NEA

Toby Mok

Contents

1 Analysis **2**
- 1.1 Background . . . 2
  - 1.1.1 Game Description . . . 2
  - 1.1.2 Current Solutions . . . 3
  - 1.1.3 Client Interview . . . 4
- 1.2 Objectives . . . 5
  - 1.2.1 Client Objectives . . . 5
  - 1.2.2 Other User Considerations . . . 6
- 1.3 Research . . . 6
  - 1.3.1 Board Representation . . . 7
  - 1.3.2 CPU techniques . . . 8
  - 1.3.3 GUI framework . . . 8
- 1.4 Proposed Solution . . . 9
  - 1.4.1 Language . . . 9
  - 1.4.2 Development Environment . . . 10
  - 1.4.3 Source Control . . . 10
  - 1.4.4 Techniques . . . 11
- 1.5 Critical Path Design . . . 11

2 Design **13**
- 2.1 System Architecture . . . 13
  - 2.1.1 Main Menu . . . 14
  - 2.1.2 Settings . . . 14
  - 2.1.3 Past Games Browser . . . 16
  - 2.1.4 Config . . . 17
  - 2.1.5 Game . . . 18
  - 2.1.6 Board Editor . . . 19
- 2.2 Algorithms and Techniques . . . 20
  - 2.2.1 Minimax . . . 20
  - 2.2.2 Minimax improvements . . . 21
  - 2.2.3 Board Representation . . . 25
  - 2.2.4 Evaluation Function . . . 29
  - 2.2.5 Shadow Mapping . . . 32
  - 2.2.6 Multithreading . . . 34
- 2.3 Classes . . . 35

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/`.

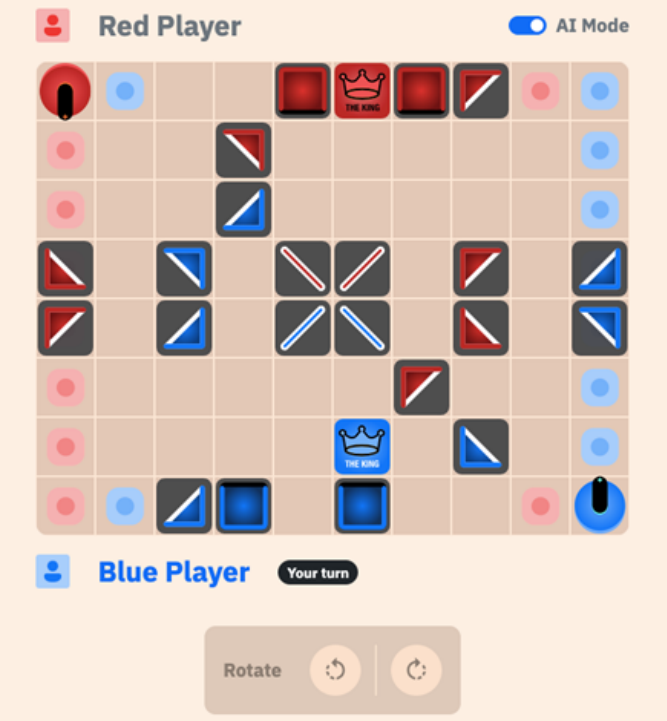


Figure 1: Online implementation on laser-chess.com

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.

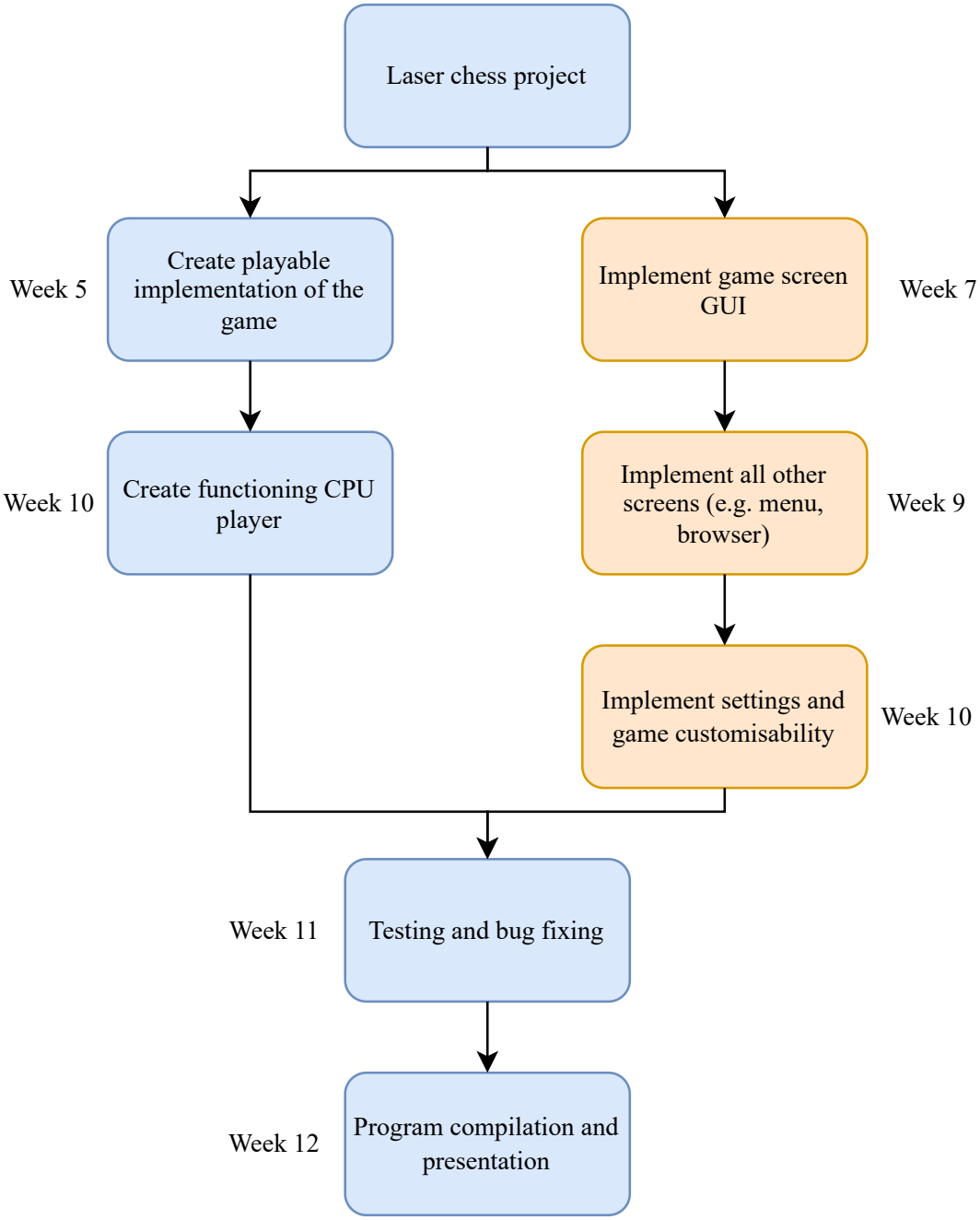
GUI framework

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 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:

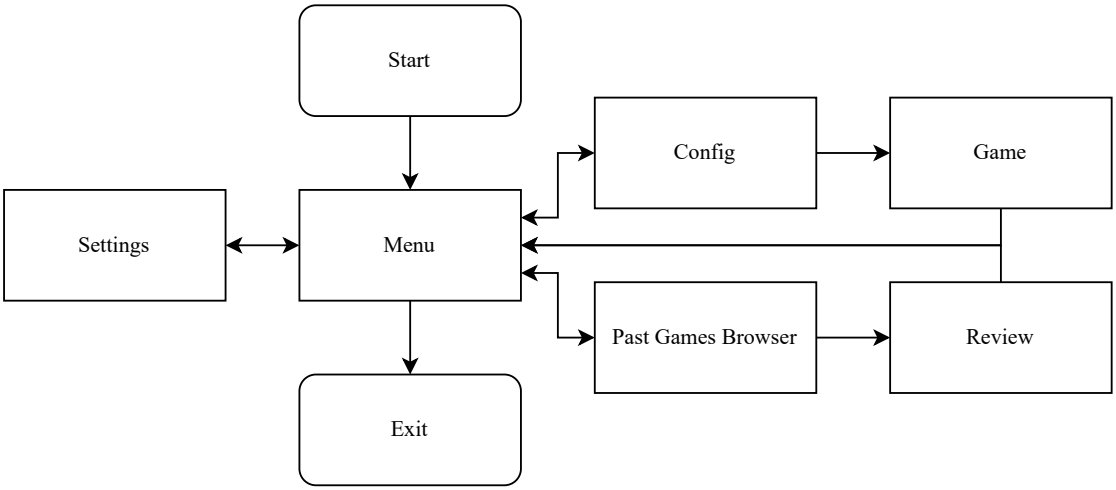


Figure 2: Flowchart for Program Overview

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

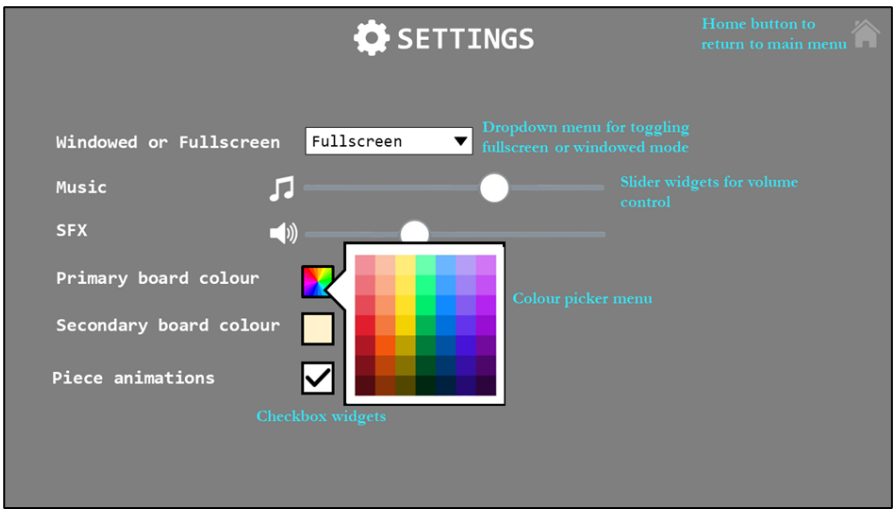


Figure 4: Settings screen prototype

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.

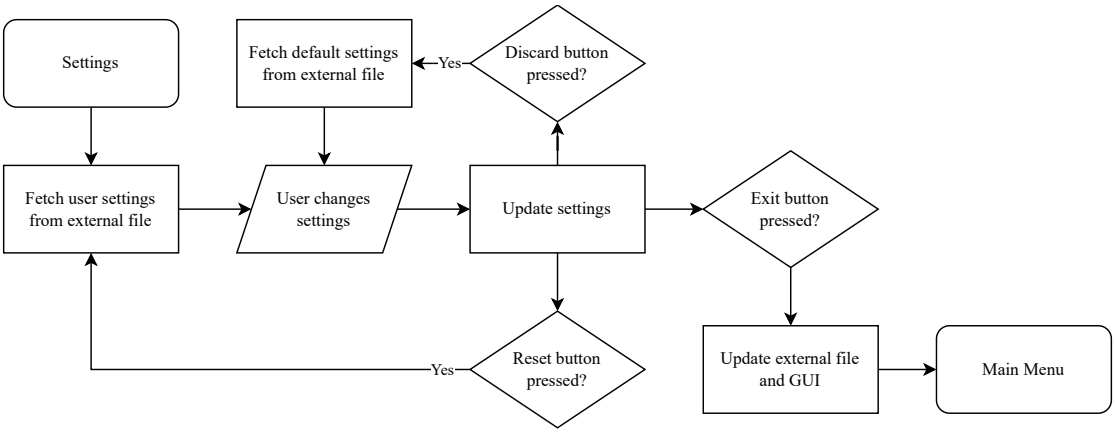


Figure 5: Flowchart for Settings

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.

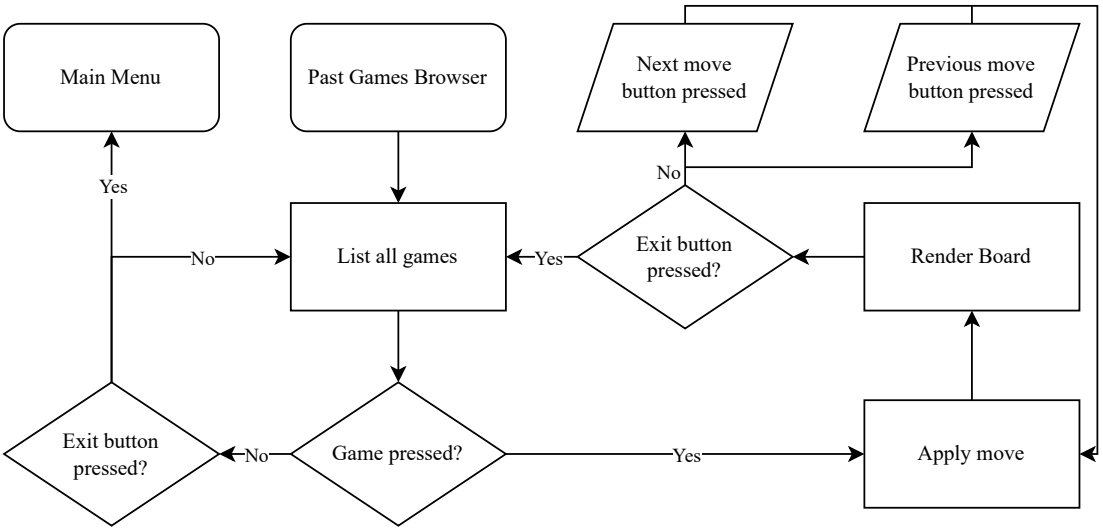


Figure 7: Flowchart for Browser

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.

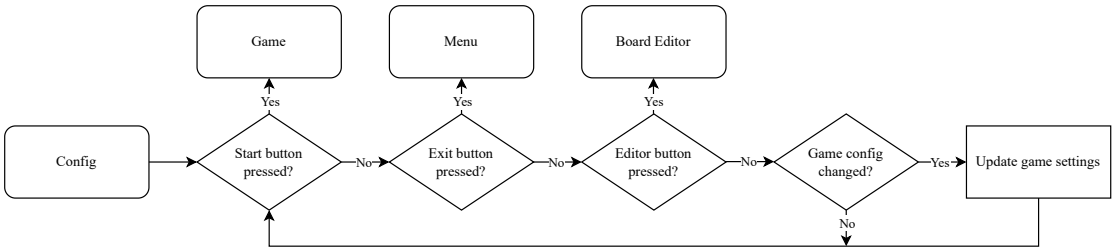


Figure 9: Flowchart for Config

2.1.5 Game

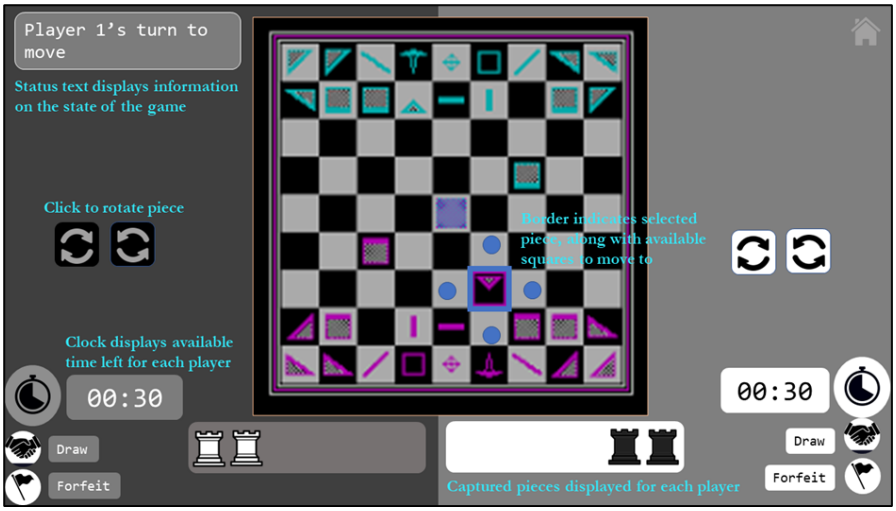


Figure 10: Game screen prototype

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.

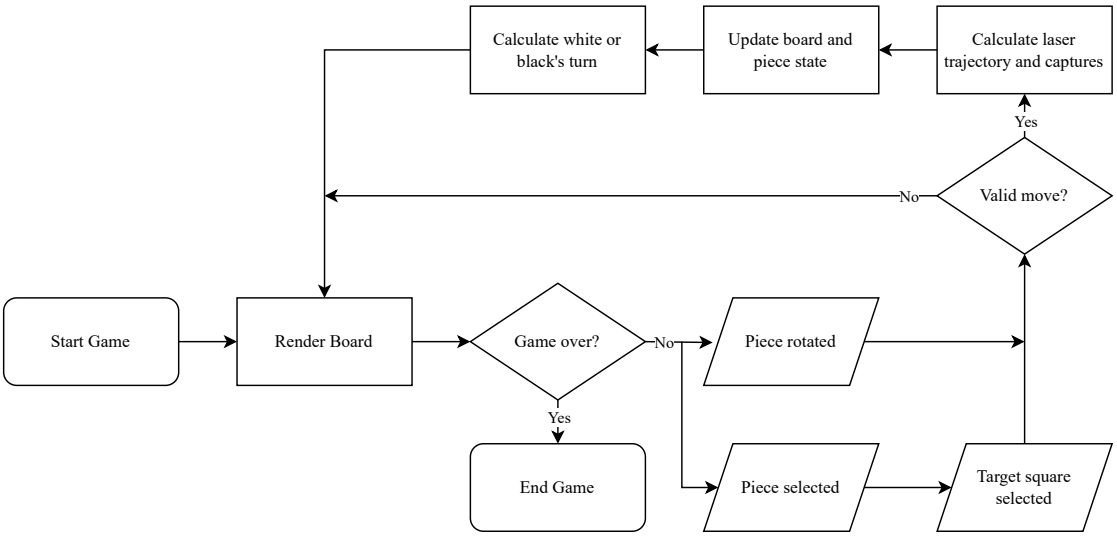


Figure 11: Flowchart for Game

2.1.6 Board Editor

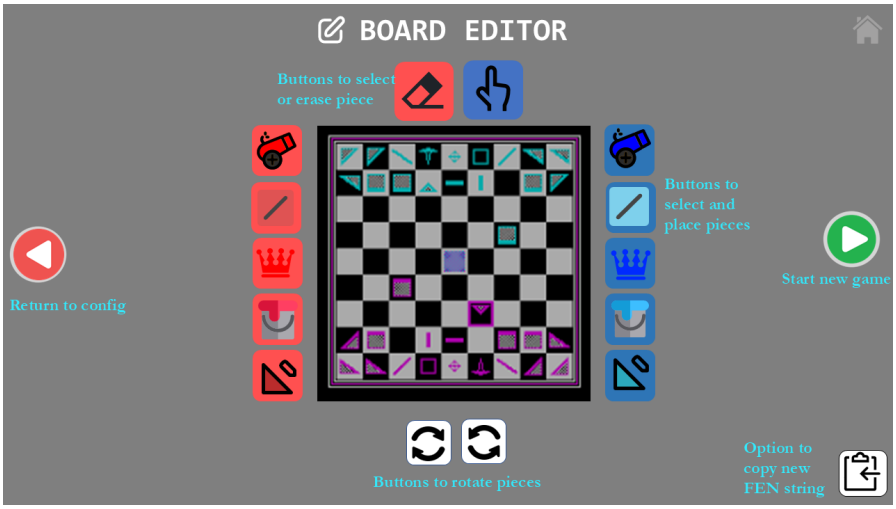


Figure 12: Editor screen prototype

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).

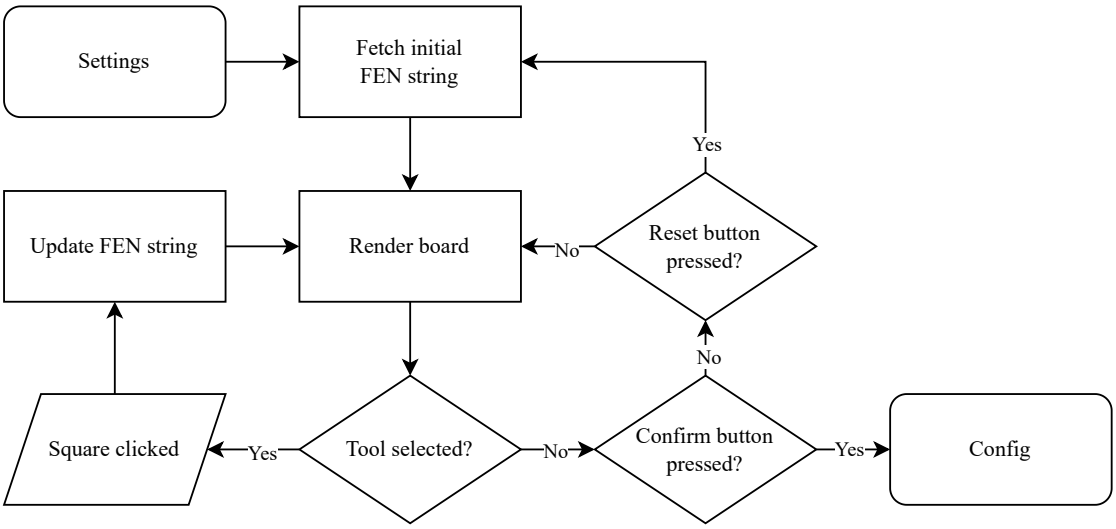


Figure 13: Flowchart for board editor

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 recursive depth-first search, alternating between minimizing and maximising the player's advantage in each recursive call.

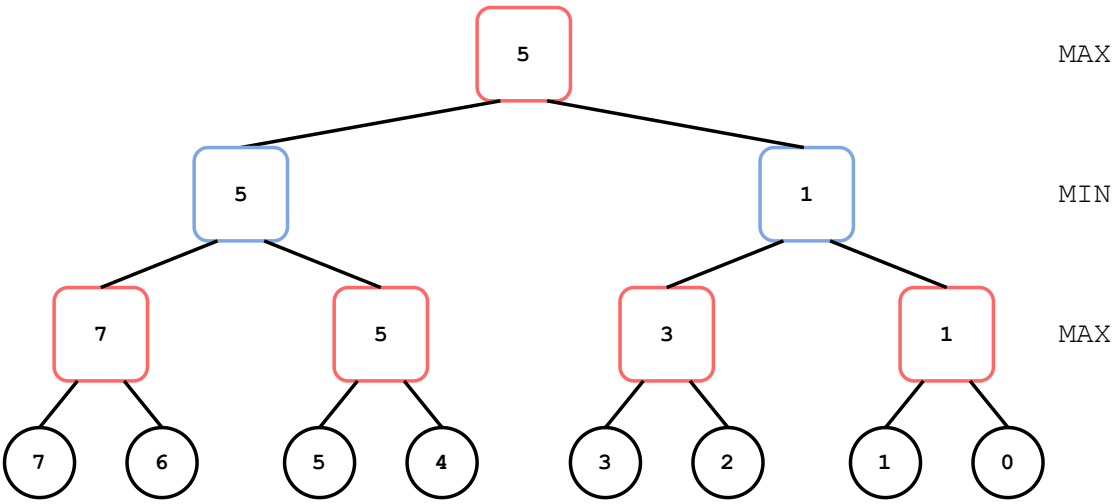


Figure 14: Example minimax tree

For the example minimax tree show in Figure 14, starting from the bottom leaf node evalua-

tions, 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:

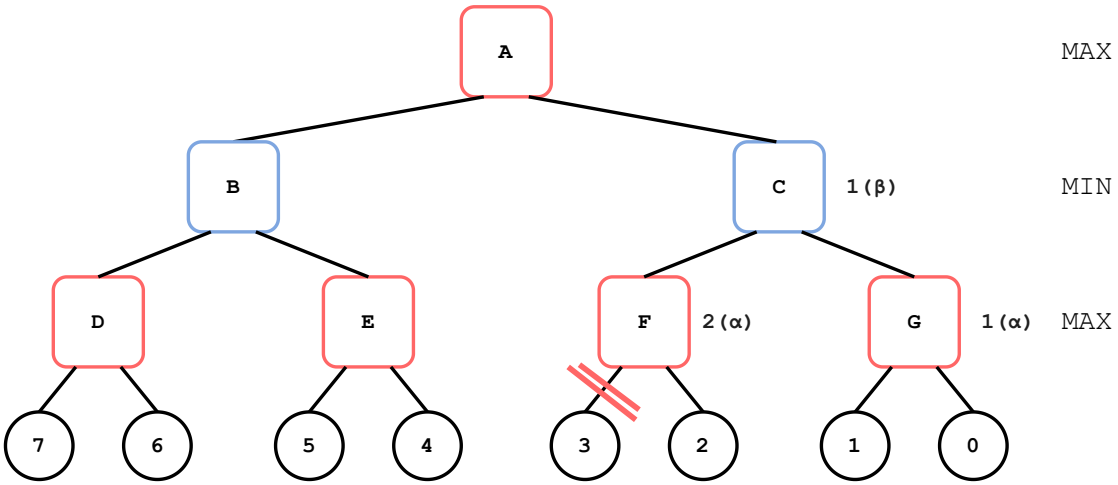


Figure 15: Example minimax tree with alpha-beta pruning

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:

all_pieces_bitboard = white_pharoah_bitboard | black_pharoah_bitboard | white_scarab_bitboard
...

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

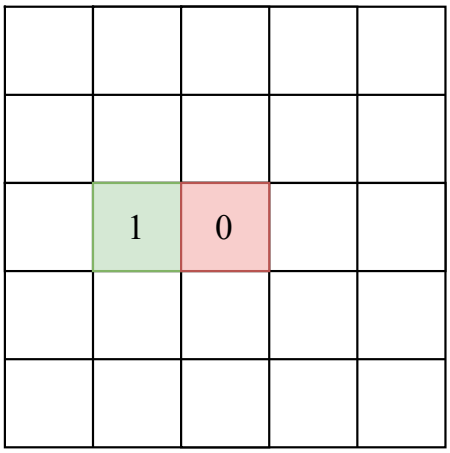


Figure 17: shifted_bitboard = piece_bitboard >> 1

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:

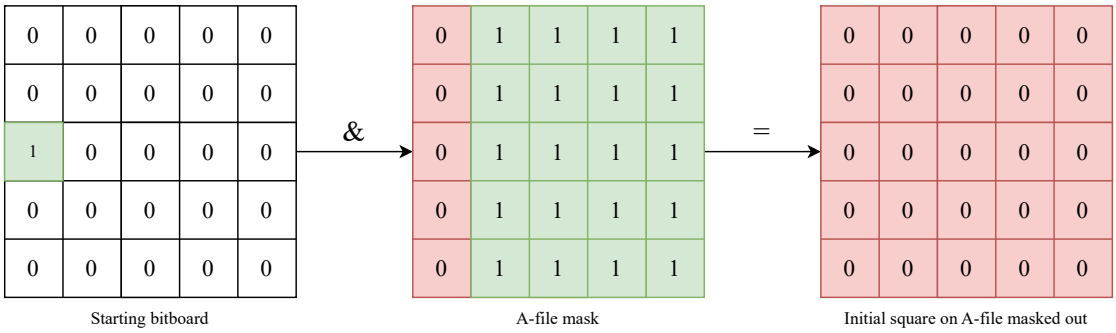


Figure 18: A-file mask example

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:

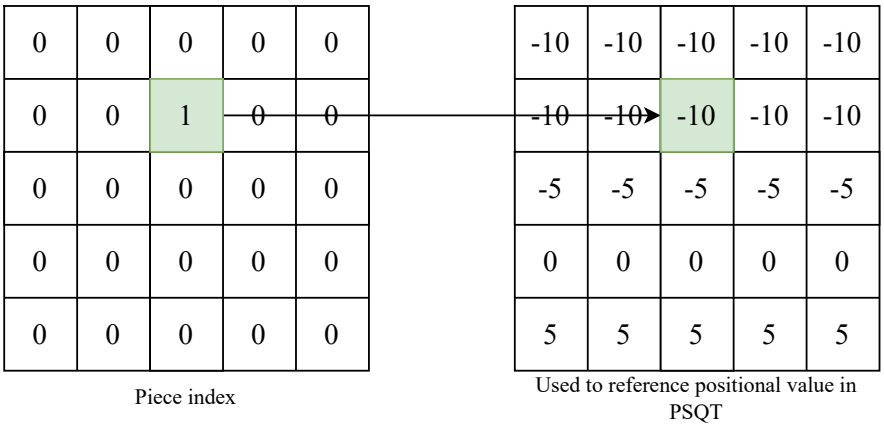


Figure 19: PSQT showing the bonus position value gained for the square occupied by a pharoah

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

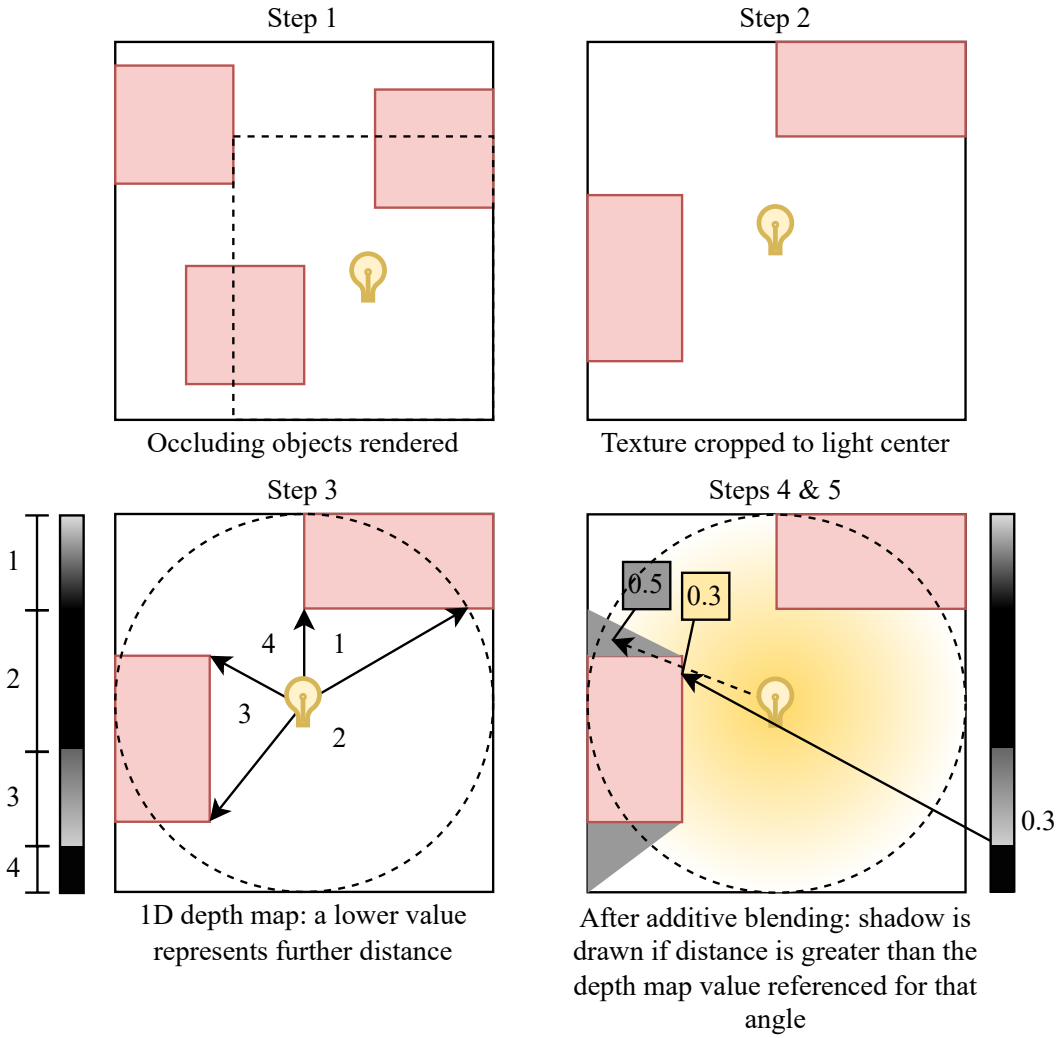


Figure 21: Workflow for 2D shadow mapping

However, 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

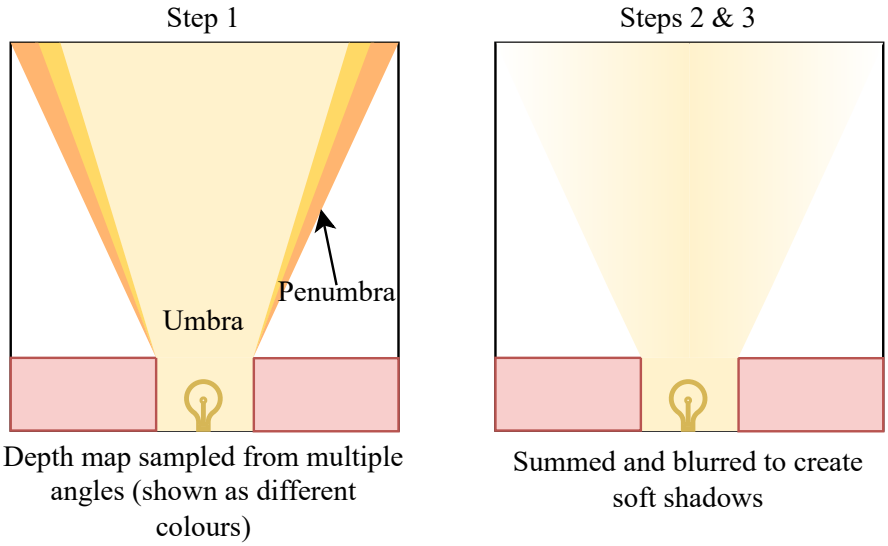


Figure 22: Workflow for 2D soft shadows

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:

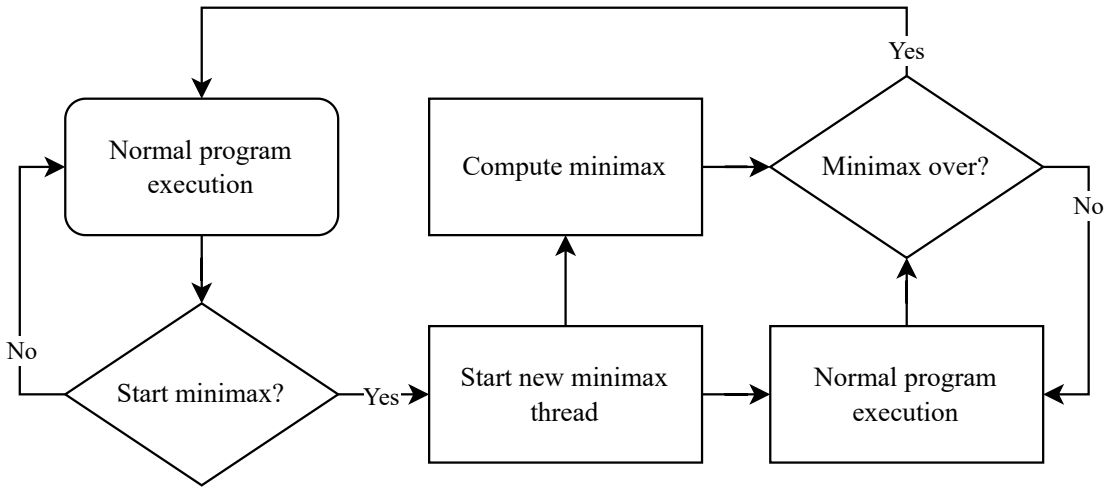


Figure 23: Multi-threading for minimax

2.3 Classes