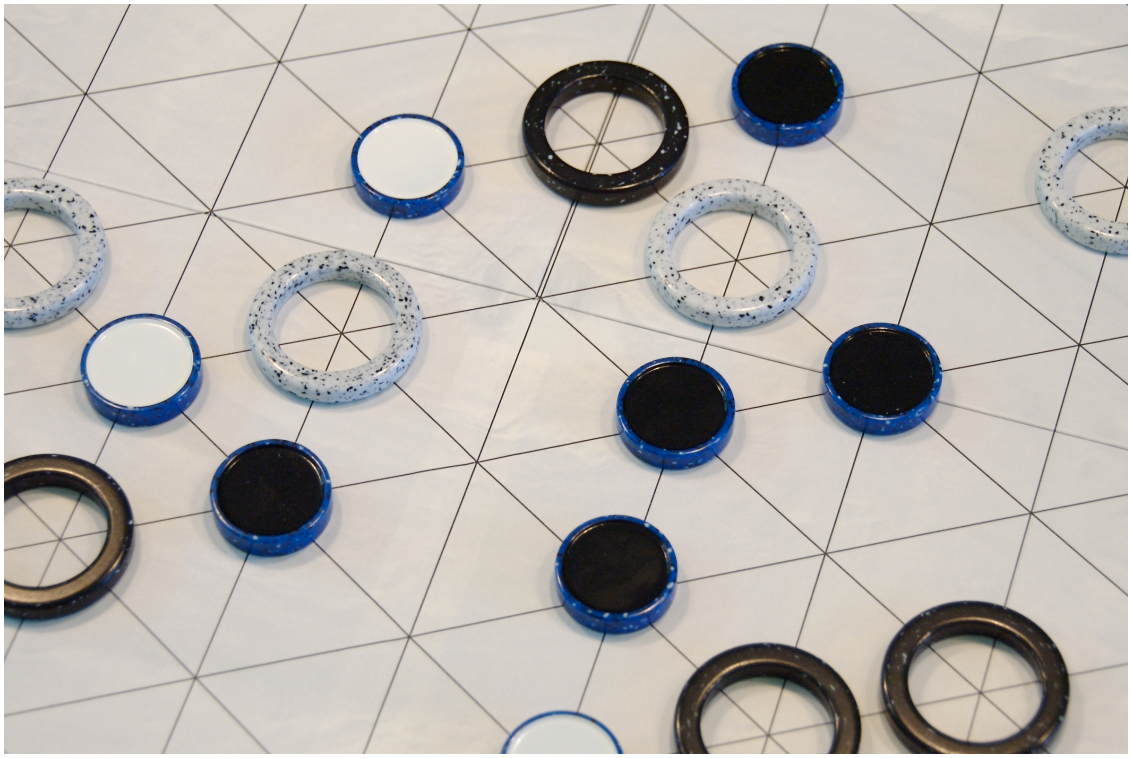
NEA Report

YINSH board game with minimax computer player

Cheung Yat Hei Eden



Contents

1 Analysis

1.1 Background

1.1.1 The Game

YINSH is a board game created by Kris Burm in 2003[1]. It is played between two players based on moving rings and flipping markers. It is currently ranked 6th in the abstract strategy category on Boardgamegeek[2]. YINSH is played between two players on a triangular grid in a hexagon with its corners cut off. Each player has five rings that can be placed on one of the 85 intersections.

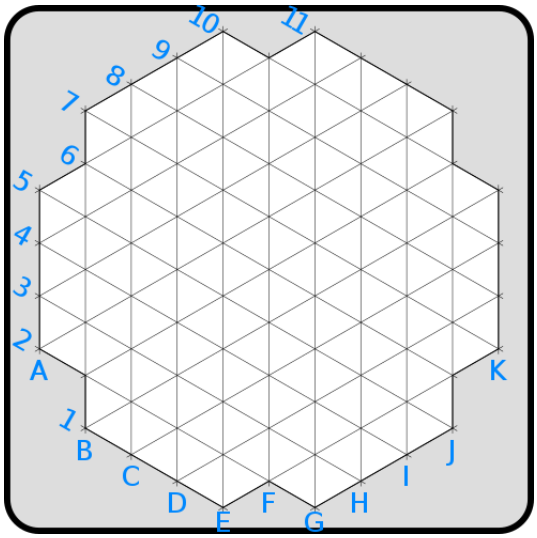


Figure 1: YINSH board with labelled coordinates

Placement phase At the start of the game, each player takes turns, with *white going first*, placing one of their rings in any unoccupied intersection of the board until all five rings of both players have been placed.

Moving phase After all the rings have been placed, in each player's turn, they have to *choose a ring of their own colour* and move it in a *straight line* in any of the six directions. When a ring is moved, it *places a marker of the same colour* on the intersection that it was moved from. It also *inverts the colour of all the markers* that it jumps over. There are some restrictions on how the rings can be moved:

1. A ring cannot move past another ring in the chosen direction.
2. A ring can jump over one or more consecutive markers in a line, regardless of their colour, but must land on the first empty square after the array of markers.

Aim of the game When a *line (a row of 5)* of any colour is created by either player, the player with the colour of the line will remove the 5 markers and a ring of their colour. If multiple lines are created simultaneously, the player can choose any of them to remove. The goal of the game is to *remove 3 lines* of your own markers before your opponent. If two players win at the same time, the player that formed the lines is the winner.

1.1.2 End User(s)

Mr. Chris Walker (Teacher of Digital Creativity)
And other students that are interested in YINSH

1.2 Client Interview

Question: Why do you want this application?

Mr. Walker: Since YINSH is not quite a popular game, it is hard for many people to get their hands on a copy of the board game. I wish to be able to improve my skills by having the *option of playing against a computer player* when there sometimes is no one free to play. An *adjustable difficulty* will allow me to improve gradually, instead of only playing against a fixed skill when playing with another person.

Question: Have you tried playing on other applications like this?

Mr. Walker: I have tried an online version that only allowed you to play YINSH against a computer player (see Figure 2). When the computer player makes a move, *everything happens instantly*, which sometimes leave me confused at which ring was moved and where it moved to. When it is my turn, the game *did not allow me to deselect a ring*; so when I have selected a ring and had not moved it, it does not allow me to select another ring. This is particularly annoying when trying to strategize as you often click onto rings. Sometimes, the AI *takes a long time*, and on some occasions, does not make a move and *the game just halts*. There is also *no additional user interface* that keep tracks of what moves have been played and how many lines have been made, which is a bit inconvenient and confusing.

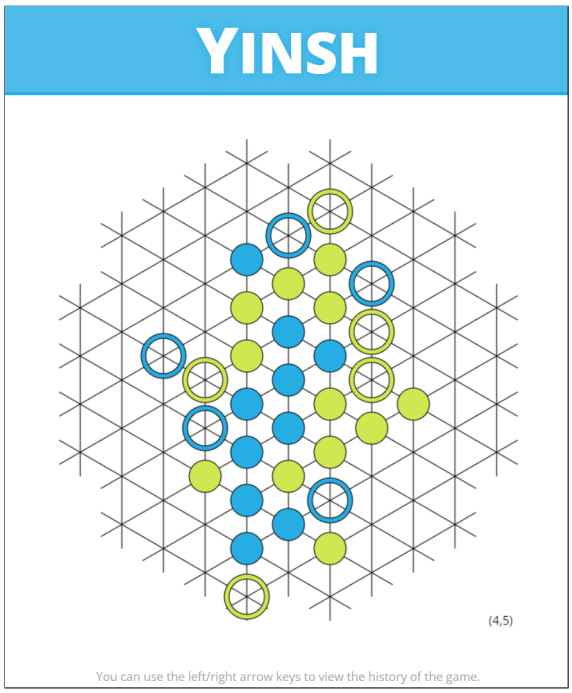


Figure 2: Example of a game being played on `https://david-peter.de/yinsh/`

The code of the computer player is written in Haskell and open source on ***`https://github.com/sharkdp/yinsh/`****. It uses a minimax algorithm, with the evaluation function heavily biased towards removing a ring, and then focused on solely getting more of the computer's markers on the board.*

There is also this game client that you can install, and it allows you to play with different computer player difficulties or other players online (see Figure 3). I found it really annoying to *have to sign up for an account* and do a whole series of verification just to be able to start up the client. This game client has a 3D UI, but again, the *moves made by the AI are instant*, making it really hard to understand. It has a *clock system*, giving each player a

time limit on how long their move has to be, which is quite nice as it forces my opponent to stop stalling. The *UI of the application provided quite a lot of information*, but it was unfortunately really *messy and hard to read*.

Figure 3: Example of game being played on `https://www.boardspace.net/`

Question: So, why do you want me to make this piece of software? How would you like the user experience to be different?

Mr. Walker: I want the game to be *bug-free and be smooth* to play on. The movement and placement of pieces should be easy to understand, and it would be great if *the piece movements can be animated* so that it is easy to see exactly what is going on. I would also like a *clean user interface* with sufficient information, like game time, how much time each player has remaining, whose turn it is, how many lines each player has formed, and a history of the moves. I also want to be *able to move backwards and forwards in time* to see the state of the board at that moment in time so that I can retrace my moves.

Question: Who do you want to play this game with and in what format(online, locally, or with bots)?

Mr. Walker: I see myself mainly playing this game *locally with a friend*, or when I am alone, with an AI. It would be nice if there are *varying difficulties of AIs* with different playing styles so that players like myself can slowly improve by player progressively stronger AIs.

Question: What additional functionality from the software would be useful for you?

Mr. Walker: It would be useful if the software can *store games I've played*. I can then look at the statistics of each game and replay the moves inside, reviewing the mistakes I've made and learn how to improve.

1.3 Project Objectives

1. The user should have the choice of playing against another human player, or against the computer.
   (a) A screen should appear before a game starts for the user to modify settings of the game.
   (b) The user should be able to change the time limit for each player.
   (c) The user should be able to choose who plays as which colour (white plays first).
   (d) The user should be able to select whether a player is a human player or a computer
   (e) The user should be able to select the difficulty of the computer.
2. The user should be able to play a game of YINSH.
   (a) There should be some indication of which intersection/ring/marker the user is hovering on.
   (b) During the placement phase, the user should be able to place rings only on legal intersections.
   (c) During the movement phase, the user should be able to select a ring and a legal intersection to move the ring to.
   (d) The software should allow users to deselect rings after selecting them.
   (e) The software should show all possible moves after a ring is selected.
   (f) After selecting a ring and an intersection to move the ring to, the software should animate the ring jumping over all the markers in between and showing the markers being flipped.
   (g) The software should alert the player when line(s) are created and highlight the line(s).
   (h) The player should be allowed to select which line to remove.
   (i) The player should be allowed to select which ring to remove after removing a line.
   (j) When a player achieves three lines, the software should display which player has won.
3. The software should show the user relevant information in the game screen.
   (a) The software should display what players/computer is playing the current game
   (b) The software should display whose turn it is to play and what they have to do.
   (c) The software should display how many rings each player has left and how many lines they have already previously formed
   (d) The software should display a table of all the previous moves of the game in algebraic notation
   (e) The user should be able to undo and redo moves.
   (f) At the end of a game, the software should display who has won, why they won, and offer a replay for the user to view the game.
4. The software should be able to import and export games and replay them.
   (a) The software should allow games to be exported into a JSON file.
   (b) The software should be able to import game data from a JSON file.
   (c) A history of all the moves played should be displayed in algebraic notation.
   (d) The software should allow the user to go forward and backwards through the moves played in a game.

1.4 Research

1.4.1 Computer Player

Since there are a very large number of positions in YINSH, it is impossible to find the store the best moves of every position in a database. Therefore, we must employ a heuristic algorithm such as minimax.

Minimax algorithm Minimax is an algorithm that is used in AIs for making decisions in two-player games. The algorithm uses a function to score each position, where more advantageous positions are given higher scores. Different elements can be used to create this function. For example, in this case, we can use some combination of the number of hoops on the board and the number of markers in the player's colour. The algorithm examines all possible combinations of moves between it and its opponent until the end of the game, or until some limit to lower the time it takes to run the algorithm. The algorithm examines all possible moves in the future, and scores each position at the maximum depth using the scoring function. At each step, the opponent is assumed to pick the move that results in the minimum score for the AI, whilst the AI will always choose the move that results in the maximum score. This repeats until the root node (current position) is reached, allowing the AI to make a decision. Here is an example of the minimax algorithm decision tree:

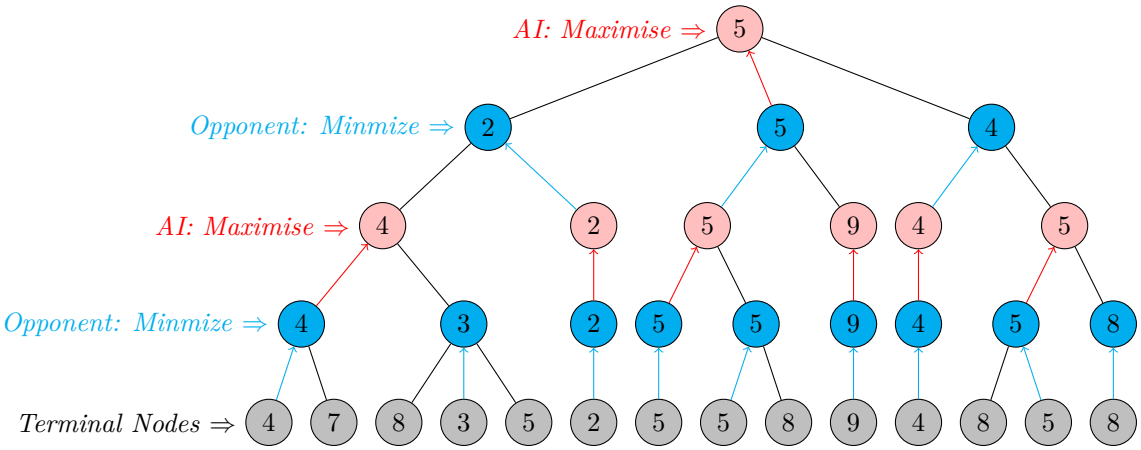


Figure 4: Example of a minimax tree of depth 4

The algorithm starts at the terminal nodes at the bottom of the tree, which will be the possible states of the game after it reaches its maximum depth. The value assigned to the nodes indicate the Then, recursively searching the tree using depth-first search, depending on whose turn it is, the algorithm assigns the parent of nodes the minimum/maximum value in its children. In Figure 4 for example, the cyan coloured nodes are the moves that the opponent can make, and since the algorithm assumes that the opponent will choose the move that results in the lowest evaluation score for the AI, they are assigned the minimum value of that node's children (or, in other words, the possible move from that position). When it is the AI's move (red nodes), the algorithm chooses the child node with the maximum value. Holistically speaking, the algorithm tries to minimize its maximum loss by evaluating all possible moves that the opponent can make in response until it reaches its maximum depth or the end of the game.
If d is the maximum depth and b is the branching factor, or what the maximum amount of moves there are at any position, the worse case time complexity of minimax is $O(b^d)$. This is because there is a maximum of b^d amount of nodes present, and each node has to be searched once.

Alpha-beta Pruning Alpha-beta pruning is a method used in the minimax algorithm to eliminate unnecessary evaluations of nodes/subtrees. While recursively iterating through the tree, two values, α and β are stored and passed around. α is the highest score that is guaranteed for the maximizing player (the AI) and β is the lowest score that is guaranteed for the minimizing player (the opponent).
Let's use Figure 5 as a demonstration of alpha-beta pruning (ignore all other subtrees). After the algorithm analyses the left subtree, the β value is set to 4, as the minimizing player is guaranteed to be able to choose a move that results in an evaluation of 4 or below. When examining the right subtree, when the algorithm reaches the node 8, it is pruned as it is more than β, and is irrelevant as the minimizing player can always pick the node with value 5. After evaluating the node with value 5, the parent of these two nodes are assigned value 5. The value of α in the right subtree is now set to 5, as the maximizing player in the right subtree is guaranteed a score of 5 or greater. However, since the value of α in the right subtree is greater than β in

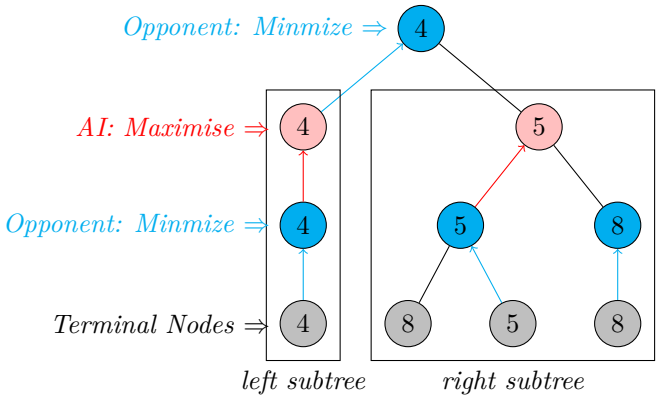


Figure 5: A subtree of the tree in Figure 4

the entire tree so far, the whole right subtree can be pruned as regardless of the values other nodes in the right subtree, it is better for the minimizing player to pick the left subtree. Using alpha-beta pruning on the whole tree, we can see its effects in Figure 6.

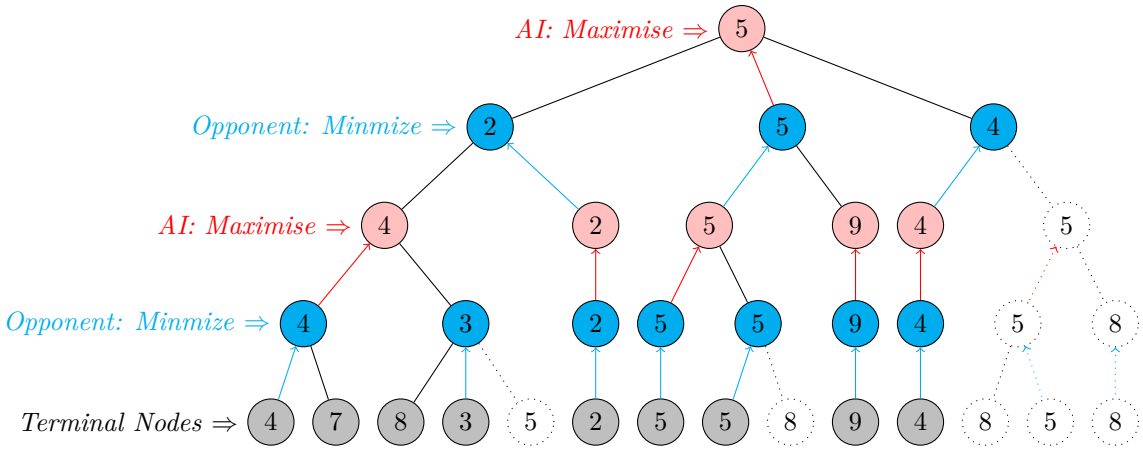


Figure 6: Minimax tree in Figure 4 with alpha-beta pruning

We can spot that after reaching a high value subtree (middle subtree), it allows us to prune a large subtree to the right. We can heuristically optimize the amount of nodes we are able to prune by searching moves that generally yield a higher evaluation score, such as captures in chess, or flipping markers in YINSH.

1.4.2 Choice of Technology

1. **Programming Language**

   Python is a popular high-level programming language created in the late 1980s by Guido van Rossum known for its simplicity, readability, and versatility. It supports multiple programming styles and has a large package library, making it an ideal choice for many applications.[3]

Advantages	Disadvantages
• Easy to write • Easy to read - Python is created with syntax similar to the English language. • Extensive documentation and support - A large community of people use python and is easy for anyone to find help. • Dynamically typed - Increases flexibility of the language	• Comparatively slower than other programming languages

C++ was created in 1983 by Bjarne Stroustrup, as an extension to the existing C programming language to support additional features like object-oriented programming. It is aimed towards maximising performance, and allows for low level memory manipulation, which is suitable for many applications that require speedy execution.[4]

Advantages	Disadvantages
• Statically typed - type mismatch errors can be identified by the linter and easily corrected • Memory efficient - memory can be optimized by doing low level manipulations • Quick runtime - C++ is compiled, which makes it much faster than interpreted languages like Python	• Strict syntax - this will make it slower to write code in C++ • Harder to learn

C# was created by Microsoft in 2002. It is a general purpose language that heavily uses the object-oriented paradigm. It is easier to manage memory in C# than C++ as it has automatic garbage collection.[4]

Advantages	Disadvantages
• Statically typed - type mismatch errors can be identified by the linter and easily corrected • Object oriented - may allow for more elegant programs that are easier to maintain • Quick runtime - C# is compiled, which makes it much faster than interpreted languages like Python	• Strict syntax - this will make it slower to write code in C# • All functionality has to be contained in classes

Hence, I will be choosing python for flexibility, extensive support, and how fast and easy it is to program with it.

2. GUI Library

PyQT is used for making powerful and customizable graphical interfaces in Python. It combines the versatility of Python with the robustness of Qt, offering tools to create user-friendly applications for various platforms. PyQt simplifies complex tasks, making it popular among developers for building interactive and visually appealing software.

Tkinter is a simple yet powerful library that helps create user interfaces. It's beginner-friendly, comes with Python by default, and allows developers to build graphical interfaces for applications easily. Tkinter's versatility and ease of use make it a popular choice for GUI development.

Pygame is a library for game development. It simplifies game creation with its easy-to-understand functions and handling of graphics, sound, and user input. Pygame's flexibility and simplicity make it a go-to choice for learning game programming and building 2D games swiftly.

Since I would like to animate my game elements in real-time, Pygame would be the best as it offers the most flexibility.

1.5 Project Timeline

As Mr. Walker would like to test the software and play with it over the Christmas holidays, we have a timeframe of 4 months, or 16 weeks. Here is a Gantt chart detailing which weeks will be used on which aspects of development of the software.

Task	Wk1	Wk2	Wk3	Wk4	Wk5	Wk6	Wk7	Wk8	Wk9	Wk10	Wk11	Wk12	Wk13	Wk14	Wk15	Wk16
Create the game loop of YINSH for two players to play on the same screen																
Create the algorithm(s) for a computer player and have the computer player interact with the board																
Create the different menus and navigation between pages																
Implement the replay screen for the user to move forward and backward in the moves																
Implement Storing and loading game data into JSON file																

2 Documented Design

2.1 Program Overview

As the program has many elements and pages, we can use abstraction to simplify the program into one flowchart, and dividing the program into parts that can be designed individually and recombined.

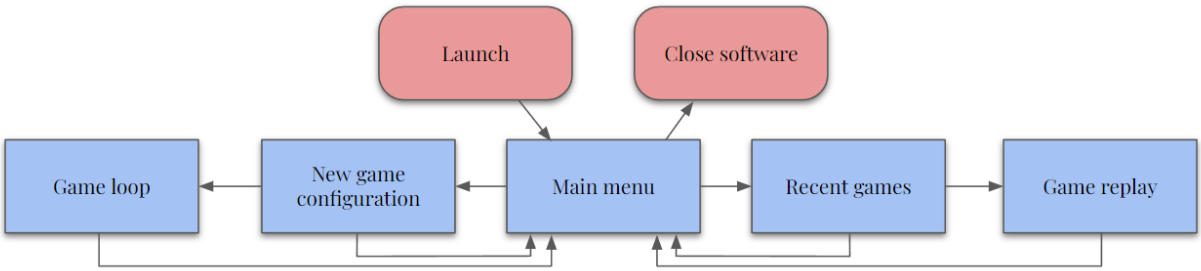


Figure 7: Overview of the whole system flow

2.2 User Interface

2.2.1 Main Menu

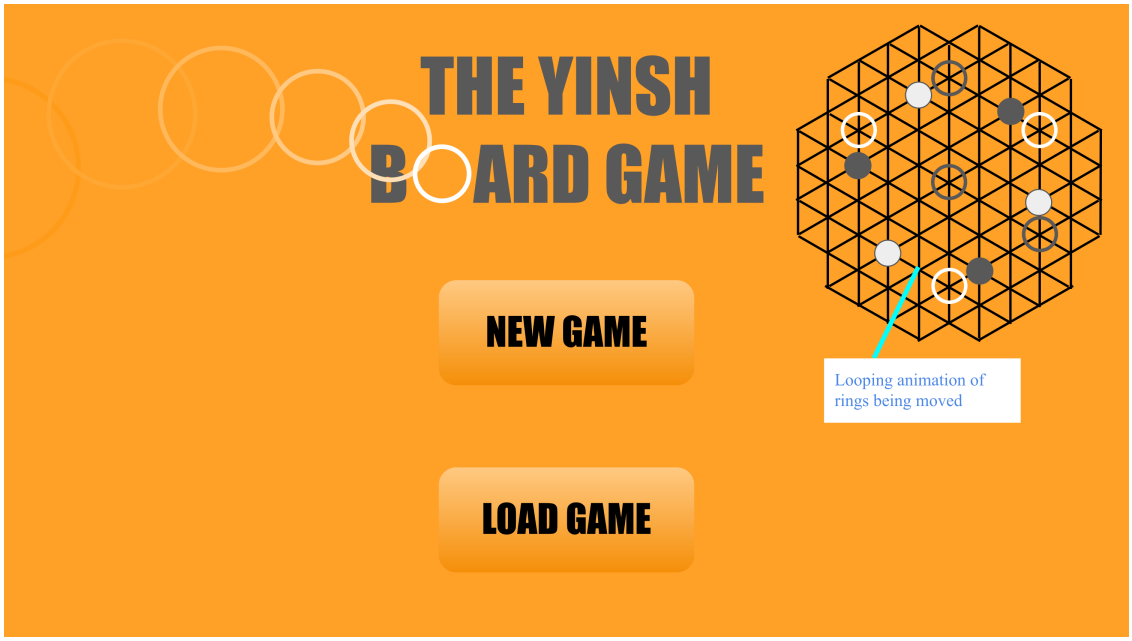


Figure 8: The main menu

2.2.2 New Game Configuration

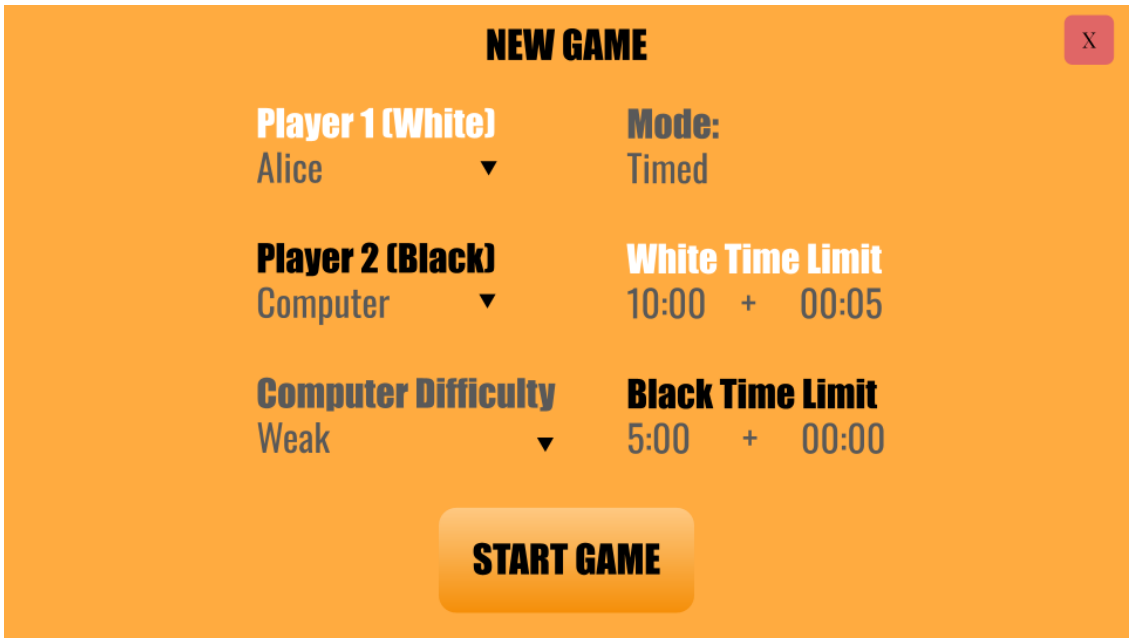


Figure 9: The main menu

2.2.3 Gameplay

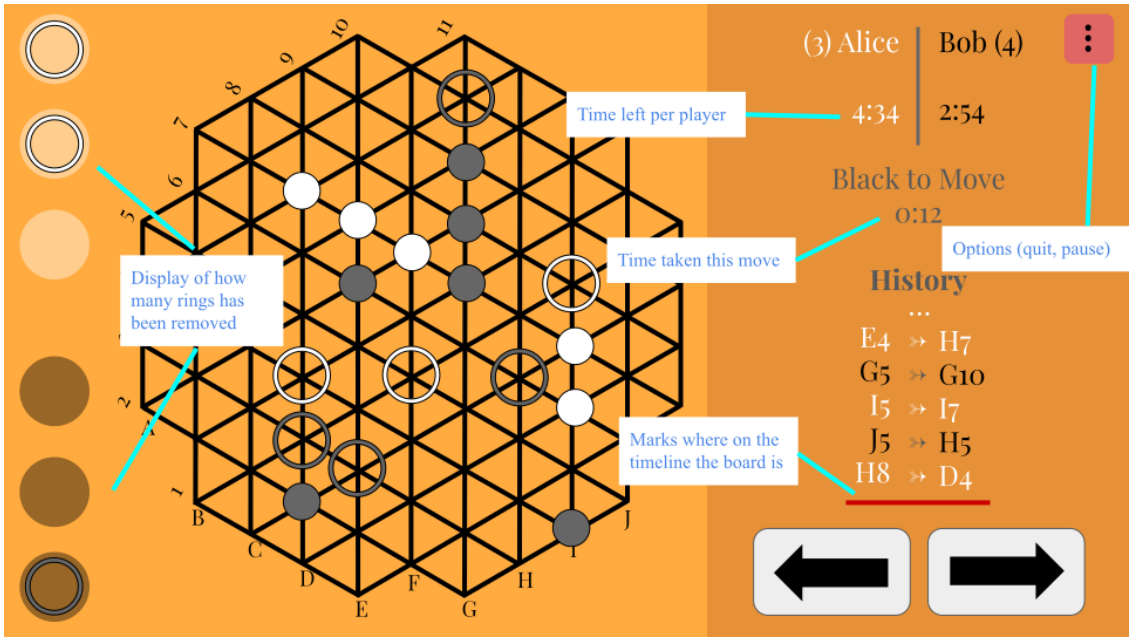


Figure 10: Gameplay UI

2.3 YINSH

2.3.1 Game Logic

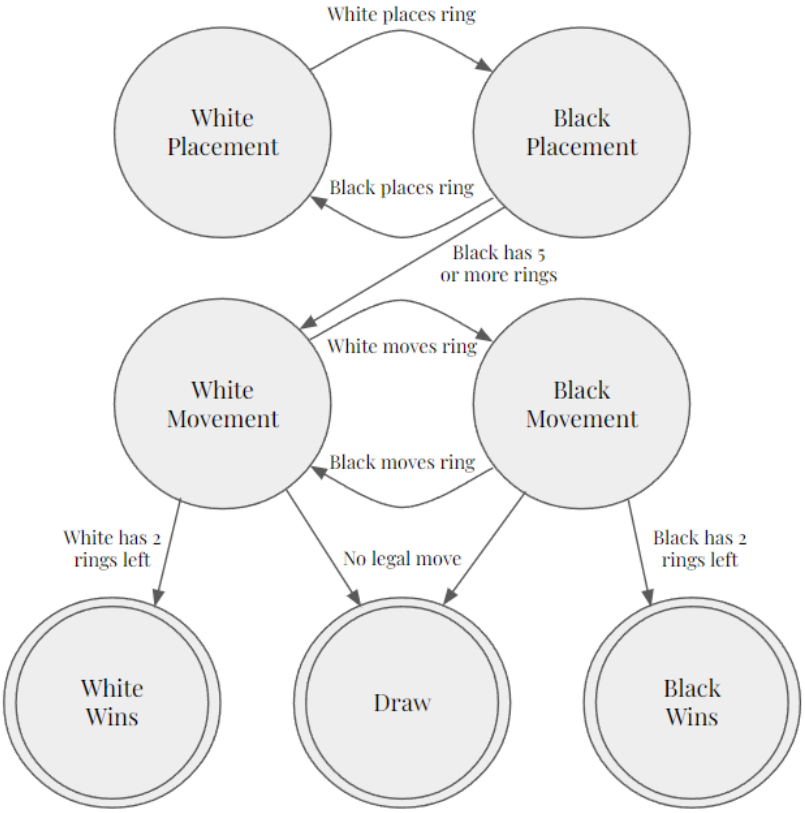


Figure 11: Finite-state machine outlining the main game flow

Since there are multiple states that the game is in, the finite-state machine in Figure 11 illustrates how the program transitions between different states. The detailed operations of each state will be detailed below. Also note that there are some special cases not included in this overview, such as when both players have removed their 3rd ring at the same time.

White/Black placement phase

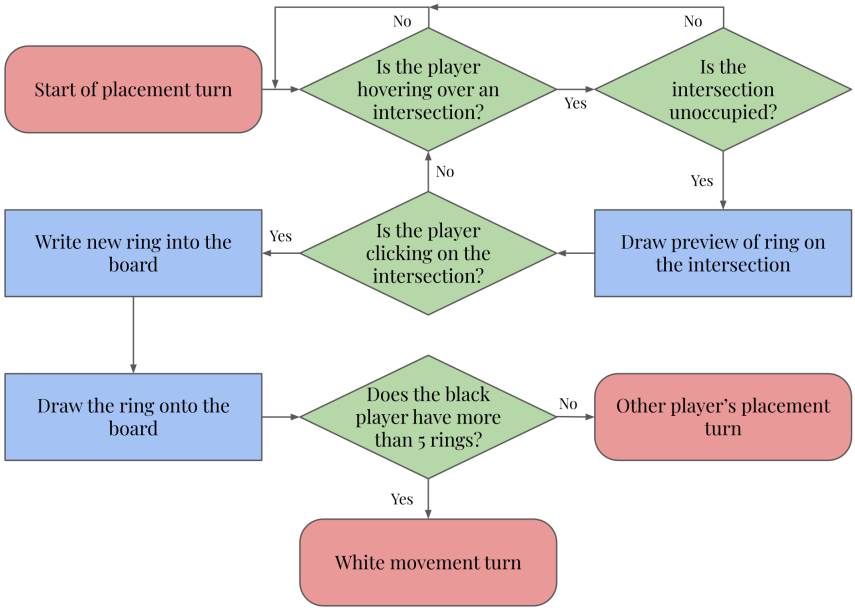


Figure 12: Flowchart of a player's turn in the placement phase

White/Black movement phase

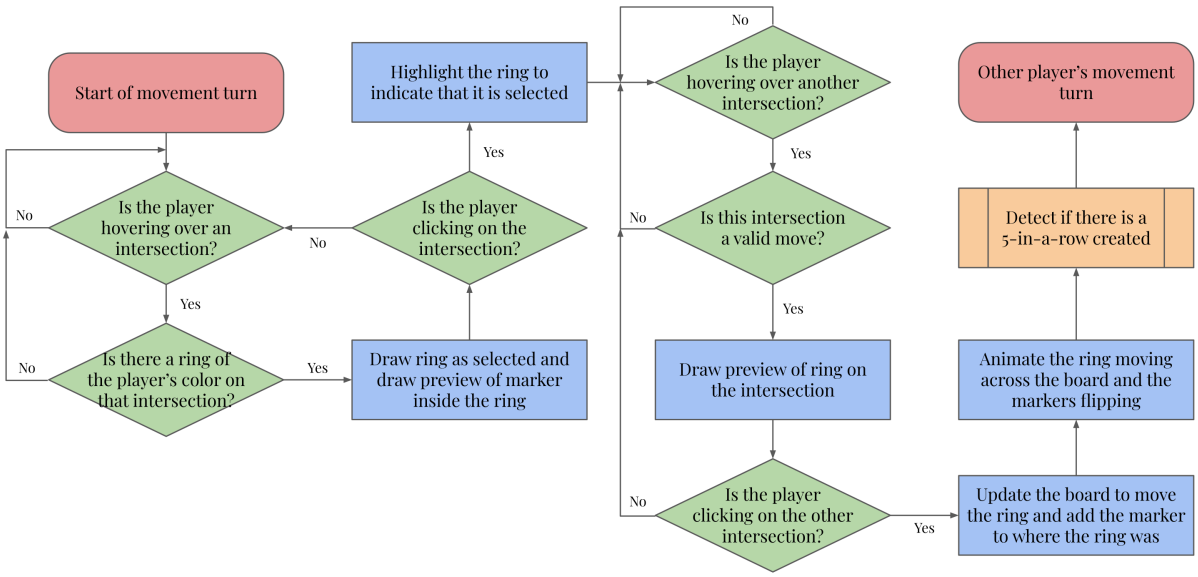


Figure 13: Flowchart of a player's turn in the movement phase

2.3.2 Drawing the Board

If we look closely at the YINSH board, it is constructed by three set of lines that are angled differently (0°, 120° and 240°). In Figure 14, distinguished by colour into red, green and blue, we can see these three sets creating the yinsh board. As a result, we can just construct one set of these lines (Figure 15) and draw it three times but rotated differently.

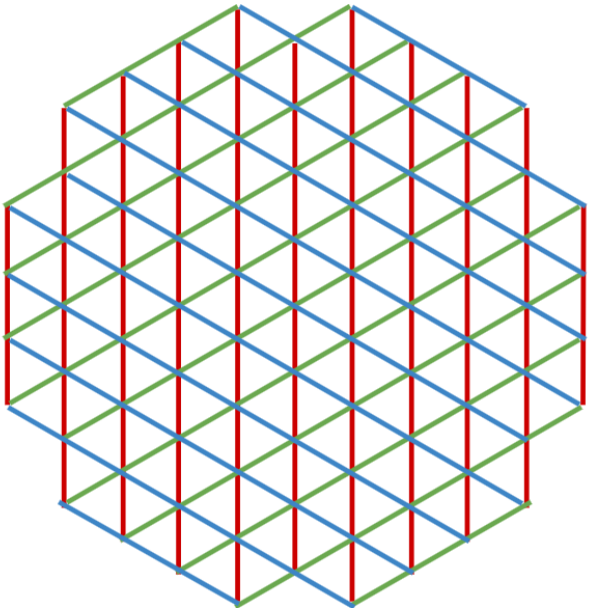


Figure 14: The lines that the YINSH board is constructed by coloured by their direction

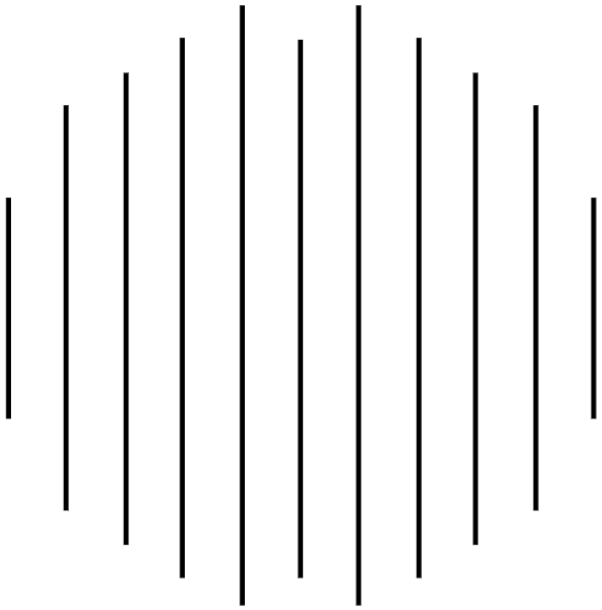


Figure 15: The vertical lines in Figure 14

Each line is defined by two points, and each point has an x and y coordinate. To rotate each line by a certain angle, we can treat its end points as two 2D vectors, and multiply them each by a rotation matrix:

$$R(\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

In this case, we need to rotate the lines by 0°, 120° and 240°. Naturally, $R(0°)$ would not modify the vector at all, which means it will be the identity matrix.

$$R(0°) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, R(120°) = \begin{bmatrix} -\frac{1}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & -\frac{1}{2} \end{bmatrix}, R(240°) = \begin{bmatrix} -\frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & -\frac{1}{2} \end{bmatrix}$$

2.3.3 Board Representation

Inspired from the algebraic notation used by YINSH, we can use the same coordinate system. Each intersection can be defined from the vertical line and the line travelling from top-left to bottom-right intersecting at that position (see Figure 16). Using this, any intersection can be represented by a 2D vector, or a array with two elements. The board can then be represented with a 2D array of size 11*11, with many positions that are inaccessible, such as (11, 3), (1, 6) etc. This will require a getter/setter to make sure the program is accessing/writing to a valid intersection on the 2D array.

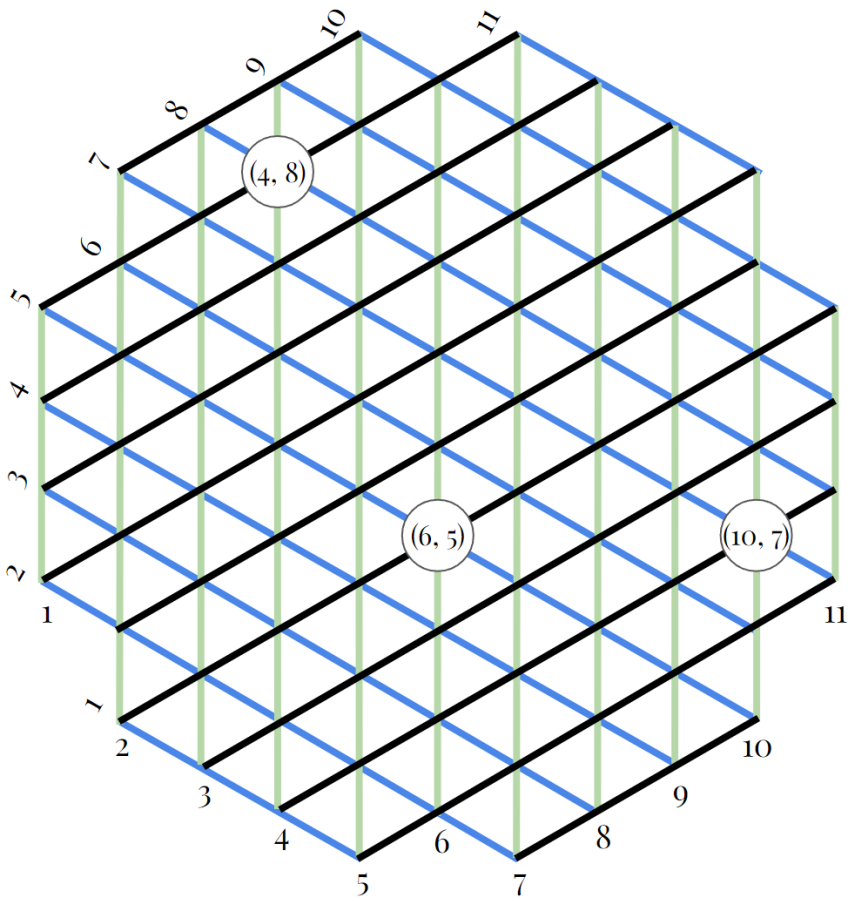
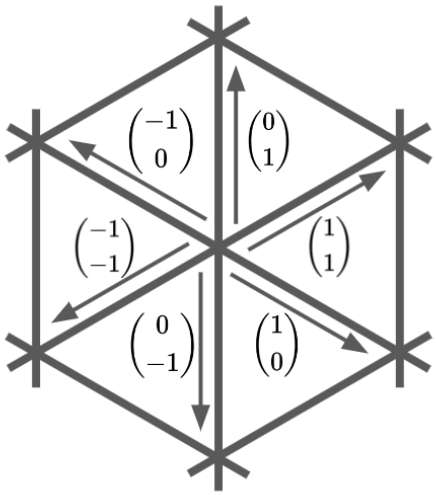


Figure 16: Coordinate system of the YINSH board with example positions

There are 3 directions of movement in this board, which can be represented by 3 pairs of vectors (Figure 2.3.3). Moving in the direction of the blue and green lines in Figure 16 will be just adding/subtracting one from a component of the coordinates, whereas the other direction will be a combination.



2.3.4 Computing Legal Moves

To be able to show the player where they can move the ring to, and for the computer player to perform minimax, we need an algorithm to return all the legal moves for a certain ring.

Algorithm 1 Find all legal moves

```
for direction ∈ directions do          ▷ directions is the set of the six direction vectors in Figure 2.3.3
    searching ← ring position
    searching ← searching + direction
    in marker sequence ← False
    while searching ∈ valid intersections do
        if searching.piece = ring then
            break
        else if searching.piece = marker then
            if in marker sequence = False then
                in marker sequence ← True
            end if
        else if searching.piece = nothing then
            valid moves ← valid moves ∪ searching
            if in marker sequence = True then
                break
            end if
        end if
        searching ← searching + direction
    end while
end for
```

2.3.5 Evaluation Function

The evaluation function is how the computer player measures how favorable the position is to it, hence its strength is entirely dependent on the evaluation function. There are many ways to evaluate a YINSH board. For simplicity, we will assume that the computer player is white and the opponent (human) is black.

Number of Markers and Placement

A naive evaluation method may solely favor a board with more white markers. However, having more white markers, especially in a line, will give the opponent an easier time in flipping markers into black. Therefore, the value of a white marker will need to be evaluated with regard to its positioning and if it can be easily flipped to black. For example, markers in the center of the board may have a higher weighting as they can possibly form more lines. Connected markers, or markers at the edge of the board and cannot be flipped may also have a higher weighting.

Placement of Rings

Rings placed in the center of the board will have the most mobility as they have the most number of possible moves. The mobility of a ring is also determined by whether its path is blocked by other rings and other factors. As a result, rings should be placed in different rows to maximise their mobility.

2.3.6 Computer Player

The idea behind minimax and alpha-beta pruning is explained in 1.4.1. This section covers the algorithm behind minimax, and with alpha-beta pruning.

Minimax

Algorithm 2 Minimax

```
function MINIMAX(board, depth, player)
    if depth = 0 or board is won then
        return EVALUATE(board)
    end if
    if player = maximizing then
        evaluation ← −∞
        for new board ∈ MOVES(board, player) do
            evaluation ← MAX(evaluation, MINIMAX(new board, depth−1, other player))
        end for
        return evaluation
    else if player = minimizing then
        evaluation ← +∞
        for new board ∈ MOVES(board, player) do
            evaluation ← MIN(evaluation, MINIMAX(new board, depth−1, other player))
        end for
        return evaluation
    end if
end function
MINIMAX(current board, depth, maximizing)
```

Alpha-Beta Pruning

Algorithm 3 Minimax with alpha-beta pruning

```
function MINIMAX(board, depth, player, α, β)
    if depth = 0 or board is won then
        return EVALUATE(board)
    end if
    if player = maximizing then
        evaluation ← −∞
        for new board ∈ MOVES(board, player) do
            evaluation ← MAX(evaluation, MINIMAX(new board, depth−1, α, β, minimizing))
            if evaluation > β then
                break
            end if
            α ← MAX(α, evaluation)
        end for
        return evaluation
    else if player = minimizing then
        evaluation ← +∞
        for new board ∈ MOVES(board, player) do
            evaluation ← MIN(evaluation, MINIMAX(new board, depth−1, α, β, maximizing))
            if evaluation < α then
                break
            end if
            β ← MIN(β, evaluation)
        end for
        return evaluation
    end if
end function
MINIMAX(current board, depth, −∞, +∞, maximizing)
```

Optimizations

Promote extreme moves. To reduce the running time of the minimax algorithm, we have to maximize the amount of branches that can be pruned by alpha-beta pruning. This can be done by exploring moves that may cause a larger change in the board, which can be sorted by the number of markers flipped in each move. This gives us a larger chance of achieving a more extreme α or β value, pruning off later branches with less extreme values.

Multi-threading. As minimax will take a noticeable amount of time to run when exploring a large tree, it may cause the game to freeze when the computer player is generating its move. Therefore, the minimax algorithm should be run in another thread, allowing users to be able to interact with the board.

Local evaluation The minimax algorithm can first evaluate all the moves at a depth, then choose the best ones to continue exploring, reducing the number of moves that needs to be explored.

2.3.7 Animations

As requested by Mr. Walker, since one move can change a lot on the board, he wishes that the movement of the pieces are animated so that it is easy to see what has happened to the board.

Movement of a Ring

When a ring is moved, it needs to look like it is physically flying across the board. It should simulate the movement of a projectile, which brings us to the projectile equations.

$$x(t) = v_x t$$
$$y(t) = v_y t - \frac{1}{2} g t^2$$

$x(t)$ is a linear function in t, and $y(t)$ is a quadratic equation in t. $x(t)$ will be used to determine the distance of the ring from its initial position to its destination, whereas $y(t)$ will be used to scale the ring to give the effect of the ring moving above the board. After determining the distance(D) of the movement of the ring, we can derive a suitable equation for $y(t)$.

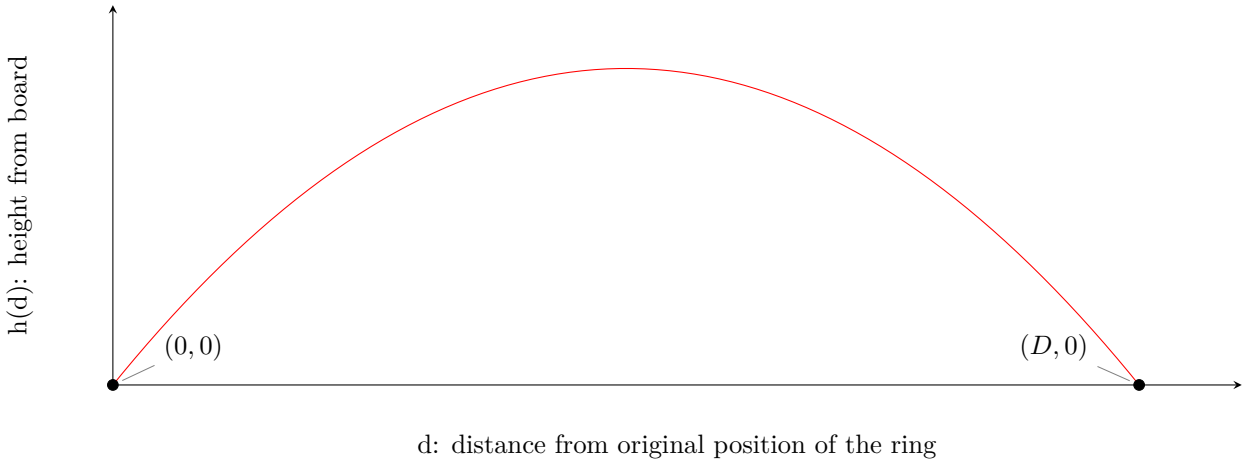


Figure 17: Trajectory of the ring moving across the board

As seen in Figure 17, we want the trajectory to intersect two key points ($(0,0)$ and $(D,0)$), which is when the ring is first launched from the board in its initial position, and when the ring contacts the board at the

destination. This tells us that $h(d) = -A(d-0)(d-D)$, where A is the constant we can tweak around to control the maximum height of the trajectory. Now we have the equations of distance and the height over time.

$$distance(t) = v_x t$$

$$height(t) = -A(v_x t)(v_x t - D) = A(Dx_v t - (v_x t)^2)$$

2.4 Class Diagrams

2.4.1 Representation of Board

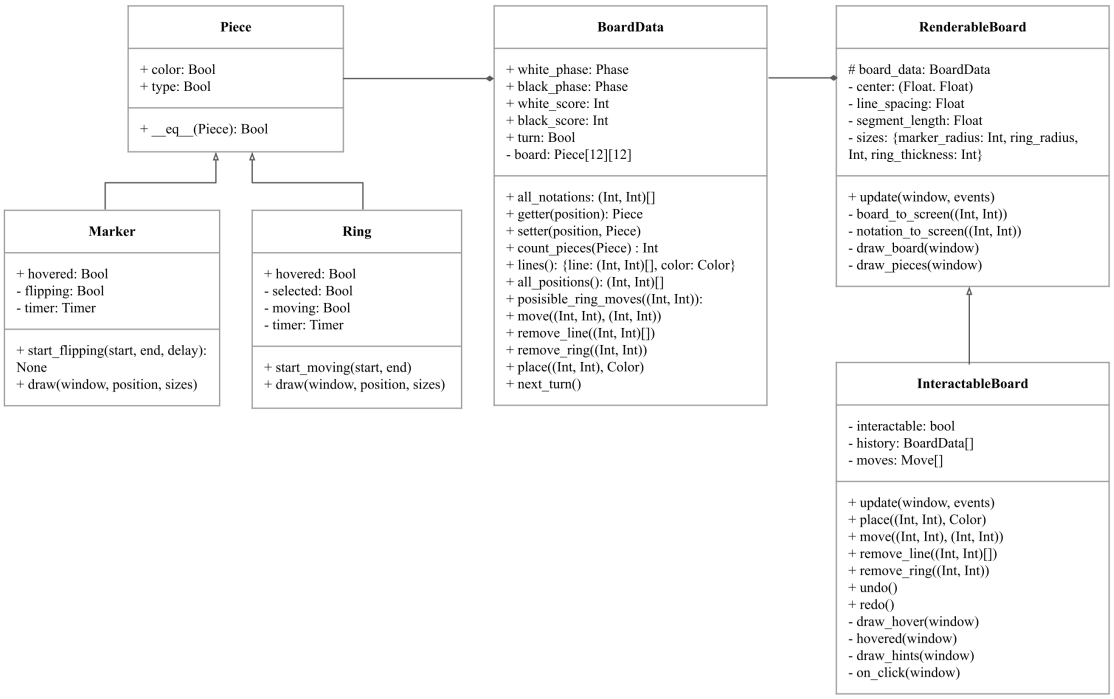


Figure 18: Class diagram of the playing board

2.4.2 User Interface Elements

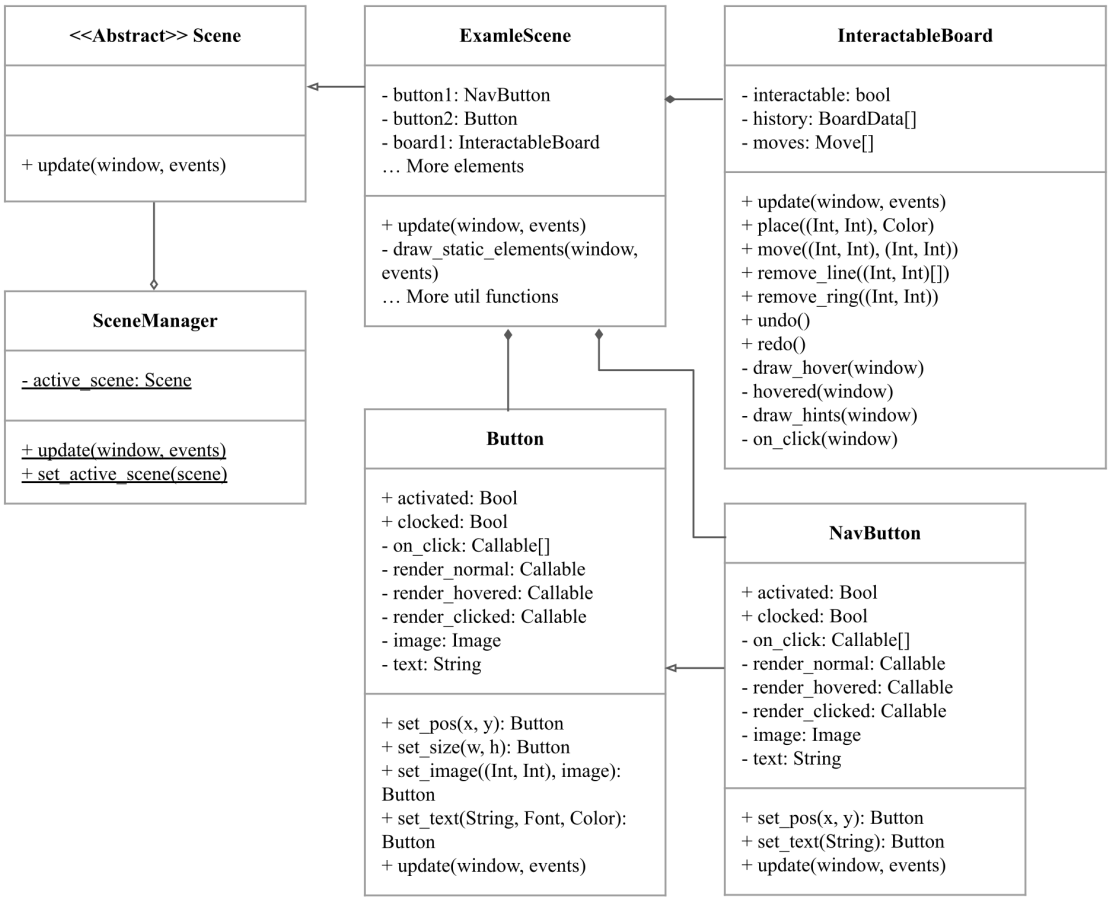


Figure 19: Class diagram of the UI elements

3 Technical Solution

3.1 Highlights of Techniques used

- Encapsulation to ensure accessing valid positions (Section 3.3.2)
- Animations - modeled with projectile motion (Section 3.5.2)
- Minimax algorithm with local evaluation and alpha beta pruning (Section 3.6.3)
- Merge sort (Section 3.6.4)
- Polymorphism of scenes by inheriting from an abstract class (Section 3.7.1 and Section 3.9)
- Game loop management and multithreading (Section 3.8 and Section 3.9.2, in `__run_computer_player()`)

3.2 Overview

3.2.1 File structure

```
YINSH
├── .venv
├── assets
│   ├── image1.png
│   ├── image2.png
│   ├── font1.ttf
│   └── ...
├── board
│   ├── piece
│   │   ├── piece.py
│   │   ├── ring.py
│   │   └── marker.py
│   ├── board_data.py
│   ├── renderable_board.py
│   └── interactive_board.py
├── brain
│   ├── minimax.py
│   ├── evaluation.py
│   ├── moves.py
│   └── merge_sort.py
├── gameUI
│   ├── button.py
│   ├── nav_button.py
│   ├── scene_manager.py
│   └── scene.py
├── scenes
│   ├── title_scene.py
│   ├── game_scene.py
│   ├── leaderboard_scene.py
│   ├── replay_scene.py
│   ├── player_management_scene.py
│   └── history_scene.py
├── main.py
├── util.py
├── constants.py
└── saves
    └── game1.json
```

```
├── game2.json
├── game3.json
└── ...
```

The `.venv` folder is used to store all the packages used in the software, so that the whole software is independent of the system. The `saves` folder is used to store the save files of previous games.I have structured the software in folders that group related files with related functionality together. `util.py` has some utility functions, while `constants.py` contains constant definitions such as colours and enums, and `main.py` is where the main loop is run.

3.2.2 `main.py`

```
# !/usr/bin/env pypy

import os
import pygame
import thorpy as tp
pygame.init()

window = pygame.display.set_mode((0,0), pygame.FULLSCREEN)
pygame.display.set_caption('YINSH')
tp.set_default_font("josefinsansregular", 30)
tp.init(window, tp.theme_human)

from gameUI.scene_manager import SceneManager

events = []

FPS = 60

def main():
  """The game loop that runs the game
  """
  run = True
  clock = pygame.time.Clock()

  while run:
    clock.tick(FPS)
    events = pygame.event.get()

    SceneManager.update(window, events)

    for event in events:
      if event.type == pygame.QUIT:
        run = False

    pygame.display.update()

  exit()

main()
```

3.2.3 `constants.py`

```
from enum import Enum

import pygame

GREY = (100, 100, 100)
WHITE = (200, 200, 200)
PURE_WHITE = (255, 255, 255)
BLACK = (30, 30, 30)
RED = (255, 59, 59)

```

```
BACKGROUND = (78, 89, 173)
FOREGROUND = (105, 119, 224)

PIECE_WHITE = (193, 208, 214)
PIECE_BLACK = (41, 46, 48)

DIRECTIONS = [
  (1, 0),
  (-1, 0),
  (0, 1),
  (0, -1),
  (1, 1),
  (-1, -1)
]

class Phase(Enum):
  placement = 0
  movement = 1
  removal = 2

grid_string = \
"""            5,10     7,11
         4,9     6,10     8,11
      3,8     5,9     7,10     9,11
   2,7     4,8     6,9     8,10     10,11
      3,7     5,8     7,9     9,10
   2,6     4,7     6,8     8,9     10,10
1,5     3,6     5,7     7,8     9,9     11,10
   2,5     4,6     6,7     8,8     10,9
1,4     3,5     5,6     7,7     9,8     11,9
   2,4     4,5     6,6     8,7     10,8
1,3     3,4     5,5     7,6     9,7     11,8
   2,3     4,4     6,5     8,6     10,7
1,2     3,3     5,4     7,5     9,6     11,7
   2,2     4,3     6,4     8,5     10,6
      3,2     5,3     7,4     9,5
   2,1     4,2     6,3     8,4     10,5
      3,1     5,2     7,3     9,4
         4,1     6,2     8,3
            5,1     7,2 """

jose_font_reg20 = pygame.font.SysFont('josefinsansregular', 20)
jose_font_reg20u = pygame.font.SysFont('josefinsansregular', 20)
jose_font_reg20u.underline = True
jose_font_reg30 = pygame.font.SysFont('josefinsansregular', 30)
jose_font_reg40 = pygame.font.SysFont('josefinsansregular', 40)
jose_font_reg50 = pygame.font.SysFont('josefinsansregular', 50)
jose_font_bold50 = pygame.font.SysFont('josefinsanssemibold', 50)
jose_font_bold100 = pygame.font.SysFont('josefinsanssemibold', 100)
```

This file contains constant values such as colour values and fonts which are used across different python files. This avoids the same value being defined in multiple places, and makes it easier to change a value across the whole software. The grid_string is a hexagonal grid with the coordinates of each intersection in the position of the string. This allows code to replace these coordinates with symbols that represent board pieces (rings or markers). Here is an example of a string that represents a board for debugging:

```
            -     -
          -     -     -
        -     -     -     -
      -     -     -     -     -
        -     -     -     -
      -     -     X     -     -
-     -     -     -     -     -
      -     O     -     -     -
-     -     o     O     -     -
      -     -     O     -     -
-     -     o     X     -     -
      -     O     X     O     -
-     -     x     X     -     -
      -     x     X     -     -
        -     -     -     -
      -     -     -     -     -
        -     -     -     -
          -     -     -
            -     -
```

An `O` represents a white ring, and a `X` represents a black ring. A `o` represents a white marker, a `x` represents a black marker, and a `-` represents an empty intersection. This is useful for debugging purposes, as it allows me to see the state of the board at a certain point in time. This is especially useful for debugging the minimax algorithm, as the state of the board can be printed out at each recursion level along with the evaluation to see if the algorithm is working as intended.

3.3 YINSH Board

3.3.1 `__init__()`

The board is represented using an 11 by 11 2D list. See Figure 16 for the board layout.

3.3.2 `__get_item__()` and `__set_item__()`

These are the getter and setter of the board data itself to **encapsulate** the contents of the board. Since not all positions in the 2D list are used to represent the board, to prevent accidentally reading or writing to a position that does not exist, the getter and setter first checks if the position being accessed is a valid position by using the `all_notations()` method before writing or reading a position in the board. If the position does not exist, it will raise an error to prevent the program from continuing with invalid data.

3.3.3 `lines()`

This method is used to detect the presence of lines and return the position and colour of the lines if there are any. To avoid the same line to be counted twice, we iterate through all the possible positions on the board, but then only look for three directions.

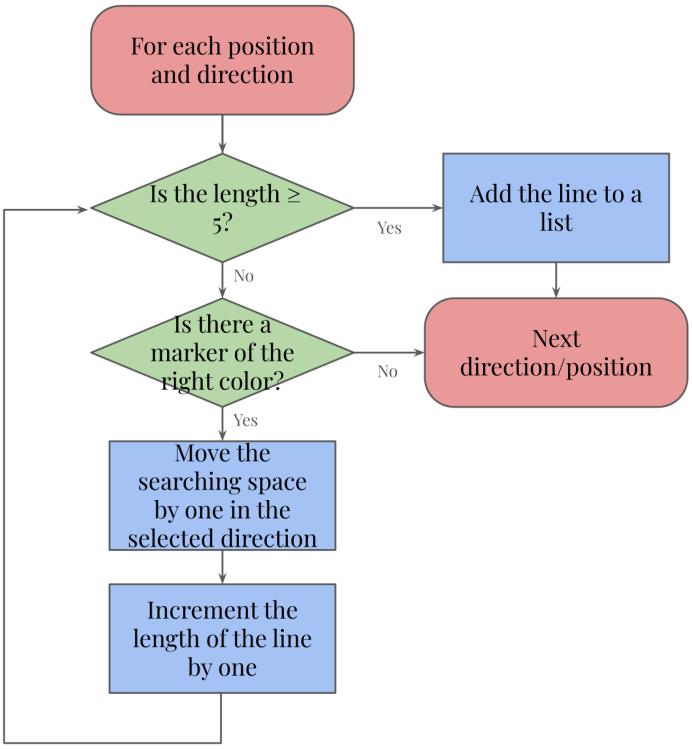


Figure 20: The logic behind detecting if a line exists given a position and a direction

3.3.4 `possible_ring_moves()`

This method returns a list of all possible positions a certain ring can move to. This is used for rendering hints for the user as well as for the computer player to search for moves.

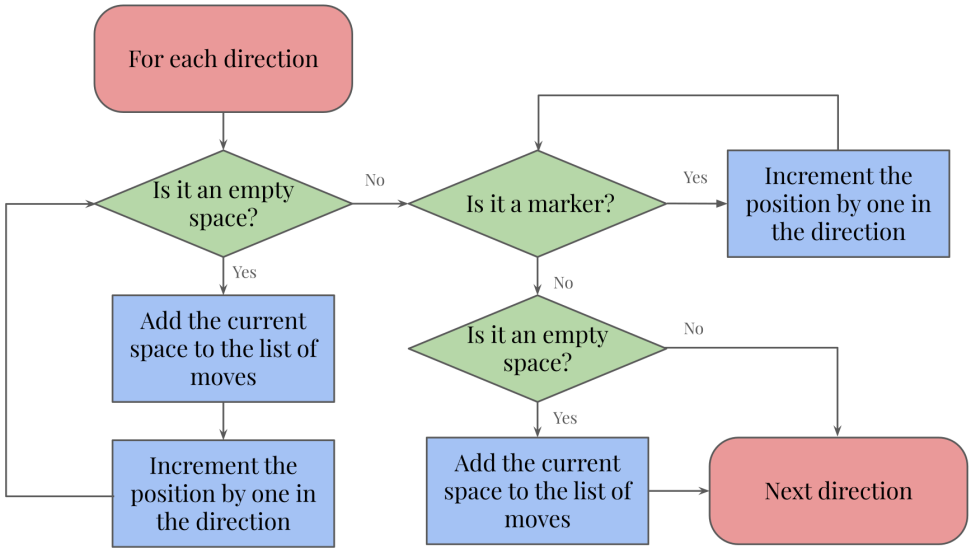


Figure 21: The logic behind finding where a ring can move to

3.3.5 `move()`

It changes the data stored in the board, then triggers to animation of the movement of the ring to start, and also checks if any new lines are created and flagging them to be removed if there are any.

3.3.6 `__repr__()`

This uses the formatted string in constants.py (see Section 3.2.3) to produce a string based representation of the board for ease of debugging the board.

3.3.7 Other methods

Refer to the doc strings in the methods to understand what they do. They are mostly utility methods and methods that allow a user/computer player interact with the board.

3.3.8 Code

board_data.py

```
from copy import deepcopy
from types import SimpleNamespace

from constants import DIRECTIONS, PIECE_BLACK, PIECE_WHITE, Phase, grid_string

from .piece import Marker, Ring


class BoardData:
  def __init__(self):
    """The constructor of BoardData.
    """
    self.__board = [[None] * 11 for _ in range(11)]

    self.phase = {
      "white": Phase.placement,
      "black": Phase.placement
    }

    self.score = {
      "white": 5,
      "black": 5
    }

    self.turn = "white"
    self.__lines_creator = None

  def __repr__(self):
    """Returns the ASCII representation of the board which can be printed into the terminal
                                                for debugging purposes.

    Returns:
        string: the ASCII representation of the board.
    """
    string = grid_string
    for position in self.all_notations():
      pos_str = f"{position[0]},{position[1]} "
      if self[position] == None:
        string = string.replace(pos_str, ". ")
      elif self[position] == Ring("white"):
        string = string.replace(pos_str, "0 ")
      elif self[position] == Marker("white"):
        string = string.replace(pos_str, "o ")
      elif self[position] == Ring("black"):
        string = string.replace(pos_str, "X ")
      elif self[position] == Marker("black"):
        string = string.replace(pos_str, "x ")
```

```
    return string

  def all_notations(self):
    """Returns a list of all the notations of intersections in the board.

    Returns:
        [Tuple[int, int]]: List of notations.
    """
    notations = []
    for y in range(2, 6): notations.append((1, y))
    for y in range(1, 8): notations.append((2, y))
    for y in range(1, 9): notations.append((3, y))
    for y in range(1, 10): notations.append((4, y))
    for y in range(1, 11): notations.append((5, y))
    for y in range(2, 11): notations.append((6, y))
    for y in range(2, 12): notations.append((7, y))
    for y in range(3, 12): notations.append((8, y))
    for y in range(4, 12): notations.append((9, y))
    for y in range(5, 12): notations.append((10, y))
    for y in range(7, 11): notations.append((11, y))
    return notations

  def __getitem__(self, notation):
    """Getter for the board. Checks if the notation is valid before accessing the data.

    Args:
        notation (Tuple[int, int]): The notation of the postion to be accessed.

    Raises:
        ValueError: If the notation given is invalid.

    Returns:
        Piece: The piece located at the given positon. Can be None.
    """
    notation = tuple(notation)
    if notation not in self.all_notations():
      raise ValueError(f"Coordinate {notation} is invalid.")
    else:
      return self.__board[notation[0]-1][notation[1]-1]

  def __setitem__(self, notation, value):
    """Setter for the board. Checks if the notation is valid before accessing the data.

    Args:
        notation (Tuple[int, int]): The notation of the postion to be accessed.
        value (Piece): The piece to be written at the given position.

    Raises:
        ValueError: If the notation given is invalid.
        ValueError: If the value given is not a valid Piece.
    """
    if notation not in self.all_notations():
      raise ValueError(f"Coordinate {notation} is invalid.")
    if value in [None, Ring("white"), Ring("black"), Marker("white"), Marker("black")]:
      self.__board[notation[0]-1][notation[1]-1] = value
    else:
      raise ValueError(f"Value {value} is invalid.")

  def count_pieces(self, piece):
    """Counts the number of pieces on the board with the same type as the given piece.

    Args:
        piece (Piece): The piece to be counted.

    Returns:
        int: The number of occurences.
    """
    count = 0
```

```
    for row in self.__board:
      for thing in row:
        if thing == piece:
          count += 1
    return count

  def lines(self):
    """Computes the lines that are present in the current board.

    Returns:
        [TypedDict("Line", {"positions": [Tuple[int, int], "color": string]})]: A list of
                                              lines, where each line consists of the
                                              positions of the markers that constitutes it,
                                              and the color of the markers.
    """
    lines = []
    # Iterate over all possible notations
    for start in self.all_notations():
      if self[start] == None or self[start].type != "marker":
        continue
      start_color = self[start].color
      # Iterate over the 3 possible directions of a line
      for direction in [(1, 0), (1, 1), (0, 1)]:
        line = [start]
        failed = False
        # Traverse the line and check to see if all the pieces are markers
        for steps in range(1, 5):
          searching = (start[0]+direction[0]*steps, start[1]+direction[1]*steps)
          line.append(searching)
          # If searching out of range of the board
          if searching not in self.all_notations():
            failed = True
            break
          piece = self[searching]
          # If position being searched is empty, not a marker, or not the same color
          if piece == None or piece.type != "marker" or piece.color != start_color:
            failed = True
            break
        # Try another direction if this failed
        if failed: continue
        # If not failed, line exists
        lines.append(SimpleNamespace(**{"positions": line, "color": start_color}))
    return lines

  def all_positions(self):
    """Returns a list of all the coordinates of intersections.

    Returns:
        [Tuple[int, int]]: List of coordinates.
    """
    intersections = []
    for notation in self.all_notations():
      intersections.append(self.notation_to_screen(notation))
    return intersections

  def possible_ring_moves(self, position):
    """Returns a list of all the possible ring moves given the position of a ring

    Args:
        position (Tuple[int, int]): Position of a ring.

    Returns:
        [Tuple[int, int]]: List of positions that the ring can travel to
    """
    moves = []
    # Iterate over the 5 directions
    for direction in DIRECTIONS:
      searching = [position[0] + direction[0], position[1] + direction[1]]
```

```
      in_marker_line = False # True if searching is still within a line of markers
      while tuple(searching) in self.all_notations():
        # If position is empty
        if self[searching] == None:
          moves.append(tuple(searching.copy()))
          # If was searching a line of markers and reaching an empty space, add the next
                                                empty space but stop afterwards
          if in_marker_line: break
        # If another ring is found, break immediately as rings cannot join over each other
        elif self[searching].type == "ring":
          break
        # If a marker is found, indicate that it was searching inside a line of markers
        elif self[searching].type == "marker":
          in_marker_line = True
        searching[0] += direction[0]
        searching[1] += direction[1]
    return moves

  def move(self, start, end):
    """Moves the ring

    Args:
        start (start): The starting position of the ring
        end (end): The position the ring is moving to
    """
    new_board = deepcopy(self)

    # Check if any markers needs to be flipped
    direction = [end[0] - start[0], end[1] - start[1]]
    for i in range(2):
      if direction[i] >= 1: direction[i] = 1
      elif direction[i] <= -1: direction[i] = -1

    searching = list(deepcopy(start))
    dist = 1

    while tuple(searching) != end:
      if self[searching] != None and self[searching].type == "marker":
        start_color = self[searching].RGB
        end_color = PIECE_BLACK if self[searching].RGB == PIECE_WHITE else PIECE_WHITE
        self[searching].color = "white" if self[searching].color == "black" else "black"
        new_board[searching].color = "white" if new_board[searching].color == "black" else "
                                                black"
        self[searching].start_flipping(start_color, end_color, dist * 150)
      searching[0] += direction[0]
      searching[1] += direction[1]
      dist += 1

    color = self[start].color

    # Update board with ring movements and new marker
    self[end] = Ring(color)
    new_board[end] = Ring(color)
    self[start] = Marker(color)
    new_board[start] = Marker(color)

    # Test if there are any lines created
    if any(line.color == "white" for line in self.lines()):
      self.phase["white"] = Phase.removal
      self.__lines_creator = color
      new_board.phase["white"] = Phase.removal
      new_board.__lines_creator = color
      self.turn = "white"
    if any(line.color == "black" for line in self.lines()):
      self.phase["black"] = Phase.removal
      self.__lines_creator = color
      new_board.phase["black"] = Phase.removal
      new_board.__lines_creator = color
```

```
245         self.turn = "black"
246       if any(line.color == "white" for line in self.lines()) and any(line.color == "black" for
                                                    line in self.lines()):
247         self.turn = new_board.__lines_creator
248       if len(self.lines()) == 0:
249         self.next_turn()
250         new_board.next_turn()
251
252       return new_board
253
254     def remove_line(self, line):
255       """Remove the line given the positions
256
257       Args:
258           line ([Tuple[int, int]]): A list of the positions of the markers that make up the
                                                    line
259       """
260       for position in line.positions:
261         self[position] = None
262       if line.color == "white":
263         self.score["white"] -= 1
264       else:
265         self.score["black"] -= 1
266
267     def remove_ring(self, position):
268       """Removes a ring
269
270       Args:
271           position (Tuple[int, int]): The position of the ring
272       """
273       color = self[position].color
274       other_color = "white" if color == "black" else "black"
275       self[position] = None
276       # If all lines and rings already removed
277       if self.score[color] == self.count_pieces(Ring(color)):
278         self.phase[color] = Phase.movement
279         if self.phase[other_color] != Phase.removal:
280           if self.__lines_creator == "white": self.turn = "black"
281           else: self.turn = "white"
282         else:
283           self.next_turn()
284
285     def place(self, position, color):
286       """Places down a ring
287
288       Args:
289           position (Tuple[int, int]): The position of the ring
290           color (string): The color of the ring
291       """
292       if self[position] == None:
293         self[position] = Ring(color)
294         self.next_turn()
295       if self.count_pieces(Ring(color)) >= 5:
296         self.phase[color] = Phase.movement
297
298     def next_turn(self):
299       """Give control to the other player
300       """
301       if self.turn == "white": self.turn = "black"
302       else: self.turn = "white"
```

3.4 Rendering and Interacting with the YINSH Board

3.4.1 `RenderableBoard.update()` and `InteractableBoard.update()`

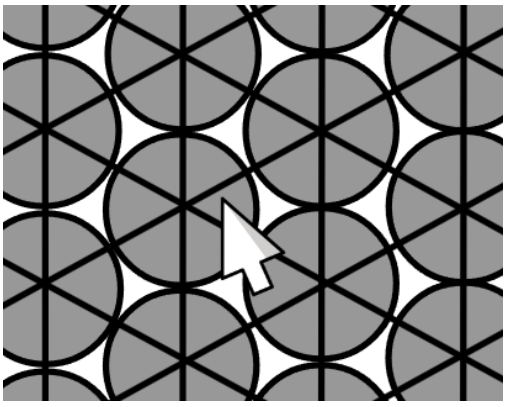
The update method should be present on all objects that is rendered onto the screen. In this case, the `RenderableBoard.update()` method is responsible for invoking the methods `__draw_board()` and `__draw_pieces()`. The `InteractableBoardupdate()` method will invoke the `RenderableBoard.update()` method from the inherited `RenderableBoard` class to draw the board elements, as well as invoking methods for interaction with the user such as `__draw_hover()`, `__draw_hints()`, or `__on_click()` if the board is clicked.

3.4.2 Handling hovering and clicking

Algorithm 4 Find the position that the cursor is on

```
min distance ← ∞
closest position ← Null
for position ∈ all positions do
    screen position ← ScreenPosition(position)
    distance ← √((screen position.x − cursor.x)² + (screen position.y − cursor.y)²)
    if distance < min distance then
        min distance ← distance
        closest position ← position
    end if
end for
if closest position ≠ Null then
    distance from cursor ← √((closest position.x − cursor.x)² + (closest position.y − cursor.y)²)
    if distance from cursor ≤ line spacing / 2 then return closest position
    elsereturn Null
    end if
end if
```

This is equivalent to imagining small circles centered at each intersection, and when the cursor enters any one of these circles, only then will the program recognize the cursor being on that intersection.



3.4.3 Undo, Redo, and Making Moves

Instead of using two stacks for undoing and redoing moves, I have decided to stay with a list structure. This is to allow for new moves to be made and new boards to be added when the user is undoing moves, allowing the computer player(s) to play even when the user not focused on the latest board.

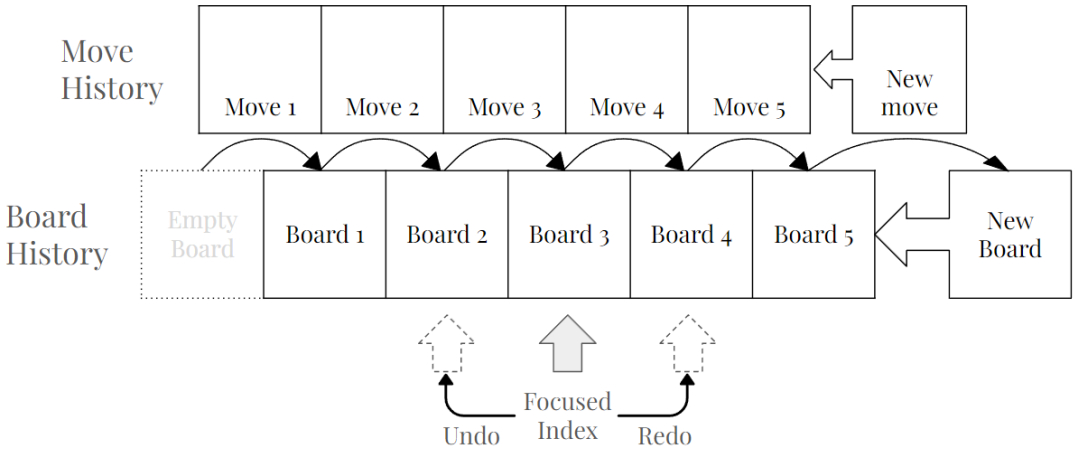


Figure 22: Representation of the move and board history lists

3.4.4 `__hovered()`

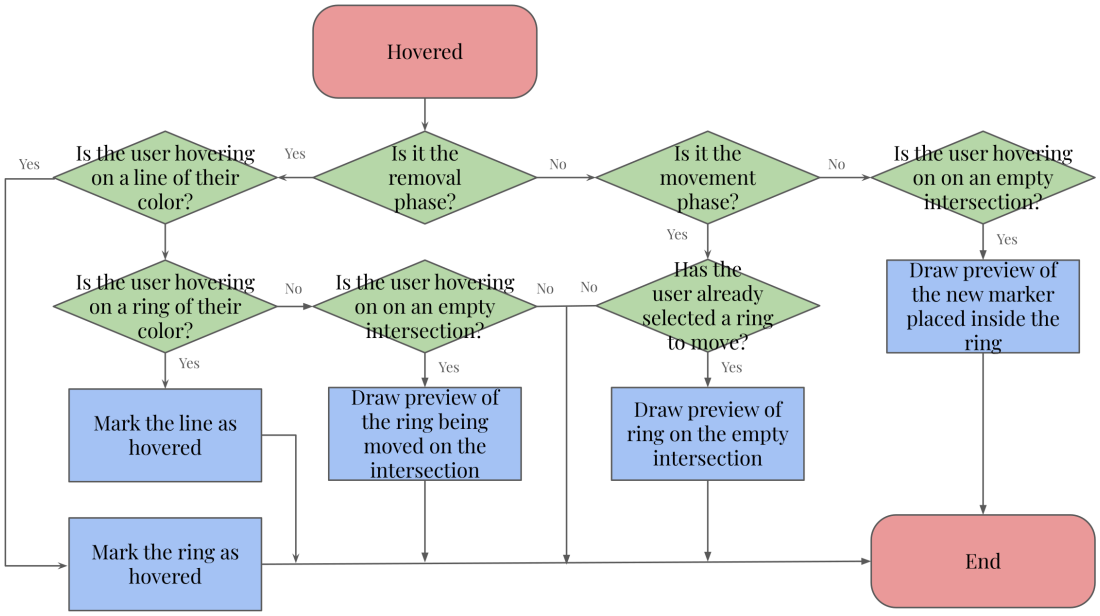


Figure 23: Flowchart of rendering pieces that the user is hovering

3.4.5 `__on_click()`

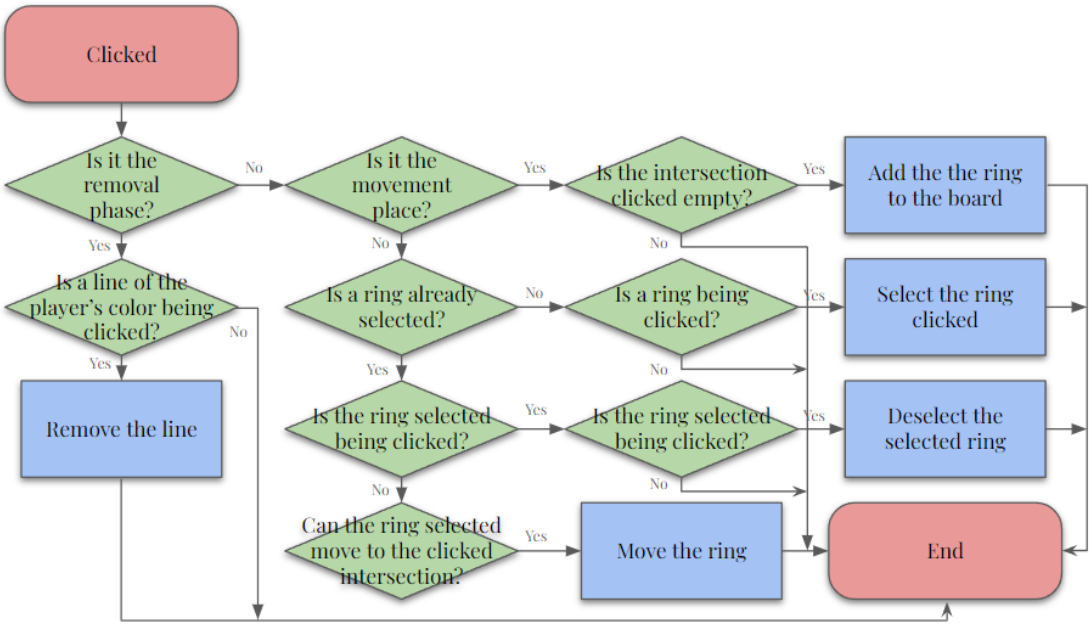


Figure 24: Flowchart of handling a click

3.4.6 Code

renderable_board.py

```python
import math
from types import SimpleNamespace

import numpy
import pygame

from board.board_data import BoardData
from constants import BLACK


class RenderableBoard:
  def __init__(self, x, y, width):
    """Constructor of RenderableBoard.

    Args:
        x (float): x-coordinate of the playing board.
        y (float): y-coordinate of the playing board.
        width (float): width of the playing board.
    """
    self._board_data = BoardData()
    self.__center = {
      "x": x + width / 2,
      "y": y + width / 2
    }
    self._line_spacing = (width * 0.85)/10
    self.__segment_length = 2/math.sqrt(3) * self._line_spacing
    self._sizes = SimpleNamespace(**{
      "marker_radius": self.__segment_length * 0.3,
      "ring_radius": self.__segment_length * 0.45,
      "ring_thickness": round(self.__segment_length * 0.1)
    })

```

```
  def set_board_data(self, board_data):
    """
    Set the board data for the renderable board.

    Args:
        board_data: The new board data to set.
    """
    self._board_data = board_data

  def get_active_board(self):
    """
    Returns the active board data.

    Returns:
        The active board data.
    """
    return self._board_data

  def __repr__(self):
    """Convert the board data into a string

    Returns:
        str: The string representation of the board's data
    """
    return self._board_data.__repr__()

  def update(self, window, events, board_data = None):
    """Update the board.

    Args:
        window (pygame.Surface): The window that the game is in.
        events (list): List of events
    """
    self.__draw_board(window)
    self.__draw_pieces(window, board_data)

  def __board_to_screen(self, coordinates):
    """Converts a coordinate relative to the board to a coordinate positioned absolutely on
                                                the screen.

    Args:
        coordinates (Tuple[float, float]): A coordinate relative to the board.

    Returns:
        Tuple[float, float]: The coordinate positioned absolutely on the screen.
    """
    x, y = coordinates
    return (self.__center["x"] + x, self.__center["y"] + y)

  def _notation_to_screen(self, notation):
    """Converts the notation of an intersection to a coordinate positioned absolutely on the
                                                screen.

    Args:
        notation (Tuple[int, int]): A notation of an intersection on the board.

    Returns:
        Tuple[float, float]: The coordinate positioned absolutely on the screen.
    """
    x, y = notation
    board_x = self._line_spacing * (x - 6)
    board_y = self.__segment_length * (-y + x/2 + 3)

    return self.__board_to_screen((board_x, board_y))

  @staticmethod
  def notation_string_to_notation(notation):
    """Converts a notation to a human readable notation string
```

```
98
99      Args:
100         notation (Tuple[int, int]): A notation
101     """
102     return (ord(notation[0]) - ord('A') + 1, int(notation[1]))
103
104   @staticmethod
105   def notation_to_notation_string(notation):
106     """Converts a notation to a human readable notation string
107
108     Args:
109         notation (Tuple[int, int]): A notation
110     """
111     if notation == None: return ''
112     return chr(notation[0] + ord('A') - 1) + str(notation[1])
113
114   def __draw_board(self, window):
115     """Draws the board.
116
117     Args:
118         window (pygame.Scene): The window that the game is in
119     """
120     for degrees in range(0, 360, 120):
121       for line_order in range(-5, 6):
122         length = 10 * self.__segment_length - abs(line_order) * self.__segment_length
123         if line_order == 0 or line_order == 5 or line_order == -5:
124           length -= 2 * self.__segment_length
125
126         board_x = line_order * self._line_spacing
127         board_y_top = length / 2
128         board_y_bottom = - length / 2
129
130         # perform rotation
131         theta = numpy.radians(degrees)
132         rotation_matrix = [
133           [numpy.cos(theta), -numpy.sin(theta)],
134           [numpy.sin(theta), numpy.cos(theta)]
135         ]
136
137         x1, y1 = numpy.matmul(rotation_matrix, [board_x, board_y_top]).tolist()
138         x2, y2 = numpy.matmul(rotation_matrix, [board_x, board_y_bottom]).tolist()
139
140         pygame.draw.line(
141           surface=window,
142           color=BLACK,
143           start_pos=self.__board_to_screen((x1, y1)),
144           end_pos=self.__board_to_screen((x2, y2)),
145           width=3
146         )
147
148   def __draw_pieces(self, window, board_data = None):
149     """Draws the pieces on the board.
150
151     Args:
152         window (pygame.Scene): The window that the game is in
153     """
154     if board_data == None: board_data = self._board_data
155     s = str(board_data)
156     for notation in board_data.all_notations():
157       if board_data[notation] == None: continue
158       board_data[notation].draw(window, self._notation_to_screen(notation), self._sizes)
```

interactive_board.py

```python
from copy import deepcopy
import math
import re
from types import SimpleNamespace

import pygame
from board.board_data import BoardData

from constants import BLACK, PIECE_BLACK, PIECE_WHITE, Phase
from util import draw_circle_alpha

from .piece import Ring
from .renderable_board import RenderableBoard

class InteractiveBoard(RenderableBoard):
  @staticmethod
  def move_to_string(move, include_color=False):
    """
    Converts a move object to a string representation.

    Parameters:
      move (tuple): A tuple representing a move. The tuple should have the following structure:
        - For "place" moves: ("place", color, position)
        - For "move" moves: ("move", color, start, end)
        - For "remove" moves: ("remove", color, positions, ring_position)

      include_color (bool, optional): Whether to include the color in the string representation.
        Defaults to False.

    Returns:
      str: The string representation of the move.

    Examples:
    >>> move_to_string(("place", "black", (3, 4)), include_color=True)
    'black:C4'
    >>> move_to_string(("move", "white", (1, 2), (3, 4)), include_color=False)
    'A2-C4'
    >>> move_to_string(("remove", "black", [(3, 1), (3, 2), (3, 3), (3, 4), (3, 5)], (5, 6)), include_color=True)
    'black:xC1-C5;E6'
    """
    color_string = ''
    if include_color:
      color_string = f'{move[1]}:'
    match move:
      case "place", color, position:
        return color_string + RenderableBoard.notation_to_notation_string(position)
      case "move", color, start, end:
        return color_string + f'{RenderableBoard.notation_to_notation_string(start)}-{RenderableBoard.notation_to_notation_string(end)}'
      case "remove", color, positions, ring_position:
        return color_string + f'x{RenderableBoard.notation_to_notation_string(positions[0])}-{RenderableBoard.notation_to_notation_string(positions[-1])};{RenderableBoard.notation_to_notation_string(ring_position)}'

  @staticmethod
  def string_to_move(string):
    """
    Converts a string representation of a move into a structured move object.

    Args:
      string (str): The string representation of the move.
```

```
58
59      Returns:
60        list: A list representing the move object with the following structure:
61          - For a "place" move: ["place", color, position]
62          - For a "move" move: ["move", color, start_position, end_position]
63          - For a "remove" move: ["remove", color, positions, ring_position]
64
65      Examples:
66        >>> string_to_move("white:A1")
67        ["place", "white", (1, 1)]
68
69        >>> string_to_move("black:B2-C3")
70        ["move", "black", (2, 2), (3, 3)]
71
72        >>> string_to_move("white:xD4-E8;F6")
73        ["remove", "white", [(4, 4), (4, 5), (4, 6), (4, 7), (4, 8)], "F6"]
74      """
75      place_regex = r'^(white|black):([A-Z][0-9]+)$'
76      move_regex = r'^(white|black):([A-Z][0-9]+)-([A-Z][0-9]+)$'
77      remove_regex = r'^(white|black):x([A-Z][0-9]+)-([A-Z][0-9]+);([A-Z][0-9]+)$'
78
79      if re.match(place_regex, string):
80        color, notation = re.match(place_regex, string).groups()
81        return ["place", color, RenderableBoard.notation_string_to_notation(notation)]
82      elif re.match(move_regex, string):
83        color, start, end = re.match(move_regex, string).groups()
84        return ["move", color, RenderableBoard.notation_string_to_notation(start),
                                                RenderableBoard.notation_string_to_notation(
                                                end)]
85      elif re.match(remove_regex, string):
86        match = re.match(remove_regex, string)
87        color = match.group(1)
88        positions = [match.group(2), match.group(3)]
89        ring_position = match.group(4)
90        return ["remove", color, [RenderableBoard.notation_string_to_notation(positions[0]),
                                                RenderableBoard.notation_string_to_notation(
                                                positions[1])], RenderableBoard.
                                                notation_string_to_notation(ring_position)]
91
92    def __init__(self, x, y, width):
93      """Constructor of InteractiveBoard.
94
95      Args:
96          x (float): x-coordinate of the playing board.
97          y (float): y-coordinate of the playing board.
98          width (float): width of the playing board.
99      """
100     super(InteractiveBoard, self).__init__(x, y, width)
101     self.__last_hovered = []
102     self.__selected_ring_position = None
103     self.__interactable = True
104     self.__history = []
105     self.__moves = []
106
107     self.__is_undoing = False
108     self.__focused_index = -1
109     self.__peeked_board = None
110
111   def get_active_board(self):
112     if self.__is_undoing: return self.__peeked_board
113     else: return self._board_data
114
115   def set_interactable(self, interactable):
116     self.__interactable = interactable
117
118   def get_is_undoing(self):
119     return self.__is_undoing
120
```

```
121   def import_moves(self, moves):
122     """Import a list of moves to the board, and generate the board history
123
124     Args:
125         moves (List): A list of moves to be imported to the board.
126     """
127     self.__moves = moves
128     self.__history = []
129     board_data = BoardData()
130     for move in moves:
131       match move:
132         case "place", color, position:
133           board_data.place(position, color)
134         case "move", color, start, end:
135           board_data.move(start, end)
136         case "remove", color, line, ring:
137           if line != None: board_data.remove_line(SimpleNamespace(**{"positions": line, "
                                                    color": color}))
138           if ring != None: board_data.remove_ring(ring)
139       self.__history.append(deepcopy(board_data))
140     self._board_data = deepcopy(board_data)
141
142   def get_moves(self):
143     """
144     Returns the list of moves made on the board.
145
146     Returns:
147         list: A list of moves made on the board.
148     """
149     return self.__moves
150
151   def get_focused_index(self):
152     """
153     Returns the index of the currently focused board on the interactive board.
154
155     Returns:
156         int: The index of the currently focused board.
157     """
158     if self.__is_undoing:
159       return self.__focused_index
160     else:
161       return len(self.__history)
162
163   def update(self, window, events):
164     """Updates the board.
165
166     Args:
167         window (pygame.Surface): The surface to draw the board on.
168         events (list): List of events.
169     """
170     super().update(window, events, self.get_active_board())
171     self.__draw_hints(window)
172     self.__draw_hover(window)
173     if not self.__interactable: return
174     for event in events:
175       if event.type == pygame.MOUSEBUTTONDOWN:
176         self.__on_click(pygame.mouse.get_pos())
177
178   def __nearest_notation(self, coordinates):
179     """Gets the notation of the nearest intersection to a given screen coordinate.
180
181     Args:
182         coordinates (Tuple[float, float]): Coordinates of the target position.
183
184     Returns:
185         Tuple[int, int]: The notation for the nearest intersection. Can be None when the
                                                    nearest intersection is too far.
186     """
```

```
    x, y = coordinates
    closest_distance = 1e18
    closest_notation = None
    for notation in self._board_data.all_notations():
      intersection_x, intersection_y = self._notation_to_screen(notation)
      distance = math.sqrt((x-intersection_x)**2 + (y-intersection_y)**2)
      if distance < closest_distance:
        closest_distance = distance
        closest_notation = notation
    if closest_distance <= self._sizes.ring_radius:
      return closest_notation
    else:
      return None

  def __draw_hover(self, window):
    """Called every frame to check if an intersection is hovered.

    Args:
        window (pygame.Surface): The window that the game is in.
    """
    if self.__is_undoing: return
    notation_hovered = self.__nearest_notation(pygame.mouse.get_pos())
    # De-hover previously hovered item
    if len(self.__last_hovered) != 0:
      for item in self.__last_hovered:
        item.hovered = False
      self.__last_hovered = []

    if not self.__interactable: return
    # Call __hovered() if the cursor is on an intersection
    if notation_hovered != None: self.__hovered(notation_hovered, window)

  def __hovered(self, notation_hovered, window):
    """Called when an intersection is hovered.

    Args:
        notation_hovered (Tuple[int, int]): The intersection being hovered.
        window (pygame.Surface): The window that the game is in.
    """
    # self.board_data.turn is the person who created the lines, so they should always get
                                                priority.

    # REMOVAL PHASE
    if self._board_data.phase["white"] == Phase.removal or self._board_data.phase["black"] =
                                                = Phase.removal:
      # If there are any lines of the priority player
      other_color = "white" if self._board_data.turn == "black" else "black"
      if any(line.color == self._board_data.turn for line in self._board_data.lines()):
        for line in self._board_data.lines():
          if line.color != self._board_data.turn: continue
          if notation_hovered in line.positions:
            for position in line.positions:
              self._board_data[position].hovered = True
              self.__last_hovered.append(self._board_data[position])
            break
      elif self._board_data.count_pieces(Ring(self._board_data.turn)) > self._board_data.
                                                score[self._board_data.turn]:
        if self._board_data[notation_hovered] == Ring(self._board_data.turn):
          self.__last_hovered.append(self._board_data[notation_hovered])
          self._board_data[notation_hovered].hovered = True
      # If there no lines of priority player and the other player has lines
      elif any(line.color != self._board_data.turn for line in self._board_data.lines()):
        for line in self._board_data.lines():
          if line.color == self._board_data.turn: continue
          if notation_hovered in line.positions:
            for position in line.positions:
              self._board_data[position].hovered = True
              self.__last_hovered.append(self._board_data[position])
```

```
            break
        elif self._board_data.count_pieces(Ring(other_color)) > self._board_data.score[
                                                other_color]:
          if self._board_data[notation_hovered] == Ring(other_color):
            self.__last_hovered.append(self._board_data[notation_hovered])
            self._board_data[notation_hovered].hovered = True

      # PLACEMENT PHASE
      elif (self._board_data.turn == "white" and self._board_data.phase["white"] == Phase.
                                                placement)\
        or (self._board_data.turn == "black" and self._board_data.phase["black"] == Phase.
                                                placement):
        if self._board_data[notation_hovered] == None:
          draw_circle_alpha(surface=window,
                            color=(*(PIECE_WHITE if self._board_data.turn == "white" else
                                                PIECE_BLACK),128),
                            center=self._notation_to_screen(notation_hovered),
                            radius=self._sizes.ring_radius,
                            width=self._sizes.ring_thickness)

      # MOVEMENT PHASE
      else:
        # If no ring selected
        if self.__selected_ring_position == None:
          # Skip if not hovering ring of player's color
          if self._board_data[notation_hovered] == None: return
          if self._board_data[notation_hovered].type == "marker" or self._board_data[
                                                notation_hovered].color != self._board_data.
                                                turn: return

          # Set last hovered to currently hovered to be de-hovered later
          self.__last_hovered.append(self._board_data[notation_hovered])
          self._board_data[notation_hovered].hovered = True
        # If ring is selected
        else:
          if notation_hovered in self._board_data.possible_ring_moves(self.
                                                __selected_ring_position):
            # Preview where the ring will move to
            draw_circle_alpha(surface=window,
                              color=(*(PIECE_WHITE if self._board_data.turn == "white" else
                                                PIECE_BLACK),128),
                              center=self._notation_to_screen(notation_hovered),
                              radius=self._sizes.ring_radius,
                              width=self._sizes.ring_thickness)

    def __draw_hints(self, window):
      """Draw the possible intersections that the selected ring can move to.

      Args:
          window (pygame.Surface): The window that the game is in.
      """
      if self.__is_undoing: return
      if (self._board_data.turn == "white" and self._board_data.phase["white"] == Phase.
                                                movement)\
        or (self._board_data.turn == "black" and self._board_data.phase["black"] == Phase.
                                                movement):
          if self.__selected_ring_position != None:
            for possible_move in self._board_data.possible_ring_moves(self.
                                                __selected_ring_position):
              pygame.draw.circle(surface=window,
                                 color=BLACK,
                                 center=self._notation_to_screen(possible_move),
                                 radius=self._sizes.marker_radius * 0.3)

    def __on_click(self, position):
      """Called when the board is clicked. This is used for placing and moving rings, as well
                                                as removing lines and rings.

```

```
    Args:
        position (Tuple[int, int]): The position clicked.
    """
    notation_clicked = self.__nearest_notation(position)
    if notation_clicked == None: return

    # REMOVAL PHASE
    if self._board_data.phase["white"] == Phase.removal or self._board_data.phase["black"] =
                                                    = Phase.removal:
      # Remove line if a line is clicked
      for line in self._board_data.lines():
        if notation_clicked in line.positions:
          self.remove_line(line)
          break
      # Remove rings if a ring is clicked
      piece_clicked = self._board_data[notation_clicked]

      excess_white_rings = self._board_data.count_pieces(Ring("white")) - self._board_data.
                                                    score["white"]
      white_lines = sum(1 for line in self._board_data.lines() if line.color == "white")
      if piece_clicked == Ring("white"):
        if excess_white_rings > 0 and white_lines == 0:
          self.remove_ring(notation_clicked)

      excess_black_rings = self._board_data.count_pieces(Ring("black")) - self._board_data.
                                                    score["black"]
      black_lines = sum(1 for line in self._board_data.lines() if line.color == "black")
      if piece_clicked == Ring("black"):
        if excess_black_rings > 0 and black_lines == 0:
          self.remove_ring(notation_clicked)


    # PLACEMENT PHASE
    elif self._board_data.turn == "white" and self._board_data.phase["white"] == Phase.
                                                    placement:
      if self._board_data[notation_clicked] == None: self.place(notation_clicked, "white")
    elif self._board_data.turn == "black" and self._board_data.phase["black"] == Phase.
                                                    placement:
      if self._board_data[notation_clicked] == None: self.place(notation_clicked, "black")

    # MOVEMENT PHASE
    else:
      # If no ring selected
      if self.__selected_ring_position == None:
        if self._board_data[notation_clicked] != Ring(self._board_data.turn): return
        self._board_data[notation_clicked].selected = True
        self.__selected_ring_position = notation_clicked

      # If ring is already selected
      else:
        # If ring selected clicked again, deselect
        if self.__selected_ring_position == notation_clicked:
          self.__selected_ring_position = None
          self._board_data[notation_clicked].selected = False
        # Other positions clicked
        else:
          if notation_clicked not in self._board_data.possible_ring_moves(self.
                                                    __selected_ring_position): return
          self.move(self.__selected_ring_position, notation_clicked)
          self.__selected_ring_position = None

  def place(self, position, color):
    """
    Place a piece of the specified color at the given position on the board.

    Args:
      position (Tuple[int, int]): The position on the board where the piece should be placed
                                                    .
```

```
      color (str): The color of the piece to be placed.
    """
    self._board_data.place(position, color)
    self.__history.append(deepcopy(self._board_data))
    self.__moves.append(['place', color, position])

  def remove_ring(self, position):
    """
    Removes a ring from the board at the specified position.

    Args:
      position (Tuple[int, int]): The position of the ring to be removed.
    """
    self.__moves[-1][3] = position
    self._board_data.remove_ring(position)
    self.__history.append(deepcopy(self._board_data))

  def remove_line(self, line):
    """
    Removes a line from the board.

    Args:
      line (Line): The line to be removed from the board.
    """
    self.__moves.append(["remove", self._board_data.turn, line.positions, None])
    self._board_data.remove_line(line)

  def move(self, start, end):
    """Move a ring and initiate its animation.

    Args:
        start (Tuple[int, int]): The notation of the intersection where the ring was
                                 originally.
        end (Tuple[int, int]): The notation of the intersection that the ring is moving to.
    """
    self.__moves.append(["move", self._board_data.turn, start, end])

    new_board = self._board_data.move(start, end)

    self.__history.append(deepcopy(new_board))

    # Animate the ring moving
    self._board_data[end].start_moving(
      self._notation_to_screen(start),
      self._notation_to_screen(end)
    )

  def undo(self):
    """
    Undo the previous move on the board.
    """
    if self.__is_undoing == False:
      self.__is_undoing = True
      self.__focused_index = max(0, len(self.__history) - 1)
    if self.__focused_index > 0:
      self.__focused_index -= 1
    self.__peeked_board = deepcopy(self.__history[self.__focused_index])

  def redo(self):
    """
    Redo the previously undone move on the board.
    """
    if not self.__is_undoing: return
    self.__focused_index += 1
    if self.__focused_index == len(self.__history) - 1:
      self.__is_undoing = False
    else:
      move = self.__moves[self.__focused_index]
```

```
436
437         match move:
438           case "place", color, position:
439             self.__peeked_board.place(position, color)
440           case "move", color, start, end:
441             self.__peeked_board.move(start, end)
442             self.__peeked_board[end].start_moving(
443               self._notation_to_screen(start),
444               self._notation_to_screen(end)
445             )
446           case "remove", color, line, ring:
447             self.__peeked_board.remove_line(SimpleNamespace(**{"positions": line, "color":
                                                 color}))
448             self.__peeked_board.remove_ring(ring)
```

3.5 Board Elements

3.5.1 Rendering and Animations

A `Ring` object stores flags such as `__moving` or `__selected`, which is used in the update method to determine what style to render the ring as.

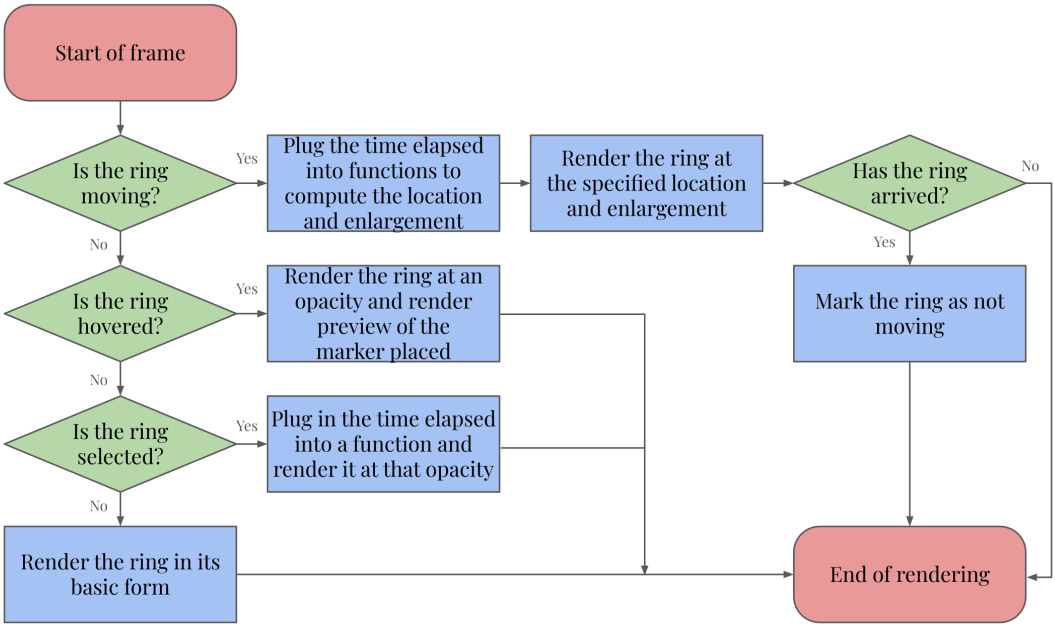


Figure 25: Flowchart of rendering a ring every frame

A `Marker` object uses a very similar logic to render itself in different states.

3.5.2 Ring Movement Animation

The ring's movement is based on two functions, *pos* and *scale*. They are calculated as follows:

$$x = vt$$

Where v is an arbitrary constant to alter the speed

$$y = -Ax(x - d) = -Avt(vt - d)$$

Where A is an arbitrary constant to alter to the max height of the movement, and d is the distance between the start and end points

$$scale = 1 + y = 1 - Avt(vt - d)$$

$pos(t)$ is just a linear interpolation from the coordinates of the starting point to the coordinates to the ending point, as the horizontal speed of projectiles are constant.

$$pos = \begin{pmatrix} x \\ y \end{pmatrix} = \left(\frac{t}{Period}\right)\begin{pmatrix} start_x \\ start_y \end{pmatrix} + \left(1 - \frac{t}{Period}\right)\begin{pmatrix} end_x \\ end_y \end{pmatrix}$$

3.5.3 Code

piece.py

```
import constants


class Piece:
  def __init__(self, color, type):
    self.color = color
    self.type = type

  @property
  def RGB(self):
    """Returns the color of the piece in RGB

    Returns:
        tuple(int, int, int): the color of the piece
    """
    if self.color == "white": return constants.PIECE_WHITE
    else: return constants.PIECE_BLACK

  def __eq__(self, value):
    if value == None: return False
    return self.color == value.color and self.type == value.type

  def __repr__(self):
    """Returns a string-based representation of the piece.

    Returns:
        str: the representation of the piece
    """
    s = f'{"W" if self.color == "white" else "B"} {self.type}'
    if self.hovered: s += ' H'
    if self.selected: s += ' S'
    if self.flipping: s += ' F'
    return s
```

ring.py

```
import math

import pygame

from util import Timer, draw_circle_alpha

from .piece import Piece


class Ring(Piece):
  def __init__(self, color):
    """Constructor of Ring

    Args:
        color (string): color of the ring
    """
    super(Ring, self).__init__(color, "ring")
    self.hovered = False
    self.__selected = False
    self.__moving = False
    self.__timer = Timer()

  @property
  def selected(self): return self.__selected
  @selected.setter
  def selected(self, val):
    self.__selected = val
    if val: self.__timer.reset()

```

```
  @property
  def moving(self): return self.__moving

  def start_moving(self, start, end):
    """Starts the moving animation of a ring

    Args:
        start (Tuple[int, int]): Starting positon of ring
        end (Tuple[int, int]): Ending position of ring
    """
    self.__timer.reset()
    self.__moving = True

    # Calculate position and scale functions
    distance = math.sqrt((start[0] - end[0])**2 + (start[1] - end[1])**2)
    velocity = 1
    period = distance/velocity
    diff = (end[0] - start[0], end[1] - start[1])

    def x(t): return velocity * t
    def y(t): return (-.5/distance) * x(t) * (x(t)-distance)
    def pos(t):
      if period - t <= 20: self.__moving = False
      return (start[0] + diff[0] * t/period, start[1] + diff[1] * t/period)

    self.pos = pos
    self.scale = lambda t: 1 + y(t) * 0.01

  def draw(self, window, position, sizes):
    """Draws the ring

    Args:
        window (pygame.Surface): The surface to draw the ring on
        position (Tuple[int, int]): The position to draw the ring at
        sizes (dict): Dictionary of sizes of elements to refer to
    """
    if self.moving:
      t = self.__timer.time()
      pygame.draw.circle(
        surface=window,
        color=self.RGB,
        center=self.pos(t),
        radius=sizes.ring_radius * self.scale(t),
        width=round(sizes.ring_thickness * self.scale(t))
      )
    elif self.selected:
      pygame.draw.circle(
        surface=window,
        color=self.RGB,
        center=position,
        radius=sizes.marker_radius
      )
      alpha = 150 + 105 * math.cos(0.0045 * self.__timer.time())
      draw_circle_alpha(surface=window,
                        color=(*self.RGB,alpha),
                        center=position,
                        radius=sizes.ring_radius,
                        width=sizes.ring_thickness)
    elif self.hovered:
      draw_circle_alpha(surface=window,
                        color=(*self.RGB,128),
                        center=position,
                        radius=sizes.marker_radius,
                        width=0)

      draw_circle_alpha(surface=window,
                        color=(*self.RGB,128),
                        center=position,
```

```
                        radius=sizes.ring_radius,
                        width=sizes.ring_thickness)
    else:
      pygame.draw.circle(
        surface=window,
        color=self.RGB,
        center=position,
        radius=sizes.ring_radius,
        width=sizes.ring_thickness
      )
```

marker.py

```
import pygame

from util import Timer, draw_circle_alpha

from .piece import Piece


class Marker(Piece):
  def __init__(self, color):
    """Constructor of Marker

    Args:
        color (string): color of the marker
    """
    super(Marker, self).__init__(color, "marker")
    self.__flipping = False
    self.hovered = False
    self.__timer = Timer()

  def start_flipping(self, start, end, delay):
    """Initiates the flipping animation

    Args:
        start (tuple(int, int, int)): the starting color
        end (tuple(int, int, int)): the ending color
        delay (int): how long it should wait after the function call to start animating
    """
    self.__timer.reset() # reset timer

    period = 1000

    def color_func(t): # colour function of the marker during flipping
      t -= delay
      if period - t <= 20: self.__flipping = False
      return end if t > period/2 else start

    def yscale_func(t): # function of the scaling of the marker during flipping
      t -= delay
      halfp = period/2
      if t < 0: return 1
      scaling = abs((t - halfp)/halfp)
      if scaling >= 1:
        self.__flipping = False
        scaling = 1
      return scaling

    self.color_func = color_func
    self.yscale_func = yscale_func
    self.__flipping = True

  def draw(self, window, position, sizes):
    """Draws the marker

    Args:
        window (pygame.Surface): Surface to draw on
```

```
        position (Tuple[int, int]): Position to draw at
        sizes (dict): Dictionary of sizes of elements to refer to
    """
    if self.__flipping:
      t = self.__timer.time()

      # Calculate positions
      rect = pygame.Rect(0,0,0,0)
      rect.height = sizes.marker_radius * 2
      rect.width = self.yscale_func(t) * sizes.marker_radius * 2
      rect.center = position

      # Draw marker
      pygame.draw.ellipse(
        surface=window,
        color=self.color_func(t),
        rect=rect
      )
    elif self.hovered:
      draw_circle_alpha(surface=window,
                        color=(*self.RGB,128),
                        center=position,
                        radius=sizes.marker_radius,
                        width=0)
    else:
      pygame.draw.circle(surface=window,
                         color=self.RGB,
                         center=position,
                         radius=sizes.marker_radius)
```

3.6 Computer Player

3.6.1 Finding the moves

To find the moves, the program has to look at what playing phase the player is in, then search moves possible for that player in that certain configuration of the board.

moves.py

```python
from copy import deepcopy

from board.piece import Ring
from constants import Phase


def reachable_states(board_data):
  """Returns all the reachable states from a given position.

  Args:
      board_data (BoardData): The current board position.

  Returns:
      [TypedDict("State", {"board": BoardData, "interaction": Callable[[BoardData], None]})
                                              ]: For each reachable state, there is an
                                              element for the board that will be resulting
                                              from that move, and a function that can be
                                              applied to the current board to bring it to
                                              the new position.
  """
  player = board_data.turn
  if board_data.phase[player] == Phase.placement:
    return reachable_placement_states(board_data, player)
  elif board_data.phase[player] == Phase.movement:
    return reachable_movement_states(board_data, player)
  elif board_data.phase[player] == Phase.removal:
    if sum(1 for line in board_data.lines() if line.color == player) > 0:
      return reachable_remove_line_states(board_data, player)
    else:
      return reachable_remove_ring_states(board_data, player)

def reachable_placement_states(board_data, player):
  """
  Generates a list of reachable placement states on the board for a given player.

  Args:
    board_data (BoardData): The current state of the board.
    player (str): The player for whom the reachable placement states are generated.

  Returns:
    list: A list of reachable placement states, where each state is a dictionary with the
                                              following keys:
      - "board": The new board state after placing a piece.
      - "interaction": The position where the piece was placed.
  """
  states = []
  for position in board_data.all_notations():
    if board_data[position] == None:
      new_board = deepcopy(board_data)
      new_board.place(position, player)
      state = deepcopy({
        "board": new_board,
        "interaction": deepcopy(position)
      })
      states.append(deepcopy(state))
  states = list(map(lambda state: {
    "board": state["board"],
    "interaction": lambda board: board.place(state["interaction"], player)
  }, states))
```

```
  return states

def reachable_movement_states(board_data, player):
  """
  Generates a list of reachable movement states for a given player on the board.

  Args:
    board_data (BoardData): The current state of the board.
    player (str): The player for whom reachable movement states are generated.

  Returns:
    list: A list of dictionaries representing the reachable movement states. Each dictionary
                                                contains two keys:
      - "board": The new board state after the movement.
      - "interaction": A lambda function that represents the movement interaction.

  """
  states = []
  for position in board_data.all_notations():
    if board_data[position] != Ring(player):
      continue
    for ring_move in board_data.possible_ring_moves(position):
      new_board = deepcopy(board_data)
      new_board.move(position, ring_move)
      states.append({
        "board": new_board,
        "interaction": deepcopy({
          "position": position,
          "ring_move": ring_move
        })
      })
  states = list(map(lambda state: {
    "board": state["board"],
    "interaction": lambda board: board.move(
      deepcopy(state["interaction"]["position"]),
      deepcopy(state["interaction"]["ring_move"])
    )
  }, states))
  return states

def reachable_remove_line_states(board_data, player):
  """
  Generates a list of reachable states by removing a line of the specified player.

  Args:
    board_data (BoardData): The current state of the game board.
    player (str): The player whose lines should be removed.

  Returns:
    list: A list of dictionaries representing the reachable states. Each dictionary contains
                                                the following keys:
      - 'board': The new board state after removing the line.
      - 'interaction': The line that was removed.

  """
  states = []
  for line in board_data.lines():
    if line.color != player:
      continue
    new_board = deepcopy(board_data)
    for position in line.positions:
      new_board[position] = None
    new_board.remove_line(line)
    states.append({
      "board": new_board,
      "interaction": deepcopy(line)
    })
  states = list(map(lambda state: {
```

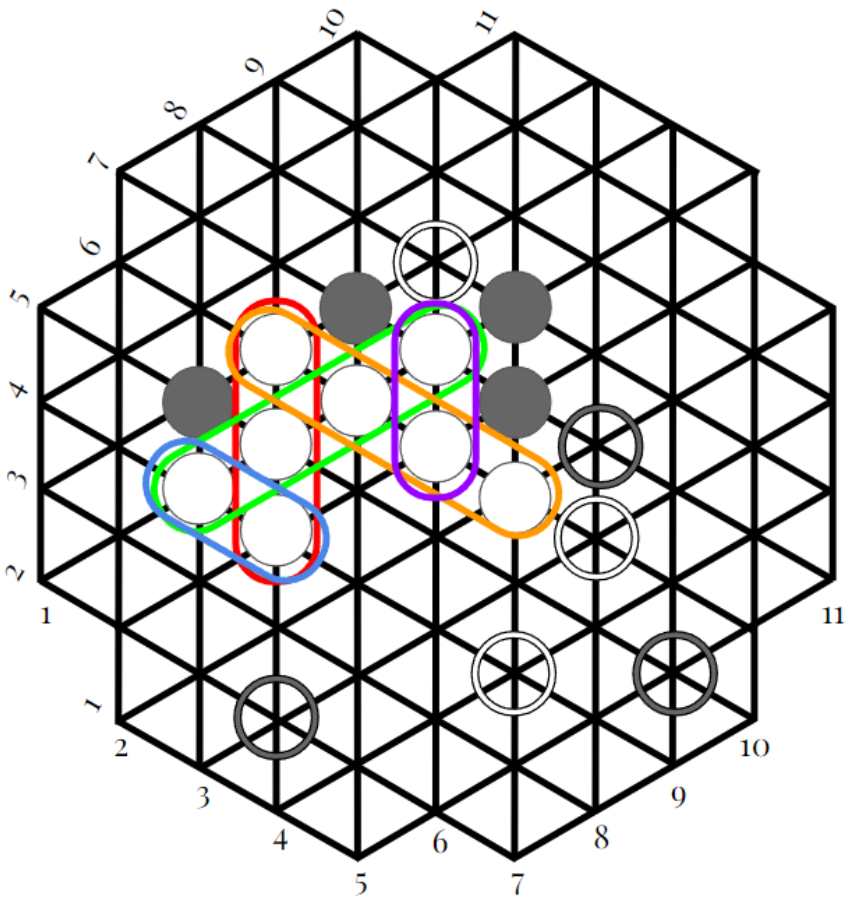
```
    "board": state["board"],
    "interaction": lambda board: board.remove_line(deepcopy(state["interaction"]))
  }, states))
  return states

def reachable_remove_ring_states(board_data, player):
  """
  Generates a list of reachable states by removing a ring for a given player.

  Args:
    board_data (Board): The current state of the game board.
    player (str): The player whose ring is being removed.

  Returns:
    list: A list of dictionaries representing the reachable states. Each dictionary contains
                                                    the following keys:
      - "board": The new state of the game board after removing the ring.
      - "interaction": The position of the removed ring.

  """
  states = []
  for position in board_data.all_notations():
    if board_data[position] != Ring(player):
      continue
    new_board = deepcopy(board_data)
    new_board.remove_ring(position)
    states.append({
      "board": new_board,
      "interaction": deepcopy(position)
    })
  states = list(map(lambda state: {
    "board": state["board"],
    "interaction": lambda board: board.remove_ring(deepcopy(state["interaction"]))
  }, states))
  return states
```

3.6.2 Evaluating a Position

There are many components of the evaluation of a position.

1. Scoring connected markers

Lines are searched for, and a score is added to the evaluation depending on the length of each line. Here is an example board:



The score of a line is exponentially correlated with the length of the connected markers. This is because a line of 4 is much easier to complete than a line of 3, and a line of 5 can already be removed and does not have to be protected like other lines.

2. Number of rings left

The fewer rings the player has left, the more likely they are to win. When the player only has 2 rings left, then this means the player has won. Hence, the weighting for 2 rings left is very large such that the computer AI will always choose to win if possible.

3. Positioning of rings

Rings are most useful in the middle of the board, as it can be used to block more moves and access more spaces on the board. The number of moves that a ring can make will also be counted and multiplied by a constant, which measures how active the ring is. The more spaces a ring can reach, the more versatile and active it is.

4. Number of markers

The more markers a player has on the board, the more advantageous it is to them. However, this is very heuristic, as markers can be easily flipped over by the opponent. Hence, this has a relatively small weight in the whole evaluation, but still serves to guide the computer player to make a move that generally improves the board.

evaluation.py

```
from board.piece import Marker, Ring

hashed = {}

line_length_scores = dict([
  (1, 0),
  (2, 30),
  (3, 500),
  (4, 2000),
  (5, 100000)
])
ring_left_scores = dict([
  (1, 1e20),
  (2, 1e19),
  (3, 1e14),
  (4, 1e9),
  (5, 0)
])
ring_distance_scores = dict([
  (0, 1),
  (1, .8),
  (2, .7),
  (3, .7),
  (4, .5),
  (5, .2)
])
# The multiplier to the number of possible moves by that ring
ring_activity_multiplier = 0.01

# The multiplier to the number of markers
marker_multiplier = 0.1

def evaluate(board_data):
  """Evalutes a board configuration.

  Args:
      board_data (BoardData): The board.

  Returns:
      float: The evaluation score.
  """
  if str(board_data) in hashed:
    return hashed[str(board_data)]
  total_score = 0
  total_score += evaluate_lines(board_data)
  total_score += evaluate_rings_left(board_data)
  total_score += evaluate_ring_position(board_data)
  total_score += evaluate_markers_score(board_data)
  hashed[str(board_data)] = total_score
  return total_score

def evaluate_lines(board_data):
  """Evaluates the score of the lines formed.

  Args:
      board_data (BoardData): The board.

  Returns:
      float: The evaluation score.
  """
  score = 0
  for start in board_data.all_notations():
    if board_data[start] == None:
      continue
    multiplier = 1 if board_data[start].color == "white" else -1

```

```
    if board_data[start].type == "ring":
      multiplier *= 0.1
    for direction in [(1, 0), (1, 1), (0, 1)]:
      length = 1
      for steps in range(1, 5):
        searching = (start[0]+direction[0]*steps, start[1]+direction[1]*steps)
        # If searching out of range
        if searching not in board_data.all_notations():
          break
        at_searching = board_data[searching]
        # If position being searched is empty, or not the same color
        if at_searching == None or at_searching.color != board_data[start].color:
          break
        elif board_data[searching].type == "ring":
          multiplier *= 0.1
        length += 1
      if length != 1:
        score += multiplier * line_length_scores[length]

  return score

def evaluate_rings_left(board_data):
  """Evaluate the score based on the number of rings left.

  Args:
      board_data (BoardData): The board.

  Returns:
      float: The evaluation score.
  """
  return ring_left_scores[board_data.score["white"]] - ring_left_scores[board_data.score["
                                          black"]]

def evaluate_ring_position(board_data):
  """Evaluates the score based on the positioning of rings.

  Args:
      board_data (BoardData): The board.

  Returns:
      float: The evaluation score.
  """
  score = 0
  for position in board_data.all_notations():
    if board_data[position] == Ring("white"):
      distance = max(abs(position[0]-6), abs(position[1]-6), abs(position[0] - position[1]))
      score += ring_distance_scores[distance]
      posible_moves = board_data.possible_ring_moves(position)
      score += ring_activity_multiplier * len(posible_moves)
    elif board_data[position] == Ring("black"):
      distance = max(abs(position[0]-6), abs(position[1]-6), abs(position[0] - position[1]))
      score -= ring_distance_scores[distance]
      posible_moves = board_data.possible_ring_moves(position)
      score -= ring_activity_multiplier * len(posible_moves)

  return score

def evaluate_markers_score(board_data):
  """Evaluates the score based on the number of markers.

  Args:
      board_data (BoardData): The board.

  Returns:
      float: The evaluation score.
  """
  return (board_data.count_pieces(Marker("white")) - board_data.count_pieces(Marker("black")
                                          )) * marker_multiplier
```

3.6.3 Minimax Algorithm

Refer to Section 2.3.6 and Section 1.4.1 for the explanation of the minimax algorithm. When first implementing the minimax algorithm, if the computer player explores all possible moves to a depth of 2-3, the time it takes to make a move is very long. This is because the branching factor of the game is very large, and the time complexity of minimax is $O(b^d)$, where b is the branching factor and d is the depth. Even after using alpha-beta pruning, the time it takes to make a move is still very long. To heuristically optimize this, each reachable board at a depth will be evaluated, then sorted and only the top 5 will be explored further. This is because the top 5 moves are the most likely to be the best moves, and the rest of the moves are not worth exploring as they are very unlikely to be the best move. This is a trade-off between the time it takes to make a move and the quality of the move.

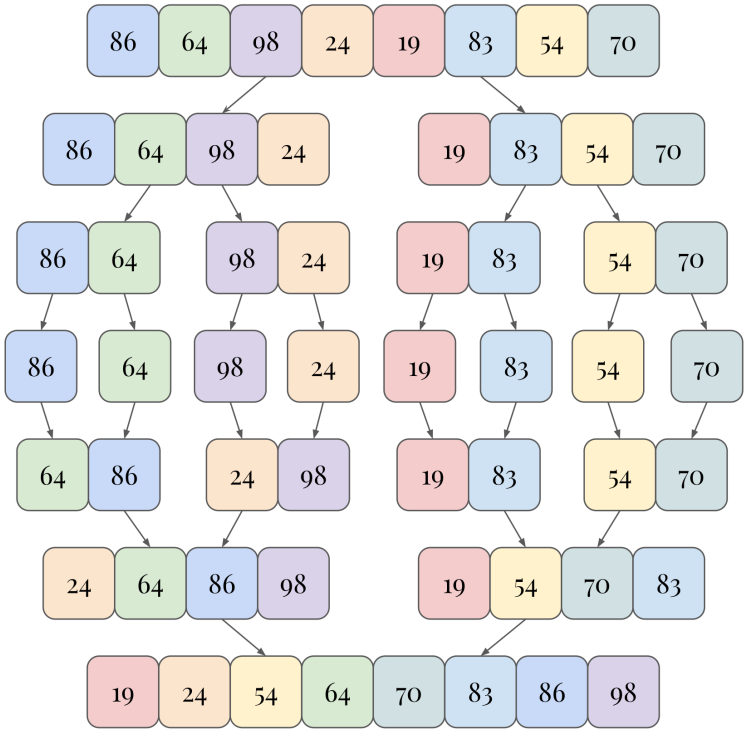
minimax.py

```
from brain.evaluation import evaluate
from brain.merge_sort import merge_sort
from brain.moves import reachable_states


# Alpha = highest guarenteed score for computer
# Beta = lowest guarentee score for opponent
def minimax(board_data, depth, player, alpha = -1e20, beta = 1e20):
  """Calculate the best move given the current board configuration for a certain player.

  Args:
      board_data (BoardData): The board
      depth (int): The minimax depth; how many moves to search ahead.
      player (string): Which player is playing.
      alpha (float, optional): Alpha value used for alpha beta pruning. Defaults to -1e20.
      beta (float, optional): Beta value used for alpha beta pruning. Defaults to 1e20.

  Returns:
      TypedDict("State", {"board": BoardData, "interaction": Callable[[BoardData], None]}):
                                          The board that will be resulting from the best
                                           move, and a function that can be applied to
                                          the current board to bring it to the new state
                                          .
  """
  is_maximising = player == "white"
  other_player = "black" if player == "white" else "white"

  if depth == 0:
    # f = open("log.txt", "a")
    # f.write(str(board_data))
    # f.write('\n')
    # f.write(str(evaluate(board_data)))
    # f.write('\n')
    return {
      "board": board_data,
      "interaction": lambda _: None
    }

  if is_maximising:
    best_evaluation = -1e20
    best_state = None
    states = reachable_states(board_data)
    states = merge_sort(states, key=lambda state: evaluate(state["board"]), reverse=True)

    for state in states[0:5]:
      new_evaluation = evaluate(minimax(state["board"], depth - 1, other_player, alpha, beta
                                          )["board"])
      if new_evaluation > best_evaluation:
        best_evaluation = new_evaluation
        best_state = state
      # If the opponent have the choice to make us have a lower score in another tree
                                          reguardless, then stop searching
      if best_evaluation > beta:
```

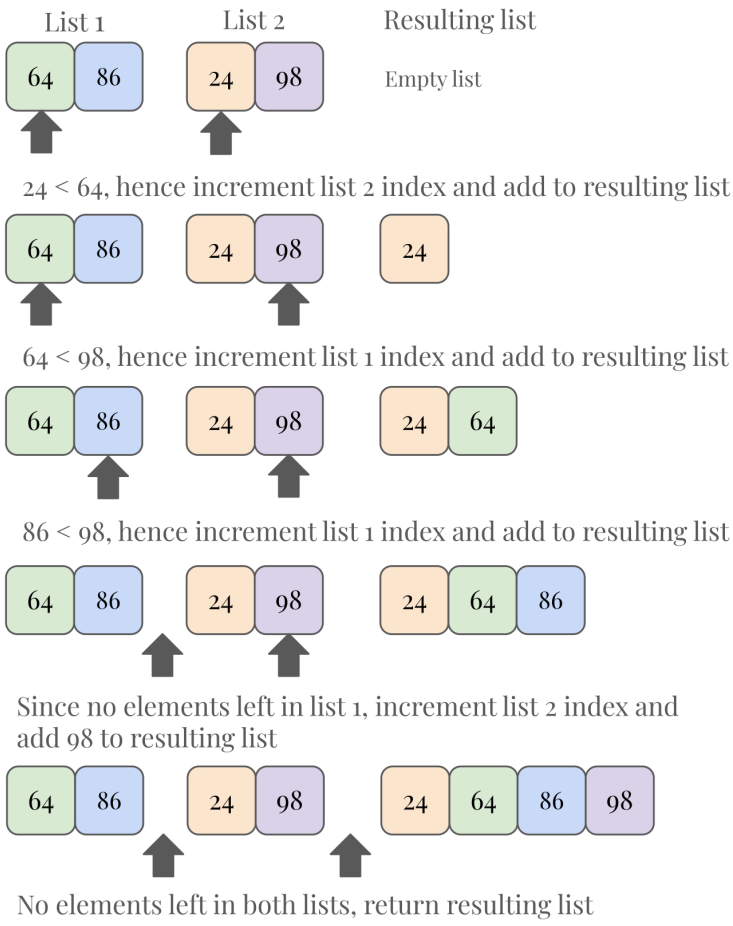
```
        break
      alpha = max(alpha, best_evaluation)
    return best_state
  else:
    # Worst as in worst for the computer player
    worst_evaluation = 1e20
    worst_state = None
    states = reachable_states(board_data)
    states = merge_sort(states, key=lambda state: evaluate(state["board"]))
    for state in states[0:5]:
      new_evaluation = evaluate(minimax(state["board"], depth - 1, other_player, alpha, beta
                                        )["board"])
      if new_evaluation < worst_evaluation:
        worst_evaluation = new_evaluation
        worst_state = state
      # If the we have the choice to get a higher score in another tree reguardless, then
                                        stop searching
      if worst_evaluation < alpha:
        break
      beta = min(beta, worst_evaluation)
    return worst_state
```

3.6.4 Merge Sort

To sort the evaluations of the reachable boards, I have implemented a merge sort algorithm. This is because merge sort has a time complexity of $O(n \log n)$, which is the best time complexity for sorting algorithms. The algorithm is also recursive, which is elegant and easy to understand and implement. A merge sort algorithm divides the list to be sorted in half, then recursively calls itself on the two halves. When the two halves are sorted, they can be merged together in linear time. Here is an example of the splitting and merging of a list to sort it using merge sort:



This how two lists are merged together:



merge_sort.py

```
def merge(list1, list2, key, reverse):
  """Merge two sorted lists into one sorted list.

  Args:
      list1 (list): The first list
      list2 (list): The second list
      key (function): The function applied to each element that returns a numerical value
                                          which elements will be compared with.
      reverse (bool): If True, the list will be sorted in descending order.

  Returns:
      list: A sorted list containing all the elements of list1 and list2
  """
  sorted_list = []
  list1_index, list2_index = 0, 0

  # Fill the sorted_list with the elements of list1 and list2
  while list1_index < len(list1) and list2_index < len(list2):
    factor = -1 if reverse else 1
    if key(list1[list1_index]) * factor < key(list2[list2_index]) * factor:
      sorted_list.append(list1[list1_index])
      list1_index += 1
    else:
      sorted_list.append(list2[list2_index])
```

```
      list2_index += 1

  # If list1 contains more elements
  while list1_index < len(list1):
    sorted_list.append(list1[list1_index])
    list1_index += 1

  # If list2 contains more elements
  while list2_index < len(list2):
    sorted_list.append(list2[list2_index])
    list2_index += 1

  return sorted_list

def merge_sort(lst, key = lambda x: x, reverse = False):
  """Sorts the list using the merge sort algorithm.

  Args:
      lst (list): The list to sort.
      key (function): The function applied to each element that returns a numerical value
                                              which elements will be compared with.
      reverse (bool): If True, the list will be sorted in descending
  """
  if len(lst) <= 1: return lst

  middle = (len(lst))//2

  left_subarray = merge_sort(lst[:middle], key, reverse)
  right_subarray = merge_sort(lst[middle:], key, reverse)

  return merge(left_subarray, right_subarray, key, reverse)
```

3.7 UI Elements

3.7.1 Scene

The scene class is an **abstract** class with no implementation details. This is to provide any scenes with template to follow.
scene.py

```
from abc import ABC, abstractmethod

class Scene(ABC):
  @abstractmethod
  def __init__(self):
    """Constructor of a scene
    """
    pass

  @abstractmethod
  def update(self, window, events):
    """Updates the scene.

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    pass
```

3.7.2 SceneManager

The `SceneManager` is a static class that is used to change between scenes. To avoid circular imports between scenes, the update methods of each scene will return a string if the scene needs to be changed. The string is then matched by the `SceneManager`, and changes to the suitable scene.
scene_manager.py

```
from scenes.title_scene import TitleScene


class SceneManager():
  __active_scene = TitleScene()

  @staticmethod
  def update(window, events):
    """Updates the scene.

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    SceneManager.__active_scene.update(window, events)

  @staticmethod
  def set_active_scene(scene):
    """Sets the active scene.

    Args:
        scene (Scene): The scene to be the new active scene.
    """
    SceneManager.__active_scene = scene
```

3.7.3 Button

This is the general button where the size, image, and text displayed on it can be customized. Notice that some configuration methods in the class return the object itself. This is so that builder notation can be

used to construct the button(see line 32-43 of `game_scene.py` and line 38-52 of `title_scene.py`). This makes it more clear which values have what meaning and easier to read and maintain.

button.py

```
import pygame


class Button():
  def __init__(self):
    self.enabled = True
    self.held = False
    self.__on_click = []
    self.__render_normal = None
    self.__render_hovered = None
    self.__render_clicked = None

    self.__image = None
    self.__text = None

  def set_pos(self, x, y):
    """Set the positon of the button.

    Args:
        x (float): The x-coordinate of the button.
        y (float): The y-coordinate of the button.

    Returns:
        Button: The object itself.
    """
    self.x, self.y = x, y
    return self

  def set_size(self, w, h):
    """Set the size of the button.

    Args:
        w (float): Width of the button.
        h (float): Height of the button.

    Returns:
        Button: The object itself.
    """
    self.w, self.h = w, h
    return self

  def set_image(self, image):
    """Set the image of the button.

    Args:
        image (pygame.Surface): The image loaded onto a surface.

    Returns:
        Button: The object itself.
    """
    self.__image = pygame.transform.scale(image, (self.w, self.h))
    self.__image.convert()
    self.__render_normal = lambda window: (self.__image.set_alpha(155),
                                           window.blit(self.__image, (self.x, self.y)))
    self.__render_hovered = lambda window: (self.__image.set_alpha(205),
                                           window.blit(self.__image, (self.x, self.y)))
    self.__render_clicked = lambda window: (self.__image.set_alpha(255),
                                           window.blit(self.__image, (self.x, self.y+5)))
    self.__render_deactivated = lambda window: (self.__image.set_alpha(50),
                                               window.blit(self.__image, (self.x, self.y)))
    return self

  def set_text(self, text, font, color):
```

```
    """Set the text displayed on the button.

    Args:
        text (string): The string to be displayed.
        font (python.Font): The font of the text.
        color (Tuple[int, int, int]): Color of the text.

    Returns:
        Button: The object itself.
    """
    self.__text = text
    self.__font = font
    self.__text_color = color
    return self

  def on_click(self, func):
    self.__on_click.append(func)
    return self

  def update(self, window, events):
    """Updates the button

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    mouse_pos = pygame.mouse.get_pos()
    hovered = mouse_pos[0] >= self.x and mouse_pos[0] <= self.x + self.w and\
              mouse_pos[1] >= self.y and mouse_pos[1] <= self.y + self.h

    for event in events:
      if event.type == pygame.MOUSEBUTTONDOWN:
        if hovered:
          if self.held == False: self.__clicked()
          self.held = True
      elif event.type == pygame.MOUSEBUTTONUP:
        self.held = False

    if not self.enabled: self.__render_deactivated(window)
    elif self.held: self.__render_clicked(window)
    elif hovered: self.__render_hovered(window)
    else: self.__render_normal(window)

    if self.__text != None:
      text_x = self.x + self.w//2 - self.__font.size(self.__text)[0]//2
      text_y = self.y + self.h//2 - self.__font.size(self.__text)[1]//2
      if self.held:
        text_y += 5
      text_surf = self.__font.render(self.__text, True, self.__text_color)
      window.blit(text_surf, (text_x, text_y))

  def __clicked(self):
    """Called when the button is clicked
    """
    if self.enabled:
      for func in self.__on_click:
        func()
```

3.7.4 NavButton

This is inherited from the base class `Button` and is used for the navigational buttons to switch between scenes. This is so that buttons of similar styles will not have to be repeatedly defined, hence reducing code redundancies.

nav_button.py

```
import os
```

```python
import pygame

from constants import *
from gameUI.button import Button


class NavButton(Button):
  def __init__(self):
    """Constructor of the default button for navigation in menus.
    """
    super().__init__()
    self.set_size(450, 125)
    self.set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/nav_button.png")))

  def set_text(self, text):
    """Set the text of the button.

    Args:
        text (string): The string to be displayed.

    Returns:
        NavButton: the object itself.
    """
    return super().set_text(text, jose_font_bold50, (255, 255, 255))
```

3.8 Game Loop Management

Since I have chosen to use Pygame to be able to draw elements manually for elements, there are no utility functions for managing different scenes and elements. Hence, there is only one main thread at a time and the control has to be passed back and forth carefully between different elements to draw themselves onto the screen, and handle events that concern them. The main game loop is shown in the figure below.

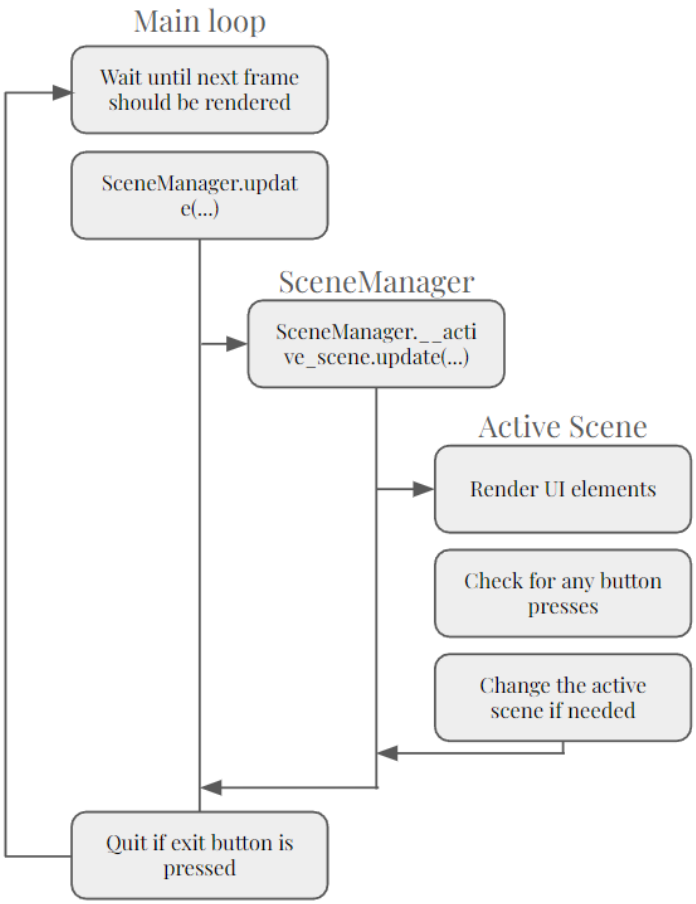


Figure 26: The main game loop

3.9 Scenes

As mentioned in the last section, we rely on the scene manager to be able to switch between scenes. Each scene has an update function that is called every function, and this is what the game scene's update function does behind the scenes. Here is how scenes transition to each other:

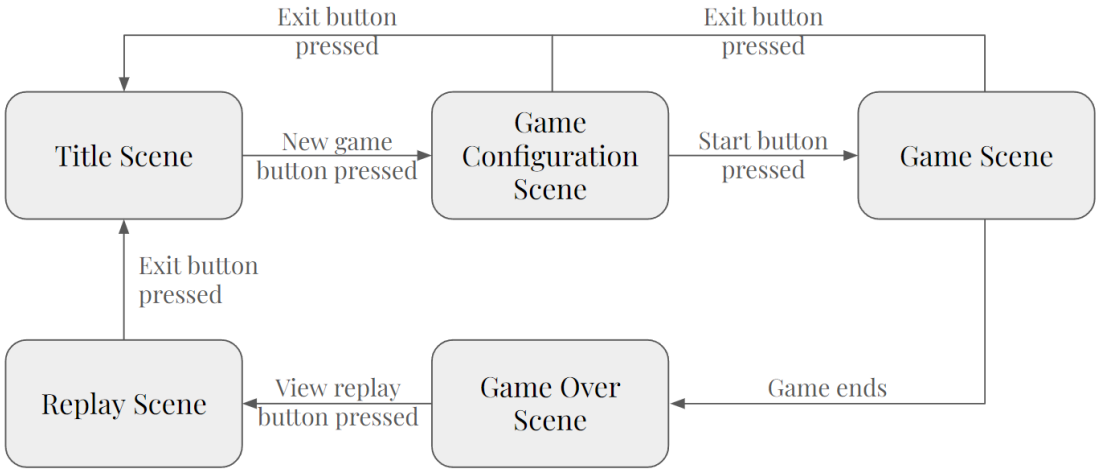


Figure 27: A state transition diagram of switching between scenes

All scenes must inherit from the abstract class `Scene` (see Section 3.7.1). They must **override** the constructor, and the update method. Using **polymorphism**, this allows the Scene manager to call the same `update` function on scenes, but expect different behaviours. Here is how the scenes are implemented.

3.9.1 Title Scene

title_scene.py

```
0  import json
1  import os
2  import pygame
3  from board.interactive_board import InteractiveBoard
4
5  from board.piece import Ring
6  from board.piece.marker import Marker
7  from board.renderable_board import RenderableBoard
8  from constants import BACKGROUND
9  from gameUI.button import Button
10 from gameUI.nav_button import NavButton
11 from gameUI.scene import Scene
12 from util import draw_circle_alpha
13 from constants import *
14 from tkinter import filedialog
15
16 UPDATE_BOARD = pygame.USEREVENT+1
17
18 class TitleScene(Scene):
19   def __init__(self):
20     """Constructor of the title scene
21     """
22     self.__decorative_board = RenderableBoard(1050, 70, 500)
23
24     pygame.time.set_timer(UPDATE_BOARD, 1000)
25
26     self.ring_positions = [(6, 3), (3, 3), (3, 6), (6, 9), (9, 9), (9, 6)]
27     self.index = 0
28
```

```
    for pos in [(3, 6), (9, 9), (6, 3)]:
      self.__decorative_board.get_active_board()[pos] = Ring("white")

    for pos in [(6, 9), (6, 6), (9, 6)]:
      self.__decorative_board.get_active_board()[pos] = Ring("black")

    for pos in [(4, 3), (5, 3), (4, 7), (5, 8), (9, 7), (9, 8)]:
      self.__decorative_board.get_active_board()[pos] = Marker("black")

    for pos in [(3, 4), (3, 5), (7, 4), (8, 5), (7, 9), (8, 9)]:
      self.__decorative_board.get_active_board()[pos] = Marker("white")

    def new_game():
      from gameUI.scene_manager import SceneManager
      from scenes.game_config_scene import GameConfigScene
      SceneManager.set_active_scene(GameConfigScene())

    def import_game():
      file_name = filedialog.askopenfile(mode="r", defaultextension="json", filetypes=[("
                                         JSON files", "*.json")], initialdir=os.path.
                                         join(os.getcwd(), "/save")).name
      try:
        with open(file=file_name, mode="r") as f:
          json_data = json.load(f)

          from gameUI.scene_manager import SceneManager
          from scenes.replay_scene import ReplayScene
          SceneManager.set_active_scene(ReplayScene(json_data["white"], json_data["black"],
                                         list(map(InteractiveBoard.string_to_move,
                                         json_data["moves"]))))
      except: pass

    self.__new_game_button = NavButton()\
                  .set_pos(525, 570)\
                  .set_text("New Game")\
                  .on_click(new_game)

    self.__import_button = NavButton()\
                  .set_pos(525, 770)\
                  .set_text("Import Game")\
                  .on_click(import_game)

    self.__cross_button = Button()\
                  .set_pos(1480, 20)\
                  .set_size(40, 40)\
                  .set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/
                                         cross_button.png")))\
                  .on_click(pygame.quit)


  def __static_elements(self, window, events):
    """Renders the static elements of the title screen

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    pygame.draw.rect(window, BACKGROUND, window.get_rect())
    window.blit(jose_font_bold100.render('THE YINSH', True, (255, 255, 255)), (430, 75))
    window.blit(jose_font_bold100.render('B  ARD GAME', True, (255, 255, 255)), (380, 230))
    draw_circle_alpha(window, (0, 0, 0, 40), (-70, 190), 120, 10)
    draw_circle_alpha(window, (0, 0, 0, 80), (120, 150), 100, 10)
    draw_circle_alpha(window, (0, 0, 0, 120), (285, 170), 80, 10)
    draw_circle_alpha(window, (0, 0, 0, 160), (405, 225), 60, 10)
    draw_circle_alpha(window, (0, 0, 0, 200), (478, 270), 40, 10)
    if any(event.type == UPDATE_BOARD for event in events):
      self.__decorative_board.get_active_board().move(self.ring_positions[self.index % 6],
                                         self.ring_positions[(self.index+1) % 6])
```

```
91          self.index -= 1
92
93      def update(self, window, events):
94          """Updates the title scene
95
96          Args:
97              window (pygame.Scene): The window that the game is in
98              events (list): List of events
99          """
100         self.__static_elements(window, events)
101         self.__decorative_board.update(window, events)
102         self.__new_game_button.update(window, events)
103         self.__import_button.update(window, events)
104         self.__cross_button.update(window, events)
```

3.9.2 Game Scene

Every time the scene is updated, the scene stores which player's turn it is last frame, then compare it to the whose turn it is in the current frame. If it has changed, it then tests whether the current player is a computer player. If it is a computer player, then the minimax algorithm is engaged and ran in a **separate thread** so that the update functions can still be called to make the software stay responsive.

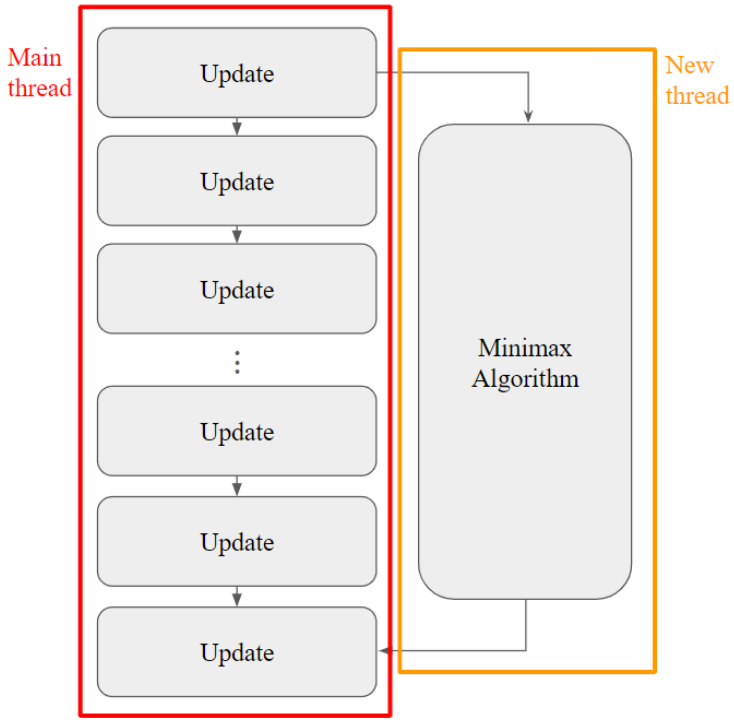


Figure 28: Diagram to demonstrate multithreading

game_scene.py

```
0  import os
1  from threading import Thread
2  from time import sleep
3
4  import pygame
5
6  from board.interactive_board import InteractiveBoard
```

```
from board.piece import Ring
from constants import *
from gameUI.button import Button
from gameUI.scene import Scene
from brain.minimax import minimax
from util import Timer, draw_circle_alpha


class GameScene(Scene):
  def __init__(self, white_name, black_name, is_white_AI, is_black_AI, white_AI_difficulty,
                                              black_AI_difficulty, white_time_lim,
                                              black_time_lim):
    """Constructor of the GameScene

    Args:
        white (string): The name of the white player
        black (string): The name of the black player
        is_white_AI (bool): Whether the white player is a computer player
        is_black_AI (bool): Whether the black player is a computer player
        white_time_lim (int): The time that the white player has in seconds
        black_time_lim (_type_): The time that the black player has in seconds
    """
    self.__white_name, self.__black_name, self.__is_white_AI, self.__is_black_AI =
                                              white_name, black_name, is_white_AI,
                                              is_black_AI
    self.__board = InteractiveBoard(150, 80, 800)

    self.__undo_button = Button()\
                         .set_pos(1180, 860)\
                         .set_size(80, 80)\
                         .set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/
                                              left_button.png")))\
                         .on_click(self.__board.undo)
    self.__redo_button = Button()\
                         .set_pos(1290, 860)\
                         .set_size(80, 80)\
                         .set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/
                                              right_button.png")))\
                         .on_click(self.__board.redo)

    def exit_to_title():
      """Exit to title scene
      """
      from gameUI.scene_manager import SceneManager
      from scenes.title_scene import TitleScene
      SceneManager.set_active_scene(TitleScene())
    self.__exit_button = Button()\
                         .set_pos(1480, 20)\
                         .set_size(40, 40)\
                         .set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/
                                              cross_button.png")))\
                         .on_click(exit_to_title)
    self.__white_time_left = white_time_lim
    self.__black_time_left = black_time_lim
    self.__elapsed_move_time = Timer()
    self.__last_turn = "black"
    self.__minimaxing = False
    self.__white_minimax_depth = 3 if white_AI_difficulty == "Hard" else 2 if
                                              white_AI_difficulty == "Medium" else 1
    self.__black_minimax_depth = 3 if black_AI_difficulty == "Hard" else 2 if
                                              black_AI_difficulty == "Medium" else 1

  def __draw_rings_removed(self, window, events):
    """Renders the rings that are removed from the board on the left side of the screen

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
```

```
    """
    white_pos = [(75, 105), (75, 230), (75, 355)]
    black_pos = [(75, 855), (75, 730), (75, 605)]
    for pos in white_pos:
      draw_circle_alpha(window, (193, 208, 214, 100), pos, 50, 0)

    for pos in black_pos:
      draw_circle_alpha(window, (41, 46, 48, 100), pos, 50, 0)

    if self.__board.get_active_board().phase["white"] != Phase.placement:
      for i in range(5-self.__board.get_active_board().count_pieces(Ring("white"))):
        pygame.draw.circle(
          surface=window,
          color=WHITE,
          center=white_pos[i],
          radius=35,
          width=8
        )

    if self.__board.get_active_board().phase["black"] != Phase.placement:
      for i in range(5-self.__board.get_active_board().count_pieces(Ring("black"))):
        pygame.draw.circle(
          surface=window,
          color=BLACK,
          center=black_pos[i],
          radius=35,
          width=8
        )

  def __draw_player_names(self, window, events):
    """Displays the names of the players

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    # Seperation line
    pygame.draw.line(
      surface=window,
      color=PURE_WHITE,
      start_pos=(1280, 100), end_pos=(1280, 250),
      width=5
    )

    # Player names
    white_name = jose_font_reg30.render(self.__white_name, True, PURE_WHITE)
    black_name = jose_font_reg30.render(self.__black_name, True, BLACK)
    white_name_size = jose_font_reg30.size(self.__white_name)

    window.blit(white_name, (1250 - white_name_size[0], 120))
    window.blit(black_name, (1310, 120))

  def __draw_time_info(self, window, events):
    """Renders the time left for each player and the time spent on the current turn

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    # Time left per player
    turn = self.__board.get_active_board().turn

    white_time_total_secs = self.__white_time_left
    black_time_total_secs = self.__black_time_left

    if turn == "white":
      white_time_total_secs -= self.__elapsed_move_time.time()//1000
    else:
```

```
      black_time_total_secs -= self.__elapsed_move_time.time()//1000

    white_time_rawtext = f"{white_time_total_secs//60}:{(white_time_total_secs%60):0>{2}}"
    black_time_rawtext = f"{black_time_total_secs//60}:{(black_time_total_secs%60):0>{2}}"

    white_time = jose_font_reg50.render(white_time_rawtext, True, PURE_WHITE)
    black_time = jose_font_reg50.render(black_time_rawtext, True, BLACK)
    white_time_size = jose_font_reg50.size(white_time_rawtext)

    window.blit(white_time, (1250 - white_time_size[0], 180))
    window.blit(black_time, (1310, 180))

    # Time spent this turn
    elapsed_time_rawtext = f"{self.__elapsed_move_time.time()//1000//60}:{(self.
                                            __elapsed_move_time.time()//1000%60):0>{2}}"
    elapsed_time = jose_font_reg50.render(elapsed_time_rawtext, True, PURE_WHITE if turn ==
                                            "white" else BLACK)
    elapsed_time_size = jose_font_reg50.size(elapsed_time_rawtext)
    window.blit(elapsed_time, (1280-elapsed_time_size[0]/2, 350))

  def __draw_instruction(self, window, events):
    """Displays the instruction for the player and the status of the game

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    turn = self.__board.get_active_board().turn

    # Instruction for player
    status_rawtext = ""

    if turn == "white":
      status_rawtext += f"White({self.__white_name}) "
    else:
      status_rawtext += f"Black({self.__black_name}) "
    if (turn == "white" and self.__is_white_AI) or (turn == "black" and self.__is_black_AI):
      status_rawtext += "is thinking..."
    elif self.__board.get_active_board().phase[turn] == Phase.placement:
      status_rawtext += "to place a ring."
    elif self.__board.get_active_board().phase[turn] == Phase.movement:
      status_rawtext += "to move a ring."
    elif self.__board.get_active_board().phase[turn] == Phase.removal:
      status_rawtext += "to remove a line\nand a ring."

    status_text = jose_font_reg20.render(status_rawtext, True, PURE_WHITE if turn == "white"
                                            else BLACK)
    status_text_size = jose_font_reg20.size(status_rawtext)
    window.blit(status_text, (1280-status_text_size[0]/2, 300))

  def __draw_history(self, window, events):
    """Renders the history of the game on the right side of the screen

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    moves = self.__board.get_moves()

    is_undoing = self.__board.get_is_undoing()
    focused_index = self.__board.get_focused_index() + 1
    if not is_undoing: focused_index = len(moves)

    white_string = []
    black_string = []
    for i, move in enumerate(moves):
      color = move[1]
      if color == "white":
```

```
        white_string.append(f'{i+1}. {InteractiveBoard.move_to_string(move)}')
        if move[0] == "remove": black_string.append('')
      elif color == "black":
        black_string.append(f'{i+1}. {InteractiveBoard.move_to_string(move)}')
        if move[0] == "remove": white_string.append('')

    focused_row = 0

    for i, string in enumerate(white_string):
      if string != '' and focused_index == int(string.split('.')[0]): focused_row = i

    for i, string in enumerate(black_string):
      if string != '' and focused_index == int(string.split('.')[0]): focused_row = i

    top_row = max(0, focused_row-13)
    bottom_row = max(focused_row, top_row + 13)

    white_string = white_string[top_row:bottom_row + 1]
    black_string = black_string[top_row:bottom_row + 1]

    for i, string in enumerate(white_string):
      if string == '': continue
      font = jose_font_reg20
      if focused_index == int(string.split('.')[0]): font = jose_font_reg20u
      render = font.render(string, True, PURE_WHITE)

      window.blit(render, (1170, 430 + i * 30))

    for i, string in enumerate(black_string):
      if string == '': continue
      font = jose_font_reg20
      if focused_index == int(string.split('.')[0]): font = jose_font_reg20u
      render = font.render(string, True, BLACK)

      window.blit(render, (1330, 430 + i * 30))

  def __draw_UI(self, window, events):
    """Draw the UI of the game scene

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    pygame.draw.rect(window, BACKGROUND, window.get_rect())
    pygame.draw.rect(window, FOREGROUND, (1000,window.get_rect()[1],window.get_rect()[2]-
                                          1000,window.get_rect()[3]))

    self.__draw_rings_removed(window, events)
    self.__draw_player_names(window, events)
    self.__draw_time_info(window, events)
    self.__draw_instruction(window, events)
    self.__draw_history(window, events)

    self.__undo_button.update(window, events)
    self.__redo_button.update(window, events)
    self.__exit_button.update(window, events)

    self.__board.update(window, events)

  def __run_computer_player(self, window, events):
    """Run the computer player(s), if it is their turn

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events

    Returns:
        str: The turn after the computer player has made a move
```

```
266     """
267     new_turn = self.__board.get_active_board().turn
268 
269     if self.__last_turn != new_turn:
270       if self.__last_turn == "white":
271         self.__white_time_left -= self.__elapsed_move_time.time() // 1000
272       else:
273         self.__black_time_left -= self.__elapsed_move_time.time() // 1000
274       self.__elapsed_move_time.reset()
275 
276     minimax_depth = self.__white_minimax_depth if new_turn == "white" else self.
                                                  __black_minimax_depth
277 
278     if (new_turn == "white" and self.__is_white_AI) or (new_turn == "black" and self.
                                                  __is_black_AI):
279       if not self.__minimaxing:
280         self.__minimaxing = True
281         self.__board.set_interactable(False)
282         def make_move():
283           new_state = minimax(self.__board.get_active_board(), minimax_depth, self.__board.
                                                  get_active_board().turn)
284           new_state["interaction"](self.__board)
285           self.__minimaxing = False
286           self.__board.set_interactable(True)
287         thread = Thread(target=make_move)
288         thread.start()
289     if (new_turn == "white" and self.__is_white_AI and self.__board.get_active_board().phase
                                                  ["white"] == Phase.removal) or\
290         (new_turn == "black" and self.__is_black_AI and self.__board.get_active_board().
                                                  phase["black"] == Phase.removal):
291       if not self.__minimaxing:
292         self.__minimaxing = True
293         self.__board.set_interactable(False)
294         def make_move():
295           sleep(2)
296           new_state = minimax(self.__board.get_active_board(), minimax_depth, self.__board.
                                                  get_active_board().turn)
297           new_state["interaction"](self.__board)
298           self.__minimaxing = False
299           self.__board.set_interactable(True)
300         thread = Thread(target=make_move)
301         thread.start()
302 
303 
304     return new_turn
305 
306   def __check_game_over(self, window, events):
307     """Check if the game is over and if so, display the game over scene
308 
309     Args:
310         window (pygame.Scene): The window that the game is in
311         events (list): List of events
312     """
313     is_game_over = False
314     win_message = ''
315 
316     turn = self.__board.get_active_board().turn
317 
318     white_time_total_secs = self.__white_time_left
319     black_time_total_secs = self.__black_time_left
320 
321     if turn == "white":
322       white_time_total_secs -= self.__elapsed_move_time.time()//1000
323     else:
324       black_time_total_secs -= self.__elapsed_move_time.time()//1000
325 
326     if white_time_total_secs <= 0:
327       is_game_over = True
```

```
      win_message = f'{self.__black_name} has won on time!'
    if black_time_total_secs <= 0:
      is_game_over = True
      win_message = f'{self.__white_name} has won on time!'
    if self.__board.get_active_board().score["white"] <= 2:
      is_game_over = True
      win_message = f'{self.__white_name} has won by removing 3 lines!'
    if self.__board.get_active_board().score["black"] <= 2:
      is_game_over = True
      win_message = f'{self.__black_name} has won by removing 2 lines!'

    if is_game_over:
      from scenes.replay_scene import ReplayScene
      replay_scene = ReplayScene(self.__white_name, self.__black_name, self.__board.
                                                  get_moves())
      from gameUI.scene_manager import SceneManager
      from scenes.game_over_scene import GameOverScene
      SceneManager.set_active_scene(GameOverScene(win_message, self.__board.get_active_board
                                                  (), replay_scene))

  def update(self, window, events):
    """Updates the game scene

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    self.__draw_UI(window, events)
    new_turn = self.__run_computer_player(window, events)
    self.__last_turn = new_turn
    self.__check_game_over(window, events)
```

game_config_scene.py

```
import os

import pygame
import thorpy as tp

from constants import *
from gameUI.button import Button
from gameUI.nav_button import NavButton
from gameUI.scene import Scene
from util import draw_rect_alpha


class GameConfigScene(Scene):
  def __init__(self):
    """Constructor of the start game scene
    """
    self.__white_name_input = tp.TextInput(text="", placeholder="Player 1")
    self.__black_name_input = tp.TextInput(text="", placeholder="Player 2")
    self.__white_time_input = tp.TextInput(text="", placeholder="600")
    self.__black_time_input = tp.TextInput(text="", placeholder="600")
    self.__white_is_ai = tp.Checkbox(value=False)
    self.__black_is_ai = tp.Checkbox(value=False)
    self.__white_ai_difficulty = tp.TogglablesPool(label="", choices=("Easy", "Medium", "
                                                Hard"), initial_value="Medium")
    self.__black_ai_difficulty = tp.TogglablesPool(label="", choices=("Easy", "Medium", "
                                                Hard"), initial_value="Medium")

    self.__white_name_input.set_topleft(665, 220)
    self.__white_name_input.set_size((300, 55))
    self.__white_name_input.stop_if_too_large = True
    self.__white_name_input.set_bck_color = WHITE

    self.__black_name_input.set_topleft(1055, 220)
    self.__black_name_input.set_size((300, 55))
```

```
32      self.__black_name_input.stop_if_too_large = True
33      self.__black_name_input.set_bck_color = WHITE
34
35      self.__white_time_input.set_only_numbers()
36      self.__white_time_input.stop_if_too_large = True
37      self.__white_time_input.set_topleft(690, 365)
38      self.__white_time_input.set_size((300, 55))
39
40      self.__black_time_input.set_only_numbers()
41      self.__black_time_input.stop_if_too_large = True
42      self.__black_time_input.set_topleft(1085, 365)
43      self.__black_time_input.set_size((300, 55))
44
45      self.__white_is_ai.set_center(710, 520)
46      self.__white_is_ai.set_size((40, 40))
47
48      self.__black_is_ai.set_center(1120, 520)
49      self.__black_is_ai.set_size((40, 40))
50
51      self.__white_ai_difficulty.set_center(730, 640)
52      self.__black_ai_difficulty.set_center(1115, 640)
53
54      self.__ui_group = tp.Group(elements=[self.__white_name_input, self.__black_name_input,
                                             self.__white_is_ai, self.__black_is_ai, self.
                                             __white_time_input, self.__black_time_input,
                                             self.__white_ai_difficulty, self.
                                             __black_ai_difficulty], mode = None)
55
56
57      def start():
58        """starts the game
59        """
60        from gameUI.scene_manager import SceneManager
61        from scenes.game_scene import GameScene
62        SceneManager.set_active_scene(GameScene(
63          self.__white_name_input.value or 'Player 1',
64          self.__black_name_input.value or 'Player 2',
65          self.__white_is_ai.value,
66          self.__black_is_ai.value,
67          self.__white_ai_difficulty.get_value(),
68          self.__black_ai_difficulty.get_value(),
69          int(self.__white_time_input.value or '600'),
70          int(self.__black_time_input.value or '600')
71        ))
72
73      self.__start_button = NavButton()\
74                            .set_text('Start')\
75                            .set_pos(1536/2 - 450/2, 790)\
76                            .on_click(start)
77
78      def exit():
79        """Exits to the title scene
80        """
81        from gameUI.scene_manager import SceneManager
82        from scenes.title_scene import TitleScene
83        SceneManager.set_active_scene(TitleScene())
84      self.__exit_button = Button()\
85                           .set_pos(1480, 20)\
86                           .set_size(40, 40)\
87                           .set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/
                                                         cross_button.png")))\
88                           .on_click(exit)
89
90    def update(self, window, events):
91      """Updates the title scene
92
93      Args:
94          window (pygame.Scene): The window that the game is in
```

```
        events (list): List of events
    """
    pygame.draw.rect(window, BACKGROUND, window.get_rect())
    draw_rect_alpha(window, (*PURE_WHITE, 100), (1536/2-250, 150, 400, window.get_rect().
                                                 height - 370))
    draw_rect_alpha(window, (*BLACK, 100), (1536/2+150, 150, 400, window.get_rect().height -
                                                 370))
    title = jose_font_bold50.render('Game Setup', True, PURE_WHITE)
    title_length = jose_font_bold50.size('Game Setup')[0]
    window.blit(title, (1536/2 - title_length/2, 50))

    self.__start_button.update(window, events)
    self.__exit_button.update(window, events)

    player_text = jose_font_reg40.render('Player Name', True, PURE_WHITE)
    window.blit(player_text, (260, 225))

    time_text = jose_font_reg40.render('Time Limit', True, PURE_WHITE)
    window.blit(time_text, (310, 370))

    time_text = jose_font_reg40.render('Computer Player?', True, PURE_WHITE)
    window.blit(time_text, (180, 495))

    time_text = jose_font_reg40.render('Computer Difficulty', True, PURE_WHITE)
    window.blit(time_text, (150, 630))

    self.__ui_group.get_updater().update(events=events)
```

game_over_scene.py

```
from board.renderable_board import RenderableBoard
from gameUI.nav_button import NavButton
from gameUI.scene import Scene
from constants import *


class GameOverScene(Scene):
  def __init__(self, win_message, ending_board_data, replay_scene):
    """Constructor of GameOverScene.

    Args:
        win_message (str): The message to be displayed in the scene.
        ending_board_data (BoardData): The board data at the end of the game.
        replay_scene (ReplayScene): The replay scene to be displayed when the replay button
                                                 is clicked.
    """
    self.__win_message = win_message
    self.__board = RenderableBoard(1536/2-350, 110, 700)
    self.__board.set_board_data(ending_board_data)

    from gameUI.scene_manager import SceneManager
    self.__replay_button = NavButton()\
                           .set_text('Watch Replay')\
                           .on_click(lambda: SceneManager.set_active_scene(replay_scene))\
                           .set_pos(1536/2-450/2, 800)

  def update(self, window, events):
    """Updates the game over scene

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    pygame.draw.rect(window, BACKGROUND, window.get_rect())

    win_message = jose_font_reg50.render(self.__win_message, True, PURE_WHITE)
    win_message_length = jose_font_reg50.size(self.__win_message)[0]
    window.blit(win_message, (1536/2-win_message_length/2, 50))
```

```
37
38      self.__replay_button.update(window, events)
39      self.__board.update(window, events)
```

replay_scene.py

```
0  import json
1  import os
2  import pygame
3
4  from datetime import datetime
5  from tkinter import filedialog
6  from board.interactive_board import InteractiveBoard
7  from board.piece import Ring
8  from constants import *
9  from gameUI.button import Button
10 from gameUI.scene import Scene
11 from util import draw_circle_alpha
12
13
14 class ReplayScene(Scene):
15   def __init__(self, white_name, black_name, moves):
16     """Constructor of the replay scene
17
18     Args:
19         white_name (str): The name of the white player
20         black_name (str): The name of the black player
21         moves (List[List]): The list of moves that were made in the game
22     """
23     self.__white_name, self.__black_name = white_name, black_name
24     self.__board = InteractiveBoard(150, 80, 800)
25     self.__board.import_moves(moves)
26     self.__board.set_interactable(False)
27
28     self.__undo_button = Button()\
29                          .set_pos(1100, 860)\
30                          .set_size(80, 80)\
31                          .set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/
                                                left_button.png")))\
32                          .on_click(self.__board.undo)
33     self.__redo_button = Button()\
34                          .set_pos(1210, 860)\
35                          .set_size(80, 80)\
36                          .set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/
                                                right_button.png")))\
37                          .on_click(self.__board.redo)
38
39     def save():
40       """Promt the user to select a file and saves the game to that file
41       """
42       current_time = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
43       save_file = filedialog.asksaveasfile(
44         mode="w",
45         defaultextension=".json",
46         filetypes=[("JSON files", "*.json")],
47         initialdir=os.path.join(os.getcwd(), "saves"),
48         initialfile=f"{current_time}.json",
49         title="Save game"
50       )
51       data = {
52         "white": self.__white_name,
53         "black": self.__black_name,
54         "moves": list(map(lambda x: InteractiveBoard.move_to_string(x, True), self.__board.
                                                get_moves())),
55       }
56       try:
57         with open(save_file.name, "w") as file:
58           json.dump(data, file, indent=2)
```

```
      except: pass

    self.__save_button = Button()\
                         .set_pos(1400, 860)\
                         .set_size(80, 80)\
                         .set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/
                                                      save_button.png")))\
                         .on_click(save)

    def exit():
      """Exit to title scene
      """
      from gameUI.scene_manager import SceneManager
      from scenes.title_scene import TitleScene
      SceneManager.set_active_scene(TitleScene())
    self.__exit_button = Button()\
                         .set_pos(1480, 20)\
                         .set_size(40, 40)\
                         .set_image(pygame.image.load(os.path.join(os.getcwd(), "assets/
                                                      cross_button.png")))\
                         .on_click(exit)

  def __draw_rings_removed(self, window, events):
    """Renders the rings that are removed from the board on the left side of the screen

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    white_pos = [(75, 105), (75, 230), (75, 355)]
    black_pos = [(75, 855), (75, 730), (75, 605)]
    for pos in white_pos:
      draw_circle_alpha(window, (193, 208, 214, 100), pos, 50, 0)

    for pos in black_pos:
      draw_circle_alpha(window, (41, 46, 48, 100), pos, 50, 0)

    if self.__board.get_active_board().phase["white"] != Phase.placement:
      for i in range(5-self.__board.get_active_board().count_pieces(Ring("white"))):
        pygame.draw.circle(
          surface=window,
          color=WHITE,
          center=white_pos[i],
          radius=35,
          width=8
        )

    if self.__board.get_active_board().phase["black"] != Phase.placement:
      for i in range(5-self.__board.get_active_board().count_pieces(Ring("black"))):
        pygame.draw.circle(
          surface=window,
          color=BLACK,
          center=black_pos[i],
          radius=35,
          width=8
        )

  def __draw_player_names(self, window, events):
    """Displays the names of the players

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    # Seperation line
    pygame.draw.line(
      surface=window,
      color=PURE_WHITE,
```

```
      start_pos=(1280, 100), end_pos=(1280, 170),
      width=5
    )

    # Player names
    white_name = jose_font_reg30.render(self.__white_name, True, PURE_WHITE)
    black_name = jose_font_reg30.render(self.__black_name, True, BLACK)
    white_name_size = jose_font_reg30.size(self.__white_name)

    window.blit(white_name, (1250 - white_name_size[0], 120))
    window.blit(black_name, (1310, 120))

  def __draw_history(self, window, events):
    """Renders the history of the game on the right side of the screen

    Args:
        window (pygame.Scene): The window that the game is in
        events (list): List of events
    """
    moves = self.__board.get_moves()

    is_undoing = self.__board.get_is_undoing()
    focused_index = self.__board.get_focused_index() + 1
    if not is_undoing: focused_index = len(moves)

    white_string = []
    black_string = []
    for i, move in enumerate(moves):
      color = move[1]
      if color == "white":
        white_string.append(f'{i+1}. {InteractiveBoard.move_to_string(move)}')
        if move[0] == "remove": black_string.append('')
      elif color == "black":
        black_string.append(f'{i+1}. {InteractiveBoard.move_to_string(move)}')
        if move[0] == "remove": white_string.append('')

    focused_row = 0

    for i, string in enumerate(white_string):
      if string != '' and focused_index == int(string.split('.')[0]): focused_row = i

    for i, string in enumerate(black_string):
      if string != '' and focused_index == int(string.split('.')[0]): focused_row = i

    top_row = max(0, focused_row-18)
    bottom_row = max(focused_row, top_row + 18)

    white_string = white_string[top_row:bottom_row + 1]
    black_string = black_string[top_row:bottom_row + 1]

    for i, string in enumerate(white_string):
      if string == '': continue
      font = jose_font_reg20
      if focused_index == int(string.split('.')[0]): font = jose_font_reg20u
      render = font.render(string, True, PURE_WHITE)

      window.blit(render, (1170, 230 + i * 30))

    for i, string in enumerate(black_string):
      if string == '': continue
      font = jose_font_reg20
      if focused_index == int(string.split('.')[0]): font = jose_font_reg20u
      render = font.render(string, True, BLACK)

      window.blit(render, (1330, 230 + i * 30))

  def __draw_UI(self, window, events):
    """Draw the UI of the game scene
```

```
193
194     Args:
195         window (pygame.Scene): The window that the game is in
196         events (list): List of events
197     """
198     pygame.draw.rect(window, BACKGROUND, window.get_rect())
199     pygame.draw.rect(window, FOREGROUND, (1000,window.get_rect()[1],window.get_rect()[2]-
                                              1000,window.get_rect()[3]))
200
201     self.__draw_rings_removed(window, events)
202     self.__draw_player_names(window, events)
203     self.__draw_history(window, events)
204
205     self.__undo_button.update(window, events)
206     self.__redo_button.update(window, events)
207     self.__exit_button.update(window, events)
208     self.__save_button.update(window, events)
209
210     self.__board.update(window, events)
211
212   def update(self, window, events):
213     """Updates the replay scene
214
215     Args:
216         window (pygame.Scene): The window that the game is in
217         events (list): List of events
218     """
219
220     self.__draw_UI(window, events)
```

4 Testing

4.1 Development Testing

Most of the testing done during the development of the software is just to run the game, play it through, and check a few edge cases and fix any errors that arise in the process.

4.1.1 Minimax

However, for the minimax algorithm, since there is so much going on behind the scene, if it is not working properly, there might not be any errors, but the computer will be making nonsensical moves. This is why when a recursive call of the minimax function reaches its base case, it logs out the board state using the string representation function mentioned in Section 3.3.6 and the evaluation associated with it. This allows me to see what boards have been evaluated and chosen, and check that the evaluation function accurately scores the board.

```
def minimax(board_data, depth, player, alpha = -1e20, beta = 1e20):
  ...

  if depth == 0:
    f = open("log.txt", "a")
    f.write(str(board_data))
    f.write('\n')
    f.write(str(evaluate(board_data)))
    f.write('\n')
    return {
      "board": board_data,
      "interaction": lambda _: None
    }

  ...
```

4.1.2 Game Interaction

To test that all the functions used to interact with the game is working properly, I have developed a few test cases that is run every so often during development to make sure I haven't made any breaking changes to the program.

```
from types import SimpleNamespace
import pygame

pygame.init()

import unittest

from board.interactive_board import InteractiveBoard
from board.piece.marker import Marker
from board.piece.ring import Ring


class TestInteractiveBoard(unittest.TestCase):
  def test_placing(self):
    board1 = InteractiveBoard(0, 0, 500)

    for notation in board1._board_data.all_notations():
      board1 = InteractiveBoard(0, 0, 500)
      board1.place(notation, "white")
      board2 = InteractiveBoard(0, 0, 500)
      board2._board_data[notation] = Ring("white")
      self.assertEqual(str(board1._board_data), str(board2._board_data))

  def test_moving1(self):
    board1 = InteractiveBoard(0, 0, 500)
    board1.place((6, 6), "white")
```

```
    board1.place((2, 2), "black")
    board1.move((6, 6), (7, 6))

    board2 = InteractiveBoard(0, 0, 500)
    board2._board_data[(7, 6)] = Ring("white")
    board2._board_data[(2, 2)] = Ring("black")
    board2._board_data[(6, 6)] = Marker("white")
    self.assertEqual(str(board1._board_data), str(board2._board_data))

  def test_moving2(self):
    board1 = InteractiveBoard(0, 0, 500)
    board1.place((10, 10), "black")
    board1.move((10, 10), (10, 5))

    board2 = InteractiveBoard(0, 0, 500)
    board2._board_data[(10, 5)] = Ring("black")
    board2._board_data[(10, 10)] = Marker("black")
    self.assertEqual(str(board1._board_data), str(board2._board_data))

  def test_removing_line(self):
    board1 = InteractiveBoard(0, 0, 500)
    line = [(2, 3), (2, 4), (2, 5), (2, 6), (2, 7)]
    board1.place((2, 2), "white")
    last_pos = (2, 2)
    for notation in line:
      board1.move(last_pos, notation)
      last_pos = notation
    board1.remove_line(SimpleNamespace(**{"positions": [(2, 2), (2, 3), (2, 4), (2, 5), (2,
                                          6)], "color": "white"}))

    board2 = InteractiveBoard(0, 0, 500)
    board2._board_data[(2, 7)] = Ring("white")
    self.assertEqual(str(board1._board_data), str(board2._board_data))

  def test_removing_ring(self):
    board1 = InteractiveBoard(0, 0, 500)
    line = [(2, 3), (2, 4), (2, 5), (2, 6), (2, 7)]
    board1.place((2, 2), "white")
    last_pos = (2, 2)
    for notation in line:
      board1.move(last_pos, notation)
      last_pos = notation
    board1.remove_line(SimpleNamespace(**{"positions": [(2, 2), (2, 3), (2, 4), (2, 5), (2,
                                          6)], "color": "white"}))
    board1.remove_ring((2, 7))

    board2 = InteractiveBoard(0, 0, 500)
    self.assertEqual(str(board1._board_data), str(board2._board_data))

unittest.main()
```

```
(.venv) C:\Users\Eden\Desktop\NEA\yinsh>py tests.py
pygame 2.5.0 (SDL 2.28.0, Python 3.10.11)
Hello from the pygame community. https://www.pygame.org/contribute.html
.....
----------------------------------------------------------------------
Ran 5 tests in 0.236s

OK
```

4.2 Final Tests

4.2.1 Objective 1

The user should have the choice of playing against another human player, or against the computer. (`https://youtu.be/bFUECCLeMh8?t=0`)

#	Input	Output	Test Result
1	Press the "New Game" button in the title screen	A configuration screen appears	Test Passed
2	Change the time limit for each player	The time limit for each player is changed	Test Passed
3	Set white to be a computer player	The game starts with the computer moving as white first	Test Passed
4	Set black to be a computer player	The game starts with the computer moving as black second after the user makes a move as white	Test Passed
5	Set white and black to be human players	The game allows the user to make moves as both white and black	Test Passed
6	Change the difficulty of the computer player to easy	Moves should be done quickly but naively compared to medium or hard	Test Passed

4.2.2 Objective 2

The user should be able to play a game of YINSH. (`https://youtu.be/bFUECCLeMh8?t=136`)

#	Input	Output	Test Result
7	Hover over an empty intersection during placement phase	A ghost ring appears on the intersection	Test Passed
8	Click on an occupied intersection during placement phase	The ring is not placed on the intersection	Test Passed
9	Click on an empty intersection during placement phase	The ring is placed on the intersection	Test Passed
10	Click on an empty intersection when it is the computer's turn	No ring is placed	Test passed
11	Hover over a ring during movement phase	The ring is highlighted	Test Passed
12	Click on a ring of the opposite colour during movement phase	The ring is not selected	Test Passed
13	Click on a ring of the turn's colour during movement phase	The ring is selected, and valid moves are displayed	Test Passed
14	Click on the ring after selecting it during movement phase	The ring is deselected	Test Passed
15	Click on an invalid intersection (occupied by a ring, or not in a straight line) after selecting a ring during movement phase	The ring is not moved	Test Passed
16	Click on a valid intersection after selecting a ring during movement phase	The ring is moved to the intersection. The movement should be smoothly animated, and the markers in between should display a flipping animation	Test Passed
17	Create a line and hover over the line	The line of 5 markers are highlighted	Test Passed
18	After a line is formed, click on markers not part of the line	The markers are not removed	Test Passed
19	Click on a marker in the line	The 5 markers in the line is removed	Test Passed
20	After removing a line, click on a ring of the opposite colour	The ring is not removed	Test Passed

21	After removing a line, click on a ring of the colour of the line	The ring is removed	Test Passed
22	Create two lines at the same time and hover on different markers on the line	Only 5 markers in a line should be highlighted	Test Passed
23	Remove 3 lines of a player	The game ends, and a game ended screen is shown, displaying who won and why	Test Passed

4.2.3 Objective 3

The software should show the user relevant information in the game screen.
(`https://youtu.be/bFUECCLeMh8?t=413`)

#	Input	Output	Test Result
24	Start a game	The game displays what players/computer is playing the current game	Test Passed
25	Start a game and make a few moves	The game displays whose turn it is to play and who should place a ring	Test Passed
26	Place 5 rings for each player and make a few moves	The game displays whose turn it is to move a ring	Test Passed
27	Create a line of 5 markers	The game prompts the users to remove that line and remove a ring	Test Passed
28	Remove a ring in the game	The game displays that a ring was removed	Test Passed
29	Start a game with white to be a computer player	When it is the computer player's turn, the game displays that it is thinking	Test Passed
30	Make a few moves in a game	The game displays a table of all the previous moves of the game in algebraic notation	Test Passed
31	Click on the undo button	The game undoes the last move	Test Passed
32	Click on the redo button	The game redoes the last move undone	Test Passed
33	Lose by waiting out the time limit of a player	The game displays who won and that the other player has lost on time	Test Passed
34	Remove 3 lines for a player	The game displays who won by removing 3 lines, and a replay is offered	Test Passed

4.2.4 Objective 4

The software should be able to import and export games and replay them.
(`https://youtu.be/bFUECCLeMh8?t=622`)

#	Input	Output	Test Result
35	In the game ending screen, click on the replay button	The replay screen appears	Test Passed
36	When replaying a game, press on the save button	The game opens a file dialog window for the player to select where to save their game to and what to name it	Test Passed
37	Select a location to save the game	The game's moves are exported into a JSON file	Test Passed
38	Click on the import button in the title screen	A file dialog window appears for the user to select a JSON file	Test Passed
39	Select a JSON file to import	A replay screen appears, and the game displays the final state of the board	Test Passed

40	Import a game from a JSON file with a few moves	The moves of the game are displayed and listed in algebraic notation	Test Passed
41	In the replay screen, click on the undo button	The game undoes the previous move	Test Passed
42	In the replay screen, click on the redo button	The game redoes the previous move undone	Test Passed

5 Evaluation

5.1 Objective 1

The user should have the choice of playing against another human player, or against the computer.

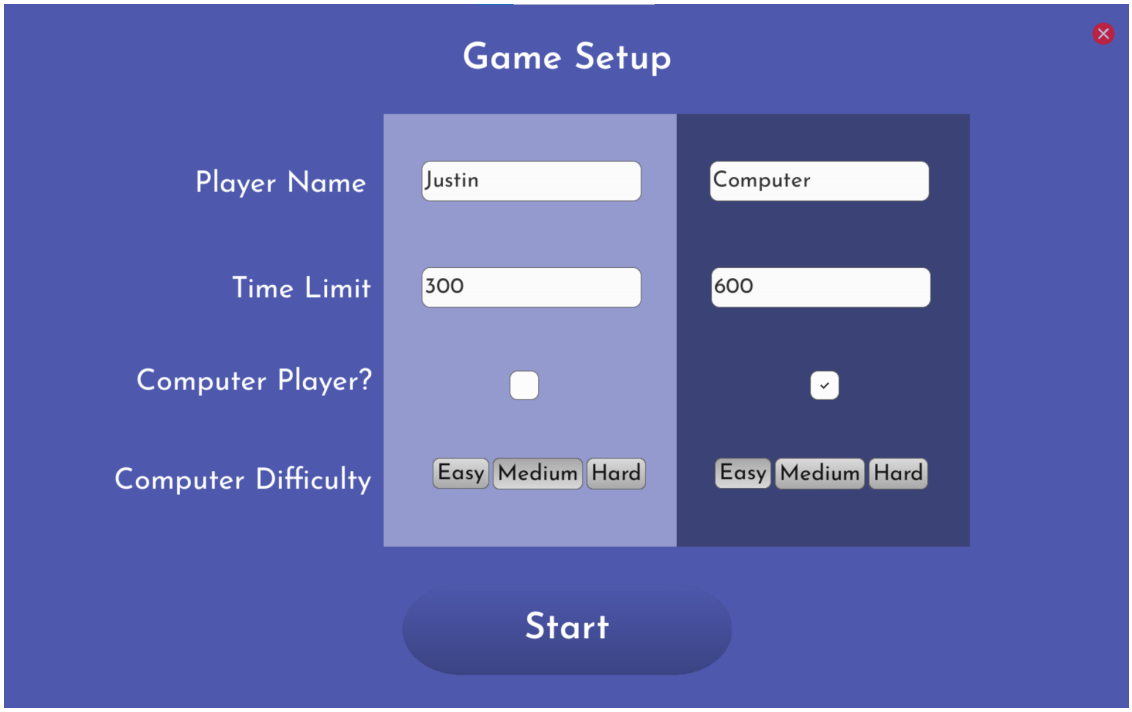


Figure 29: The game configuration screen

The game configuration screen allows the user to change the time limit for each player, select which player is a human player or a computer, and select the difficulty of the computer. The user can also select which player plays as which colour. This screen successfully meets objective 1.

5.2 Objective 2

The user should be able to play a game of YINSH.

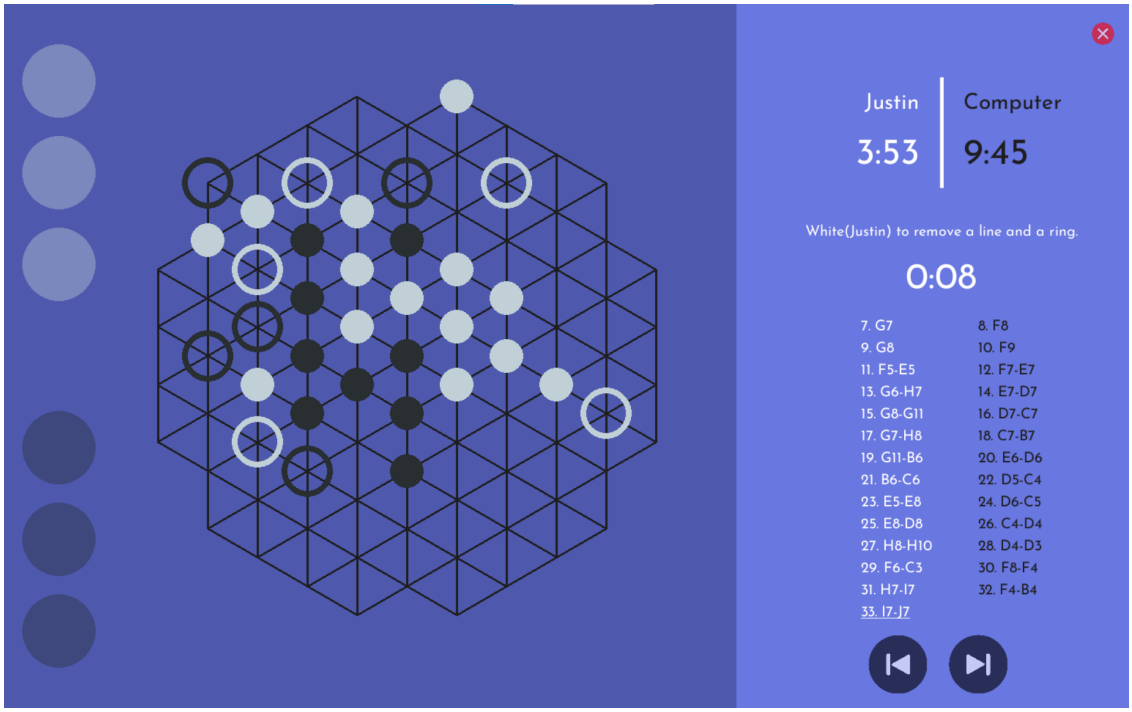


Figure 30: The game screen

The game screen allows the user to play a game of YINSH. As demonstrated in the video, the user is able to play only legal moves and the game ends when a player achieves three lines. Objective 2 is met.

5.3 Objective 3

The software should show the user relevant information in the game screen.
In Figure 30, the game screen displays the players/computer playing the current game, whose turn it is to play, how many rings each player has left, how many lines they have already previously formed, and a table of all the previous moves of the game in algebraic notation. The user is also able to undo and redo moves. At the end of a game, the software displays who has won, why they won, and offers a replay for the user to view the game. Objective 3 is met.

5.4 Objective 4

The software should be able to import and export games and replay them.
As demonstrated in the video, the software allows games to be exported into a JSON file, and is able to import game data from a JSON file. A history of all the moves played is displayed in algebraic notation, and the user is able to go forward and backwards through the moves played in a game. Objective 4 is met.

5.5 Client feedback

Figure 31: Mr. Walker trying out the software

After having Mr. Walker playing the game for a while, he has provided me with some feedback.

Product Evaluation

Did you think the product meets your original requirements?

Not Clear How To Save A Game.
Apart From That - Yes, very good!

Is there anything that you would like to have been done differently?

Make the Easy/Medium/Hard Buttons
more distinct when selected.
Make an Audio notification when Computer
makes a move.

Do you have any additional comments on the product?

Maybe a "Do you really want to quit?"
when you click on the (X)

B. O'[illegible]
Mr. Chris Walker

10.3.2024
Date

This feedback is quite fair. Saving is a bit complicated, as you are only allowed to save the game when looking at the replay. Since I've used Thorpy for some UI elements, I agree that the styling of the difficulty selection buttons are a bit hard to see. An audio notification will also be quite useful as sometimes the computer may spend 10-15 seconds on a move. To prevent accidentally deleting the whole game, a confirmation should also be in place to make sure the user actually wanted to exit the game.

5.6 Conclusion

The software has successfully met all the objectives outlined in the evaluation. If I find the time, I would definitely want to implement some quality of life features suggested by Mr. Walker. I am quite happy with the software, and it has taught me a lot about the development cycle of software, especially the planning phase by creating class diagrams so that the general structure would already have been decided during the implementation. When trying to distribute the software to Mr. Walker and other testers, I've faced multiple issues with differing python versions, or the absence of python, or missing packages. I have investigated into using different methods to bundle Python and all the packages into a folder or a single executable, and I would like to explore this further to make it as easy as possible for anyone to download this software. Overall, I am very happy with how this software turned out.

6 References

1 Wikimedia Foundation. (2023, October 30). YINSH. Wikipedia. https://en.wikipedia.org/wiki/YINSH

2 Yinsh. BoardGameGeek. (n.d.). https://boardgamegeek.com/boardgame/7854/yinsh

3 Wikimedia Foundation. (2024, March 18). Python (programming language). Wikipedia. https://en.wikipedia.org/wiki/Python_(programming_language)

4 Tomic, J. (2022, February 14). C# vs. C++: Whatś at the core?: Toptal. Toptal Engineering Blog. https://www.toptal.com/c-sharp/c-sharp-vs-c-plus-plus