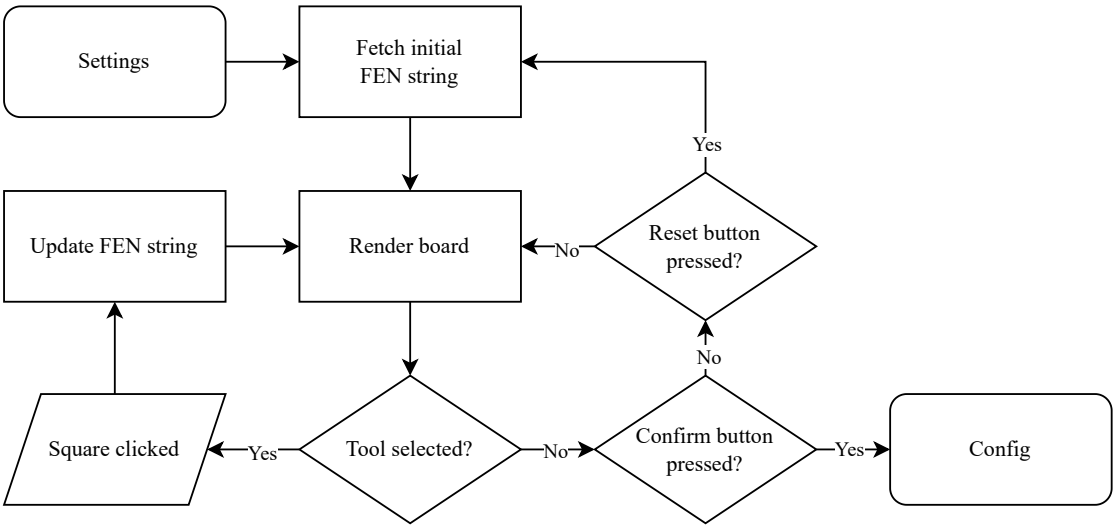


Figure 13: Flowchart for board editor

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 maximising the player's advantage in each recursive call.

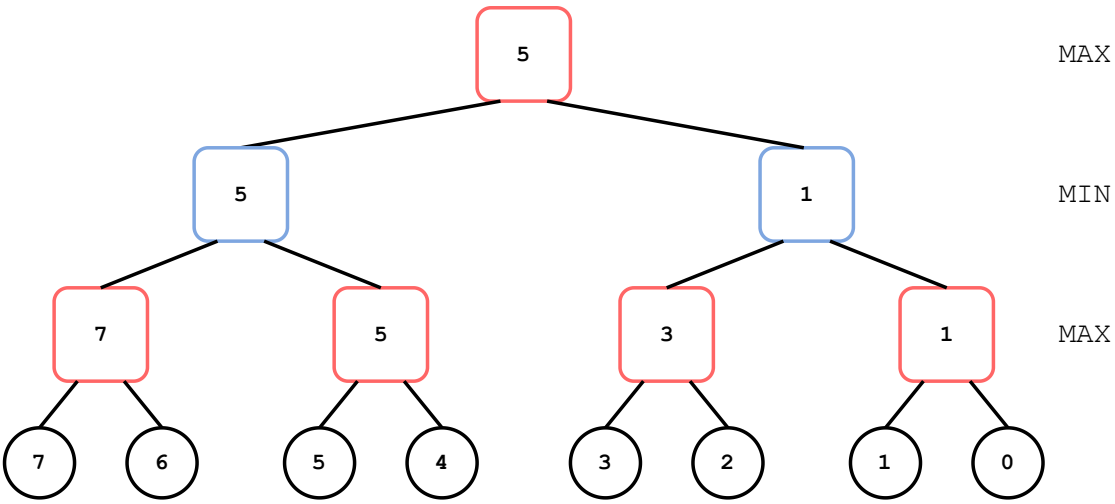


Figure 14: Example minimax tree

For the example minimax tree show in Figure 14, 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 ← −∞
        for child of node do
            value ← MAX(value, MINIMAX(child, depth − 1, false))
        end for
        return value
    else
        value ← +∞
        for child of node do
            value ← MIN(value, 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 (α), the upper bound, is the highest value that the maximising player is guaranteed of; Beta (β), 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:

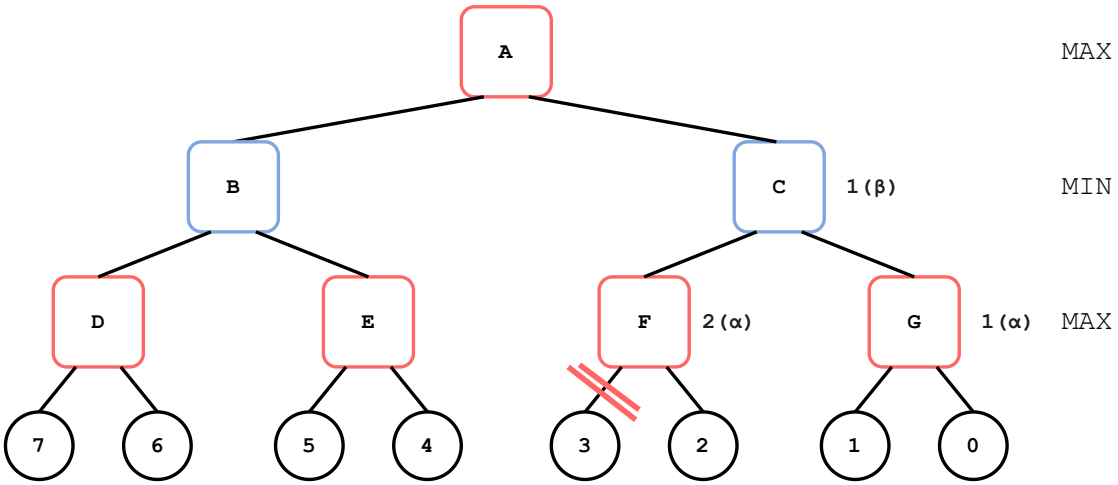


Figure 15: Example minimax tree with alpha-beta pruning

Since minimax is a depth-first search algorithm, nodes C and G and their α and β have already been searched. Next, at node F, the current α and β are $-\infty$ and 1 respectively, since the β is passed down from node C. Searching the first leaf node, the α 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 is 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 α and lower bound β. 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, α, β, maximisingPlayer)
    if depth = 0 OR node equals game over then
        return EVALUATE
    end if

    if maximisingPlayer then
        value ← −∞
        for child of node do
            value ← MAX(value, MINIMAX(child, depth − 1, α, β, false))
            if value > β then break
            end if
            α ← MAX(α, value)
        end for
        return value
    else
        value ← +∞
        for child of node do
            value ← MIN(value, MINIMAX(child, depth − 1, α, β, true))
            if value < α then break
            end if
            β ← MIN(β, 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 the 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 β, and is subsequently pruned, meaning that the true evaluation could be higher than the value stored. We are thus storing the α and not an exact value. Thus, when we fetch the cached value, we have to recheck if this value is greater than β. 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 α, 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 α. 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, α, β, maximisingPlayer)
    hash_key ← HASH(node)
    entry ← GETENTRY(hash_key)

    if entry.hash_key = hash_key AND entry.hash_key ≥ depth then
        if entry.hash_key = EXACT then
            return entry.value
        else if entry.hash_key = LOWER then
            α ← MAX(α, entry.value)
        else if entry.hash_key = UPPER then
            β ← MIN(β, entry.value)
        end if
        if α ≥ β then
            return entry.value
        end if
    end if

     ...normal minimax...

    entry.value ← value
    entry.depth ← depth
    if value ≤ α then
        entry.flag ← UPPER
    else if value ≥ β then
        entry.flag ← LOWER
    else
        entry.flag ← 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 throughh 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°, 90°, 180° and 270° 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 ← SPLIT(fen_string)
    rank ← 8
    file ← 0

    for character in part_1 do
        square ← rank × 8 + file
        if character is alphabetic then
            if character is lower then
                board.bitboards[red][character]  |  1 << character
            else
                board.bitboards[blue][character]  |  1 << character
            end if
        else if character is numeric then
            file ← file + character
        else if characteris/ then
            rank ← rank − 1
            file ← file + 1
        else
            file ← file + 1
        end if

        if part_2 is b then
            board.active_colour ← b
        else
            board.active_colour ← 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).

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

Figure 16: 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:

$$\text{all\_pieces\_bitboard} = \text{white\_pharoah\_bitboard} \mid \text{black\_pharoah\_bitboard} \mid \text{white\_scarab\_bitboard} \ldots$$

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:

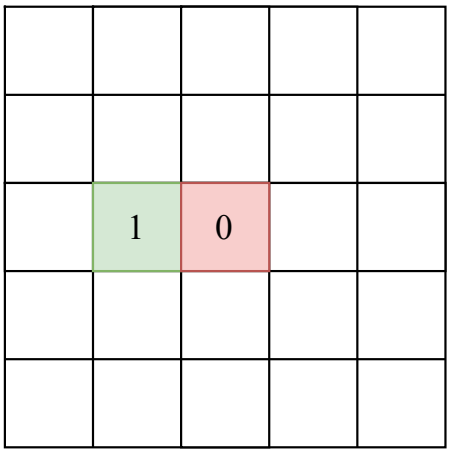


Figure 17: shifted_bitboard = piece_bitboard >> 1

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:

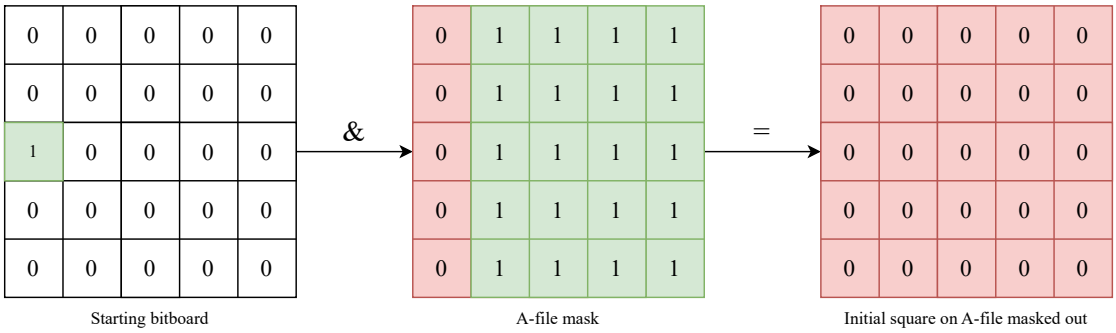


Figure 18: A-file mask example

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 ← selected_square & A_FILE_MASK
    masked_j_square ← selected_square & J_FILE_MASK

    top_left ← masked_a_square << 9
    top_left ← masked_a_square << 9
    top_middle ← selected_square << 10
    top_right ← masked_ << 11
    middle_right ← masked_ << 1
    bottom_right ← masked_ >> 9
    bottom_middle ← selected_square >> 10
    bottom_left ← masked_a_square >> 11
    middle_left ← 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:

- Pharoah - ∞
- 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 pharoah piece type on an example 5x5 board:

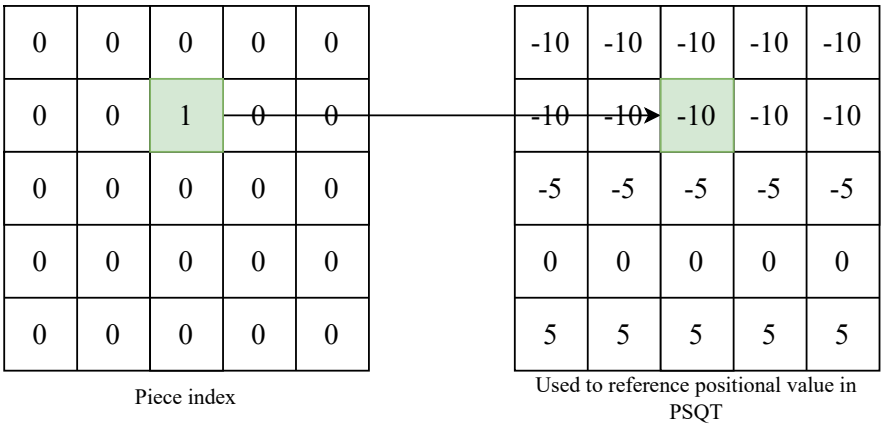


Figure 19: PSQT showing the bonus position value gained for the square occupied by a pharoah

For asymmetrical PSQTs, we would ideally like to label the board identically from both player's point of views, since currently we only have one set of PSQTs modelled from the blue perspective. We would like to flip the PSQTs to be reused from the red perspective, so that a generic algorithm can be used to sum up and calculate the total piece values for both players.

To utilise a PSQT for red pieces, a special 'FLIP' table can be implemented:

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

Figure 20: FLIP table used to map a red piece index to the blue player's perspective

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

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

The following function retrieves an array of bitb oards 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 (Pharoah safety) describes the level of protection of the pharoah, 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 pharoah, 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 pharoah 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 tecniques such as ray casting and ray marching, as its effiency 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

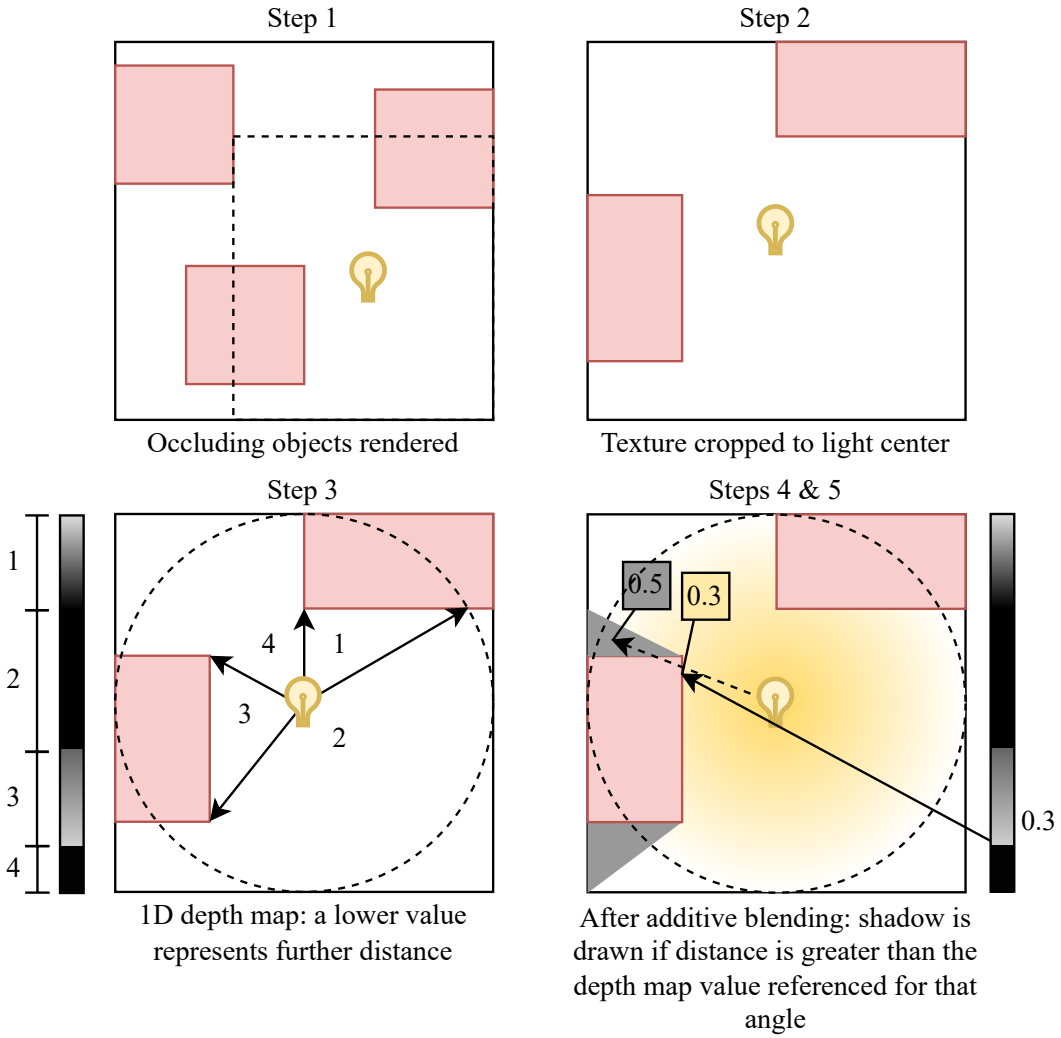


Figure 21: Workflow for 2D shadow mapping

Our method requires a coordinate transformation from Cartesian to polar, and vice versa. Polar to Cartesian transformation can be achieved with trigonemtry, 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 ($\frac{\pi}{2} < \theta < \frac{\pi}{2}$), 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

To 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 noraml distribution
3. Blur the final result proportional to the length from from the center

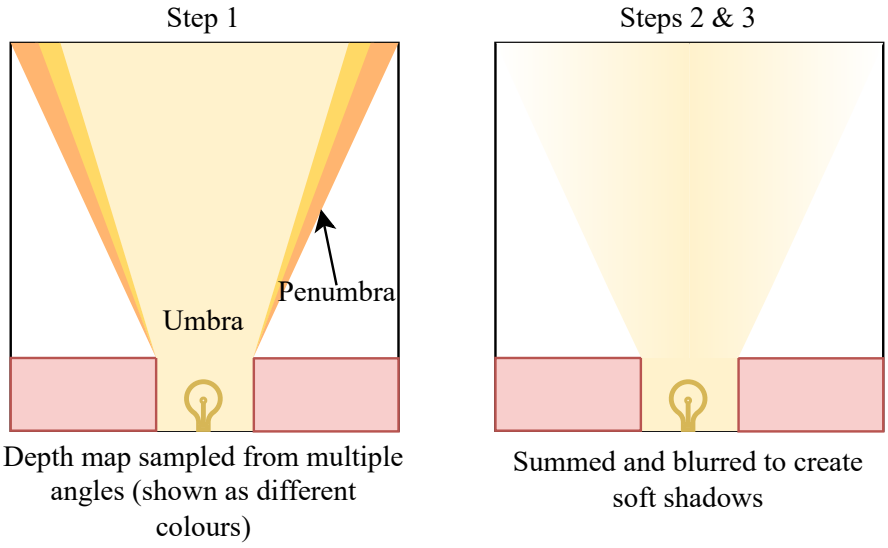


Figure 22: Workflow for 2D soft shadows

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:

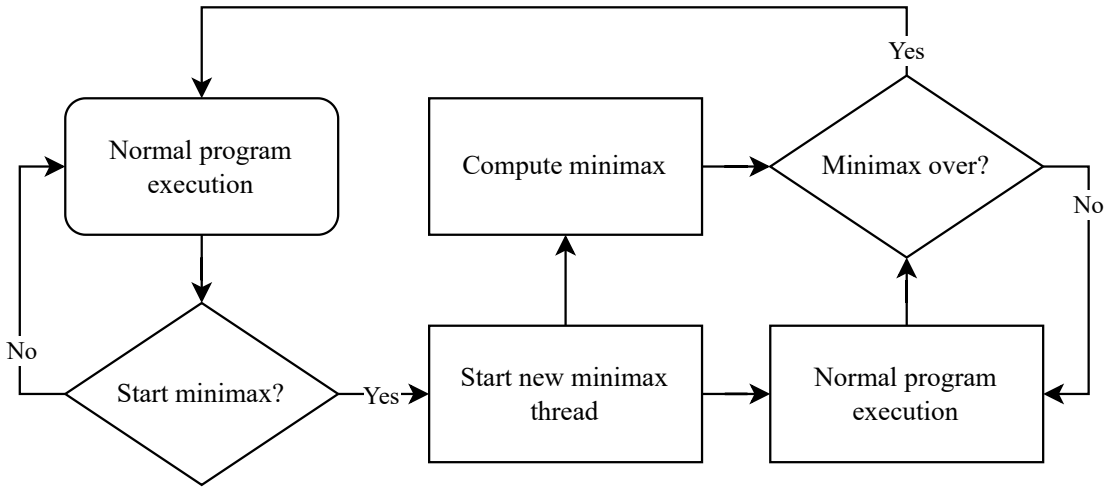


Figure 23: Multi-threading for minimax

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 3: 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[1], 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:

```
CREATE TABLE games(
    id INTEGER PRIMARY KEY,
    winner INTEGER,
    cpu_depth INTEGER,
    time real NOT NULL,
    moves TEXT NOT NULL,
    cpu_enabled INTEGER NOT NULL,
    created_dt TIMESTAMP NOT NULL,
    number_of_moves INTEGER NOT NULL,
    initial_fen_string TEXT NOT NULL,
)
```

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

```
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.

```
INSERT INTO games (
    cpu_enabled,
    cpu_depth,
    winner,
    time,
    number_of_moves,
    moves,
    initial_fen_string,
    created_dt
)
```

[1]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.

```
    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.

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

Pagination

When there are a large number of entries in the table, it would be appropiate 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.

```
    SELECT * FROM
        (SELECT ROW_NUMBER() OVER (
            PARTITION BY attribute1
            ORDER BY attribute2 ASC
        ) AS row_num, * FROM games)
    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 placeholders 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: 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 **??** 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.

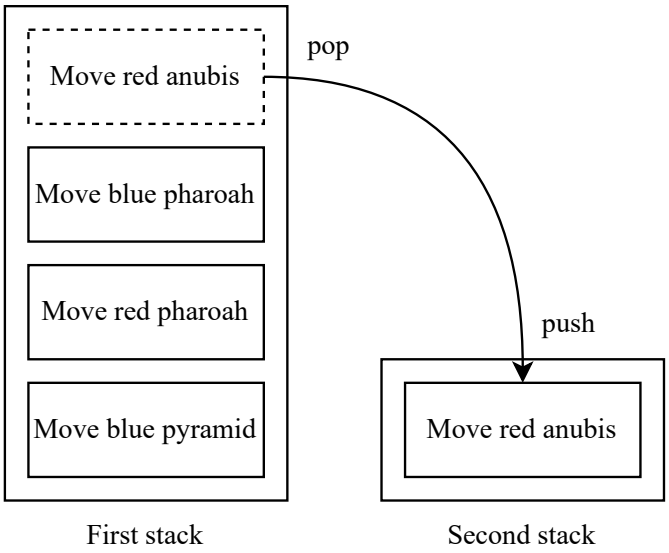


Figure 24: *Move red anubis* is undone and pushed onto the second stack

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 repetive code into a base class, as show 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, 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 4 to initalise 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**
startup	Initialise variables and functions when state set as displayed screen
cleanup	Cleanup any variables and functions when state removed from screen
draw	Draw to display
update	Update any variables for every game tick
handle_resize	Scale GUI when window resized
get_event	Receive pygame events as argument and process them

Table 4: 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 5: 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 `get_event`
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 6: 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 7: 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 8: 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 `alert_listener` is called
alert_listener	Sends event to all subscribed instances
toggle_win	Sends event for win view
toggle_pause	Sends event for pause view
...	...

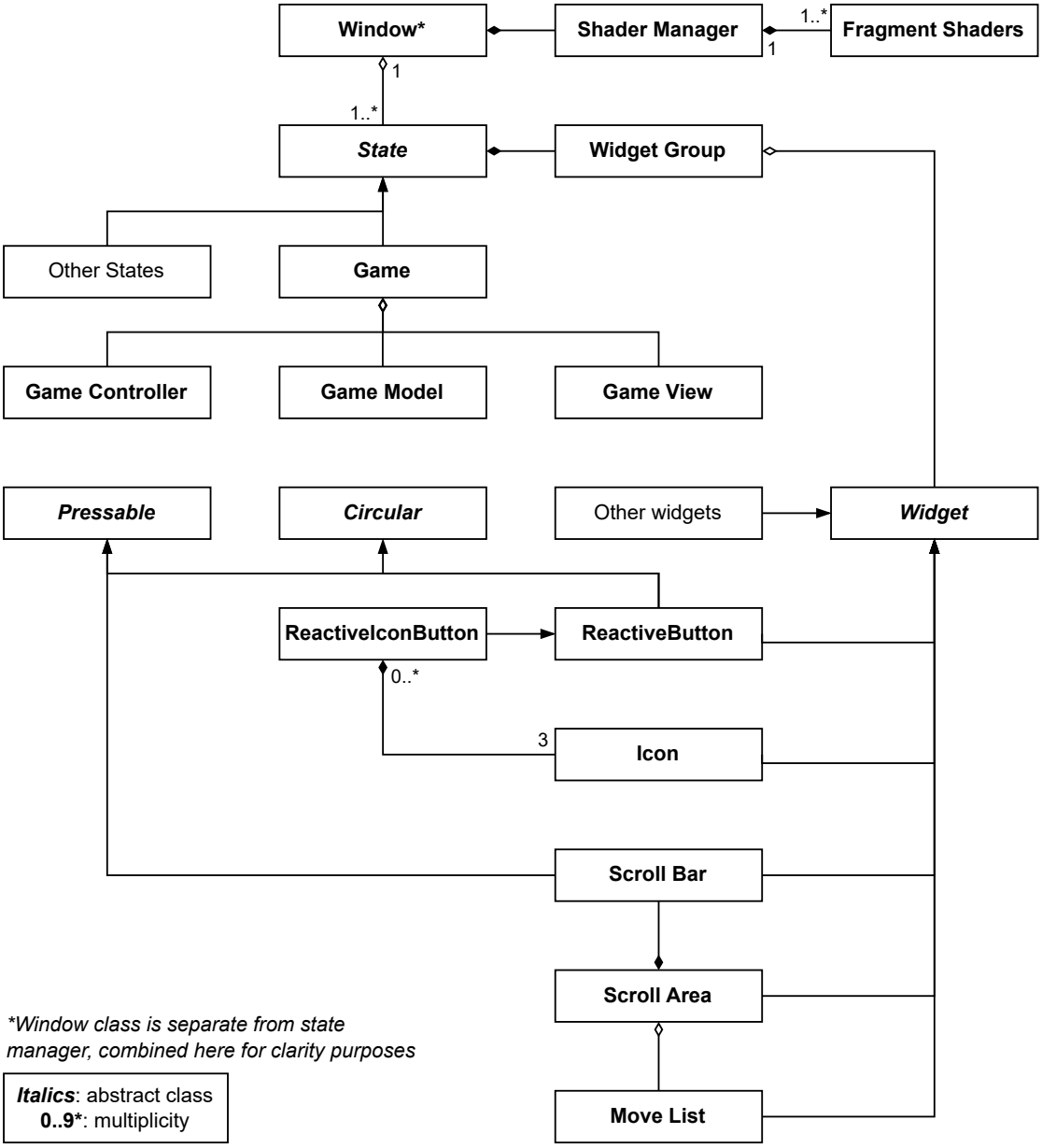
Method Name	Description

Table 9: 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



3 Technical Solution

3.1 File Tree Diagram

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

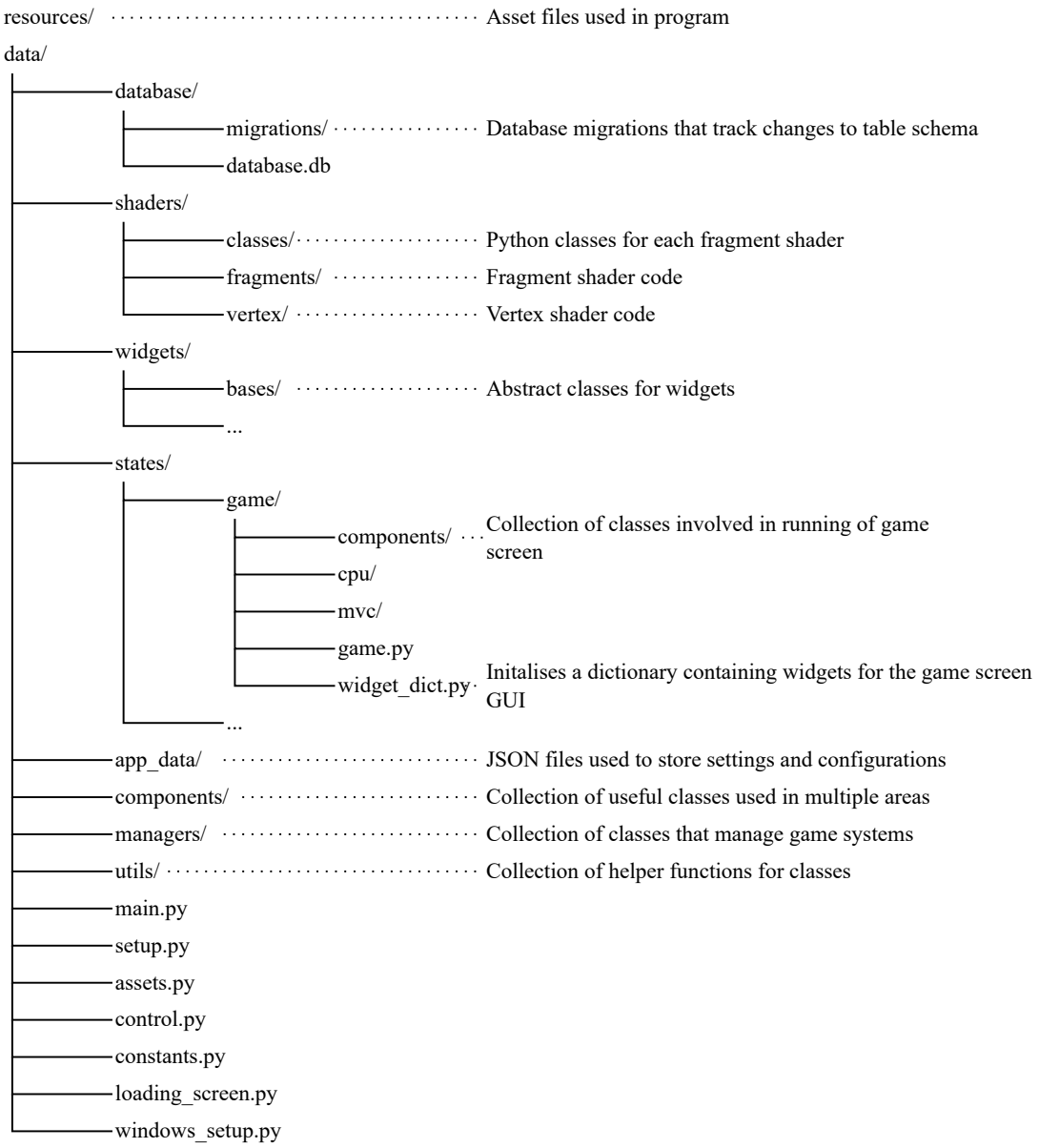


Figure 25: File tree diagram

3.2 Summary of Complexity

- Alpha-beta pruning and transposition table improvements for Minimax
- Shadow mapping and coordinate transformations
- Recursive Depth-First Search tree traversal (Theme)
- Circular doubly-linked list and stack
- Multipass shaders and gaussian blur
- Aggregate and Window SQL functions
- OOP techniques (Widget Bases and Widgets)
- Multithreading (Loading Screen)
- Bitboards
- (File handling and JSON parsing) (Helper functions)
- (Dictionary recursion)
- (Dot product) (Helper functions)

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`

```
from sys import platform
# Initialises Pygame
import data.setup

# Windows OS requires some configuration for Pygame to scale GUI continuously
    while window is being resized
if platform == 'win32':
    import data.windows_setup as win_setup

from data.loading_screen import LoadingScreen

states = [None, None]

def load_states():
    """
    Initialises instances of all screens, executed on another thread with results
    being stored to the main thread by modifying a mutable such as the states list
    """
    from data.control import Control
    from data.states.game.game import Game
    from data.states.menu.menu import Menu
    from data.states.settings.settings import Settings
    from data.states.config.config import Config
    from data.states.browser.browser import Browser
    from data.states.review.review import Review
```

```
    from data.states.editor.editor import Editor

    state_dict = {
        'menu': Menu(),
        'game': Game(),
        'settings': Settings(),
        'config': Config(),
        'browser': Browser(),
        'review': Review(),
        'editor': Editor()
    }

    app = Control()

    states[0] = app
    states[1] = state_dict

loading_screen = LoadingScreen(load_states)

def main():
    """
    Executed by run.py, starts main game loop
    """
    app, state_dict = states

    if platform == 'win32':
        win_setup.set_win_resize_func(app.update_window)

    app.setup_states(state_dict, 'menu')
    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

```
import pygame
import threading
import sys
from pathlib import Path
from data.utils.load_helpers import load_gfx, load_sfx
from data.managers.window import window
from data.managers.audio import audio

FPS = 30
start_ticks = pygame.time.get_ticks()
logo_gfx_path = (Path(__file__).parent / '../resources/graphics/gui/icons/logo/
    logo.png').resolve()
sfx_path_1 = (Path(__file__).parent / '../resources/sfx/loading_screen/
    loading_screen_1.wav').resolve()
sfx_path_2 = (Path(__file__).parent / '../resources/sfx/loading_screen/
    loading_screen_2.wav').resolve()

def easeOutBack(progress):
    """
    Represents a cubic function for easing the logo position.
    Starts quickly and has small overshoot, then ends slowly.

```

```
    Args:
        progress (float): x-value for cubic function ranging from 0-1.

    Returns:
        float: 2.70x^3 + 1.70x^2 + 0x + 1, where x is time elapsed.
    """
    c2 = 1.70158
    c3 = 2.70158

    return c3 * ((progress - 1) ** 3) + c2 * ((progress - 1) ** 2) + 1

class LoadingScreen:
    def __init__(self, target_func):
        """
        Creates new thread, and sets the load_state() function as its target.
        Then starts draw loop for the loading screen.

        Args:
            target_func (Callable): function to be run on thread.
        """
        self._clock = pygame.time.Clock()
        self._thread = threading.Thread(target=target_func)
        self._thread.start()

        self._logo_surface = load_gfx(logo_gfx_path)
        self._logo_surface = pygame.transform.scale(self._logo_surface, (96, 96))
        audio.play_sfx(load_sfx(sfx_path_1))
        audio.play_sfx(load_sfx(sfx_path_2))

        self.run()

    @property
    def logo_position(self):
        duration = 1000
        displacement = 50
        elapsed_ticks = pygame.time.get_ticks() - start_ticks
        progress = min(1, elapsed_ticks / duration)
        center_pos = ((window.screen.size[0] - self._logo_surface.size[0]) / 2, (
    window.screen.size[1] - self._logo_surface.size[1]) / 2)

        return (center_pos[0], center_pos[1] + displacement - displacement *
    easeOutBack(progress))

    @property
    def logo_opacity(self):
        return min(255, (pygame.time.get_ticks() - start_ticks) / 5)

    @property
    def duration_not_over(self):
        return (pygame.time.get_ticks() - start_ticks) < 1500

    def event_loop(self):
        """
        Handles events for the loading screen, no user input is taken except to
    quit the game.
        """
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

    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
    loading screen duration is not reached yet.
92          """
93          while self._thread.is_alive() or self.duration_not_over:
94              self.event_loop()
95              self.draw()
96              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     """
```

```
    Uses the PIL library to break down GIFs into individual frames.

    Args:
        path (str): Directory path to GIF file.

    Yields:
        PIL.Image: Single frame.
    """
    try:
        image = Image.open(path)

        first_frame = image.copy().convert('RGBA')
        yield first_frame
        image.seek(1)

        while True:
            current_frame = image.copy()
            yield current_frame
            image.seek(image.tell() + 1)
    except EOFError:
        pass

def get_perimeter_sample(image_size, number):
    """
    Used for particle drawing class, generates roughly equally distributed points around a rectangular image surface's perimeter.

    Args:
        image_size (tuple[float, float]): Image surface size.
        number (int): Number of points to be generated.

    Returns:
        list[tuple[int, int], ...]: List of random points on perimeter of image surface.
    """
    perimeter = 2 * (image_size[0] + image_size[1])
    # Flatten perimeter to a single number representing the distance from the top-middle of the surface going clockwise, and create a list of equally spaced points
    perimeter_offsets = [(image_size[0] / 2) + (i * perimeter / number) for i in range(0, number)]
    pos_list = []

    for perimeter_offset in perimeter_offsets:
        # For every point, add a random offset
        max_displacement = int(perimeter / (number * 4))
        perimeter_offset += randint(-max_displacement, max_displacement)

        if perimeter_offset > perimeter:
            perimeter_offset -= perimeter

        # Convert 1D distance back into 2D points on image surface perimeter
        if perimeter_offset < image_size[0]:
            pos_list.append((perimeter_offset, 0))
        elif perimeter_offset < image_size[0] + image_size[1]:
            pos_list.append((image_size[0], perimeter_offset - image_size[0]))
        elif perimeter_offset < image_size[0] + image_size[1] + image_size[0]:
            pos_list.append((perimeter_offset - image_size[0] - image_size[1], image_size[1]))
        else:
            pos_list.append((0, perimeter - perimeter_offset))
    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
      clockwise, given a point on the perimeter.
153
```

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

```
211         # Animated background passed in as list of surfaces, calculate_frame_index
    () used to get index of frame to be drawn
212         frame_index = calculate_frame_index(current_time, 0, len(background), fps
    =8)
213         scaled_background = scale_and_cache(background[frame_index], screen.size)
214         screen.blit(scaled_background, (0, 0))
215     else:
216         scaled_background = scale_and_cache(background, screen.size)
217         screen.blit(scaled_background, (0, 0))
218 
219 def get_highlighted_icon(icon):
220     """
221     Used for pressable icons, draws overlay on icon to show as pressed.
222 
223     Args:
224         icon (pygame.Surface): Icon surface.
225 
226     Returns:
227         pygame.Surface: Icon with overlay drawn on top.
228     """
229     icon_copy = icon.copy()
230     overlay = pygame.Surface((icon.get_width(), icon.get_height()), pygame.
    SRCALPHA)
231     overlay.fill((0, 0, 0, 128))
232     icon_copy.blit(overlay, (0, 0))
233     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
       rectangle with semicurclar cap (obround)
21     pygame.draw.rect(gradient_surface, fill_colour, border_rect, border_radius=int
       (size[1] / 2))
22     pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(
       border_width), border_radius=int(size[1] / 2))
23
24     return gradient_surface
25
26 def create_slider_gradient(size, border_width, border_colour):
27     """
28     Draws surface for colour slider, with a full colour gradient as fill colour.
29
30     Args:
31         size (list[int, int]): Image size.
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
    radius of cap (half of image height)
50              distance_from_cutoff = min(abs(first_round_end - i), abs(i -
    second_round_end))
51              draw_height = calculate_gradient_slice_height(distance_from_cutoff,
    gradient_surface.height / 2)
52
53          # Get colour from distance from left side of slider
54          color = pygame.Color(0)
55          color.hsva = (int(360 * i / gradient_surface.width), 100, 100, 100)
56
57          draw_rect = pygame.FRect((0, 0, 1, draw_height - 2 * border_width))
58          draw_rect.center = (i, gradient_y_mid)
59
60          pygame.draw.rect(gradient_surface, color, draw_rect)
61
62      border_rect = pygame.FRect((0, 0, gradient_surface.width, gradient_surface.
    height))
63      pygame.draw.rect(gradient_surface, border_colour, border_rect , width=int(
    border_width), border_radius=int(size[1] / 2))
64
65      return gradient_surface
66
67  def calculate_gradient_slice_height(distance, radius):
68      """
69      Calculate height of vertical slice of semicircular slider cap.
70
71      Args:
72          distance (float): x-distance from center of circle.
73          radius (float): Radius of semicircle.
74
75      Returns:
76          float: Height of vertical slice.
77      """
78      return sqrt(radius ** 2 - distance ** 2) * 2 + 2
79
80  def create_slider_thumb(radius, colour, border_colour, border_width):
81      """
82      Creates surface with bordered circle.
83
84      Args:
85          radius (float): Radius of circle.
86          colour (pygame.Color): Fill colour.
87          border_colour (pygame.Color): Border colour.
88          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,
        width=int(border_width))
95      pygame.draw.circle(thumb_surface, colour, (radius, radius), (radius -
        border_width))
96
97      return thumb_surface
98
99  def create_square_gradient(side_length, colour):
100     """
101     Creates a square gradient for the colour picker widget, gradient transitioning
         between saturation and value.
102     Uses smoothscale to blend between colour values for individual pixels.
103
104     Args:
105         side_length (float): Length of a square side.
106         colour (pygame.Color): Colour with desired hue value.
107
108     Returns:
109         pygame.Surface: Square gradient surface.
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]),
        border_radius=int(size[1] / 2))
146
147     return switch_surface
148
149 def create_text_box(size, border_width, colours):
150     """
```

```
    Creates bordered textbox with shadow, flat, and highlighted vertical regions.

    Args:
        size (list[int, int]): Image size.
        border_width (float): Border width.
        colours (list[pygame.Color, ...]): List of 4 colours, representing border colour, shadow colour, flat colour and highlighted colour.

    Returns:
        pygame.Surface: Textbox surface.
    """
    surface = pygame.Surface(size, pygame.SRCALPHA)

    pygame.draw.rect(surface, colours[0], (0, 0, *size))
    pygame.draw.rect(surface, colours[2], (border_width, border_width, size[0] - 2 * border_width, size[1] - 2 * border_width))
    pygame.draw.rect(surface, colours[3], (border_width, border_width, size[0] - 2 * border_width, border_width))
    pygame.draw.rect(surface, colours[1], (border_width, size[1] - 2 * border_width, size[0] - 2 * border_width, border_width))

    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

```
from data.utils.data_helpers import get_themes, get_user_settings

themes = get_themes()
user_settings = get_user_settings()

def flatten_dictionary_generator(dictionary, parent_key=None):
    """
    Recursive depth-first search to yield all items in a dictionary.

    Args:
        dictionary (dict): Dictionary to be iterated through.
        parent_key (str, optional): Prefix added to every key. Defaults to None.

    Yields:
        dict | tuple[str, str]: Another dictionary or key, value pair.
    """
    for key, value in dictionary.items():
        if parent_key:
            new_key = parent_key + key.capitalize()
        else:
            new_key = key

        if isinstance(value, dict):
            yield from flatten_dictionary(value, new_key).items()
        else:
            yield new_key, value

def flatten_dictionary(dictionary, parent_key=''):
    return dict(flatten_dictionary_generator(dictionary, parent_key))

class ThemeManager:
    def __init__(self):
```

```
        self.__dict__.update(flatten_dictionary(themes['colours']))
        self.__dict__.update(flatten_dictionary(themes['dimensions']))

    def __getitem__(self, arg):
        """
        Override default class's __getitem__ dunder method, to make retrieving an instance attribute nicer with [] notation.

        Args:
            arg (str): Attribute name.

        Raises:
            KeyError: Instance does not have requested attribute.

        Returns:
            str | int: Instance attribute.
        """
        item = self.__dict__.get(arg)

        if item is None:
            raise KeyError('(ThemeManager.__getitem__) Requested theme item not found:', arg)

        return item

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

```
import pygame
from data.utils.board_helpers import coords_to_screen_pos
from data.constants import EMPTY_BB, ShaderType, Colour
from data.managers.animation import animation
from data.managers.window import window
from data.managers.audio import audio
from data.assets import GRAPHICS, SFX
from data.constants import LaserType

type_to_image = {
    LaserType.END: ['laser_end_1', 'laser_end_2'],
    LaserType.STRAIGHT: ['laser_straight_1', 'laser_straight_2'],
    LaserType.CORNER: ['laser_corner_1', 'laser_corner_2']
}

GLOW_SCALE_FACTOR = 1.5

class LaserDraw:
    def __init__(self, board_position, board_size):
        self._board_position = board_position
        self._square_size = board_size[0] / 10
        self._laser_lists = []

    @property
    def firing(self):
        return len(self._laser_lists) > 0

```

```
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
     info.
34              laser_colour (Colour.RED | Colour.BLUE): Active colour of laser.
35          """
36          laser_path = laser_result.laser_path.copy()
37          laser_types = [LaserType.END]
38          # List of angles in degree to rotate the laser image surface when drawn
39          laser_rotation = [laser_path[0][1]]
40          laser_lights = []
41
42          # Iterates through every square laser passes through
43          for i in range(1, len(laser_path)):
44              previous_direction = laser_path[i-1][1]
45              current_coords, current_direction = laser_path[i]
46
47              if current_direction == previous_direction:
48                  laser_types.append(LaserType.STRAIGHT)
49                  laser_rotation.append(current_direction)
50              elif current_direction == previous_direction.get_clockwise():
51                  laser_types.append(LaserType.CORNER)
52                  laser_rotation.append(current_direction)
53              elif current_direction == previous_direction.get_anticlockwise():
54                  laser_types.append(LaserType.CORNER)
55                  laser_rotation.append(current_direction.get_anticlockwise())
56
57              # Adds a shader ray effect on the first and last square of the laser
     trajectory
58              if i in [1, len(laser_path) - 1]:
59                  abs_position = coords_to_screen_pos(current_coords, self.
     _board_position, self._square_size)
60                  laser_lights.append([
61                      (abs_position[0] / window.size[0], abs_position[1] / window.
     size[1]),
62                      0.5,
63                      (0, 0, 255) if laser_colour == Colour.BLUE else (255, 0, 0),
64                  ])
65
66          # Sets end laser draw type if laser hits a piece
67          if laser_result.hit_square_bitboard != EMPTY_BB:
68              laser_types[-1] = LaserType.END
69              laser_path[-1] = (laser_path[-1][0], laser_path[-2][1].get_opposite())
70              laser_rotation[-1] = laser_path[-2][1].get_opposite()
71
72              audio.play_sfx(SFX['piece_destroy'])
73
74          laser_path = [(coords, rotation, type) for (coords, dir), rotation, type
     in zip(laser_path, laser_rotation, laser_types)]
75          self._laser_lists.append((laser_path, laser_colour))
76
77          window.clear_effect(ShaderType.RAYS)
78          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
    , self._square_size)
108
109             image = GRAPHICS[type_to_image[type][laser_colour]]
110             rotated_image = pygame.transform.rotate(image, rotation.to_angle())
111             scaled_image = pygame.transform.scale(rotated_image, (self.
    _square_size + 1, self._square_size + 1)) # +1 to prevent rounding creating
    black lines
112             laser_list.append((scaled_image, (square_x, square_y)))
113
114             # Scales up the laser image surface as a glow surface
115             scaled_glow = pygame.transform.scale(rotated_image, (self._square_size
     * GLOW_SCALE_FACTOR, self._square_size * GLOW_SCALE_FACTOR))
116             offset = self._square_size * ((GLOW_SCALE_FACTOR - 1) / 2)
117             glow_list.append((scaled_glow, (square_x - offset, square_y - offset))
    )
118
119         # Scaled glow surfaces drawn on top with the RGB_ADD blend mode
120         if glow:
121             screen.fblits(glow_list, pygame.BLEND_RGB_ADD)
122
123         screen.blits(laser_list)
124
125     def draw(self, screen):
126         """
127         Draws all lasers on the screen.
128
129         Args:
130             screen (pygame.Surface): The screen to draw on.
131         """
132         for laser_list in self._laser_lists:
133             self.draw_laser(screen, laser_list)
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.
```

```
            board_size (tuple[int, int]): The new size of the board.
        """
        self._board_position = board_position
        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

```
import pygame
from random import randint
from data.utils.asset_helpers import get_perimeter_sample, get_vector,
    get_angle_between_vectors, get_next_corner
from data.states.game.components.piece_sprite import PieceSprite

class ParticlesDraw:
    def __init__(self, gravity=0.2, rotation=180, shrink=0.5, opacity=150):
        self._particles = []
        self._glow_particles = []

        self._gravity = gravity
        self._rotation = rotation
        self._shrink = shrink
        self._opacity = opacity

    def fragment_image(self, image, number):
        image_size = image.get_rect().size
        """
        1. Takes an image surface and samples random points on the perimeter.
        2. Iterates through points, and depending on the nature of two consecutive
     points, finds a corner between them.
        3. Draws a polygon with the points as the vertices to mask out the area
    not in the fragment.

        Args:
            image (pygame.Surface): Image to fragment.
            number (int): The number of fragments to create.

        Returns:
            list[pygame.Surface]: List of image surfaces with fragment of original
     surface drawn on top.
        """
        center = image.get_rect().center
        points_list = get_perimeter_sample(image_size, number)
        fragment_list = []

        points_list.append(points_list[0])

        # Iterate through points_list, using the current point and the next one
        for i in range(len(points_list) - 1):
            vertex_1 = points_list[i]
            vertex_2 = points_list[i + 1]
            vector_1 = get_vector(center, vertex_1)
            vector_2 = get_vector(center, vertex_2)
            angle = get_angle_between_vectors(vector_1, vector_2)

            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
    on the same side
51                  corners_to_draw = 4
52
53              elif abs(vertex_1[0] - vertex_2[0]) == image_size[0] or abs(vertex_1
    [1] - vertex_2[1]) == image_size[1]: # Points on opposite sides
54                  corners_to_draw = 2
55
56              elif angle < 180: # Points on adjacent sides
57                  corners_to_draw = 3
58
59              else:
60                  corners_to_draw = 1
61
62              corners_list = []
63              for j in range(corners_to_draw):
64                  if len(corners_list) == 0:
65                      corners_list.append(get_next_corner(vertex_2, image_size))
66                  else:
67                      corners_list.append(get_next_corner(corners_list[-1],
    image_size))
68
69              pygame.draw.polygon(cropped_image, (0, 0, 0, 0), (center, vertex_2, *
    corners_list, vertex_1))
70
71              fragment_list.append(cropped_image)
72
73          return fragment_list
74
75      def add_captured_piece(self, piece, colour, rotation, position, size):
76          """
77          Adds a captured piece to fragment into particles.
78
79          Args:
80              piece (Piece): The piece type.
81              colour (Colour.BLUE | Colour.RED): The active colour of the piece.
82              rotation (int): The rotation of the piece.
83              position (tuple[int, int]): The position where particles originate
    from.
84              size (tuple[int, int]): The size of the piece.
85          """
86          piece_sprite = PieceSprite(piece, colour, rotation)
87          piece_sprite.set_geometry((0, 0), size)
88          piece_sprite.set_image()
89
90          particles = self.fragment_image(piece_sprite.image, 5)
91
92          for particle in particles:
93              self.add_particle(particle, position)
94
95      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.BLUE | Colour.RED): The active colour of the sparks.
```

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

```
156                 end_radius = (1 - self._shrink) * particle[1][0]
157                 target_radius = particle[1][0] - particle[4] * (particle[1][0] -
    end_radius)
158 
159                 updated_image = pygame.Surface((target_radius * 2, target_radius *
     2), pygame.SRCALPHA)
160                 pygame.draw.circle(updated_image, particle[1][1], (target_radius,
    target_radius), target_radius)
161 
162             # Update opacity
163             alpha = 255 - particle[4] * (255 - self._opacity)
164 
165             updated_image.fill((255, 255, 255, alpha), None, pygame.
    BLEND_RGBA_MULT)
166 
167             particle[0] = updated_image
168 
169     def draw(self, screen):
170         """
171         Draws the particles, indexing the surface and position attributes for each
     particle.
172 
173         Args:
174             screen (pygame.Surface): The screen to draw on.
175         """
176         screen.blits([
177             (particle[0], particle[2]) for particle in self._particles
178         ])
```

3.4.3 Widget Bases

Widget bases are the base classes for 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 overriden. 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.

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,
       remains constant.
```

```
16          parent (_Widget, optional): The parent widget position and size is relative to.
17
18          Relative to current surface:
19          relative_position (tuple[float, float]): The position of the widget relative to its surface.
20          relative_size (tuple[float, float]): The scale of the widget relative to its surface.
21
22          Remains constant, relative to initial screen size:
23          relative_font_size (float, optional): The relative font size of the widget.
24          relative_margin (float): The relative margin of the widget.
25          relative_border_width (float): The relative border width of the widget.
26          relative_border_radius (float): The relative border radius of the widget.
27
28          anchor_x (str): The horizontal anchor direction ('left', 'right', 'center').
29          anchor_y (str): The vertical anchor direction ('top', 'bottom', 'center').
30          fixed_position (tuple[int, int], optional): The fixed position of the widget in pixels.
31          border_colour (pygame.Color): The border color of the widget.
32          text_colour (pygame.Color): The text color of the widget.
33          fill_colour (pygame.Color): The fill color of the widget.
34          font (pygame.freetype.Font): The font used for the widget.
35          """
36          super().__init__()
37
38          for required_kwarg in REQUIRED_KWARGS:
39              if required_kwarg not in kwargs:
40                  raise KeyError(f'(_Widget.__init__) Required keyword "{required_kwarg}" not in base kwargs')
41
42          self._surface = None # Set in WidgetGroup, as needs to be reassigned every frame
43          self._raw_surface_size = DEFAULT_SURFACE_SIZE
44
45          self._parent = kwargs.get('parent')
46
47          self._relative_font_size = None # Set in subclass
48
49          self._relative_position = kwargs.get('relative_position')
50          self._relative_margin = theme['margin'] / self._raw_surface_size[1]
51          self._relative_border_width = theme['borderWidth'] / self._raw_surface_size[1]
52          self._relative_border_radius = theme['borderRadius'] / self._raw_surface_size[1]
53
54          self._border_colour = pygame.Color(theme['borderPrimary'])
55          self._text_colour = pygame.Color(theme['textPrimary'])
56          self._fill_colour = pygame.Color(theme['fillPrimary'])
57          self._font = DEFAULT_FONT
58
59          self._anchor_x = kwargs.get('anchor_x') or 'left'
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.
    surface_size[0]) / self.surface_size[1], (kwargs.get('relative_size')[1] *
    self.surface_size[0]) / self.surface_size[1])
70                  case 'both':
71                      self._relative_size = ((kwargs.get('relative_size')[0] * self.
    surface_size[0]) / self.surface_size[1], kwargs.get('relative_size')[1])
72                  case _:
73                      raise ValueError('(_Widget.__init__) Unknown scale mode:',
    scale_mode)
74          else:
75              self._relative_size = (1, 1)
76
77          if 'margin' in kwargs:
78              self._relative_margin = kwargs.get('margin') / self._raw_surface_size
    [1]
79
80              if (self._relative_margin * 2) > min(self._relative_size[0], self.
    _relative_size[1]):
81                  raise ValueError('(_Widget.__init__) Margin larger than specified
    size!')
82
83          if 'border_width' in kwargs:
84              self._relative_border_width = kwargs.get('border_width') / self.
    _raw_surface_size[1]
85
86          if 'border_radius' in kwargs:
87              self._relative_border_radius = kwargs.get('border_radius') / self.
    _raw_surface_size[1]
88
89          if 'border_colour' in kwargs:
90              self._border_colour = pygame.Color(kwargs.get('border_colour'))
91
92          if 'fill_colour' in kwargs:
93              self._fill_colour = pygame.Color(kwargs.get('fill_colour'))
94
95          if 'text_colour' in kwargs:
96              self._text_colour = pygame.Color(kwargs.get('text_colour'))
97
98          if 'font' in kwargs:
99              self._font = kwargs.get('font')
100
101     @property
102     def surface_size(self):
103         """
104         Gets the size of the surface widget is drawn on.
105         Can be either the window size, or another widget size if assigned to a
    parent.
106
107         Returns:
108             tuple[int, int]: The size of the surface.
109         """
110         if self._parent:
111             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
```

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

```
    def set_geometry(self):
        """
        Sets the position and size of the widget.
        """
        self.rect = self.image.get_rect()

        if self._anchor_x == 'left':
            if self._anchor_y == 'top':
                self.rect.topleft = self.position
            elif self._anchor_y == 'bottom':
                self.rect.topleft = self.position
            elif self._anchor_y == 'center':
                self.rect.topleft = self.position
        elif self._anchor_x == 'right':
            if self._anchor_y == 'top':
                self.rect.topleft = self.position
            elif self._anchor_y == 'bottom':
                self.rect.topleft = self.position
            elif self._anchor_y == 'center':
                self.rect.topleft = self.position
        elif self._anchor_x == 'center':
            if self._anchor_y == 'top':
                self.rect.topleft = self.position
            elif self._anchor_y == 'bottom':
                self.rect.topleft = self.position
            elif self._anchor_y == 'center':
                self.rect.topleft = self.position

    def set_surface_size(self, new_surface_size):
        """
        Sets the new size of the surface widget is drawn on.

        Args:
            new_surface_size (tuple[int, int]): The new size of the surface.
        """
        self._raw_surface_size = new_surface_size

    def process_event(self, event):
        """
        Abstract method to handle events.

        Args:
            event (pygame.event.Event): The event to process.
        """
        raise NotImplementedError
```

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

circular.py

```
from data.components.circular_linked_list import CircularLinkedList

class _Circular:
    def __init__(self, items_dict, **kwargs):
        # The key, value pairs are stored within a dictionary, while the keys to access them are stored within circular linked list.
        self._items_dict = items_dict
        self._keys_list = CircularLinkedList(list(items_dict.keys()))

    @property
    def current_key(self):
        """
```

```
        Gets the current head node of the linked list, and returns a key stored as the node data.
        Returns:
            Data of linked list head.
        """
        return self._keys_list.get_head().data

    @property
    def current_item(self):
        """
        Gets the value in self._items_dict with the key being self.current_key.

        Returns:
            Value stored with key being current head of linked list.
        """
        return self._items_dict[self.current_key]

    def set_next_item(self):
        """
        Sets the next item in as the current item.
        """
        self._keys_list.shift_head()

    def set_previous_item(self):
        """
        Sets the previous item as the current item.
        """
        self._keys_list.unshift_head()

    def set_to_key(self, key):
        """
        Sets the current item to the specified key.

        Args:
            key: The key to set as the current item.

        Raises:
            ValueError: If no nodes within the circular linked list contains the key as its data.
        """
        if self._keys_list.data_in_list(key) is False:
            raise ValueError('(_Circular.set_to_key) Key not found:', key)

        for _ in range(len(self._items_dict)):
            if self.current_key == key:
                self.set_image()
                self.set_geometry()
                return

            self.set_next_item()
```

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

circular_linked_list.py

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.previous = None

```

```
class CircularLinkedList:
    def __init__(self, list_to_convert=None):
        """
        Initializes a CircularLinkedList object.

        Args:
            list_to_convert (list, optional): Creates a linked list from existing items. Defaults to None.
        """
        self._head = None

        if list_to_convert:
            for item in list_to_convert:
                self.insert_at_end(item)

    def __str__(self):
        """
        Returns a string representation of the circular linked list.

        Returns:
            str: Linked list formatted as string.
        """
        if self._head is None:
            return '| empty |'

        characters = '| -> '
        current_node = self._head
        while True:
            characters += str(current_node.data) + ' -> '
            current_node = current_node.next

            if current_node == self._head:
                characters += '|'
                return characters

    def insert_at_beginning(self, data):
        """
        Inserts a node at the beginning of the circular linked list.

        Args:
            data: The data to insert.
        """
        new_node = Node(data)

        if self._head is None:
            self._head = new_node
            new_node.next = self._head
            new_node.previous = self._head
        else:
            new_node.next = self._head
            new_node.previous = self._head.previous
            self._head.previous.next = new_node
            self._head.previous = new_node

            self._head = new_node

    def insert_at_end(self, data):
        """
        Inserts a node at the end of the circular linked list.

        Args:
            data: The data to insert.
```

```
        """
        new_node = Node(data)

        if self._head is None:
            self._head = new_node
            new_node.next = self._head
            new_node.previous = self._head
        else:
            new_node.next = self._head
            new_node.previous = self._head.previous
            self._head.previous.next = new_node
            self._head.previous = new_node

    def insert_at_index(self, data, index):
        """
        Inserts a node at a specific index in the circular linked list.
        The head node is taken as index 0.

        Args:
            data: The data to insert.
            index (int): The index to insert the data at.

        Raises:
            ValueError: Index is out of range.
        """
        if index < 0:
            raise ValueError('Invalid index! (CircularLinkedList.insert_at_index)'
    )

        if index == 0 or self._head is None:
            self.insert_at_beginning(data)
        else:
            new_node = Node(data)
            current_node = self._head
            count = 0

            while count < index - 1 and current_node.next != self._head:
                current_node = current_node.next
                count += 1

            if count == (index - 1):
                new_node.next = current_node.next
                new_node.previous = current_node
                current_node.next = new_node
            else:
                raise ValueError('Index out of range! (CircularLinkedList.
    insert_at_index)')

    def delete(self, data):
        """
        Deletes a node with the specified data from the circular linked list.

        Args:
            data: The data to delete.

        Raises:
            ValueError: No nodes in the list contain the specified data.
        """
        if self._head is None:
            return

        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:

- One
- Two
- Three
- Four
- Five
- Six

3.5 Game

3.5.1 Database

3.6 Shaders