NEA Report

A Game of Pylos

Ming Chau Chan

Contents

Chapter 1

Analysis

1.1 Background and Problem Description

William Chittick is a student at Tonbridge School. He is also a friend of mine and, more importantly, an avid Pylos enjoyer. Pylos is a two-player board game that uses a small board and spherical pieces. To play Pylos, he requires a board, pieces to place, and opponents to play against. Unfortunately, these elements are not always available to him. Therefore, he has requested I help him solve this problem through programming.

1.1.1 The Game

Pylos is usually played on a 4x4 board. Each player has 15 pieces to start with, and they place their pieces in turn on the board. Four pieces create a square, on which another piece can be placed. This creates a pyramid consisting of 4 layers of 4x4, 3x3, 2x2, and 1x1 pieces. The objective of the game is to place a piece on the top of the pyramid. The player who does so wins the game.

There are also additional rules regarding retrieving and relocating pieces:

1. You may retrieve two pieces if you
    - complete a square of your own colour or
    - place four pieces of your own in a row.
2. If there is a free spot on a higher level than one of your pieces on the board, you can move that piece up to the free spot (given the piece is not supporting other pieces).

This is a game where you try to conserve pieces and stop your opponent from retrieving or conserving pieces. Traditionally, it is played on a 4x4 board, but this can be extended to larger dimensions. The rules can also be altered. Taking away rules such as being able to retrieve two pieces upon completing a square of your own colour can allow for easier gameplay for beginners.

1.1.2 Client Interview

Q: Why do you want this application?
A: I have been playing Pylos for a long time and I don't plan on stopping. Travelling with it is cumbersome so I would like an electronic version of this game. Given I am used to the real version, I would like the electronic version to emulate the real game while providing additional functionality.

Mobility and emulating the real game are important to the user. They will likely play this when travelling or otherwise when they do not have access to a real version. The client wants a substitute for the physical game so try to make it similar to the real thing.

Q: How do you usually expect the game to be played?
A: When I play with someone, we usually decide on the ruleset once we have found a board and pieces to play with. The

changes to the ruleset are mainly the methods of retrieving pieces. Against beginners, I like to take it easy on them and remove all rules of retrieval. Against better players, we can make the game more interesting by adding ways of retrieving pieces such as making squares of the same colour or making four in a row.

The client normally plays with variable rules. Therefore, the game class should be able to handle different rules of the game.

Q: What experiences have you had with other products of a similar nature? Why is this necessary?
A: I have tried multiple versions of the game online, most with a command line interface. The best graphical user interface I have seen was a two-dimensional representation of the game board with terrible choices for colours and low resolution. I would prefer a proper user interface with a good representation of the board and more game features.

Again, the client is used to the real game so it is difficult to get used to electronic versions which try to use two-dimensional representations of the board. A graphical user interface with a three-dimensional representation of the board is therefore necessary.

Q: What aspects did you like then? Are there any of these that you would like to be incorporated into this new version?
A: Some of the versions I have tried offer replays. They allow you to watch a game you previously played. I found this to be very helpful in reviewing and improving my gameplay. It would be great if you could include this as a feature. I hope the replay can be viewed with perspective and you can move around the board. Some of the websites also offer multiplayer gameplay. However, none of my friends is as interested in Pylos as I am. Also, to make up for the lack of perspective, the websites often had other tools that helped communicate additional information. For example, the position being hovered over by the cursor would be highlighted if it is a legal input; ghost pieces appear in places where pieces can be placed; and some pieces are brighter in colour to show that they can be promoted. I quite liked this.

The client takes Pylos very seriously so a replay option is an important feature. A computer player to practice against is also desirable since William has no friends. This computer player should be able to play at the client's level to offer a suitable challenge. The client liked the visual tools for communicating the legality of moves, positions where pieces may be placed, and promotable pieces. These can be implemented in my version of the game.

Q: What additional functionalities are you looking for? Perhaps things specific to an electronic version could improve the experience?
A: I hope the colours will be customizable. That might freshen the gameplay a little. Also, my computer is not very powerful, so it must not be very demanding in terms of computational power. This would allow me to save battery and also have fluid gameplay.

The colour scheme can play a part in bringing variety to the gameplay and making things more exciting. Each player can choose the colour of their pieces and the background can change as well. The client has limited resources in terms of computing power, so I should optimise the program well.

1.2 Past Solutions

There are very few versions of the game online, mainly just the version on https://en.boardgamearena.com/gamepanel?game=pylos. The interface is rudimentary with a 2D representation of the board as shown in figure 1.1.

This version of the game is hosted on a board game website (boardgamearena.com) and supports online multiplayer gameplay. This is just one of many different games on the website, and the developers most likely have not devoted much effort to making the game as good as possible. This is reflected in the poor user interface design. This lack of thought in design is prevalent in nearly all forms of this game found online.

My client, William Chittick, is an avid enjoyer of Pylos. However, the board game requires a board and 30 wooden pieces in the form of spheres, which can be difficult to transport from school to home or play when travelling. Therefore, they tried to find an electronic version of the game. After playing multiple versions similar to the one above, they found that:

- The pieces are hard to differentiate due to the colour scheme.
- The positions of the pieces in 3D space are difficult to perceive due to the lack of perspective.

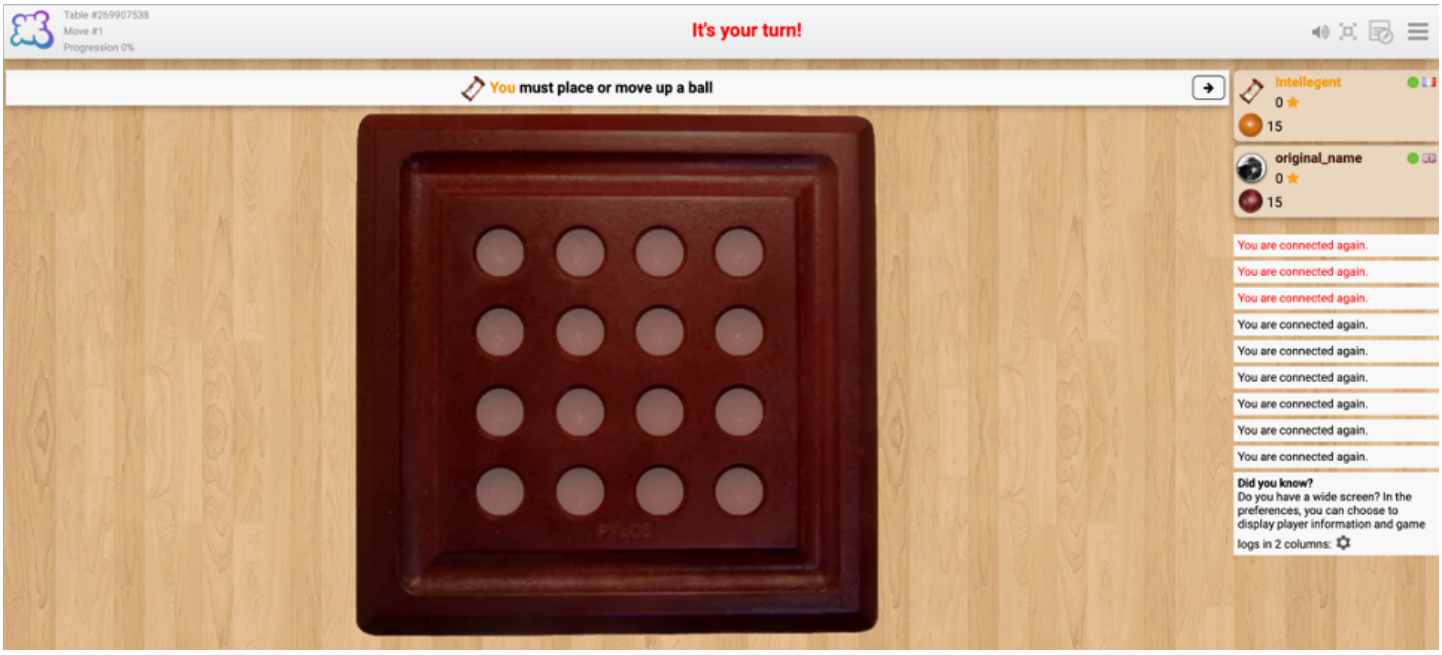


Figure 1.1: Pylos game from boardgamearena.com

- The colours of the pieces and the board are dull and cannot be changed.
- The rules are much simpler in this version, disallowing the complex strategies and tactics which my client is fond of.
- Navigating through each site is confusing and difficult.

It is precisely due to this lack of a good electronic version that my client requires a better, improved, version. It should provide perspective, be easy to use, and have additional customizability. All the versions simply lack a good user interface and intuitive game menus. Moreover, none of the versions currently available have computer players to practice against. Hence, my client has requested that some form of computer player be included in this game.

However, there were redeeming features that the client found. These are a replay option; highlighting pieces being hovered over; differences in colour for pieces which can be promoted; and ghost pieces (translucent pieces which appear at locations where a piece can be placed).

1.3 Other End Users

Other parties who would be potentially interested include board game websites and other Pylos players. Currently, board game websites all have no proper versions of Pylos. They would likely be willing to take the program and host it on their websites if it is better than their version and computationally inexpensive. However, this version does not support multiplayer. Interested parties would have to implement multiplayer independently if it is a desired feature. They would need to take measures to prevent cheating such as encryption and preventing direct access to the game class. I will aim to make my code sustainable, easy to read, and easily understood so interested third parties may edit and utilise my code.

1.4 Objectives

1. The player should be able to play a game of Pylos.
   (a) Players should be able to place pieces when the rules permit.
   (b) They should be able to retrieve pieces when the rules permit.
   (c) Players should not be able to make illegal moves such as placing, retrieving, or moving their pieces when it is not their turn.
   (d) They should be able to win by placing a piece on top of the pyramid.

Figure 1.2: Example of the board during a game.

(e) Players should be able to look through the moves that have been played throughout the game. This means they can undo moves to look at the state of the board some moves ago and then redo them when they want to.

(f) When moves have been undone, the players should not be able to interact with the game.

2. The player should be able to customise the settings for each game.

   (a) There should be a screen before each game where the player can customise the settings for that game.

   (b) The colour of each player's pieces and the background colour should be decided by the player.

   (c) Custom rules regarding the retrieval of pieces should be able to be toggled on or off ("Square" and "Alignment").

   (d) The players should be able to alter the size of the board.

3. The player should be able to play against a computer player.

   (a) The player should be able to choose whether to play against a computer player or another person.

   (b) When a player chooses to play against the computer player, they should choose whether the computer moves first or second.

   (c) The CPU player should move quickly (within 5 seconds of the human player making a move).

4. The player should be able to replay recent games.

   (a) The player should be able to see the recently played games along with relevant data such as the winner and loser, the duration, and the number of moves played.

   (b) The user can then choose a game and replay it, allowing them to go through the moves made in the game. They may move forwards or backwards through the game.

5. When in-game or watching a replay, the player should be able to see a three-dimensional rendering of the game board and manoeuvre the camera around it.

   (a) Placed pieces appear as spheres of the player's chosen colour.

   (b) Perspective exists. This means closer pieces appear bigger, and pieces which are further from the camera appear smaller.

   (c) The user should be able to rotate the board around the vertical axis.

   (d) Similarly, the user should be able to tilt the camera up or down to view the board at different angles.

   (e) The camera should be able to zoom in or out.

6. The interface should communicate relevant information to the player.

   (a) A white circle highlights the position that the user is hovering over to indicate if they can retrieve that piece or place a piece at that location.

(b) Pieces that cannot be moved should be darker in colour to indicate they are not movable.
(c) On the other hand, pieces that can be moved or retrieved should be brighter to indicate they are movable.
(d) Ghost pieces (Transparent pieces) indicate locations where the player may place a piece.
(e) Time elapsed for each player;
(f) The number of pieces remaining for each player;
(g) The player whose turn it is;
(h) Status message (Place a piece, retrieve a piece, square completed);
(i) And error message (Invalid input, piece not retrievable, spot lower than selected piece).

1.5 Research

I have identified three major subproblems: creating a game of Pylos, creating a CPU player, and rendering the game board. I spent about a week researching these and I will briefly describe the pros and cons of different techniques I came across in my research. The following information is my summary of the readily available content on the internet that I read/ watched. The sources of which are mostly Wikipedia and GeeksforGeeks, with other helpful websites here and there.

1.5.1 Pylos

The standard approach is to use object-oriented programming (OOP). This involves creating classes for each object, allowing them to interact with each other in an organised fashion. This enables the programmer to easily edit and plan out the program as well as for third parties to understand, edit, or utilise particular portions of the code.

In such an approach, there are usually separate classes for the game and the user interface. The game class holds an instance of the game while allowing moves to be made according to the rules and storing data related to the game. These include things such as checking whether an input is valid, placing and retrieving pieces, and storing moves which have been made over the course of the game. The user interface class handles user input and communicates information such as the game state or a menu. In this case, the user interface will be a graphical one, meaning it will involve buttons to be pressed, textboxes to be written in, and a representation of the game board. To coordinate all the interactions effectively, more classes will be created for objects such as buttons and text boxes.

Regarding the game state representation, a 3D representation of the board is required as per the client's request. The other implementations of Pylos have conventional displays of information which are reasonable. Similarly, I will use text over the background to show status messages, error messages, etc.

1.5.2 CPU player

There exist many different optimization strategies that can be utilised in the CPU player. In order to decide on the best approach to implement a computer player for my client to play against, here I document multiple approaches and consider the pros and cons of each.

Is Pylos a solved game?

Pylos is a solved game to the extent that with less than $3^{30} \approx 2 \times 10^{14}$ positions, a database can hold all positions and the next optimal move. However, this would require an infeasible amount of memory. Explored below are some alternatives and their respective pros and cons.

Minimax algorithm

A minimax algorithm, also known as a maximin algorithm, works by exploring many turns ahead. The algorithm takes all actions possible and after some k turns it evaluates the game state using some evaluation function. The algorithm then plays the move that offers the maximum reward at the end for the CPU player assuming both players play the best possible moves. This algorithm maximises the minimum reward after k turns.

The evaluation function is crucial to the success of the decision-making of a minimax algorithm. Continuing on from above, a more optimal strategy is to store desirable positions from which a win can be forced and implement the use of a minimax algorithm that aims for those positions. To achieve this, a database of desirable positions must first be generated. This can be done by reversing all the moves from a final position (a completed pyramid). This would be precomputed and stored in the game files. The computer can then evaluate the difference between the current position and all desired positions.

This gives an optimal move in reasonable time and space complexity. However, it still requires an evaluation function that measures the distance and an efficient database to store the moves.

Occam's razor; In this case, not much more can be deduced from the state of the game as it often offers complex situations that require even more moves ahead to be predicted. Therefore, an alternative is to let the value (the advantage of the computer player) = No. pieces of the computer player − No. pieces of the human player.

Finally, alpha-beta pruning is a popular pruning method to improve the minimax algorithm. When implemented, the computation time can be greatly shortened. This allows for a deeper exploration of the game tree and thus permits the algorithm to find better moves to play.

Monte-Carlo Tree Search (MCTS)

MCTS involves random moves being played out until the game ends. Doing this many times and counting the number of wins can be a good evaluation function for the minimax. However, this is unrealistic as the number of moves which would need to be taken to finish the game is at least of order 10^1 and at least 10^2 paths would have to be explored to provide any significant and reliable evaluation. These paths would have to be explored for approximately 10^6 leaf nodes. This gives a total of more than 10^9 moves made by the MCTS algorithm which would be extremely slow to compute. In this case, some sort of greedy choosing heuristic can be employed, but this may not be reliable if the opponent has a strategy that beats the greedy heuristic.

Machine Learning

A Feedforward Neural Network comprises layers of nodes that you can train with a big dataset to give answers to complex problems (such as a state in a game of Pylos). This approach offers fast computation time (once the neural network has been trained) and sustainability (the neural network can learn as it plays against new players).

However, the main difficulty of this approach is the large dataset required to train the neural network in the first place. A neural network would require a large variety of game states and a loss function that approximates the reward and loss of each state.

There are also other learning strategies such as a genetic algorithm, NEAT learning, and others. However, these all come with the problem of training. I simply do not have access to a large enough dataset for training.

1.5.3 3D rendering

Ray Tracing

Ray tracing is often used in movies and games to create realistic images. This is done by tracing a "ray of light" from the camera to the scene. This involves ray-marching and shading methods. The reflections and diffusion can be computed to simulate real conditions. The resulting light values can come together to render the board from whichever perspective the user is looking at it from. Note that the number of rays one needs to simulate is exactly the resolution of the screen. Therefore, ray tracing on reflective surfaces can become quite expensive at high resolutions and bounds. For example, this approach is highly impractical for around $1000 * 1000 = 10^6$ pixels as a frame has to be rendered in at $\frac{1000}{fps} = \frac{1000}{50} = 20$ milliseconds.

One solution is to make the surfaces matte or non-reflective. This allows for quicker computation as one can simply calculate the dot product of the light and the ray for the intensity of the light on a particular spot. This can allow for a decent resolution while still maintaining smooth gameplay. Lowering the resolution is also an option, but it impacts the visual clarity and takes away from the user experience of the game.

Surface Point Rendering

This is a creative solution that may be used due to the simplicity of the shapes being rendered. The spheres can be reduced to points on their surfaces. One can then iterate over the points and draw a line to the camera and colour the pixel that the line crosses. This solution saves computation and is easy to implement due to the pieces being spheres (so points on the surface are easily generated by iterating over two angles). However, once the number of points gets large, the computation time slows down again.

Approximatation Methods

The methods mentioned above are "True 3D" rendering methods. "True 3D" means that the resulting images rendered are proper projections of the objects in the scene onto the screen in front of the camera. These often require intense calculations

for life-like accuracy. However, these calculations can be simplified if you are willing to sacrifice a little bit of accuracy. Instead of conic sections, by approximating the ellipse with a circle, the calculations required are much simpler and result in less computation time.

There exist methods which are not "True 3D", but they are often sufficient. For example, the rendering class iterates over all the pieces and approximates their projections onto the screen using circles. When the spheres are far away enough, this is convincing enough to seem 3D. This works because the ellipses are mostly circular (to the point where we wouldn't notice much difference). This allows the program to run much faster.

1.6 Proposed Solution

1.6.1 Language

I have chosen Python as the programming language for this project. There were many factors which contributed to this choice. I mainly considered three programming languages: Python, C, and C++. Below are the pros and cons of each and the final justification of my choice.

Python	
Pros	Cons
Easy to use as memory management is automatic and Python uses dynamic data typing. Supports object-oriented programming. Large standard library and publicly available libraries (Good for GUI Frameworks and math modules). More portable as it is interpreted.	Very slow due to additional features (e.g. garbage collection, dynamic typing).

C++	
Pros	Cons
Faster than Python. Safe due to modifiers and well-designed exception handling. Supports object-oriented programming.	Large standard library but fewer GUI frameworks available than Python. Harder to work with than Python due to static typing, more complex syntax, and more memory management involved.

C	
Pros	Cons
Extremely fast due to the lack of overhead processing.	Procedural so it does not support object-oriented programming. Time-consuming to use due to the manual allocation and freeing of memory. Difficult to use due to risk of memory leaks, static data typing, etc.

Firstly, there are many modules available for Python. No other community has such an extensive selection of libraries and modules. The community has also made available large quantities of resources and support, making the libraries easy to use. The ease of use of these libraries will be helpful in a project as large as this. For example, an essential requirement for my game is a good graphics engine/ GUI framework. Python immediately offers a wide variety of options for me, along with modules such as numpy to assist in other areas like computation.

Secondly, I would have chosen a faster language if the program required speed. However, the client requested the program to be computationally inexpensive. Therefore, the method of rendering I have decided upon requires little computational power. As a result, there are almost no disadvantages to using Python.

Thirdly, Python supports object-oriented programming (OOP). OOP allows the programmer to minimise the amount of work they have to do. It is also straightforward to plan out and execute. Therefore, Python and C++ have the upper hand against C.

Lastly, I am familiar with Python. For an extended project, it would be useful to work with a language I am already familiar with to avoid unforeseen bugs and issues arising from a lack of understanding of the language. Moreover, despite

knowing languages such as C, C++, Java, and Rust, I find that Python is the easiest to use due to its simplicity from things like dynamic typing and its syntax. Therefore, Python is my language of choice.

1.6.2 Development Environment

There are two popular IDEs for Python development. They are VS Code and Pycharm. Note that VS Code is not an IDE. It is a code editor that offers a similar experience through extensions. This is because the default installation does not have enough to qualify as an IDE.

Pycharm	
Pros	Cons
The pro version is available for free with the GitHub student developer pack. The environment is fully set up upon installation (no need to install plugins manually) Code management is well developed, with features such as "search everywhere" and "smart editor"	More memory is required for installation (400 Mb) than VS Code.

VS Code	
Pros	Cons
Lightweight and customisable. Open-source with features like version control and bug tracking coming out of the box for free.	Need to manually install any plugins you want I am unfamiliar with the hotkeys and shortcuts.

As seen above, both have their strengths and weaknesses. I felt that I wanted an IDE well-configured for Python. Pycharm is well documented and has good support online from the JetBrains website. I have no specific need for the customizability of VS Code and it would take time to install the necessary extensions and plugins. Therefore, I made the choice to use Pycharm over the course of this project.

1.6.3 Techniques

I outlined the benefits and drawbacks in the research section above. From these, I have drawn the conclusion as to which techniques best suit my objectives and the project's goals the most.

Object Oriented Programming

This makes the code easy to plan out and edit for future programmers. It is standard to use such programming. In the design chapter, I plan to elaborate on the classes and outline the interactions so that no unforeseen errors occur during implementation.

CPU Player

Objective 3 states that the player should be able to play against a computer player. I have decided to implement a minimax algorithm for the player to play against. Along with alpha-beta pruning, it should be able to consider seven or eight moves ahead in each game state on a standard four-by-four board.

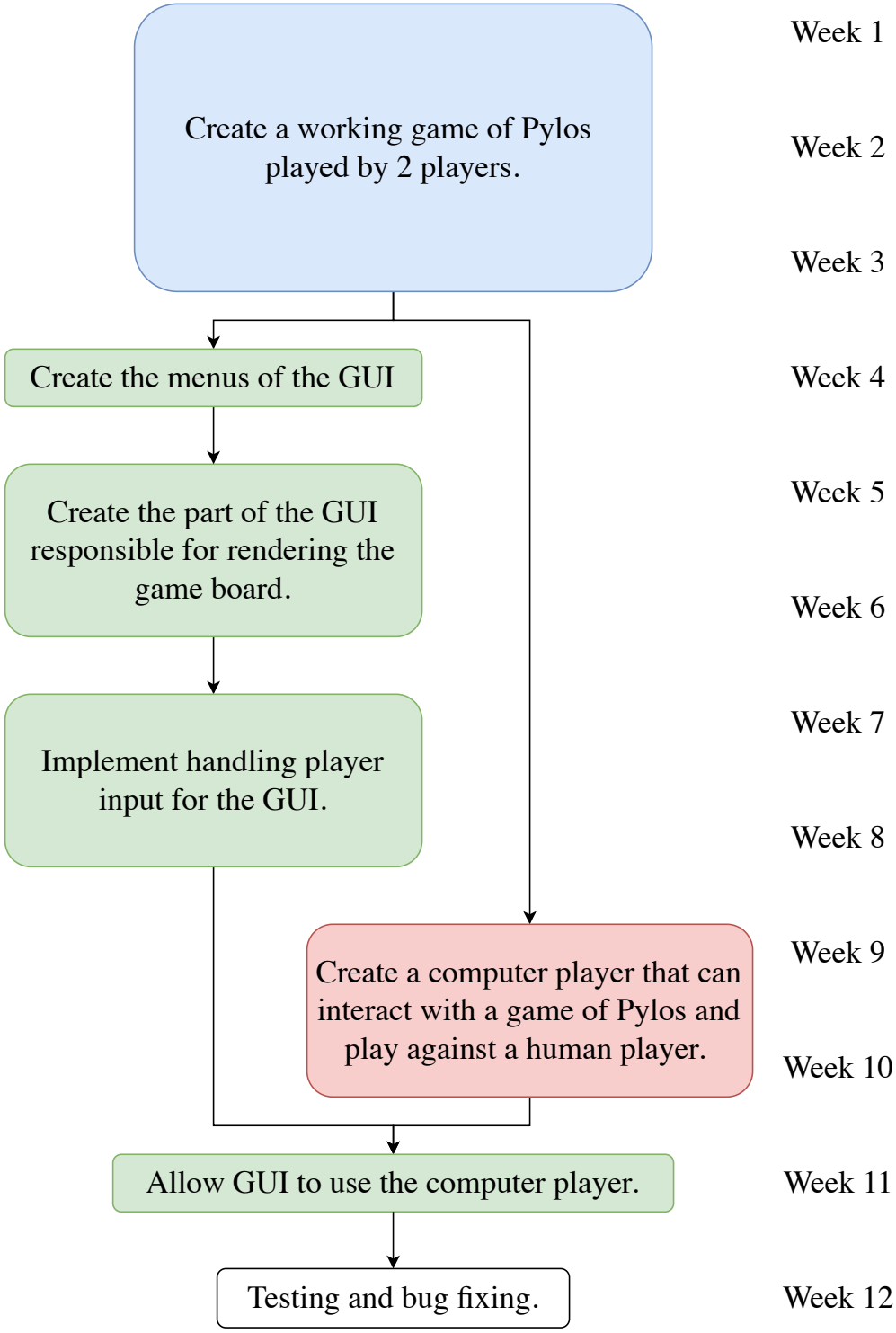
Rendering

Objective 5 states that the player should be able to see a rendering of the game board. I have chosen to use approximations of the 3D objects in the scene. This means using circles to approximate ellipses. I will also use ideas from z-buffering to order the pieces.

The main consideration was computational power and time complexity. The client simply does not possess the facilities for actual rendering techniques. In this method, each piece is considered once only per frame so very few operations are made per second compared to other options.

1.7 Critical Path Diagram

In an extended project such as this one, one must plan ahead lest one loses track of their progress and misses the deadline. As my client wants this done before Easter in order to enjoy the finished product over the holidays, I have approximately 12 weeks of development time from now until then. I have designed a critical path diagram to keep myself on track during development in order to finish this project in good time.



Chapter 2

Design

2.1 System Design

This section of the design chapter aims to **specify the layout of the program**. I will create an overview and then design each element of the plan in detail. I will work from the top level down. This makes planning easier and allows me to code in a concise and organised manner. I will detail the processes behind each of the key elements from the level before. This will be the structure of the code, so great care must be taken to ensure no mistakes are made in this step of progress.

This is known as abstraction. **Abstraction** is a good tool to use for solving problems like this. This is because it allows one to break down a complex and convoluted problem into many simpler ones which are much more straightforward. Solving all of these subproblems gives rise to a grand solution to the original.

2.1.1 Top-Level

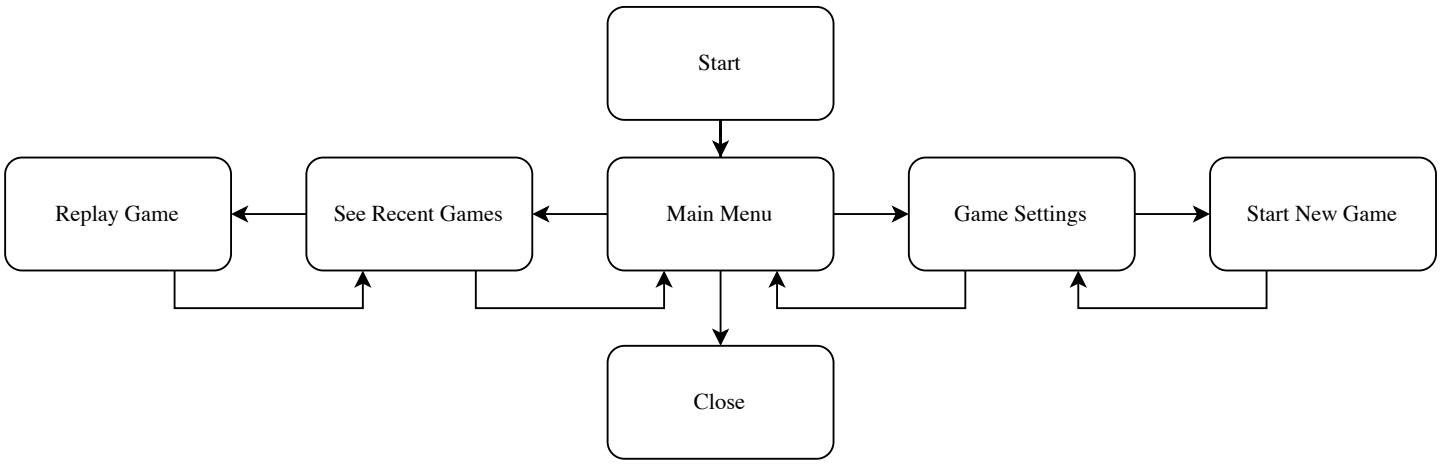


Figure 2.1: System Overview

This is the **top-level overview** (Figure 2.1). This describes what happens in general when the program is running and the key points of a user's experience. These main elements have to be linked together now so that later on when they are elaborated upon, I remain mindful of their connections with each other. Moreover, critical elements can be identified and focused on. This assists in avoiding major bugs or issues later in development.

2.1.2 Game Settings

This is where the user may specify the settings for a new game. These settings are stated under **objective 2**. Various settings may be altered in this state. These settings will be in the form of buttons and text boxes. Once the game starts, they will be saved, meaning they will be identical the next time a user goes to Game Settings again. As shown in Figure 2.1, Game Settings goes to Start New Game once the player decides to start the game.

2.1.3 Start New Game

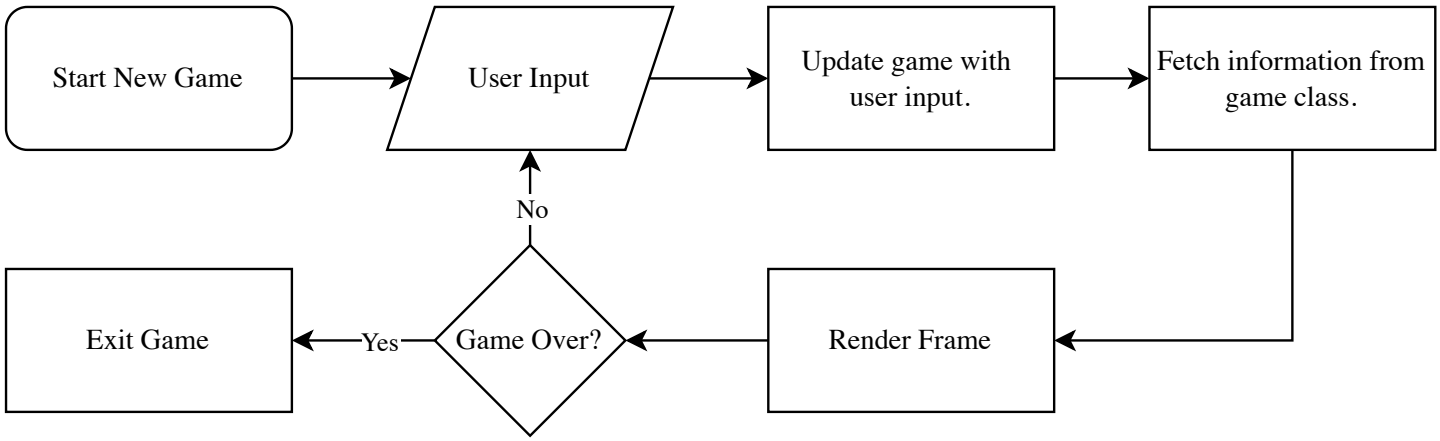


Figure 2.2: Flowchart for Start New Game

There are many possible inputs from the user. The process is complex enough that I have created another flow chart for User Input (Figure 2.3).

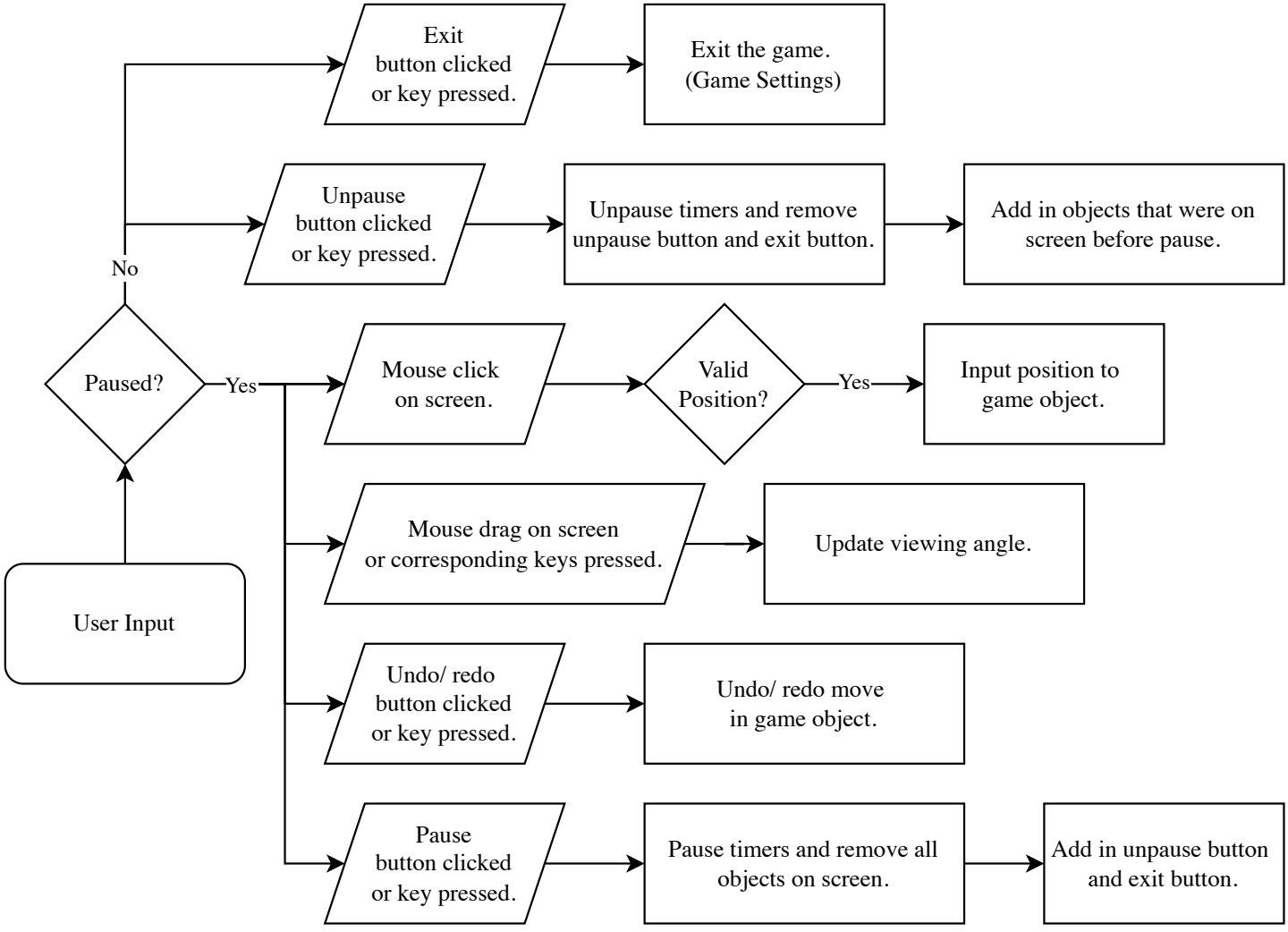


Figure 2.3: Flowchart for User Input

2.1.4 See Recent Games

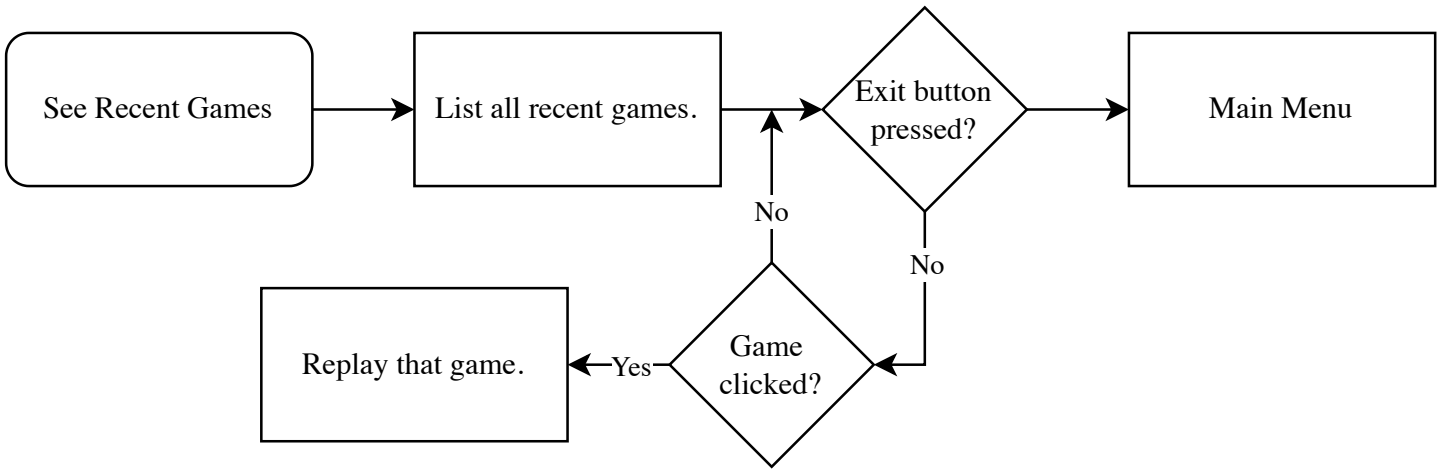


Figure 2.4: Flowchart for See Recent Games

"See Recent Games" serves the purpose of laying out information for the user to see. The only interaction the user will have in this state is to select one of the games displayed or to exit and go back to the main menu.

2.1.5 Replay Game

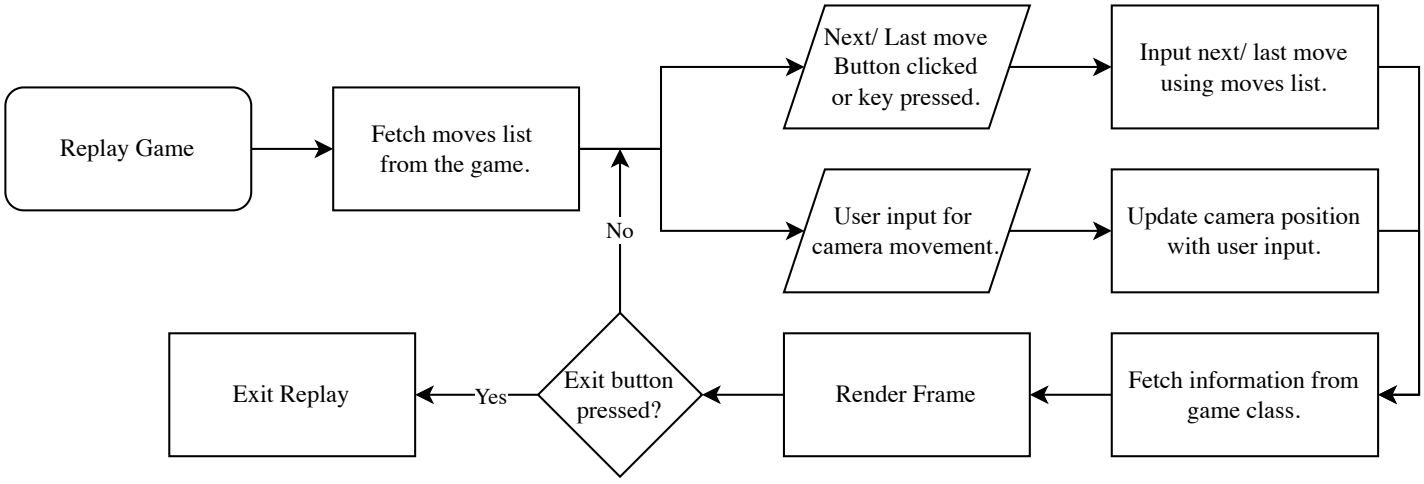


Figure 2.5: Flowchart for Replay Game

The user inputs here are quite similar to those in Start New Game except the user cannot directly input into the game. Instead, they may only input the recorded moves from the game they chose to watch.

2.2 User Interface

Referring back to the previous section, each element in the top-level overview must have its own separate screen. To avoid confusing the element and its screen, I have assigned each element and its screen individual names.

Element	Screen
Main Menu	Home
Game Settings	Start
Start New Game	Game
See Recent Games	Watch
Replay Game	Replay

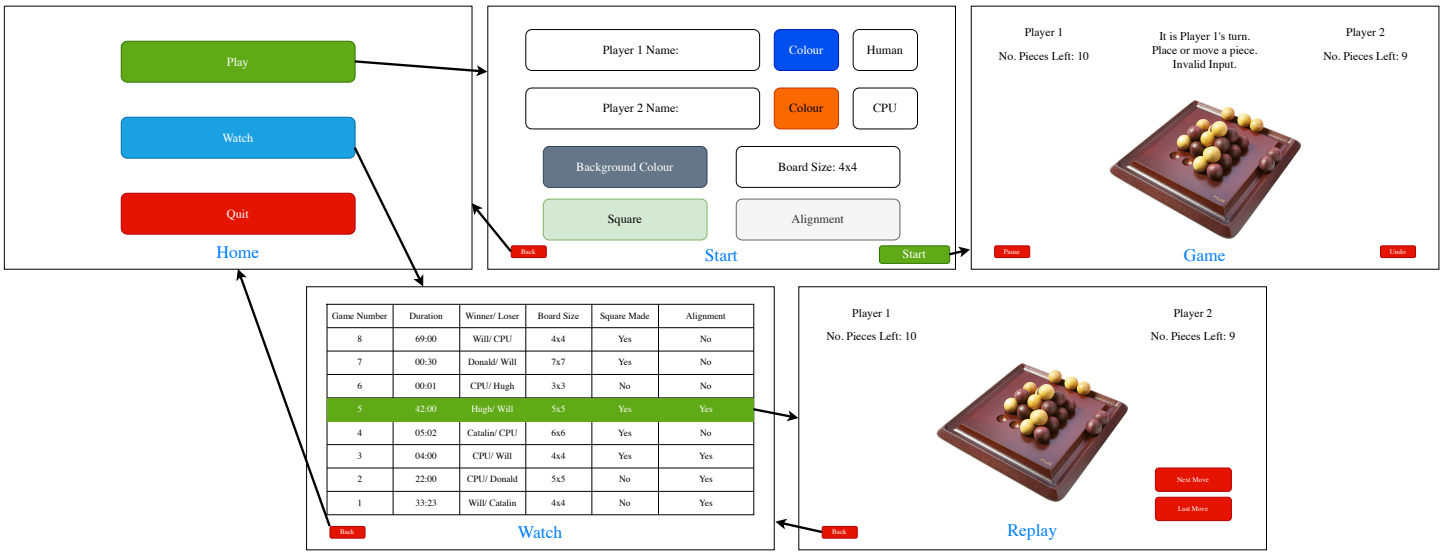


Figure 2.6: Overview of GUI

2.2.1 Home

The home screen (Figure 2.7) consists of buttons to manoeuvre into more specific areas of the menu. These buttons include:

- Play button (to start choosing settings for a game)
- Watch button (to choose a game to replay)
- Quit button (to quit the game and terminate the program)

The home screen will serve as the main screen that players return to so it should not be too cluttered and complicated. It should be clear and easy to read so that the user can navigate the menus easily. The buttons will be different colours to signify other functions and make them easier to differentiate. Each button should have text on them to clearly direct the user and display its function.

2.2.2 Start

This screen (Figure 2.8) appears when the play button on the home screen is pressed. This screen allows you to select the settings for the game you are about to play. For example:

- The two colours of the pieces
- The background colour
- The presence of a CPU player (and whether they move first or not)
- The board size (3x3 to 7x7)
- The rules (Square made, alignment)

There should be a "New Game" button to begin a new game. Also, the user may want to exit the game and change colours. Therefore, if a game is in progress, there should be a "Resume" button to resume the game with new colours selected.

2.2.3 Game

This screen (Figure 2.9) is when the user is in the game. The user should be able to see a 3D representation of the board. For clarity, this will be placed in the middle of the screen. The user also needs to see messages and prompts from the game. This will be displayed at the top of the screen. Each player's timer will be placed at the screen's top left and top right corners for intuitive understanding. Additional Information will be displayed in free space.

I have given more thought to the controls of the camera. It should be intuitive so the player does not have to put too much thought into it. The camera should be moved in proportion to the cursor movement. The player simply has to hold the left mouse button and move the cursor to move the camera. The player should also be able to use keys to control camera movement. The standard is "W", "A", "S", and "D", so the player will use these four keys to move the camera up, left, down, and right respectively. To place pieces or retrieve pieces, the player will use a right click. This avoids confusion between camera movement and game input. I will allow the user to zoom in and out with the mouse wheel.

2.2.4 Watch

(Figure 2.10). The client wished to be able to watch previous games in the hopes of improving and learning. This is supported by storing the moves of the previous eight games. This part of the menu opens upon the pressing of the watch button on the home screen. Information about recent games is displayed. The information displayed includes:

- The names of the winner and the loser
- The duration of the game
- The board size
- The number of moves Played

2.2.5 Replay

Similar to the play screen, the replay screen (Figure 2.11) displays a 3-dimensional representation of the board in the centre for the user to view game states in past games. The user will be able to iterate through the game states using buttons.

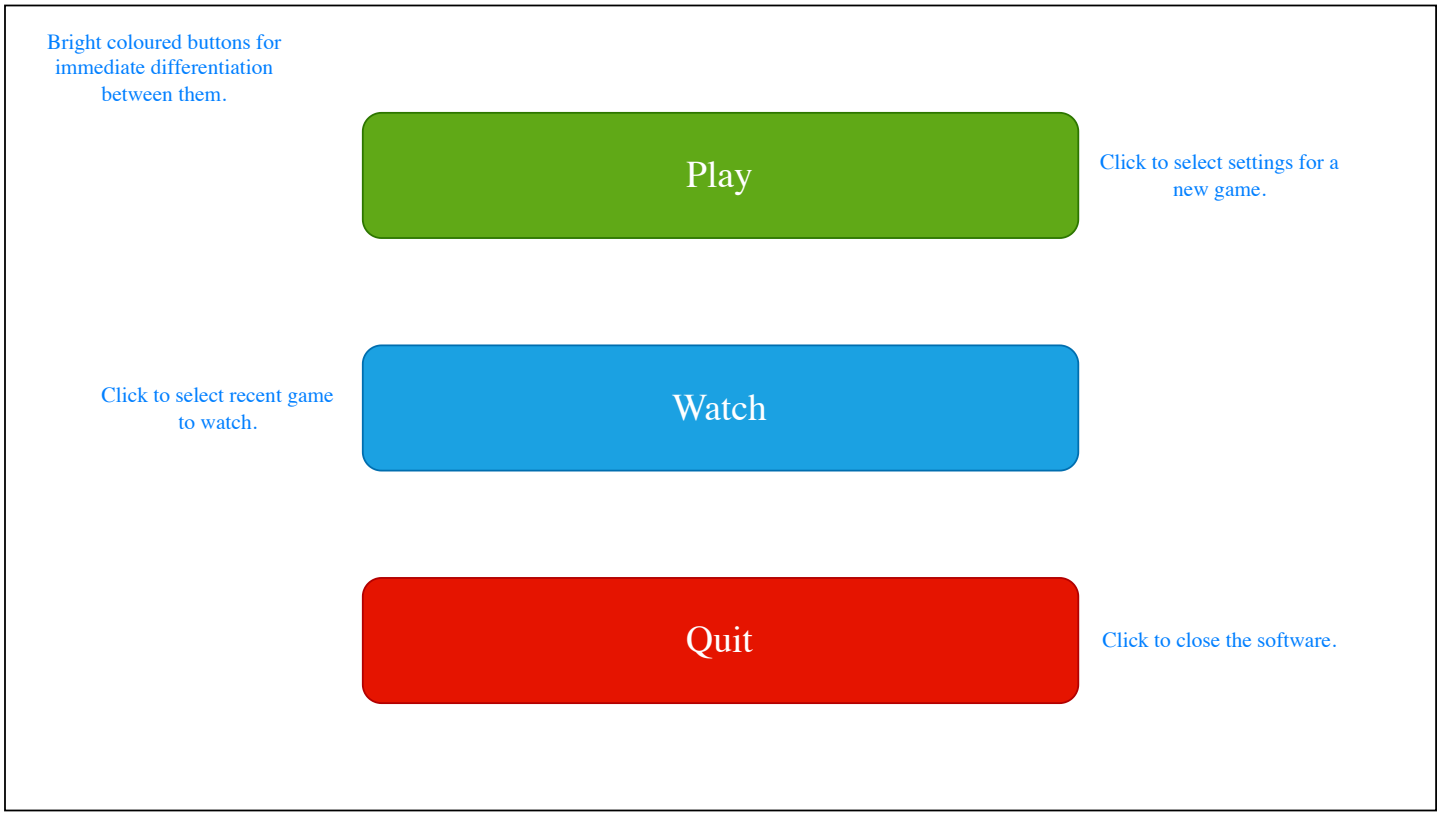


Figure 2.7: Home screen

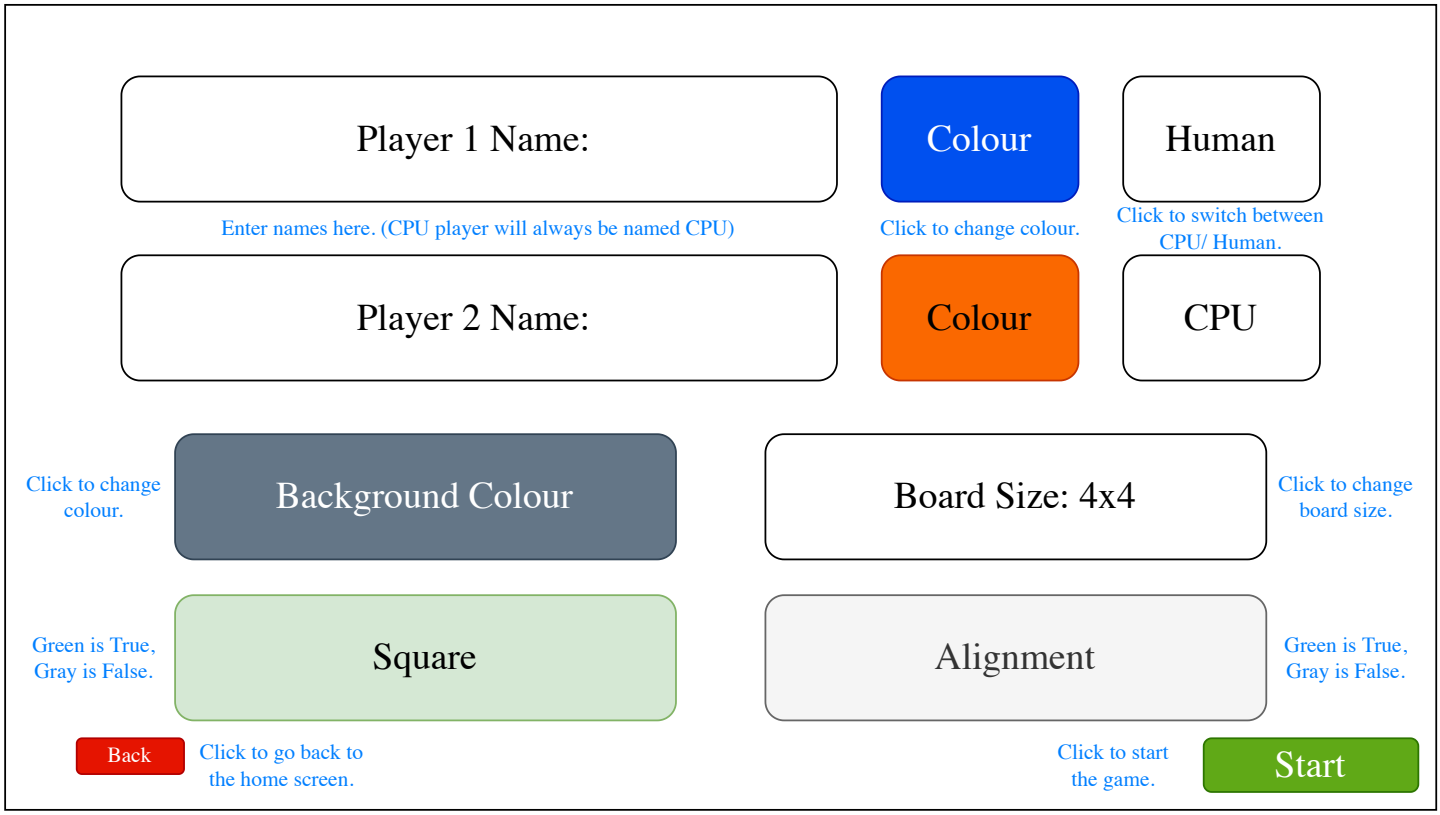


Figure 2.8: Start screen

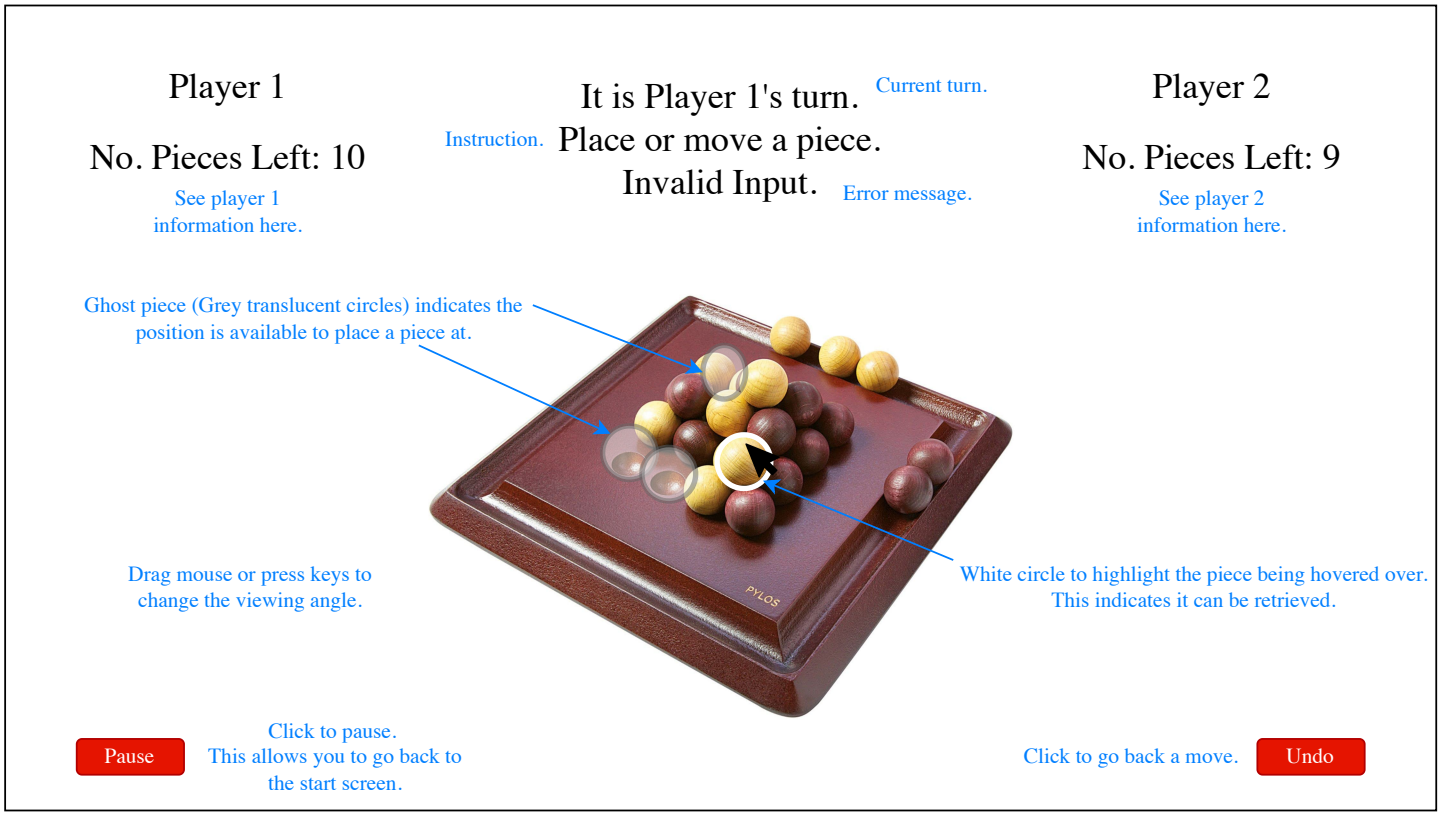


Figure 2.9: Game screen

Game Number	Duration	Winner/ Loser	Board Size	Square Made	Alignment
8	69:00	Will/ CPU	4x4	Yes	No
7	00:30	Donald/ Will	7x7	Yes	No
6	00:01	CPU/ Hugh	3x3	No	No
5	42:00	Hugh/ Will	5x5	Yes	Yes
4	05:02	Catalin/ CPU	6x6	Yes	No
3	04:00	CPU/ Will	4x4	Yes	Yes
2	22:00	CPU/ Donald	5x5	No	Yes
1	33:23	Will/ Catalin	4x4	No	Yes

Back

Click to go back to the home screen.

Click a game to watch that game.

Figure 2.10: Watch screen

Player 1

No. Pieces Left: 10

See player 1 information here.

Player 2

No. Pieces Left: 9

See player 2 information here.

Drag mouse or press keys to change the viewing angle.

Click to go to next move.

Next Move

Click to go back a move.

Last Move

Back

Click to go back to watch screen.

Figure 2.11: Replay screen

2.3 Algorithms

2.3.1 Minimax

Minimax is commonly used in many games for minimizing loss and finding the optimal move. It works by alternating maximizing and minimizing the computer player's "advantage" between layers of the game tree. This simulates both the computer player and its opponent playing the best moves possible. The computer player simply has to play the best move possible according to the tree. In figure 2.12, red is to minimize, blue is to maximize, and purple is to evaluate that state.

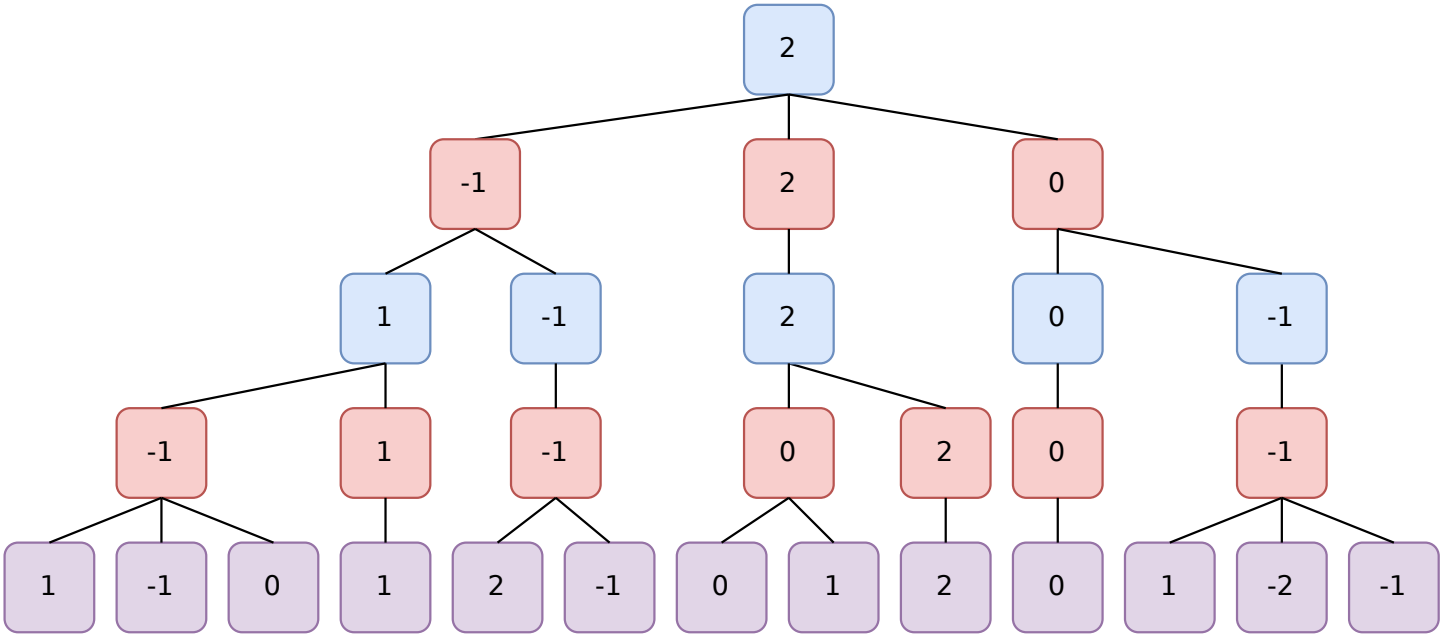


Figure 2.12: Minimax Tree

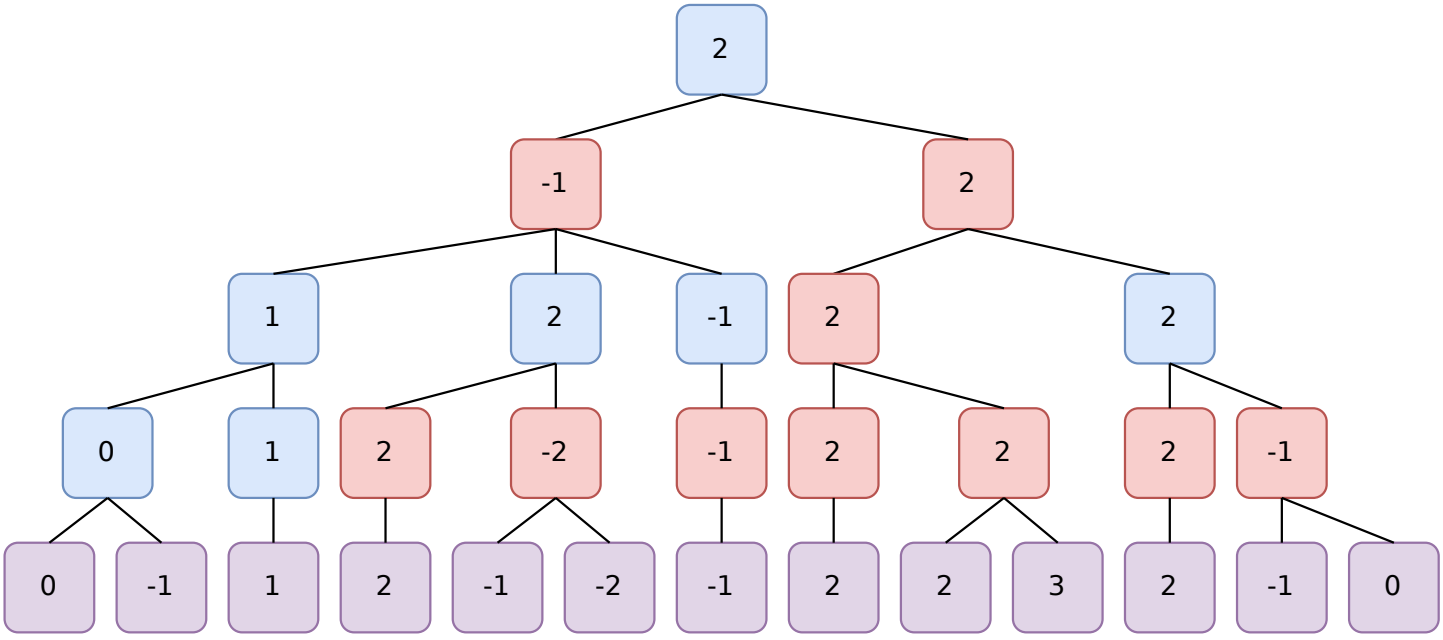


Figure 2.13: Minimax Tree for Pylos

As shown in Figure 2.13, the minimax algorithm I intend to utilize is slightly altered for Pylos. Due to the nature of the game not being in turn with rules such as retrieval of pieces after placement; moving pieces requiring two inputs; It is easier

to simply maximize the advantage when it is the AI's turn instead of alternating between turns as it is not guaranteed to be your move every other turn. See the implementation logic in Algorithm 1.

Algorithm 1 Minimax for Pylos

```
function MINIMAX(state, depth, α, β)
    if state is won OR depth is 0 then
        return EVALUATE(state)
    end if
    if current player is computer then
        for all next state from state do
            reward ← max(reward, MINIMAX(next state, depth−1))
        end for
    else
        for all next state from state do
            reward ← min(reward, MINIMAX(next state, depth−1))
        end for
    end if
    return reward
end function
```

Alpha-beta pruning

The depth of the minimax tree is a major determining factor in its success against a player. This is because the depth determines how many moves it can see ahead. The further the computer can see ahead, the more things it will consider. Consequently, it will make better decisions. However, because each node in a layer gives several nodes in the next layer equal to the number of possible moves from that state, the time taken will increase approximately exponentially ($O(k^d)$ where k is the approximate number of possible moves each turn and d is the depth of the tree).

This pruning method eliminates paths that would originally have been explored. It does this by considering an expression such as $\max(a, \min(b, x))$ and using the observation that after evaluating b, the function $\min(b, x)$ can only return values $\leq b$. Therefore if $a \geq b$, the evaluation of x is unnecessary. Using the example in figure 2.13, the tree explored under alpha-beta pruning would look like figure 2.14.

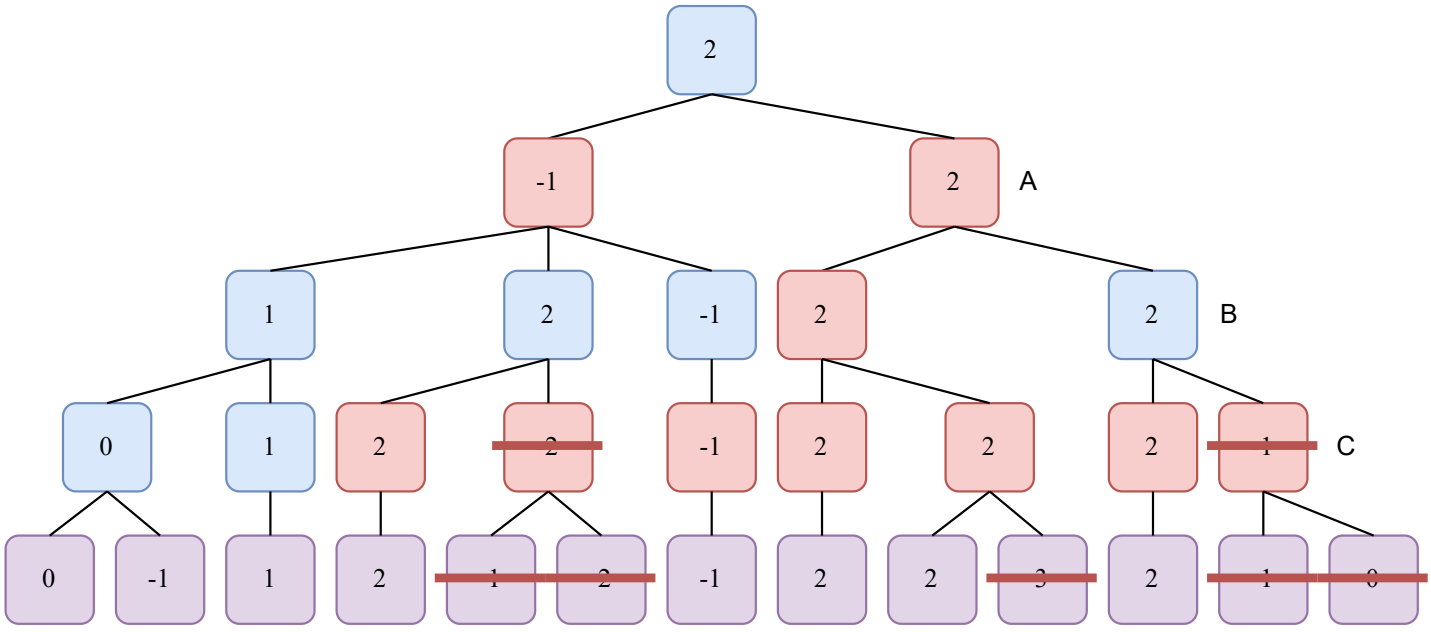


Figure 2.14: Minimax Tree for Pylos after alpha-beta pruning

There is no reason to explore node C, as node B would return values ≥ 2 to node A which would return values ≤ 2.

Therefore, it is not explored, and computational time is shortened.

The minimax algorithm determines whether to continue exploring by keeping track of the values α and β, the maximum value, and the minimum value which have been obtained in the path to the current node. If $\alpha \geq \beta$, any further exploration from that node would be pointless by the same logic as above. This results in greater optimization and offers The implementation shown in Algorithm 2.

Algorithm 2 Minimax for Pylos with Alpha-Beta Pruning

```
function MINIMAX(state, depth, α, β)
    if state is won OR depth is 0 then
        return EVALUATE(state)
    end if
    if current player is computer then
        for all next state from state do
            reward ← max(reward, MINIMAX(next state, depth−1, α, β))
            α ← max(α, reward)
            if α ≥ β then
                break
            end if
        end for
    else
        for all next state from state do
            reward ← min(reward, MINIMAX(next state, depth−1, α, β))
            β ← min(β, reward)
            if β ≤ α then
                break
            end if
        end for
    end if
    return reward
end function
```

Multi-Threading

If both processes run on the same thread, the player will not be able to interact with the game while the minimax algorithm is running. Multi-threading can solve this. This technique allows processes to be run simultaneously. As a result, the user can still interact with the GUI while the minimax algorithm is calculating the next move.

2.3.2 Rendering

The main feature of the game is to represent the game in three dimensions. This requires the program to capture the scene in each frame and display it. This is called rendering. Think of it as taking a "picture" of the game board and showing the picture to the user. Just like how videos are filmed, the game can be animated by many of these "pictures" (or renders as they are more often called). The scene is rendered every frame and when the framerate is high enough it looks smooth.

Projection

Projection means projecting three-dimensional objects onto a two-dimensional plane (call this the projection plane). This can be thought of as drawing lines from the player's eyes to every pixel on the display in front of them. The intersections of these lines and objects in the scene determine the values of the pixels. This is the main principle behind all photo-realistic rendering techniques. Therefore, comes down to two issues: the representation of three-dimensional objects and the calculation of intersections/projections. In my case, the only objects in the scene are spheres. For spherical pieces as we will have here, examples of their projections are shown in Figure 2.15.

As you can already tell, projection is not easy. This is because it involves calculating the intersection of a plane and a cone. This will be very difficult to compute, represent, and draw on the display. Alternatively, the intersection of a line with a sphere is easy to calculate. This can be done by finding the perpendicular distance (shortest distance) from the line to the centre of the sphere. If the perpendicular distance is less than the radius of the sphere, the line has to intersect the

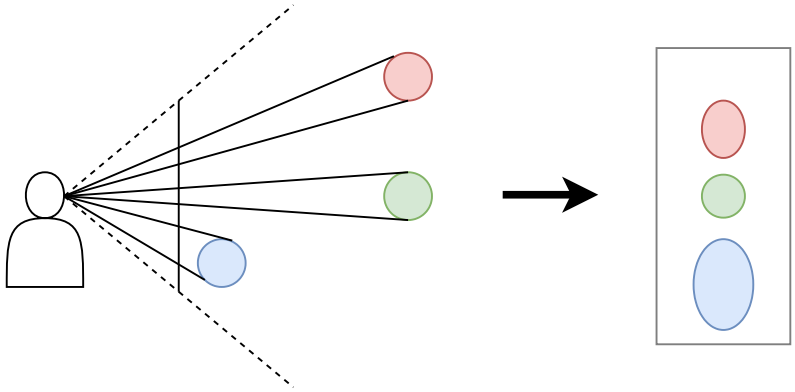


Figure 2.15: Projecting spheres onto a 2D plane.

sphere. However, this has to be done for every single pixel on the screen. This is impractical for my client's low-performance computer. It lacks the computational power for a decent rendering of the scene.

Fortunately, projection is simplified by my decision to use approximation. This is because only simple arithmetic is required instead of calculating a conic section. This is illustrated in Figures 2.16 and 2.17. In Figure 2.16, $\frac{x}{B} = \frac{1}{A}$ by similar triangles so $x = \frac{B}{A}$. The approximate radius of the projected circle can be calculated in a similar fashion.

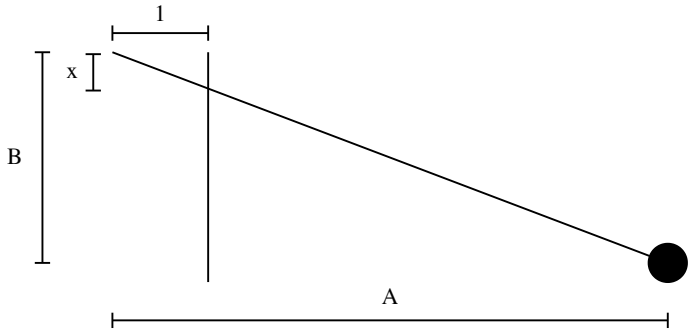


Figure 2.16: Approximating the position of the projection of a piece.

When the point is far away from the screen, the angle between the actual projection plane and the approximated plane is small. This makes the approximations in Figures 2.16 and 2.17 justified.

Generating Three-dimensional Objects

As mentioned above, the representation of the spheres is a question that needs answering. There are many ways of representing a sphere due to its simple nature. Hence, it is important in choosing a way that is most useful. I first considered what I need from this representation.

Objective 6 states that the camera should be able to manoeuvre around the board to view it at different angles. It is much easier to move objects around than to move the camera and the screen around, yet it achieves the same desired effect. Therefore, the representation of the spheres should be easily transformable (rotations and translations are comfortably applicable). With this in mind, I decided to represent the spheres as points (their centres) and their radii (uniform radii as they are all identical spherical pieces). The points are represented by their cartesian coordinates (x, y, z position/ the coefficients of its composition of i, j, k basis vectors). Rotations can then be applied with relative ease through the use of matrices. An example of a rotated point is shown in Figure 2.18. The point is rotated around the x-axis and the z-axis. These are applications of the rotational matrice R_x and R_z respectively. They are shown below. Note this means that in order to keep track of the camera's position, I only need to keep track of the two angles θ and ϕ.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, R_z(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

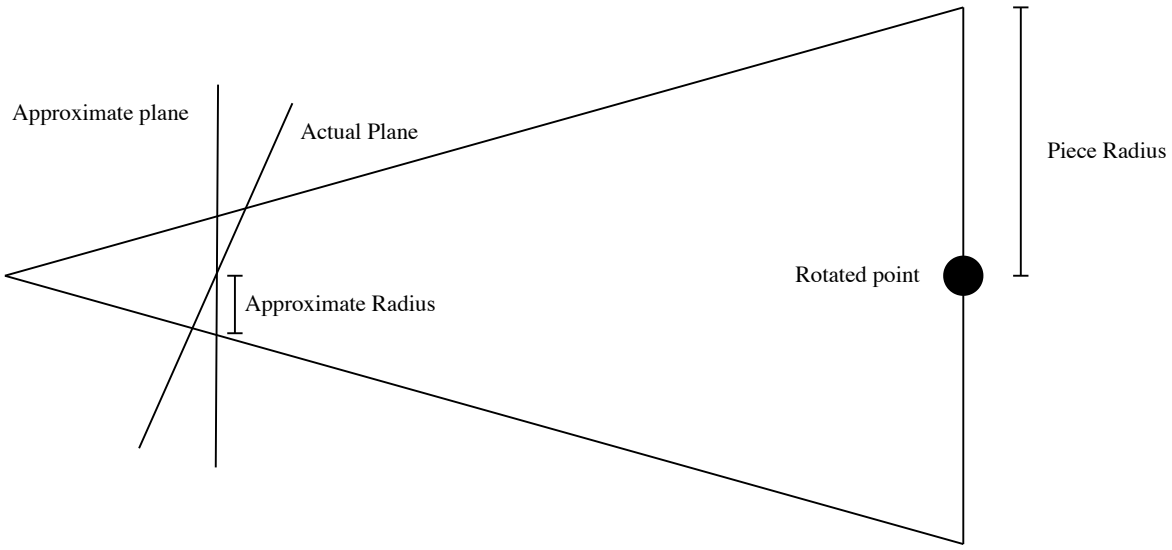


Figure 2.17: Approximating the radius of the projection of a piece.

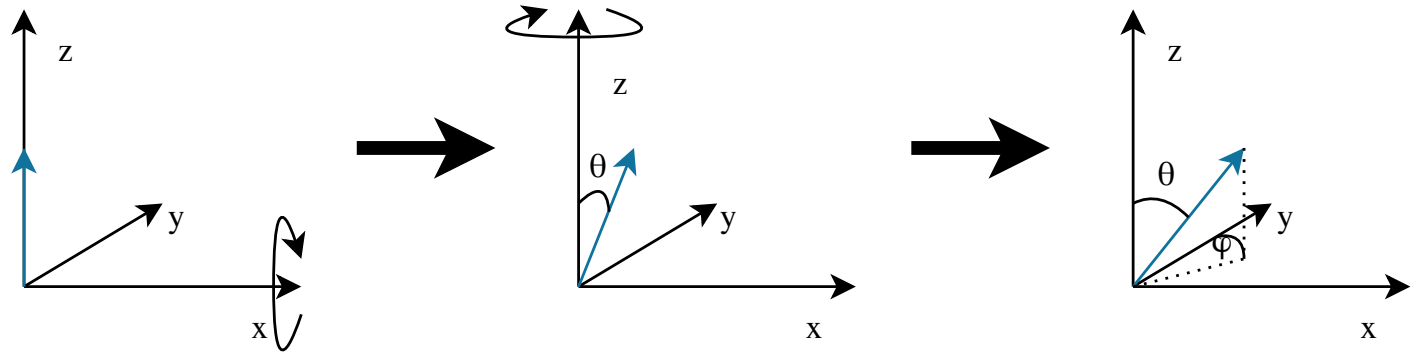


Figure 2.18: Rotating a point.

I now have a method of approximating the projection of a single sphere onto a plane in front of the camera. When projections overlap, it is important to know which takes priority over the other. Observe that spheres with centres closer to the camera will always cover the ones behind them with centres further away from the camera. This is an idea used in z-buffering, where the vertexes are drawn only if it is the closest to the camera along some line. In our case, since we can draw circles easily. It is best to draw the circles furthest away first. This is so that the circles closer to the camera can cover the ones behind them. This can be done by sorting the circles from the longest to the shortest distance to the camera and drawing them onto the screen one by one.

To summarise, transforming a piece to its correct position relative to the involves rotating it around two axes, and then translating it so that the camera at the origin is far enough away. Once a piece has been transformed, it can be approximated and the results are a coordinate for the centre of the approximated circle on the plane and its radius. These circles can then be sorted by distance to the camera to ensure the correct order. Figure 2.19 outlines the process.

Communicating Information

Objective 6 involves the visual communication of information. The power of a good representation of a game board is great. Possible moves should be communicated to the player clearly and efficiently.

Firstly, a circle must appear in the position the player's cursor is hovering over. Since we can already draw pieces, it is easy to draw a white outline over the piece to highlight it. The problem is determining whether the cursor is intersecting the piece or not. The equation for the distance from a line can be used. The line is represented in the vector form $a + \lambda\vec{d}$ and the point with position vector $\vec{b}$. The perpendicular distance is given by $\frac{\vec{d}\times(\vec{b}-\vec{a})}{|\vec{d}|}$. The line in our case is the line from

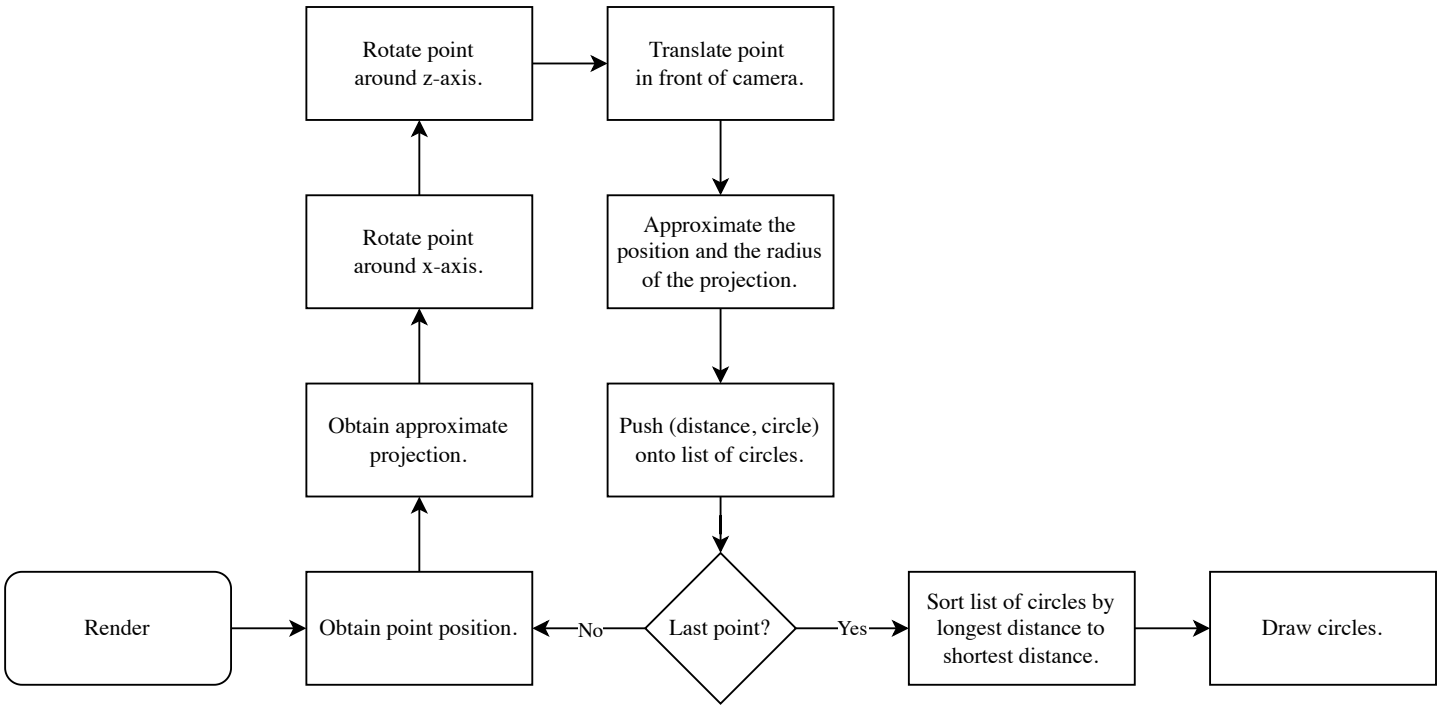


Figure 2.19: Flowchart of the rendering process.

the origin to the cursor position on the display. This means that the line always passes through the origin ($a = (0, 0, 0)$) and that the direction vector is from the origin to the position of the cursor on the projection plane. Assuming the distance of the plane from the origin is 1, the line (in vector form) is: $\lambda \begin{pmatrix} m_x \\ 1 \\ m_y \end{pmatrix}$ where m_x and m_y are the x and y positions of the cursor (relative to the centre of the screen and scaled to fit the projection plane) respectively.

If the direction vector is normalised ($|\vec{d}| = 1$) then the expression for the distance between the line and the centre of a piece becomes just $|\vec{d} \times \vec{b}|$. The cross-product between vectors $\vec{a}$ and $\vec{b}$ represented by arrays of length 3 is given by:

$$\begin{pmatrix} a[1] \times b[2] - a[2] \times b[1] \\ a[2] \times b[0] - a[0] \times b[2] \\ a[0] \times b[1] - a[1] \times b[0] \end{pmatrix}$$

Ghost pieces should be present in locations where pieces may be placed. Since pieces can be rendered easily, they should be rendered in a similar fashion. The positions of all ghost pieces can be generated before rendering begins. Then, only the ghost pieces in valid positions are rendered in each frame.

2.4 Classes

Object-oriented programming (OOP) organises the program around data and objects. The templates for these objects are called classes. A class contains attributes and methods. Some attributes are read-only. Conventionally, these attributes are read through a getter method. For example, the height of a Rectangle class might be read through a method such as `Rectangle.get_height()`.

2.4.1 Pylos

The Pylos class holds a game of Pylos. This means that no game logic should be executed outside of the class. I have to find a way of communicating information between the game class and an interface class (the GUI class in this case).

Input

The main problem is inputting moves into the game. I would like for there to be one input method. This makes it so that when the user clicks a position, the GUI class can input that position directly to the game class without any game logic executed in the GUI class. Therefore, it makes sense for the input method to take a position on the board as its input. All inputs can be represented this way.

Once input in the form of a position is given to the input method, it must determine the validity of the move. Of course, this depends on the moves allowed from the position that the current player is in. Therefore, a method for validating input positions is needed.

Then, changes need to be made to the board. This means placing and retrieving pieces from the specified location on the board. Hence, checking whether a piece can be placed there or whether the piece there can be retrieved is also needed.

Finally, the game has to go to the next state. This means the instructions given to the player change. For example, if a player has placed a piece such that, according to the chosen rules, they may retrieve two pieces from the board, the game must know that and only allow the player to retrieve pieces. To that end, the game must also be able to check whether either a square of the same colour has been made or four pieces of the same colour have been placed in a line.

Essentially, the class needs to determine whether an input is valid according to the chosen rules, then transition between the different states of the game while making appropriate changes to the board, and keep track of important information such as how many pieces are left for each player. Figure 2.20 illustrates the process in its entirety.

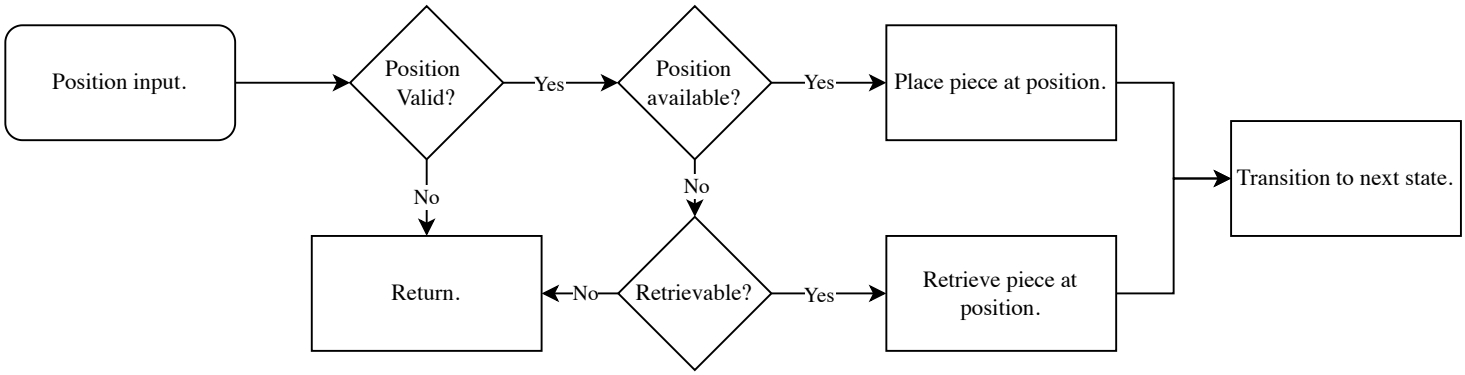


Figure 2.20: Flowchart of input processing.

If the whole process is programmed in the input method, the code would be very cluttered and disorganised, making it difficult to read and understand. The position validation may need to use processes like checking the availability and retrievability of a position. Therefore, it makes sense for each of these components of the process to have a method of its own. These methods are named below. The input method is aptly named `input` with the single argument of a position input.

Process	Method(Arguments)
Validating a position.	`valid(position)`
Checking whether a position is available.	`available(position)`
Checking whether a position has a piece that is retrievable.	`retrievable(position)`
Placing a piece in a position.	`place(position)`
Retrieving a piece in a position.	`retrieve(position)`
Checking whether pieces can be retrieved by having placed a piece in a position.	`check(position)`

Since these processes do not need to be called from outside the class (it may even cause problems if they are), they will be protected methods. In python, this is done by adding two underscores in front of the methods.

Other methods

The GUI displays the game board to the player. Therefore, it needs some way of getting information about the game board. We can call this method `peek`. It will take an argument of a position on the board and return the player of the piece occupying that position.

However, when the minimax algorithm makes moves to the game, the player should not be able to see these moves. Therefore, it is necessary for the Pylos class to know when the computer player is making moves and to give the GUI the correct game board information. It can do this by having the computer player communicate that it is beginning its evaluation and making a copy of the current game state to give to the GUI while the minimax algorithm is running. The aforementioned `peek` method will therefore return the occupancy of a position from the copied board when the minimax algorithm is running and from the real board otherwise.

Lastly, **Objective 1(e)** states that the player should be able to undo and redo moves during the game. These will be methods of their own, `undo` and `redo`. These are public methods as they will be called by the GUI class when the user decides to undo or redo moves by pressing buttons or keys.

Attributes

From the above, some attributes are required. These keep track of the game state and allow other classes to communicate information to the game class.

A public attribute for the computer player to communicate that it has begun its evaluation. We shall call this

Attribute Name	Access Specifier	Description
`thinking`	Public	The computer player sets this to true when it begins its evaluation so that the game class gives the GUI class the correct game board information.
`board`	Protected	Holds the current state of the game board.
`board_copy`	Protected	Holds the state of the game board without moves made by the minimax algorithm during its evaluation.
`state`	Protected	Holds the number of the current state (to be assigned).
`current_player`	Protected	Holds the number of the current player (0 for player 1, 1 for player 2).
`selected_piece`	Protected	Holds the position of a piece to be moved.
`state`	Protected	Holds the size of the game board.
`square`	Protected	Holds a boolean for whether the "Square" rule is enabled or not.
`alignment`	Protected	Holds a boolean for whether the "Alignment" rule is enabled or not.

2.4.2 GUI

The GUI class will handle user interaction and be responsible for displaying everything. The GUI class will have the `run` method to begin the game. It must display menus, the details of which are discussed in the UI section above. These objects include buttons, textboxes, just text, timers, and the three-dimensional game board mentioned in **Objective 5**.

These objects may change appearance from frame to frame. To allow this, the program must iterate through each object on the screen and render them in every frame. Since they all have to be "drawn", they can each have a class of their own, with a `draw` method. This way, they can all be held in a list in the GUI class which enables a loop function to iterate over them and call the `draw` method in each frame.

The objects may call functions themselves such as the quit button quitting the game upon being pressed. To transition between different screens, buttons will be used. A stack will be used to keep track of the current screens. Figure 2.21 shows an example of how the call stack operates.

Each screen has a function associated with it. These functions will be responsible for generating the objects on the screen. These objects will be stored in a list as an attribute of the GUI class. We can call this protected attribute `entities`. As shown in Figure 2.21, whenever a change is made to the call stack, the function at the top of the stack should be called to generate the new screen. It is easy to see this works well. At the start of the program, the only function in the call stack is `home`. `home` can never be popped from the call stack. An empty call stack means there will be no function to call so no screen will be generated.

In pygame, pygame event objects are pushed onto the event queue when they happen. The program can use `pygame.event.get()` to fetch these events and process them every frame. In order for each object to handle these events, the GUI will pass each event as an argument to some event-handling method in each object. These methods will all be named `handle`.

Objective 4 states that the user should be able to replay recent games. This means the games must be stored outside of the GUI class. I will store it in a text file, then load all the games into the GUI class in the protected attribute `__games`. In the `quit` method of the GUI, it will write all of the games to the text file so they may be fetched the next time the program is launched.

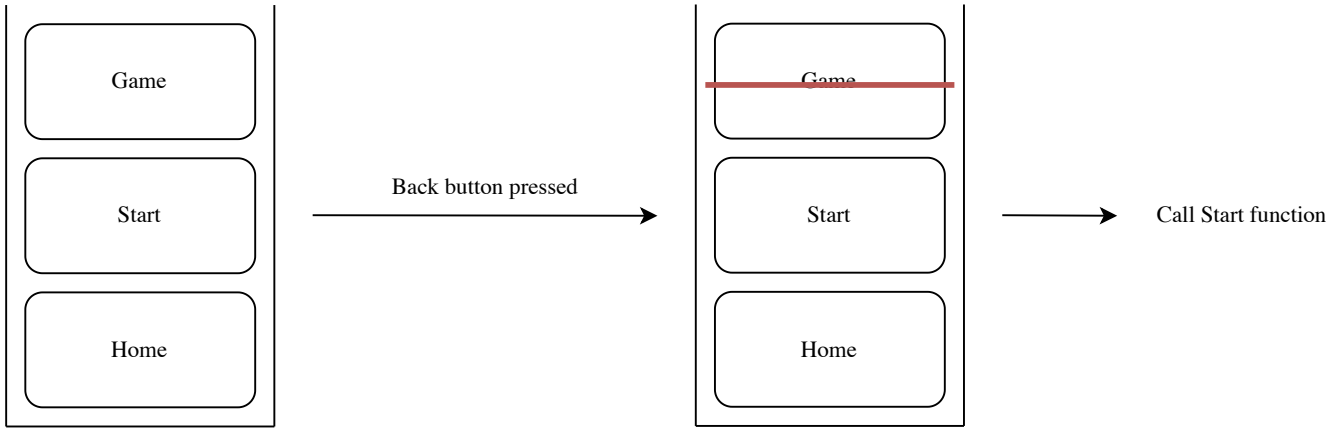


Figure 2.21: An example of call stack operation

2.4.3 Other Classes

These classes are contained by the GUI. They all have a `handle` method and a `draw` method for handling pygame events and rendering themselves respectively. I intend to make all of these objects scalable along with the size of the display window. This can be done by storing the object's position relative to the dimensions of the window. Then every time the window is resized, the new position of the object (in terms of pixels) can be recalculated.

Label

This is simply some text displayed on the screen. The text should be able to be changed (for variable information like the number of pieces a player has left).

Button

When the cursor hovers over the button, the button should look like it is being pressed. When it is clicked, the button calls the function it has been assigned (stored in the public attribute `func`). The button should also be able to be activated by a keypress. The button on the keyboard responsible for this will be stored in the `key` attribute.

Textbox

The `handle` method is similar to the button in that it has to detect when it is clicked, upon which its text can be edited. When the text can be altered (the user has clicked on the textbox), a bar should appear at the end of the text to show that it can now be edited. All keys should cause their symbols to be added to the text on the textbox except the backspace key which should remove the last character of the text. Additionally, it would be more convenient for the user to be able to delete many characters quickly by holding down the backspace key.

Timer

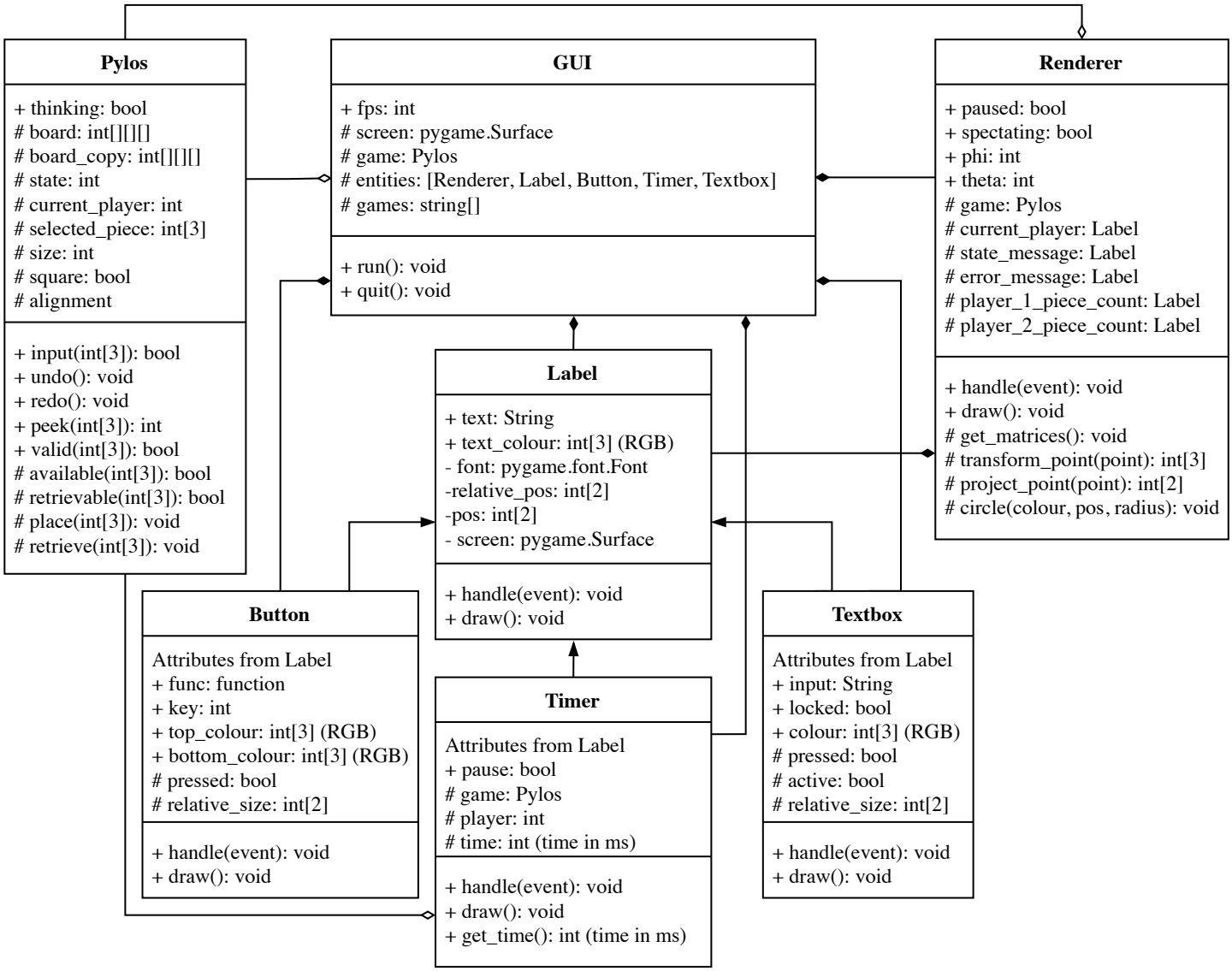
A timer is just a label that has the time elapsed as its text. Therefore, it only needs to keep track of time and update the text accordingly. It should also have a getter method to allow the time to be read from outside the class. Additionally, it should be able to be paused (in case the game is paused). The rest is exactly the same as its superclass, the Label.

Renderer

The Renderer class utilises the algorithm described above for rendering the game board. It should also take input from the user for camera movement and adjust correspondingly. Therefore, it needs to keep track of θ and ϕ.

It needs to contain the current game (for rendering the game board). It also needs to be able to be paused and have a setting where the player cannot interact with the game (for when the CPU player is thinking or when a game is being replayed). Since the information rendering (displaying the appropriate information to the user) is only dependent on the game class, it can also be handled by the renderer class.

2.4.4 Class Relation Diagram



Chapter 3

Technical Solution

I have made many comments throughout the code. Each class has a docstring which documents the attributes and methods that a class has. I have placed similarly formatted comments at the start of each method for supplementary information on arguments and usage. The docstrings are intentionally written such that it gives the information required to use the function without knowing the details of implementation. This is so any programmer may read the docstring and use the function immediately. I also wrote comments along some lines of code to give specific explanations and to give a sense of structure to the code.

3.1 Pylos

3.1.1 `__init__`

The game class and all attributes are initialised within the `__init__` function.

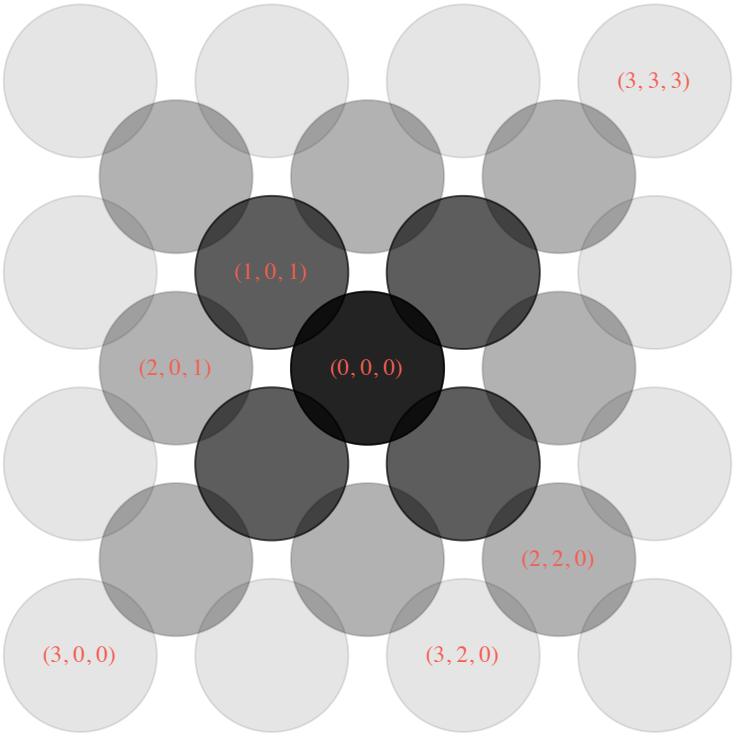


Figure 3.1: Game board indexing

The board is represented as a 3-dimensional list. This is illustrated in Figure 3.1. The copy of the board (`__board_copy`) has the same structure. `__board` will be used to hold the pieces occupying each position on the board so `__board_copy` will be

useful when the CPU player interacts with the board and the minimax algorithm makes moves that should not be displayed to the player. The copy of the board is updated by the setter of `thinking` each time `thinking` is set to `True`.

3.1.2 `valid`

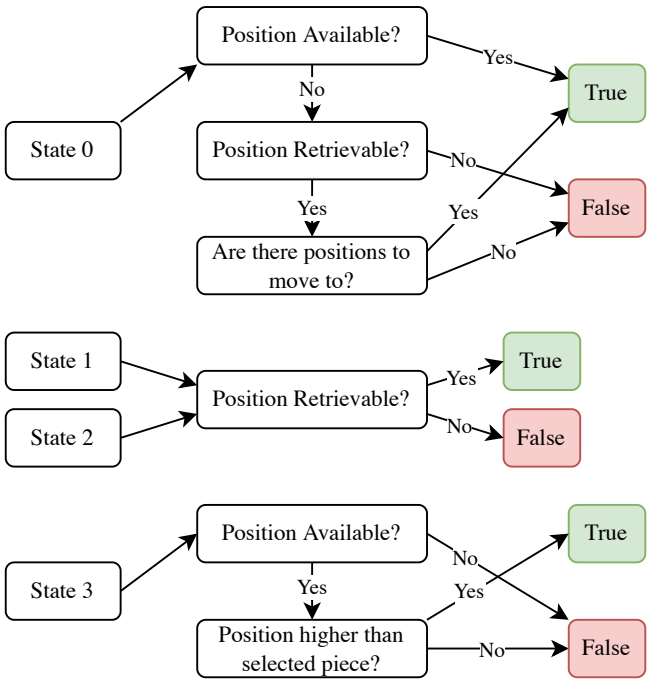


Figure 3.2: Input validation logic

`valid` checks an input (logic shown in Figure 3.2) and updates the error message (`__error_message`) if necessary. `__error_message` can be read through the getter but has no setter so it cannot be changed from outside the class. This is unused in the GUI as the validity of moves is also checked by the GUI. Instead, this information is conveyed through visual cues like ghost pieces and highlighting a piece when the cursor hovers over it if it can be retrieved. However, this is still a helpful error-checking tool for other sorts of user interfaces to make use of in the future.

3.1.3 `peek` and `thinking_peek`

`peek` and `thinking_peek` allow the user access to the private variables `__board` and `__board_copy` respectively. This is important because the minimax algorithm is making moves and changing the state of the board while it "thinks". This means that the GUI will be accessing various different possible states of the board several moves ahead and trying to render them. `__board_copy` prevents this. However, it has to be copied at the right time otherwise it would be very inefficient. This is when the setter is useful. This is because it is only when `thinking` changes from false to true do the board have to be copied.

`__board` and `__board_copy` are private in order to avoid outside classes from changing the values on the board without using the methods of the `Pylos` class. This ensures the correct moves are made and reduces the chance of bugs occurring.

3.1.4 `input`

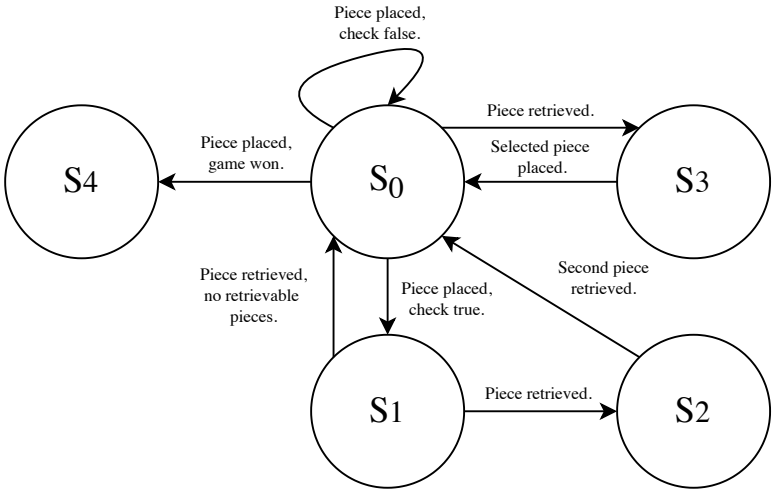


Figure 3.3: Game states transition diagram

`input` uses `valid` to validate the input it receives. The function then executes the state transition logic shown in Figure 3.3. The moves made are pushed onto `__moves_stack` in the format of `[state, current player, position input]`. This allows the undo function below to simply pop the last move off of the stack to undo its effects.

Since the stack is implemented as a list, all the moves made can be kept track of and returned at the end of the game for a replay. The property `history` of the class returns the list of moves made from `__moves_stack`. The resulting list of moves is stored by the GUI.

3.1.5 `undo`

As mentioned above, because the stack structure obeys "First In, Last Out", the undo function can simply pop the last move of the `__moves_stack` and undo the move. It then pushes the move onto `__undone_moves_stack`. This allows `redo` to undo the undone moves.

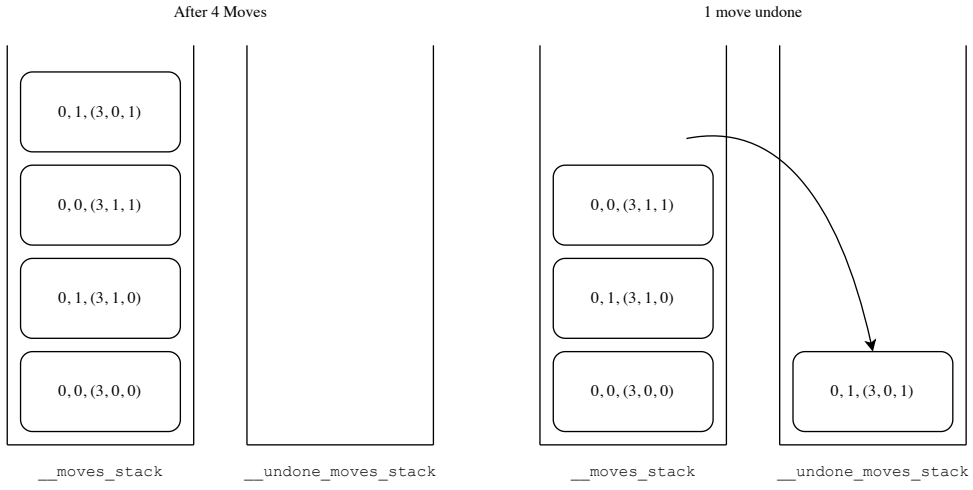


Figure 3.4: What happens when moves are undone.

3.1.6 `redo`

Similar to `undo`, `redo` pops the last move from `__undone_moves_stack` and inputs it to the game. This has the effect of redoing a move. When a move that isn't the move at the top of `__undone_moves_stack` is input to the game, `input` clears

the `__undone_moves_stack`. This is so the moves can no longer be redone and ensures no illegal moves are made.

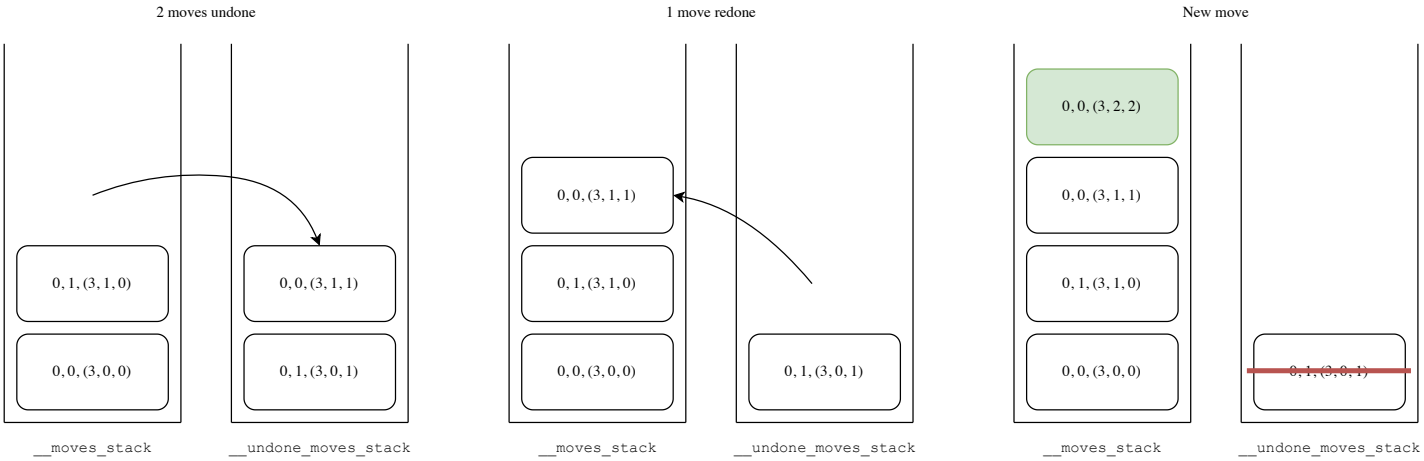


Figure 3.5: What happens when moves are redone.

3.1.7 Other Methods

The remaining methods are getter methods or protected methods (only accessible within the class). The protected methods do exactly what is described in their doc strings and I don't think there is anything of note to say about them. The getter methods offer access to private or protected variables. This is usually to ensure they cannot be changed from outside of the class and/ or its subclasses.

Getter Method	Variable
get_current_player	__current_player
get_thinking_player	__current_player or __thinking_player
get_winner	__winner
get_state	__state
get_piece_count	__piece_count
get_size	__size
get_state_message	__state_reference[__state]
get_error_message	__error_message
get_square	__square
get_alignment	__alignment
get_history	__moves_stack + __undone_moves_stack

3.1.8 Pylos.py

```
class Pylos:
    """A game of Pylos.

        Attributes:
            thinking: Whether the minimax algorithm is making moves.

        Methods:
            valid(pos): Returns True if a position (pos) is valid else False.
            peek(pos): Returns the piece occupying pos (-1 for empty, 0 for player 1, 1 for player 2).
            thinking_peek(pos):
                Returns the piece occupying pos (-1 for empty, 0 for player 1, 1 for player 2)
                without the moves made by the minimax algorithm.
            input(pos): Input pos to the game.
            undo(): Undo the last move.
            get_current_player(): Returns 0 or 1 if it is player 1 or 2's turn to make a move.
            get_winner(): Returns winner of the game (-1 if the game is still going, 0 if player 1, 1 if player 2).
            get_moves_list(): Returns a list of all possible moves from the current game state.
            get_thinking_moves_list(): Returns moves list without the moves made by minimax
```

```
            get_state():
                Returns the state of the game.
                0: Place or move a piece.
                1: Retrieve the first piece.
                2: Retrieve the second piece.
                3: Promote the selected piece.
                4: Game won.
            get_piece_count(): List of [No. pieces player 1 has, No. pieces player 2 has].
            get_size(): Size of the game board (bottom layer is size x size).
            get_state_message(): State message. (e.g. "Place or promote a piece.", "Retrieve a piece.")
            get_error_message(): Error message. (e.g. "Invalid input.", piece cannot be placed here.)
            get_history(): List of moves made.
    """

    def __init__(self, size, square=True, alignment=False):
        """
            Initialises the game and sets all variables to default values.

            Args:
                size: The base (the bottom layer) of the board will be size x size.
                square: Whether pieces may be retrieved if a square of the same colour is made.
                alignment: Whether pieces may be retrieved if a row of four of the same colour is made.
        """
        self.__state = 0
        self.__state_reference = {0: "Place or promote a piece.",
                                  1: "Retrieve the first piece.",
                                  2: "Retrieve the second piece.",
                                  3: "Promote selected piece.",
                                  4: "Game Won."}
        self.__error_message = ""
        self.__current_player = 0
        starting_pieces = ((size * (size + 1) * (2 * size + 1)) / 6 + 1) // 2
        self.__piece_count = [starting_pieces, starting_pieces]
        self.__moves_stack = []
        self.__undone_moves_stack = []

        self.__selected_piece = None
        # Game settings (rules)
        # If a player makes a square of their colour, they can retrieve two pieces
        self.__square = square
        # If a player aligns four or more pieces, in any orientation, they can retrieve two pieces
        self.__alignment = alignment
        self.__size = size
        self.__board = [[[-1 for _ in range(i + 1)] for _ in range(i + 1)] for i in range(size)]
        self.__board_copy = [[[[spot for spot in row] for row in layer] for layer in self.__board], self.__state, self.
    __current_player]

        self.__thinking = False
        self.__thinking_player = 0
        self.__old_moves_list = []

        self.__winner = -1

        # Board structure:
        # [layer, row, column]
        # The ith layer has i + 1 rows and i + 1 columns.
        # The number of layers is indicated by the size argument.
        # Layers are numbered as 0, 1, ..., size - 1.

    def valid(self, pos, update_error_message=False):
        """Checks whether pos is a valid position on board.

            Args:
                pos: Position [layer, row, column] to be checked.
                update_error_message:
                    Whether to update the error_message according to the input.

            Returns:
                True if the position pos is a valid input else False.
        """
        ok = True
        error = ""
        if self.__state == 4:
            error = "Game is over."
            ok = False
        if not (0 <= pos[0] < self.__size and pos[0] >= pos[1] >= 0 <= pos[2] <= pos[0]):
            error = "Invalid Input."
            ok = False
        if self.__state == 0:
            if not self.__available(pos):
                if not self.__retrievable(pos):
                    error = "Invalid Input."
                    ok = False
                promotable = False
```

```
                self.__retrieve(pos)
                for i in range(pos[0]):
                    for j in range(i + 1):
                        for k in range(i + 1):
                            if self.__available((i, j, k)):
                                promotable = True
                self.__place(pos)
                if not promotable:
                    error = "Invalid Input. No available spot above this piece."
                    ok = False
        if 1 <= self.__state <= 2 and not self.__retrievable(pos):
            error = "Piece is not retrievable."
            ok = False
        if self.__state == 3 and (pos[0] >= self.__selected_piece[0] or not self.__available(pos)):
            if pos[0] >= self.__selected_piece[0]:
                error = "Spot is not higher than the retireved piece."
            if not self.__available(pos):
                error = "Spot is not available."
            ok = False
        if update_error_message:
            self.__error_message = error
        return ok

    def peek(self, pos):
        """Returns the occupancy of the spot at pos without minimax moves.

        Args:
            pos: Position [layer, row, column] to peek at.

        Returns:
            -1, 0, 1 for empty, player 1, and player 2 respectively (without the moves the minimax algorithm has made).
        """
        if self.__thinking:
            return self.__board_copy[0][pos[0]][pos[1]][pos[2]]
        return self.__board[pos[0]][pos[1]][pos[2]]

    def input(self, pos):
        """Call to make any move.

        Args:
            pos: [layer, row, column] indicates the position to be operated on.

        Returns: True if the move was made else False.
        """

        # Checks whether the input is valid or not.
        self.__error_message = ""
        if not self.valid(pos, True):
            if self.__state == 3 and not self.__thinking:
                self.undo(False)
            if not self.__thinking:
                print(self.__error_message)
            return False

        self.__moves_stack.append((self.__state, self.__current_player, pos))
        if not self.__undone_moves_stack or pos != self.__undone_moves_stack[-1][2]:
            self.__undone_moves_stack.clear()
            if not self.__thinking:
                print("Move input: ", pos)
                print(self.get_error_message())
                print(self.get_state_message())

        # Logic for game state transitions and changes to the board.
        if self.__state == 0:
            if self.__retrievable(pos):
                self.__retrieve(pos)
                self.__selected_piece = pos
                self.__state = 3
            else:
                self.__place(pos)
                if self.__check(pos):
                    self.__state = 1
                else:
                    self.__next_turn()
        elif self.__state == 1:
            self.__retrieve(pos)
            if len(self.get_moves_list()) > 0:
                self.__state = 2
            else:
                self.__state = 0
                self.__next_turn()
        elif self.__state == 2:
            self.__retrieve(pos)
            self.__state = 0
```

```
186             self.__next_turn()
187         elif self.__state == 3:
188             self.__place(pos)
189             if self.__check(pos):
190                 self.__state = 1
191             else:
192                 self.__state = 0
193                 self.__next_turn()
194         return True
195 
196     def undo(self, redo=True):
197         """Undoes last move."""
198         if self.__moves_stack:
199             self.__error_message = ""
200             move = self.__moves_stack.pop()
201             if redo:
202                 self.__undone_moves_stack.append(move)
203             self.__state, self.__current_player, pos = move
204             if self.__available(pos):
205                 self.__place(pos)
206             else:
207                 self.__retrieve(pos)
208             print("Move undone.")
209         else:
210             self.__error_message = "No moves to undo."
211 
212     def redo(self):
213         if self.__undone_moves_stack:
214             self.__error_message = ""
215             _, _, pos = self.__undone_moves_stack[-1]
216             self.input(pos)
217             self.__undone_moves_stack.pop()
218             print("Move redone.")
219         else:
220             self.__error_message = "No moves to redo."
221 
222     def __peek(self, pos):
223         """Returns the occupancy of the spot at pos.
224 
225        Args:
226            pos: Position [layer, row, column] to peek at.
227 
228        Returns:
229            -1, 0, 1 for empty, player 1, and player 2 respectively.
230        """
231         return self.__board[pos[0]][pos[1]][pos[2]]
232 
233     def __available(self, pos):
234         """Checks whether a piece can be placed at the spot at pos.
235 
236        Args:
237            pos: Position [layer, row, column] to check if a piece can be placed at.
238 
239        Returns:
240            True if a piece can be placed at pos else False.
241        """
242         if self.__board[pos[0]][pos[1]][pos[2]] != -1 or self.__piece_count[self.__current_player] <= 0:
243             return False
244         if pos[0] == self.__size - 1:
245             return True
246         for dx in range(2):
247             for dy in range(2):
248                 nx, ny = pos[1] + dx, pos[2] + dy
249                 if self.__peek((pos[0] + 1, nx, ny)) == -1:
250                     return False
251         return True
252 
253     def __retrievable(self, pos):
254         """Checks whether there exists a piece at pos that is retrievable.
255 
256        Args:
257            pos: Position [layer, row, column] to check if a piece can be retrieved from.
258 
259        Returns:
260            True if a piece exists at pos and can be retrieved else False.
261        """
262         if self.__peek(pos) == -1 or self.__peek(pos) != self.__current_player:
263             return False
264         if pos[0] == 0:
265             return True
266         for dx in range(2):
267             for dy in range(2):
268                 nx, ny = pos[1] - dx, pos[2] - dy
269                 if pos[0] > nx >= 0 <= ny < pos[0] and self.__peek((pos[0] - 1, nx, ny)) != -1:
```

```
270                     return False
271             return True
272
273     def __place(self, pos):
274         """Places piece at pos.
275
276            Args:
277                pos: [layer, row, column] position to place a piece.
278
279            Returns:
280                True if placed else False
281         """
282         if not self.__available(pos):
283             return False
284         self.__piece_count[self.__current_player] -= 1
285         self.__board[pos[0]][pos[1]][pos[2]] = self.__current_player
286         if pos == (0, 0, 0):
287             self.__state = 4
288             self.__winner = self.__current_player
289         return True
290
291     def __retrieve(self, pos):
292         """Retrieves a piece from pos if the current player occupies it.
293
294        Args:
295            pos: [layer, row, column] indicates the position of the piece to be retrieved.
296
297        Returns:
298            True if retrieved else False.
299         """
300         if not self.__retrievable(pos):
301             return False
302         self.__piece_count[self.__current_player] += 1
303         self.__board[pos[0]][pos[1]][pos[2]] = -1
304         return True
305
306     def __check(self, pos):
307         """Checks if pieces can be retrieved according to the rules (Square and Alignment).
308
309        Args:
310            pos: [layer, row, column] indicates the position of the piece to be checked.
311
312        Returns:
313            True if pieces may be retrieved else False.
314         """
315         if self.__square:
316             for mx, my in [(1, 1), (1, -1), (-1, 1), (-1, -1)]:
317                 if not 0 <= pos[1] + mx <= pos[0] >= pos[2] + my >= 0:
318                     continue
319                 for dx, dy in [(1, 0), (1, 1), (0, 1)]:
320                     npos = [pos[0], pos[1] + (dx * mx), pos[2] + (dy * my)]
321                     if self.__peek(npos) != self.__current_player:
322                         break
323                 else:
324                     return True
325         if self.__alignment:
326             for dx, dy in ((1, 0), (0, 1)):
327                 count = 0
328                 for checking_pos in zip((pos[0] for _ in range(4)), (pos[1] + i * dx for i in range(4)),
329                                         (pos[2] + i * dy for i in range(4))):
330                     if not (0 <= checking_pos[1] <= checking_pos[0] >= checking_pos[2] >= 0) or self.__peek(checking_pos)
    != self.__current_player:
331                         break
332                     else:
333                         count += 1
334                 for checking_pos in zip((pos[0] for _ in range(1, 4)), (pos[1] - i * dx for i in range(1, 4)),
335                                         (pos[2] - i * dy for i in range(1, 4))):
336                     if not (0 <= checking_pos[1] <= checking_pos[0] >= checking_pos[2] >= 0) or self.__peek(checking_pos)
    != self.__current_player:
337                         break
338                     else:
339                         count += 1
340                 if count >= 4:
341                     return True
342         return False
343
344     def __next_turn(self):  # Changes player
345         self.__current_player = 1 - self.__current_player
346         if len(self.get_moves_list()) == 0:
347             self.__current_player = 1 - self.__current_player
348
349     @property
350     def thinking(self):
351         return self.__thinking
```

```
352
353     @thinking.setter
354     def thinking(self, b):
355         if not self.__thinking and b:
356             self.__thinking_player = self.__current_player
357             self.__board_copy = [[[[spot for spot in row] for row in layer] for layer in self.__board], self.__state,
        self.__current_player]
358             self.__old_moves_list = self.get_moves_list()
359         if self.__thinking and not b:
360             self.__board = [[[spot for spot in row] for row in layer] for layer in self.__board_copy[0]]
361             self.__state = self.__board_copy[1]
362             self.__current_player = self.__board_copy[2]
363         self.__thinking = b
364
365     def get_current_player(self):  # Integer of 0 or 1 if the current player is player 1 or player 2 respectively.
366         return self.__current_player
367
368     def get_thinking_player(self):
369         if self.thinking:
370             return self.__thinking_player
371         return self.__current_player
372
373     def get_winner(self):  # Returns winner if game is won, else -1
374         return self.__winner
375
376     def get_moves_list(self):  # List of available moves from the current game state.
377         return [(i, j, k)
378                 for i in range(self.__size)
379                 for j in range(i + 1)
380                 for k in range(i + 1)
381                 if self.valid((i, j, k))]
382
383     def get_thinking_moves_list(self):  # moves_list without moves made by minimax algorithm.
384         if self.__thinking:
385             return self.__old_moves_list
386         return self.get_moves_list()
387
388     def get_state(self):
389         return self.__state
390
391     def get_piece_count(self):
392         return list(map(int, self.__piece_count))
393
394     def get_size(self):
395         return self.__size
396
397     def get_state_message(self):
398         return self.__state_reference[self.__state]
399
400     def get_error_message(self):
401         return self.__error_message
402
403     def get_square(self):
404         return self.__square
405
406     def get_alignment(self):
407         return self.__alignment
408
409     def get_history(self):
410         return [move for state, player, move in self.__moves_stack + self.__undone_moves_stack[::-1]]
411
412     def undone(self):
413         return len(self.__undone_moves_stack) > 0
```

3.2 Renderer

The renderer handles the rendering of the game board. The procedure is described in the design chapter. Here, it is implemented as a multitude of functions, each carrying out a step of the process. When all processes are concluded, we are left with the description of a circle somewhere on the screen which approximates the projection of a spherical playing piece.

Functions such as `get_magnitude`, `unit`, `cross`, `matmul` are all defined outside of the class but used within the class. Alternatively, I could have implemented them as static methods but I did not since they were only going to be used within the Renderer class.

3.2.1 handle

The renderer stores Labels for the purpose of rendering game information such as the number of pieces left for each player and instructions for play. These are held in the protected attribute `__entities`. In this `handle` method, it iterates through its entities to allow the Labels to handle events.

The renderer also keeps track of the camera's position using the angles θ and ϕ (discussed in the Design chapter). When the left mouse button is pressed, the protected attribute `__pressed` is set to true. This tells the `draw` method to update the angles proportional to the change in mouse position. Keypresses are also available for camera control. This method checks whether the event is the input of "W", "S", "A", or "D". The rate of change of the camera angle is held in the protected attribute `delta_angles`. For example, the pressing of the W key increments the change in angle x by some set amount (held in `__d_theta` and `__d_phi`). Holding these keys down produces a steady movement in the camera as the changes between frames of the angles θ and ϕ are set to a constant.

3.2.2 draw

Updates

If `__pressed` is true, the difference between the cursor's current position and its position last frame (held in the attribute `__last_pos`) is calculated. θ and ϕ are incremented in proportion to that difference. Otherwise, the changes to the angles specified by `__delta_angles` are executed. If any change to the angles is made, the rotational matrices (stored in `__rot_x` and `__rot_z`) are updated by calling the `__get_matrices` method.

Animation

I have implemented an animation into the game to avoid the abrupt placement and retrieval of pieces. It works by changing the position of the playing pieces gradually. Instead of the pieces seemingly popping into existence, they start off high above the board and descend onto the board when the player places them. Similarly, the pieces retreat quickly to their positions above the board and out of view when the pieces are retrieved. I tried to find a function that could make the placement of the pieces seem natural. This means slowing down when the piece is near the board and speeding up as the piece gets further away. I ended up decreasing the z position (z-axis goes vertically up and down) of the piece by $e^{\frac{1}{n}}$ (where n is the number of frames since the piece has been placed) each frame after the piece has been placed. The pieces exponentially slow down as they approach the correct position, achieving the desired effect.

Projection

Within the `draw` function, the process shown in Figure 2.19 is executed. It also calculates the intersection of a line and a point mentioned in the design chapter to determine whether to highlight a piece.

`__get_matrices` updates the rotational matrices with angles θ and ϕ. `__transform_point(point)` rotates the point given and translates it to the correct position. It applies the rotational matrices on the point using the `matmul` function outside the class. The translation applied to one point is the same for all other points so this translation is stored in the protected attribute `__origin_distance`. `__transform_point(point)` essentially returns $\vec{a} + R_z(\phi)R_x(\theta)\vec{p}$ where $\vec{p}$ is the given point and $\vec{a}$ is the vector from the camera to the centre of rotation of the board. `__project_point(point)` approximates the position of the projection.

The following code (lines 368-374 in GUI.py) checks for the intersection between the cursor and a position/ piece in the scene. This is in a loop iterating through all the positions. It allows the computer to highlight a position if the piece in the position can be retrieved or a piece can be placed in the position.

```
# Shortest distance from the line to the centre of a piece
perpendicular_distance = get_magnitude(cross(cursor, point))
# Check whether the line collides with the piece
if not self.__game.undone() and not self.spectating and perpendicular_distance < self.__piece_radius and distance <
    closest:
    self.__hovered_pos = pos
    hover_circle = (coord, radius)
    closest = distance
```

`closest` holds the shortest distance from the camera to a piece that is hovered upon until the current piece. By only changing `__hovered_pos` and `hover_circle` when the distance to the current piece is less than `closest`, it ensures the highlighted piece is the frontmost piece in all pieces intersected by the line traced from the origin to the cursor.

Drawing

Each rendered piece (circle represented by coordinates of its centre and its radius) is appended onto a list with its y-value. This list is sorted from largest to smallest y-value. The list is then iterated over and each circle is drawn onto the pygame display surface held in `Renderer` as `__screen` using the `__circle` method. As mentioned in the Design chapter, this concept is borrowed from z-buffering applied on a smaller scale. The `__circle` method works around pygame's interactions with RGBA values. A surface must be specifically flagged with alpha values for the module to draw transparent shapes onto it. Therefore, a new surface is created each time, with the transparent circle on it, and drawn onto the `__screen` using the `pygame.__screen.blit` method. This works well (see the testing chapter). After this, all the Labels within `__entities` are drawn as well. This is so that the Labels appear over the pieces and not the other way around.

3.3 Label

The label displays text on the screen. Since objects such as a button, a textbox, and a timer all need to render text, it makes sense for them to inherit from the Label class. They can call `super().draw()` and `super().handle(event)` to render the text and handle any events relating to the text. The private attribute `_text` has a getter and a setter so it can be accessed and changed from outside of the class using `Label.text` (I used Python's `property` decorator). The other private attributes are private so that Label's subclasses (Button, Timer, and Textbox) can use them. In pygame, the text has to be rendered onto a surface and then drawn onto the main screen using a `Rect` object as its target. I created the protected attributes `text_surf` and `text_rect` for this.

The `handle` method resizes `text_surf` and `text_rect` when the window is resized. Events are stored as pygame event objects with attribute `type` set to `pygame.WINDOWRESIZED`. When the window is resized, `handle` updates the position of the private attribute `pos` using `relative_pos` which is the position of the Label relative to the width and height of the screen.

3.4 Button

This can be thought of as a pressable button with a label on it. The button consists of two rectangles with rounded corners. The top rectangle is elevated when the cursor is not on top of it. The `Rect` class in pygame stores a rectangle. The top and bottom rectangles are instances of `Rect` held in `__top` and `__bottom` respectively.

The `handle` method of Button calls the `handle` method of its superclass (Label). This repositions its text upon window resizing. It also handles the repositioning of `__top` and `__bottom` rectangles in a similar fashion. The widths and heights of the two rectangles are scaled accordingly as well. The width and height relative to that of the screen are stored in the protected attribute `__relative_size`. It checks whether the key assigned to the button has been pressed. If it has, it calls `func`.

When the cursor hovers over the button, the button looks like it is being pressed. Every frame, the `draw` method checks whether the cursor is on the button using `Rect.collidepoint(pygame.mouse.get_pos())`. The position of the top `Rect` object is updated accordingly. When the button is clicked, it calls the function it has been assigned (stored in the public attribute `func`). The `draw` method also checks whether the mouse button has been clicked when the cursor is hovering over the button. If it is, the method calls `func`. Finally, the `draw` calls `draw` of its superclass to render the text on the button. It updates the position of the text to the centre of the top rectangle before this.

The colour of the top and bottom rectangles can be changed using the public attributes `top_colour` and `bottom_colour` respectively.

3.5 Textbox

The textbox takes text input. However, it must only take input when clicked. `__active` is set to true if the conditions to input text are fulfilled. The method used is exactly that in the `Button` class. The `Rect` object representing the textbox is stored in `__rect`. The `handle` method also checks whether keys have been pressed and adds the Unicode symbol input by the keypress to `__input` only if `__active` is true. As long as `__active` is managed properly, the textbox will operate correctly.

3.6 Timer

The timer stores the time elapsed in milliseconds in the protected `__time`. Since the timer depends heavily on the game state (whether it is the timed player's turn), the Timer class holds the `Pylos` class in the protected attribute `__game`. The timer may also be paused by setting the public attribute `pause` to true and unpaused by setting it to false. The `draw` method is

called every tick. Depending on the game conditions and whether the timer is paused, it increments the time. The text is then updated and the `draw` method of its superclass, `Label`, is called to render the text.

3.7 GUI

The GUI is the class which holds everything. To start the game, create an instance of the class with suitable parameters. Then, start the game by calling `GUI.run()`.

Each screen has its respective function defined within `GUI.run`. The functions update the protected attributes `entities` and `call_stack` (how the call stack operates is explained in the design chapter and an example is given in Figure 2.21). They may also define some new functions to be called by the Button objects they put in `entities`. For example, on the Start screen, there are many buttons with functions that change other objects on the screen. These are defined in the `start` function.

The `loop` function is the main program loop. It iterates over all events in every frame to pass the events to the `handle` method of each object in `entities`. Then, it iterates over each object and calls the `draw` method. This way, all the objects handle any events that have happened since the last frame, and are then drawn on the screen with updates.

3.7.1 GUI.py

```
import numpy as np
import Pylos
import pygame
import Brain

pygame.init()

clock = pygame.time.Clock()


def get_magnitude(vector):
    """Gets the magnitude of a vector.
    Args:
        vector: a vector (the magnitude of which is to be measured)
    Returns:
        the magnitude of the vector given (|vector|)
    """
    return np.sqrt(sum(component ** 2 for component in vector))


def unit(vector):
    """Gets normalised vector of the vector given.
    Args:
        vector: A vector (a normalised version of which will be returned).
    Returns:
        A unit vector (magnitude = 1) in the same direction as the given vector.
    """
    magnitude = get_magnitude(vector)
    return [i / magnitude for i in vector]


def cross(a, b):
    """Calculates the cross product of two length 3 vectors.
    Args:
        a: Left vector.
        b: Right vector.
    Returns:
        The cross product of a and b.
    """
    return [a[1] * b[2] - a[2] * b[1], a[2] * b[0] - a[0] * b[2], a[0] * b[1] - a[1] * b[0]]


def matmul(m, v):
    """Calculates the product of a matrix and vector
        Args:
            m: Left nxn matrix.
            v: Right nx1 vector.
        Returns:
            The product of m and v.
    """
    if len(m) != len(v):
        return None

    n = len(v)
    product = [0] * n
    for i in range(n):
```

```
        for j in range(n):
            product[i] += m[i][j] * v[j]
    return product


class Renderer:
    """Holds the image of the game and handles interactions with the board.
    Attributes:
        pause: Whether the rendering is paused.
        spectating: Whether the player is spectating (player cannot interact with the game).

    Methods:
        handle(event): Handles pygame event.
        draw(): Renders frame, call every frame.
    """

    def __init__(self, screen, game, x0=0, y0=0, width=1600, height=1000,
                 angle_x=0, angle_z=0, scale=400,
                 background_colour=(255, 255, 255, 255),
                 piece_colours=((255, 0, 0, 255), (0, 255, 0, 255), (100, 100, 100, 100)),
                 spectating=False,
                 player_names=("Player 1", "Player 2")):
        """Initialises the Renderer class.

        Args:
            screen: Reference to the pygame surface to draw on.
            game: Pylos game object for the renderer to interact with.
            x0: Horizontal shift to the right for the rendered image.
            y0: Vertical downward shift for the rendered image.
            width: Width of the rendered image.
            height: Height of the rendered image.
            angle_x: Starting vertical angle of board.
            angle_z: Starting horizontal angle of board.
            scale: The ratio of display width/ width of screen in front of camera.
            background_colour: Background colour
            piece_colours: the RGBA values of pieces of player 1, player 2, and ghost piece in that order.
            spectating: If true, the renderer will not detect input to the game.
            player_names: The display names of player 1, player 2 respectively.
        """

        self.__screen = screen
        self.__game = game
        self.spectating = spectating

        self.__player_names = player_names

        self.pause = False
        self.__last_pos = [0, 0]
        self.__hovered_pos = False
        self.__pressed = 0
        self.__d_angles = [0, 0]
        self.__time = 0

        # Rotational matrices
        self.theta = angle_x
        self.phi = angle_z
        self.__rot_x = None
        self.__rot_z = None
        self.__get_matrices()

        # Display settings
        self.__origin_distance = np.array([0., 5., 1.])
        self.__offset = np.array([x0, y0])
        self.__zoom = 1
        self.__scale = scale
        self.__back_ground_colour = background_colour
        self.__colours = piece_colours

        # Animation settings
        self.__d_theta = 0.05
        self.__d_phi = 0.05
        self.__animation_frames = 20
        self.__animation_speed = 10
        self.__animation_magnitude = 0.15

        # Animation is - magnitude * e^((frames - t)/ speed) every frame for placing
        # and + magnitude * e^(t / speed) every frame for lifting.

        # height = magnitude * (e^(1/s) + e^(2/s) + e^(3/s) + ... + e^(frames/s))
        # = magnitude * (e^(frames + 1 / s) - e^(1/s))/(e^(1/s) - 1)
        animation_height = self.__animation_magnitude * \
                           (np.exp((self.__animation_frames + 1) / self.__animation_speed)
                            - np.exp(1 / self.__animation_speed)) / (np.exp(1 / self.__animation_speed) - 1)

```

```
141         # Labels for text display
142         self.__font = pygame.font.Font("Assets/Font/EBGaramond-Normal.ttf", 30)
143         self.__current_player = Label(screen,
144                                       (width / 2, height / 10),
145                                       f"Current player: {self.__player_names[self.__game.get_current_player()]}",
146                                       font_size=30 / 1600,
147                                       text_colour=(255, 255, 255))
148         self.__status_message = Label(screen,
149                                       (width / 2, height * 2 / 10),
150                                       self.__game.get_state_message(),
151                                       font_size=40 / 1600,
152                                       text_colour=(255, 255, 255))
153         self.__error_message = Label(screen,
154                                      (width / 2, height * 3 / 10),
155                                      self.__game.get_error_message(),
156                                      font="Assets/Font/EBGaramond-Italic.ttf", font_size=30 / 1600,
157                                      text_colour=(255, 255, 255))
158         self.__player_1_piece_count = Label(screen,
159                                             (width / 8, height / 9),
160                                             f"{self.__player_names[0]}: {self.__game.get_piece_count()[0]}",
161                                             font_size=40 / 1600, text_colour=piece_colours[0])
162         self.__player_2_piece_count = Label(screen,
163                                             (width * 7 / 8, height / 9),
164                                             f"{self.__player_names[1]}: {self.__game.get_piece_count()[1]}",
165                                             font_size=40 / 1600,
166                                             text_colour=piece_colours[1])
167         self.__entities = [self.__current_player, self.__status_message, self.__error_message,
168                            self.__player_1_piece_count, self.__player_2_piece_count]
169 
170         # Board configurations
171         self.__piece_radius = 2 / game.get_size()
172         self.__pieces = [[[np.array([(j - i * 0.5) * 4 / game.get_size(),
173                                      (k - i * 0.5) * 4 / game.get_size(),
174                                      (4 / game.get_size()) * np.sqrt(0.5) * (
175                                              1 + i - game.get_size()) - animation_height])
176                            for k in range(i + 1)]
177                           for j in range(i + 1)]
178                          for i in range(game.get_size())]
179 
180         self.__step = [[[-self.__animation_frames - 1
181                          for _ in range(i + 1)]
182                         for _ in range(i + 1)]
183                        for i in range(game.get_size())]
184 
185         self.__last = [[[game.peek((i, j, k)) for k in range(i + 1)] for j in range(i + 1)] for i in
186                        range(game.get_size())]
187 
188         self.__ghost_pieces = [
189             [[np.array([(j - i * 0.5) * 4 / game.get_size(),
190                         (k - i * 0.5) * 4 / game.get_size(),
191                         (4 / game.get_size()) * np.sqrt(0.5) * (1 + i - game.get_size())])
192               for k in range(i + 1)]
193              for j in range(i + 1)]
194             for i in range(game.get_size())]
195 
196         for i in range(game.get_size()):
197             for j in range(i + 1):
198                 for k in range(i + 1):
199                     if self.__last[i][j][k] != -1:
200                         self.__pieces[i][j][k][2] += animation_height
201                         self.__step[i][j][k] = 0
202 
203     def __get_matrices(self):
204         """Updates matrices of self._rot_x, self._rot_z.
205         y-axis facing away from camera so rotation by x-axis goes up and down and rotation by z goes side to side.
206         """
207         # rotational matrix by _angle_z along z axis
208         self.__rot_z = np.array([[np.cos(self.phi), -np.sin(self.phi), 0],
209                                  [np.sin(self.phi), np.cos(self.phi), 0],
210                                  [0, 0, 1]])
211         # rotational matrix by _angle_x along x axis
212         self.__rot_x = np.array([[1, 0, 0],
213                                  [0, np.cos(self.theta), -np.sin(self.theta)],
214                                  [0, np.sin(self.theta), np.cos(self.theta)]])
215 
216     def __transform_point(self, point):
217         """Rotates given point and translates to actual position relative to camera."""
218         # Rotating the given point using matrix multiplication.
219         rotated_point = matmul(self.__rot_z, point)
220         rotated_point = matmul(self.__rot_x, rotated_point)
221 
222         # translate point so that the point we are rotating around is at [0, origin_distance, 0]
223         rotated_point += self.__origin_distance
224         return rotated_point
```

```
225
226     def __project_point(self, point):
227         """Projects point onto screen.
228         Args:
229             point: point to be projected onto the screen.
230         Returns: Coordinates of projected point on the screen.
231         """
232         if point[1] < 0:
233             return False
234         m = self.__scale * self.__zoom / point[1]
235         return [point[0] * m + self.__screen.get_width() / 2, point[2] * m + self.__screen.get_height() / 2]
236
237     def __circle(self, colour, pos, radius):
238         """Draws circle.
239
240         Args:
241             colour: The RGBA value of the colour of the piece.
242             pos: The position for the circle to be drawn. (0, 0) is top left.
243             radius: Radius of the circle to be drawn.
244         """
245         if pos[0] - radius >= self.__screen.get_width() or pos[1] - radius >= self.__screen.get_height() or \
246                 pos[0] <= -radius or pos[1] <= -radius or radius >= self.__screen.get_width() / 2:
247             return
248
249         image = pygame.Surface((radius * 2, radius * 2), flags=pygame.SRCALPHA)
250         pygame.draw.circle(image, colour, (radius, radius), radius)
251         pygame.draw.circle(image, (0, 0, 0, colour[3]), (radius, radius), radius, width=2)
252         self.__screen.blit(image, (pos[0] - radius, pos[1] - radius))
253
254     def handle(self, event):
255         """Handles Pygame events."""
256         for entity in self.__entities:
257             entity.handle(event)
258         if self.pause:
259             return
260         # Start camera movement.
261         if event.type == pygame.MOUSEBUTTONDOWN:
262             if event.button == 1:
263                 self.__pressed = True
264                 self.__last_pos = pygame.mouse.get_pos()
265             if event.button == 4:
266                 self.__origin_distance[1] = min(30, max(self.__origin_distance[1] - 0.04, 1))
267             if event.button == 5:
268                 self.__origin_distance[1] = min(30, max(self.__origin_distance[1] + 0.04, 1))
269         if event.type == pygame.MOUSEBUTTONUP:
270             # Place piece.
271             if event.button == 1:
272                 self.__pressed = False
273             if event.button == 3 and self.__hovered_pos and not self.__game.thinking:
274                 place_piece = pygame.mixer.Sound("Assets/SFX/Place_Piece.wav")
275                 place_piece.play()
276                 self.__game.input(self.__hovered_pos)
277                 for move in self.__game.get_moves_list():
278                     if not self.__game.valid(move, True):
279                         print(self.__game.get_error_message())
280                         print(move)
281                 self.__hovered_pos = False
282
283         # Camera movement keys
284         if event.type == pygame.KEYDOWN:
285             if event.key == pygame.K_d:
286                 self.__d_angles[0] -= self.__d_phi
287             if event.key == pygame.K_a:
288                 self.__d_angles[0] += self.__d_phi
289             if event.key == pygame.K_w:
290                 self.__d_angles[1] -= self.__d_theta
291             if event.key == pygame.K_s:
292                 self.__d_angles[1] += self.__d_theta
293         if event.type == pygame.KEYUP:
294             if event.key == pygame.K_d:
295                 self.__d_angles[0] += self.__d_phi
296             if event.key == pygame.K_a:
297                 self.__d_angles[0] -= self.__d_phi
298             if event.key == pygame.K_w:
299                 self.__d_angles[1] += self.__d_theta
300             if event.key == pygame.K_s:
301                 self.__d_angles[1] -= self.__d_theta
302
303     def draw(self):
304         """Renders frame, handles camera movement and inputs to the game.
305         Call every frame.
306         """
307
308         self.__time += clock.get_time()
```

```
309
310         if self.pause:
311             return
312
313         self.__screen.fill(self.__back_ground_colour)
314
315         pos = pygame.mouse.get_pos()
316         cursor = unit(
317             [(pos[0] - self.__screen.get_width() / 2) / self.__scale, self.__zoom,
318              (pos[1] - self.__screen.get_height() / 2) / self.__scale]
319         )
320
321         if self.__pressed:
322             self.phi += (pos[0] - self.__last_pos[0]) * self.__d_phi * 0.1
323             self.theta += (self.__last_pos[1] - pos[1]) * self.__d_theta * 0.1
324             self.__last_pos = pygame.mouse.get_pos()
325         else:
326             self.phi += self.__d_angles[0]
327             self.theta += self.__d_angles[1]
328
329         if self.__pressed or self.__d_angles != [0, 0]:
330             self.__get_matrices()
331
332         self.theta = min(0.2, max(-0.7, self.theta))
333
334         # Rendering
335         hover_circle = None
336         closest = 1000
337         distances = []
338         circles = []
339         count = 0
340         moves_list = self.__game.get_thinking_moves_list()
341         self.__hovered_pos = False
342         for i in range(self.__game.get_size()):
343             for j in range(i + 1):
344                 for k in range(i + 1):
345                     pos = (i, j, k)
346                     player = self.__game.peek(pos)
347
348                     # Check if animation needs to be started.
349                     if player != -1 and self.__step[i][j][k] == -self.__animation_frames - 1:
350                         self.__step[i][j][k] = self.__animation_frames
351                         self.__last[i][j][k] = self.__game.peek(pos)
352                     elif player == -1 and self.__step[i][j][k] == 0:
353                         self.__step[i][j][k] = -1
354
355                     # Animation
356                     if self.__step[i][j][k] != 0 and self.__step[i][j][k] != -self.__animation_frames - 1:
357                         if 0 > self.__step[i][j][k] > -self.__animation_frames - 1:
358                             player = self.__last[i][j][k]
359                         parity = abs(self.__step[i][j][k]) / self.__step[i][j][k]
360                         self.__pieces[i][j][k][2] += parity * self.__animation_magnitude * np.exp(
361                             abs(self.__step[i][j][k] / self.__animation_speed))
362                         self.__step[i][j][k] -= 1
363
364                     if pos not in moves_list and player == -1:
365                         continue
366
367                     colour = self.__colours[player]
368                     if player != -1:
369                         if pos in moves_list:
370                             dimming = 0.15 * np.cos(self.__time / 125) + 0.85
371                         else:
372                             dimming = 0.7
373                         colour = [colour[0] * dimming, colour[1] * dimming, colour[2] * dimming, colour[3]]
374
375                     if player == -1:
376                         point = self.__ghost_pieces[i][j][k]
377                     else:
378                         point = self.__pieces[i][j][k]
379
380                     point = self.__transform_point(point)
381                     distance = point[1]
382                     if point[1] < 0:
383                         continue
384                     radius = self.__piece_radius * self.__scale * self.__zoom / point[1]
385                     coord = self.__project_point(point)
386
387                     if pos in moves_list:
388                         # Shortest distance from the line to the centre of a piece
389                         perpendicular_distance = get_magnitude(cross(cursor, point))
390                         # Check whether the line collides with the piece
391                         if not self.__pressed and not self.__game.undone() and not self.spectating and \
392                                 perpendicular_distance < self.__piece_radius and distance < closest:
```

```
                            self.__hovered_pos = pos
                            hover_circle = (coord, radius)
                            closest = distance
                    distances.append((distance, count))
                    circles.append((colour, coord, radius))
                    count += 1

        distances.sort(reverse=True)
        for _, i in distances:
            colour, pos, radius = circles[i]
            self.__circle(colour, pos, radius)

        # Draw white circle around current selected piece.
        if hover_circle and not self.__game.thinking and not self.pause:
            pygame.draw.circle(self.__screen, (255, 255, 255, 255), hover_circle[0], hover_circle[1], width=5)

        # Information rendering
        if not self.__game.thinking:
            self.__current_player._text = f"Current player: {self.__player_names[self.__game.get_current_player()]}"
            self.__status_message._text = self.__game.get_state_message()
            self.__error_message._text = self.__game.get_error_message()
            self.__player_1_piece_count._text = f"{self.__player_names[0]}: {self.__game.get_piece_count()[0]}"
            self.__player_2_piece_count._text = f"{self.__player_names[1]}: {self.__game.get_piece_count()[1]}"
        self.__current_player.draw()
        self.__status_message.draw()
        self.__error_message.draw()
        self.__player_1_piece_count.draw()
        self.__player_2_piece_count.draw()


class Label:
    """Label object. Displays text on screen.

    Attributes:
        text: The text displayed.
        text_colour: The colour of the text displayed.

    Methods:
        handle(event): Handles pygame event.
        draw(): Renders the label. Call every frame.
    """

    def __init__(self, screen, pos,
                 text, text_colour=(0, 0, 0),
                 font="Assets/Font/EBGaramond-Normal.ttf", font_size=30 / 1600):
        """
        Initialises Label object.

        Args:
            screen: Pygame surface to display text on.
            pos: The position on screen of the centre of text.
            text: The string to be displayed.
            text_colour: RGB value of the text.
            font: The path to the font used for the text.
            font_size: Font size of the text.
        """
        self._screen = screen
        self._relative_pos = pos
        self._pos = [pos[0] * screen.get_width(), pos[1] * screen.get_height()]
        self._text = text
        self.text_colour = text_colour
        self._font_path = font
        self.__relative_font_size = font_size
        self._font_size = int(font_size * screen.get_width())
        self._font = pygame.font.Font(self._font_path, self._font_size)
        self.__text_surf = self._font.render(text, True, text_colour)
        self.__text_rect = self.__text_surf.get_rect(center=pos)

    def handle(self, event):
        """Handles pygame event."""
        if event.type == pygame.WINDOWRESIZED:  # Resize
            self._pos = [self._relative_pos[0] * self._screen.get_width(),
                         self._relative_pos[1] * self._screen.get_height()]
            self._font_size = int(self.__relative_font_size * self._screen.get_width())
            self._font = pygame.font.Font(self._font_path, self._font_size)

    def draw(self):
        """Renders the Label. Call every frame."""
        self.__text_surf = self._font.render(self._text, True, self.text_colour)
        self.__text_rect = self.__text_surf.get_rect(center=self._pos)
        self._screen.blit(self.__text_surf, self.__text_rect)

    @property
    def text(self):
```

```
477         return self._text
478
479     @text.setter
480     def text(self, s):
481         self._text = s
482
483
484 class Button(Label):
485     """Button object. Displays a button on screen.
486
487     Attributes:
488         text: The text displayed.
489         text_colour: The colour of the text displayed.
490         func: The function called when the button is pressed.
491         key: The key that presses the button when pressed.
492
493     Methods:
494         handle(event): Handles pygame event.
495         draw(): Renders the button. Call every frame.
496     """
497
498     def __init__(self, screen, pos, width, height, func, elevation=10/900,
499                  top_colour=(255, 165, 0), bottom_colour="#354B5E",
500                  key=-1,
501                  text="", text_colour=(0, 0, 0),
502                  font="Assets/Font/EBGaramond-Normal.ttf", font_size=40 / 1600):
503         """Initialises a Label object.
504
505         Args:
506             screen: Pygame surface to display the button on.
507             pos: The position on screen of the centre of the button.
508             width: Ratio of the button width against the screen width (button width / screen width).
509             height: Ratio of the button height against the screen height (button height / screen height).
510             func: The function called when the button is pressed.
511             elevation: The number of pixels the top of the button is elevated from the bottom.
512             top_colour: The colour of the top of the button.
513             bottom_colour: The colour of the bottom of the button.
514             key: The key that presses the button when pressed.
515             text: The string to be displayed on the button.
516             text_colour: RGB value of the text.
517             font: The path to the font used for the text.
518             font_size: Font size of the text.
519         """
520
521         super().__init__(screen, pos, text, text_colour, font, font_size)
522         self.func = func
523         self.key = key
524         self.__relative_elevation = elevation
525         self.__elevation = elevation * screen.get_height()
526         self.__border_radius = int(min(self._screen.get_width() * width, self._screen.get_height() * height) // 5)
527         self.__pressed = False
528         self.__relative_size = [width, height]
529
530         # Rectangles
531         # Top rectangle
532         self.__top = pygame.Rect(
533             (self._pos[0] - self._screen.get_width() * width / 2,  # Horizontal position.
534              self._pos[1] - self._screen.get_height() * height / 2),  # Vertical position.
535             (self._screen.get_width() * width,  # Width.
536              self._screen.get_height() * height))  # Height
537         # Bottom rectangle
538         self.__bottom = pygame.Rect((self._pos[0] - self._screen.get_width() * width / 2,  # Horizontal position.
539                                      self._pos[1] - self._screen.get_height() * height / 2 + self.__elevation),
540                                     # Vertical position.
541                                     (self._screen.get_width() * width,  # Width.
542                                      self._screen.get_height() * height))  # Height
543
544         # Colours
545         self.top_colour = top_colour
546         self.bottom_colour = bottom_colour
547
548     def handle(self, event):
549         """Handles pygame event."""
550         super().handle(event)
551         if event.type == pygame.WINDOWRESIZED:  # Resizing itself.
552             self.__top.width = self.__bottom.width = self._screen.get_width() * self.__relative_size[0]
553             self.__top.height = self.__bottom.height = self._screen.get_height() * self.__relative_size[1]
554             self.__top.center = self._pos
555             self.__elevation = self.__relative_elevation * self._screen.get_height()
556             self.__bottom.center = [self._pos[0], self._pos[1] + self.__elevation]
557             self.__border_radius = int(min(self._screen.get_width() * self.__relative_size[0],
558                                            self._screen.get_height() * self.__relative_size[1]) // 5)
559         if event.type == pygame.MOUSEBUTTONDOWN:  # Button being hovered over.
560             if self.__bottom.collidepoint(event.pos):
```

```
561                 self.__pressed = True
562                 return
563         if (event.type == pygame.MOUSEBUTTONUP and self.__bottom.collidepoint(event.pos) and self.__pressed) \
564                 or (event.type == pygame.KEYUP and event.key == self.key):  # Button pressed.
565             self.__pressed = False
566             self.func()
567 
568     def draw(self):
569         """Renders the button. Call every frame."""
570         if self.__bottom.collidepoint(pygame.mouse.get_pos()):
571             self.__top.centery = self._screen.get_height() * self._relative_pos[1]
572         else:
573             self.__top.centery = self._screen.get_height() * self._relative_pos[1] - self.__elevation / 2
574 
575         pygame.draw.rect(self._screen, self.bottom_colour, self.__bottom, border_radius=self.__border_radius)
576         pygame.draw.rect(self._screen, self.top_colour, self.__top, border_radius=self.__border_radius)
577         self._pos = self.__top.center
578         super().draw()
579 
580 
581 class Textbox(Label):
582     """Textbox object. Allows user to input text.
583 
584     Attributes:
585         colour: The colour of the textbox.
586         input: The input in the textbox.
587         locked: Whether the textbox can be interacted with.
588 
589     Methods:
590         handle(event): Handles pygame event.
591         draw(): Renders the Textbox. Call every frame.
592         get_input(): The text in the textbox.
593     """
594 
595     def __init__(self, screen, pos, width, height, colour=(255, 165, 0),
596                  text="", text_colour=(0, 0, 0),
597                  font="Assets/Font/EBGaramond-Normal.ttf", font_size=40 / 1600):
598         """Initialises a Textbox object.
599 
600         Args:
601             screen: Pygame surface to display the textbox on.
602             pos: The position on screen of the centre of the textbox.
603             width: Ratio of the textbox width against the screen width (textbox width / screen width).
604             height: Ratio of the textbox height against the screen height (textbox height / screen height).
605             colour: The colour of the textbox.
606             text: The string to be displayed on the textbox.
607             text_colour: RGB value of the text.
608             font: The path to the font used for the text.
609             font_size: Font size of the text.
610         """
611         super().__init__(screen, pos, text,
612                          text_colour,
613                          font, font_size)
614         self.__relative_size = [width, height]
615         self.__pressed = False
616         self.__active = False
617         self.__locked = False
618         self.__back_space = -1
619         self.colour = colour
620         self.__input = text
621         self.__rect = pygame.Rect(
622             (
623                 self._pos[0] - self._screen.get_width() * width / 2,
624                 self._pos[1] - self._screen.get_height() * height / 2),
625             (self._screen.get_width() * width, self._screen.get_height() * height))
626         self.__border_radius = int(min(self._screen.get_width() * self.__relative_size[0],
627                                        self._screen.get_height() * self.__relative_size[1]) // 5)
628 
629     def handle(self, event):
630         """Handles pygame event."""
631         super().handle(event)
632         if self.__locked:
633             return
634         if event.type == pygame.WINDOWRESIZED:
635             self.__rect.width = self._screen.get_width() * self.__relative_size[0]
636             self.__rect.height = self._screen.get_height() * self.__relative_size[1]
637             self.__rect.center = self._pos
638             self.__border_radius = int(min(self._screen.get_width() * self.__relative_size[0],
639                                            self._screen.get_height() * self.__relative_size[1]) // 5)
640         if event.type == pygame.MOUSEBUTTONDOWN:
641             if self.__rect.collidepoint(event.pos):
642                 if pygame.mouse.get_pressed()[0]:
643                     self.__pressed = True
644             else:
```

```
                self.__active = False
                self.__pressed = False
        if event.type == pygame.MOUSEBUTTONUP:
            if self.__rect.collidepoint(event.pos) and self.__pressed:
                self.__active = True
            else:
                self.__active = False
            self.__pressed = False
        if self.__active:
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_BACKSPACE:
                    self.__input = self.__input[:-1]
                    self.__back_space = 0
                else:
                    self.__input += event.unicode
                    if self._font.size(self.__input)[0] + 20 > self.__rect.width:
                        self.__input = self.__input[:-1]
            if event.type == pygame.KEYUP and event.key == pygame.K_BACKSPACE:
                if event.key == pygame.K_BACKSPACE:
                    self.__input = self.__input[:-1]
                    self.__back_space = -1

    def draw(self):
        """Renders the textbox. Call every frame."""
        if self.__back_space >= 0:
            if self.__back_space < 25:
                self.__back_space += 1
            else:
                self.__input = self.__input[:-1]
                self.__back_space -= 4
        self.text = self.__input + ("|" if self.__active else "")
        pygame.draw.rect(self._screen, self.colour, self.__rect, border_radius=self.__border_radius)
        super().draw()

    @property
    def input(self):
        return self.__input

    @input.setter
    def input(self, s):
        if self._font.size(s)[0] + 20 <= self.__rect.width:
            self.__input = s

    @property
    def locked(self):
        return self.__locked

    @locked.setter
    def locked(self, b):
        if b:
            self.__active = False
            self.__pressed = False
        self.__locked = b


class Timer(Label):
    """Timer object. Displays time on screen.

    Attributes:
        pause: Whether the timer is paused or not.

    Methods:
        handle(event): Handles pygame event.
        draw(): Renders the timer. Call every frame.
        get_time(): Returns the current time on the timer in milliseconds.
    """

    def __init__(self, screen, pos, game, player, font="Assets/Font/EBGaramond-Normal.ttf", font_size=40 / 1600,
                 text_colour=(255, 255, 255)):
        self.__time = 0
        self.__game = game
        self.__player = player
        self.pause = False

        super().__init__(screen, pos, "00:00", text_colour, font, font_size)

    def draw(self):
        """Renders timer. Call every frame."""
        if self.__game.get_winner() != -1 or (self.__game.get_thinking_player() != self.__player and not self.__game.
     undone()):
            self.pause = True
        elif self.__game.get_thinking_player() == self.__player and not self.__game.undone():
            self.pause = False
        if not self.pause:
```

```
728             self.__time += clock.get_time()
729         self.text = f"{0 if self.__time // 60000 < 10 else ''}{self.__time // 60000}:" \
730                     f"{0 if self.__time // 1000 % 60 < 10 else ''}{(self.__time // 1000) % 60}"
731         super().draw()
732 
733     def get_time(self):
734         return self.__time
735 
736 
737 class GUI:
738     """The Graphical User Interface (GUI) for a game of Pylos.
739 
740     Attributes:
741         width: Width of the display in pixels.
742         height: Height of the display in pixels.
743 
744     Methods:
745         run(): Call to start playing.
746         quit(): Proper way to close application.
747     """
748 
749     def __init__(self, background_colour=(255, 255, 255, 0), width=1600, height=900, fps=60):
750         """
751         Initialises GUI class.
752 
753         Args:
754             background_colour: The background_colour.
755             width: The width of the window.
756             height: The height of the window.
757             fps: Framerate in frames per second.
758         """
759         self.__background_colour = background_colour
760         self.fps = fps
761 
762         self.__player_names = ["Player 1", "Player 2"]
763 
764         self.__screen = pygame.display.set_mode((width, height), pygame.RESIZABLE)
765         pygame.display.set_caption("Pylos")
766         self.__entities = []
767         self.__call_stack = []
768         self._game = None
769         with open("Data/Games.txt", "r") as f:
770             self.__games = [game.split("|") for game in f.read().split("\n")]
771 
772         self._available_colours = [(i * 255, j * 255, k * 255, 255)
773                                    for i in range(2)
774                                    for j in range(2)
775                                    for k in range(2)]
776         self._available_background_colours = [(0, 0, 0), (128, 0, 0), (218, 165, 32),
777                                               (85, 107, 47), (47, 79, 79),
778                                               (135, 206, 235), (25, 25, 112)]
779         self._available_colours.pop(0)
780         # Game settings
781         self._ai_player = "0"
782         self.__size = 4
783         self._colours = [(255, 0, 0, 255), (0, 255, 0, 255), (100, 100, 100, 100)]
784         self._background = (0, 0, 0)
785         self._square = True
786         self._alignment = False
787         self.__brain = None
788 
789     def quit(self):
790         """Closes window and stops python program."""
791         if self.__brain is not None:
792             self.__brain.quit()
793         with open("Data/Games.txt", "w") as f:
794             f.write("\n".join("|".join(game) for game in self.__games[:9]))
795         quit(0)
796 
797     def run(self):
798         def loop():
799             while True:
800                 self.__screen.fill(self.__background_colour)
801                 for event in pygame.event.get():
802                     if event == pygame.QUIT:
803                         self.quit()
804                     for entity in self.__entities:
805                         entity.handle(event)
806                 for entity in self.__entities:
807                     entity.draw()
808                 pygame.display.update()
809                 clock.tick(self.fps)
810 
811         def back():
```

```
812             """Goes back a screen. """
813             self.__call_stack.pop()
814             self.__call_stack[-1]()
815 
816         def home():
817             """Home screen."""
818             self.__background_colour = (255, 255, 255, 255)
819             # pygame.mixer.music.load("Assets/Music/Home.wav")
820             # pygame.mixer.music.play(-1)
821             w = 1 / 9
822             h = 1 / 13
823             self.__entities = [
824                 Button(self.__screen,
825                        (1 / 2, 2.5 * h),
826                        5 * w, 3 * h,
827                        start,
828                        text="Play",
829                        top_colour=(252, 48, 3)),
830                 Button(self.__screen, (1 / 2, 6.5 * h), 5 * w, 3 * h, watch, text="Watch",
831                        top_colour=(3, 252, 65)),
832                 Button(self.__screen, (1 / 2, 10.5 * h),
833                        5 * w, 3 * h, self.quit, text="Quit")
834             ]
835 
836         def start():
837             """Start screen."""
838             if self.__call_stack[-1] != start:
839                 self.__call_stack.append(start)
840 
841             def change_player_1_colour():
842                 i = self._available_colours.index(self.__entities[1].top_colour)
843                 i += 1
844                 i %= len(self._available_colours)
845                 if self._available_colours[i] == self.__entities[4].top_colour:
846                     i += 1
847                     i %= len(self._available_colours)
848                 self.__entities[1].top_colour = self._available_colours[i]
849 
850             def change_player_2_colour():
851                 i = self._available_colours.index(self.__entities[4].top_colour)
852                 i += 1
853                 i %= 7
854                 if self._available_colours[i] == self.__entities[1].top_colour:
855                     i += 1
856                     i %= 7
857                 self.__entities[4].top_colour = self._available_colours[i]
858 
859             def change_background_colour():
860                 i = (self._available_background_colours.index(self.__entities[6].top_colour) + 1) % len(
861                     self._available_background_colours)
862                 self.__entities[6].top_colour = self._available_background_colours[i]
863 
864             def change_player_1():
865                 if self.__entities[2].text == "Player":
866                     self.__entities[2].text = "CPU"
867                     self.__entities[0].input = "CPU"
868                     self.__entities[0].locked = True
869                     if self.__entities[5].text == "CPU":
870                         self.__entities[5].text = "Player"
871                         self.__entities[3].input = "Player 2"
872                         self.__entities[3].locked = False
873                 else:
874                     self.__entities[2].text = "Player"
875                     self.__entities[0].input = "Player 1"
876                     self.__entities[0].locked = False
877 
878             def change_player_2():
879                 if self.__entities[5].text == "Player":
880                     self.__entities[5].text = "CPU"
881                     self.__entities[3].input = "CPU"
882                     self.__entities[3].locked = True
883                     if self.__entities[2].text == "CPU":
884                         self.__entities[2].text = "Player"
885                         self.__entities[0].input = "Player 1"
886                         self.__entities[0].locked = False
887                 else:
888                     self.__entities[5].text = "Player"
889                     self.__entities[3].input = "Player 2"
890                     self.__entities[3].locked = False
891 
892             def change_size():
893                 size = int(self.__entities[7].text[0])
894                 size = (size - 2) % 5 + 3
895                 self.__entities[7].text = f"{size}x{size}"
```

```
896
897             def change_square():
898                 self.__entities[8].top_colour = "grey" if self.__entities[8].top_colour == "green" else "green"
899
900             def change_alignment():
901                 self.__entities[9].top_colour = "grey" if self.__entities[9].top_colour == "green" else "green"
902
903             w = 1 / 14
904             h = 1 / 13
905             self.__entities = [
906                 Textbox(self.__screen,
907                         (w * 3.5, h * 2.5),
908                         w * 5, h * 2,
909                         text=self.__player_names[0]),
910                 Button(self.__screen,
911                        (w * 7.5, h * 2.5),
912                        w * 2, h * 2,
913                        change_player_1_colour,
914                        top_colour=self._colours[0]),
915                 Button(self.__screen,
916                        (w * 11, h * 2.5),
917                        w * 4, h * 2,
918                        change_player_1,
919                        text="Player" if self._ai_player != 0 else "CPU"),
920                 Textbox(self.__screen,
921                         (w * 3.5, h * 6.5),
922                         w * 5, h * 2,
923                         text=self.__player_names[1]),
924                 Button(self.__screen,
925                        (w * 7.5, h * 6.5),
926                        w * 2, h * 2,
927                        change_player_2_colour,
928                        top_colour=self._colours[1]),
929                 Button(self.__screen,
930                        (w * 11, h * 6.5),
931                        w * 4, h * 2,
932                        change_player_2,
933                        text="Player" if self._ai_player != 1 else "CPU"),
934
935                 Button(self.__screen,
936                        (w * 2, h * 10.5),
937                        w * 2, h * 2,
938                        change_background_colour,
939                        text="Background", text_colour=(255, 255, 255),
940                        top_colour=self._background),
941                 Button(self.__screen,
942                        (w * 4.5, h * 10.5),
943                        w * 2, h * 2,
944                        change_size,
945                        text=f"{self.__size}x{self.__size}"),
946                 Button(self.__screen,
947                        (w * 7, h * 10.5),
948                        w * 2, h * 2,
949                        change_square,
950                        text="Square",
951                        top_colour="green" if self._square else "grey"),
952                 Button(self.__screen,
953                        (w * 9.5, h * 10.5),
954                        w * 2, h * 2,
955                        change_alignment,
956                        text="Alignment",
957                        top_colour="green" if self._alignment else "grey"),
958                 Button(self.__screen,
959                        (1 / 10, 1 * 14 / 15),
960                        1 / 12, 1 / 15,
961                        back,
962                        elevation=5/900,
963                        key=pygame.K_ESCAPE,
964                        text="Exit", font_size=20 / 1600)
965             ]
966             if self._game is None:
967                 self.__entities.append(
968                     Button(self.__screen,
969                            (w * 12, h * 10.5),
970                            w * 2, h * 2,
971                            start_new_game,
972                            text="New Game",
973                            key=pygame.K_RETURN,
974                            top_colour="Gold"))
975             else:
976                 self.__entities.append(
977                     Button(self.__screen,
978                            (w * 12, h * 11.5),
979                            w * 2, h * 1.75,
```

```
                        start_new_game,
                        text="New Game",
                        key=pygame.K_n,
                        top_colour="Gold"))
                self.__entities.append(
                    Button(self.__screen,
                        (w * 12, h * 9.5),
                        w * 2.5, h * 1.75,
                        resume_game,
                        text="Resume Game",
                        key=pygame.K_RETURN,
                        top_colour="Gold"))

        def pause():
            self.__entities[0].pause = True
            self.__entities[1].pause = True
            self.__entities[2].pause = True
            self.__entities[3].pause = True

            w = 1 / 2
            h = 1 / 2
            self.__entities[1].text = "Unpause"
            self.__entities[1].func = unpause
            self.__entities.pop()
            self.__entities.pop()
            self.__entities.append(
                Button(self.__screen, (1 / 2, 1 / 2), w, h, func=back_from_game, text="Exit")
            )

        def unpause():
            self.__entities[0].pause = False
            self.__entities[1].pause = False
            self.__entities[2].pause = False
            self.__entities[3].pause = False
            self.__entities[1].text = "Pause"
            self.__entities[1].func = pause
            self.__entities.pop()
            self.__entities += [
                Button(self.__screen, (1 * 8 / 10, 1 * 9 / 10),
                       1 / 12, 1 / 15,
                       func=self._game.undo, elevation=5/900, text="Undo",
                       font_size=20 / 1600),
                Button(self.__screen, (1 * 9 / 10, 1 * 9 / 10),
                       1 / 12, 1 / 15,
                       func=self._game.redo,
                       key=pygame.K_RIGHT,
                       elevation=5/900, text="Redo",
                       font_size=20 / 1600),
            ]

        def back_from_game():
            if self._game.get_winner() != -1:
                time = self.__entities[2].get_time() + self.__entities[3].get_time()
                game = [f"{int(self.__games[1][0]) + 1}",
                        f"{0 if time // 60000 < 10 else ''}{time // 60000}:"
                        f"{0 if time // 1000 % 60 < 10 else ''}{(time // 1000) % 60}",
                        f"{self.__player_names[self._game.get_winner()]}/ "
                        f"{self.__player_names[1 - self._game.get_winner()]}",
                        f"{self._game.get_size()}x{self._game.get_size()}",
                        f"{'Yes' if self._game.get_square() else 'No'}",
                        f"{'Yes' if self._game.get_alignment() else 'No'}",
                        f"{len(self._game.get_history())}",
                        '+'.join(','.join(map(str, move)) for move in self._game.get_history()),
                        '+'.join(','.join(map(str, colour)) for colour in self._colours),
                        ','.join(map(str, self._background)),
                        ',++,++,,,+,,'.join(self.__player_names)
                        ]

                self.__games.insert(1, game)
                self.__games.pop()
                self._game = None
            back()

        def start_new_game():
            """Game screen."""
            self._ai_player = 0 if self.__entities[2].text == "CPU" \
                else 1 if self.__entities[5].text == "CPU" else -1
            self.__player_names = self.__entities[0].input if self.__entities[0].input else "Player 1", \
                                  self.__entities[3].input if self.__entities[3].input else "Player 2"
            self.__size = int(self.__entities[7].text[0])
            self._square = True if self.__entities[8].top_colour == "green" else False
            self._alignment = True if self.__entities[9].top_colour == "green" else False

            self._game = Pylos.Pylos(self.__size, self._square, self._alignment)
```

```
            if self._ai_player != -1:
                self.__brain = Brain.Brain(self._game, self._ai_player)
                self.__brain.start()

            resume_game()

        def resume_game():
            if self.__call_stack[-1] != resume_game:
                self.__call_stack.append(resume_game)
            self._colours[0] = self.__entities[1].top_colour
            self._colours[1] = self.__entities[4].top_colour
            self._background = self.__entities[6].top_colour

            self.__entities.pop()
            self.__entities = [
                Renderer(self.__screen, self._game,
                         piece_colours=self._colours,
                         width=1, height=1,
                         background_colour=self._background,
                         player_names=self.__player_names),
                Button(self.__screen, (1 / 10, 1 * 9 / 10),
                       1 / 12, 1 / 15,
                       func=pause,
                       elevation=5/900,
                       key=pygame.K_ESCAPE,
                       text="Pause", font_size=20 / 1600),
                Timer(self.__screen, (1 / 8, 1 * 2 / 9), self._game, 0),
                Timer(self.__screen, (1 * 7 / 8, 1 * 2 / 9), self._game, 1),
                Button(self.__screen, (1 * 8 / 10, 1 * 9 / 10),
                       1 / 12, 1 / 15,
                       func=self._game.undo, elevation=5/900,
                       key=pygame.K_LEFT,
                       text="Undo",
                       font_size=20 / 1600),
                Button(self.__screen, (1 * 9 / 10, 1 * 9 / 10),
                       1 / 12, 1 / 15,
                       func=self._game.redo,
                       key=pygame.K_RIGHT,
                       elevation=5/900, text="Redo",
                       font_size=20 / 1600),
            ]

        def watch():
            """Watch screen."""
            if self.__call_stack[-1] != watch:
                self.__call_stack.append(watch)
            # Game No.|Duration|Winner|Board Size|Square Made|Alignment|Moves Made|Moves
            n = 8
            print(len(self.__games))
            self.__entities = \
                [
                    Button(self.__screen, (0.5,
                                           1 * (i + 1) / (n + 3)),
                           1, 1 / 10,
                           func=lambda bound_i=i: replay(self.__games[bound_i]),
                           elevation=5 / 900,
                           text="",
                           top_colour=(255, 255, 255),
                           bottom_colour=(255, 255, 255)
                           )
                    for i in range(1, n + 1)
                ] + [
                    Button(self.__screen, (1 / 10, 1 * 9 / 10),
                           1 / 12, 1 / 15, back,
                           elevation=5 / 900, text="Exit", font_size=20 / 1600)
                ]

            self.__entities += [
                Label(self.__screen, (
                    1 * (j + 1) / n,
                    1 * (i + 1) / (n + 3)),
                      text=self.__games[i][j])
                for i in range(len(self.__games))
                for j in range(n - 1)
            ]

        def replay(game):
            """Replay screen."""
            if self.__call_stack[-1] != replay:
                self.__call_stack.append(replay)

            moves = [tuple(map(int, move.split(","))) for move in game[7].split("+")]
```

```
1148            self.__replay = Pylos.Pylos(int(game[3][0]),
1149                                        True if game[4] == "Yes" else False,
1150                                        True if game[5] == "Yes" else False)
1151            for move in moves:
1152                self.__replay.input(move)
1153            for _ in moves:
1154                self.__replay.undo()
1155
1156            self.__entities = [
1157                Renderer(self.__screen, self.__replay,
1158                         piece_colours=tuple(tuple(map(int, colour.split(","))) for colour in game[8].split("+")),
1159                         width=1, height=1,
1160                         background_colour=tuple(map(int, game[9].split(","))),
1161                         spectating=True,
1162                         player_names=game[10].split(',++,++,,,+,,')),
1163                Button(self.__screen,
1164                       (1 * 9 / 10, 1 * 9 / 10),
1165                       1 / 12, 1 / 15,
1166                       func=self.__replay.redo,
1167                       elevation=5/900,
1168                       key=pygame.K_RIGHT,
1169                       text="Next", font_size=20 / 1600),
1170                Button(self.__screen,
1171                       (1 * 8 / 10, 1 * 9 / 10),
1172                       1 / 12, 1 / 15,
1173                       func=self.__replay.undo,
1174                       elevation=5/900,
1175                       key=pygame.K_LEFT,
1176                       text="Previous", font_size=20 / 1600),
1177                Button(self.__screen,
1178                       (1 / 10, 1 * 9 / 10),
1179                       1 / 12, 1 / 15,
1180                       func=back,
1181                       elevation=5/900,
1182                       text="Exit", font_size=20 / 1600),
1183            ]
1184
1185        # Initial call stack only has home on it.
1186        self.__call_stack.append(home)
1187        home()
1188        loop()
1189
1190    @property
1191    def width(self):
1192        return self.__screen.get_width()
1193
1194    @property
1195    def height(self):
1196        return self.__screen.get_height()
```

3.8 Brain

The `Brain` class holds the computer player. It uses a minimax algorithm with alpha-beta pruning to find the "best move" from the current position. The class inherits from the `threading.Thread` class.

3.8.1 Multi-threading

The `Thread` class has a `start` method which starts a separate thread and calls the `run` method on that thread. The Brain class overrides the `run` method. This allows the GUI to call `Brain.start()` to start the CPU player on a new thread. This allows the minimax to check whether it is their move constantly. Moreover, the user can interact with the GUI like normal while the minimax algorithm is evaluated.

The `run` method, while the game is not over, checks whether it is the computer player's turn (if `Pylos.get_current_player()` is equal to the protected attribute `__player`) every constant interval of time. If it is the computer player's turn, the program calls the `__evaluate` method. Once the best move is found, it checks whether moves have been undone and waits until all moves are redone to input the best move found.

3.8.2 Minimax

The `__evaluate` method is the minimax algorithm. It resembles the pseudocode in Algorithm 2 in the design chapter. It is recursive. Once the depth reaches 0, it calls the `__evaluation_func` method. This is the evaluation function, meaning it evaluates the "advantage" the computer player holds in this state represented by some number, the reward. The evaluation function I chose is quite simple. If the state is won by the computer player, the reward is some number large enough to act

as infinity. If the state is won by the human player, the reward is some number small enough to act as negative infinity. Otherwise, the reward is given by the difference between the computer player's remaining pieces and the human player's remaining pieces. This means that the more pieces the computer player has compared to the human player, the greater the reward.

3.8.3 Brain.py

```
import time
import threading
import math


class Brain(threading.Thread):
    def __init__(self, game, player, depth=-1):
        """Initialises the Brain class.

        Args:
            game: The Pylos game class.
            player: The player the brain is acting as. (0 for player 1, 1 for player 2)
            depth: The depth of the minimax algorithm. (negative number for automatic calculation)
        """
        threading.Thread.__init__(self)
        self.__game = game
        if depth < 0:
            self.__depth = int(math.log(1e8, len(self.__game.get_moves_list())))
        else:
            self.__depth = depth
        self.__player = player
        self.__running = False
        print("Minimax algorithm started.")
        print("Player: ", self.__player)
        print("Depth: ", self.__depth)

    def __evaluation_func(self):
        """Evaluates the reward of the current game state."""
        if self.__game.get_piece_count()[1 - self.__player] == 0:
            return 1000
        if self.__game.get_winner() == self.__player:
            return 1000
        return self.__game.get_piece_count()[self.__player] - self.__game.get_piece_count()[1 - self.__player]

    def __evaluate(self, depth, alpha, beta):
        """Recursive minimax algorithm for evaluating the best move.
        Returns: maximum_value, best move
        """
        a, b = alpha, beta
        if depth == 0 or len(self.__game.get_moves_list()) == 0:
            return self.__evaluation_func(), 0
        if self.__game.get_current_player() != self.__player:
            score = 1_000_000
            for move in self.__game.get_moves_list():
                if move == (0, 0, 0):
                    return -1000, (0, 0, 0)
                if not self.__game.input(move):
                    continue
                # Recursion used here.
                score = min(score, self.__evaluate(depth - 1, a, b)[0])
                self.__game.undo(False)
                b = min(b, score)
                if b <= a:
                    break
            return score, 0
        else:
            score = -1_000_000
            best_move = None
            for move in self.__game.get_moves_list():
                if move == (0, 0, 0):
                    return 1000, (0, 0, 0)
                if not self.__game.input(move):
                    continue
                # Recursion used here.
                new_score = self.__evaluate(depth - 1, a, b)[0]
                if new_score > score:
                    score = new_score
                    best_move = move
                self.__game.undo(False)
                a = max(a, score)
                if a >= b:
                    break
            return score, best_move

```

```
75      def quit(self):
76          """Stops Brain.run and terminates the instance of Brain."""
77          self.__running = False
78
79      def run(self):
80          """Called when Brain.start() is called."""
81          self.__running = True
82          while self.__game.get_winner() == -1 and self.__running:
83              if self.__game.get_current_player() == self.__player and not self.__game.undone():
84                  self.__game.thinking = True
85                  print("Minimax has started evaluating.")
86                  best_move = self.__evaluate(self.__depth, -1_000_000, 1_000_000)[1]
87                  self.__game.thinking = False
88                  print("Best move found: ", best_move, "\n")
89                  while self.__game.undone():
90                      time.sleep(0.1)
91                  self.__game.input(best_move)
92                  time.sleep(0.5)
93              time.sleep(0.1)
94          print("Game over. Terminating self.")
95          del self
```

Chapter 4

Testing

This chapter is to ensure the product has the functionality planned in the design chapter.

4.1 Tests During Development

Regrettably, there is not much evidence on these early phases of testing. This is because most of it was simply running the program and seeing which parts are amiss or malfunctioning. These bugs were quickly found as the program was not very complex. During the development of the `Pylos` class, I developed a command line interface to interact with the game without having a GUI. This helped to test as I could input the game class in a structured manner. It also helped determine what else the GUI class may need from the game class. The Python shell helped in a similar way.

```
from Pylos import Pylos


class CLI:
    def __init__(self):
        self.game = Pylos(4, True, False)

    def show(self):
        for i in range(self.game.get_size()):
            for j in range(i + 1):
                for k in range(i + 1):
                    print(self.game.peek((i, j, k)), end=' ')
                print()
            print()
        print("\n")

    def input_move(self):
        pos = (-1, -1, -1)
        while not self.game.valid(pos):
            print("Player " + str(self.game.get_current_player() + 1) + "'s turn. \n")
            inp = input()
            try:
                pos = tuple(map(int, inp.split()))
            except:
                print("Invalid input.\n")
        return pos

    def play(self):
        print("Welcome to Pylos!\n")
        while self.game.get_winner() == -1:
            print(self.game.get_error_message())
            self.show()
            print(self.game.get_state_message())
            pos = self.input_move()
            self.game.input(pos)
        self.win()

    def win(self):
        print("\nPlayer " + str(self.game.get_winner() * 0.5 + 1.5) + " has won!!!\n Press enter to play again. \n")
        _ = input()
        self.play()
```

Listing 4.1: Command Line Interface

During the development of the `Brain` class, I had the class print a message whenever the minimax started evaluating and the best move found when it finished its evaluation. I left it in the code so it is easier to debug later on.

4.2 Final Tests

Below are the final tests I did on the product. I have outlined the method of testing and the expected output.

4.2.1 Objective 1

The player should be able to play a game of Pylos.

No.	Input	Output	Outcome
1	Press the "New Game" button on the Start screen.	The game board shows.	Test passed.
2	Right click on a position where a piece can be placed.	A piece is placed at the desired position.	Test passed.
3	Right click a piece which can be retrieved.	The piece is retrieved.	Test passed.
4	Right click a position where a piece cannot be placed.	A piece is not placed at the position.	Test passed.
5	Right click a piece that cannot be retrieved.	The piece is not retrieved.	Test passed.
6	Right click the position at the top of the pyramid when a piece can be placed there.	The game is won by the current player.	Test passed.
7	Press the undo button by either clicking on the button or pressing the left key.	A move is undone.	Test passed.
8	Press the redo button by either clicking on the button or pressing the right key.	A move is redone.	Test passed.
9	Right click a position that a piece can be placed at when moves have been undone.	A piece is not placed.	Test passed.
10	Right click a piece that can be retrieved when moves have been undone.	The piece is not retrieved.	Test passed.

4.2.2 Objective 2

The player should be able to customise the settings of each game.

11	Click the "Play" button on the home screen.	The Start screen appears.	Test passed.
12	Change Player 1's colour on the Start screen and click the "New Game" button.	Player 1's pieces are displayed in the appropriate colour.	Test passed.
13	Change Player 2's colour on the Start screen and click the "New Game" button.	Player 2's pieces are displayed in the appropriate colour.	Test passed.
14	Change the background colour option on the Start screen and click the "New Game" button.	The background is the appropriate colour.	Test passed.
15	Enable the "Square" option on the Start screen and click the "New Game" button. Make a square of the same colour.	The game prompts the player to retrieve two pieces.	Test passed.
16	Disable the "Square" option on the Start screen and click the "New Game" button. Make a square of the same colour.	The game does not prompt the player to retrieve two pieces and instead goes to the next turn.	Test passed.
17	Enable the "Alignment" option on the Start screen and click the "New Game" button. Make a line with four pieces of the same colour.	The game prompts the player to retrieve two pieces.	Test passed.
18	Disable the "Alignment" option on the Start screen and click the "New Game" button. Make a line with four pieces of the same colour.	The game does not prompt the player to retrieve two pieces and instead goes to the next turn.	Test passed.
19	Set the size of the board to 3x3 on the Start screen and click the "New Game" button.	A 3x3 game board is shown and can be played on.	Test passed.
20	Set the size of the board to 7x7 on the Start screen and click the "New Game" button.	A 7x7 game board is shown and can be played on.	Test passed.

4.2.3 Objective 3

The player should be able to play against a computer player.

21	Change Player 1 to CPU on the Start screen and click the "New Game" button.	The CPU player moves first.	Test passed.
22	Change Player 2 to CPU on the Start screen and click the "New Game" button. Input a move.	The CPU player moves second.	Test passed.
23	Change Player 1 to CPU on the Start screen and click the "New Game" button. Play a game against the CPU player.	The CPU player moves within 5 seconds for every single move it makes.	Test passed.

4.2.4 Objective 4

The user should be able to replay recent games played.

24	Click the "Watch" button on the Home screen.	The Watch screen appears. Information about the most recent eight games is displayed.	Test passed.
25	Click on a game.	The game board appears with relevant information such as the player names and the number of pieces left for each player.	Test passed.
26	Click the "Next" button.	The replay goes forward one move.	Test passed.
27	Click the "Previous" button.	The replay goes back one move.	Test passed.
28	Play a game of Pylos. Go back to the Home screen and click the Watch button. Select the most recent game played. Iterate through the moves.	The moves are exactly the same as in the game played.	Test passed.

4.2.5 Objective 5

When in-game or watching a replay, the user should be able to see a three-dimensional rendering of the game board and manoeuvre the camera around it.

29	Click the "Play" button on the Home screen then click the "New Game" button on the Start screen. Place a piece for each player.	The game board appears with no coloured spheres. Then a sphere in the player's chosen colour appears when each player places a piece.	Test passed.
30	Click the "Watch" button on the Home screen then select "Watch" on a game.	The game board appears with no coloured spheres. Then a sphere in the player's chosen colour appears when each player places a piece.	Test passed.
31	Same as in either test above.	The game board appears three-dimensional. Closer pieces appear larger and pieces further away appear smaller. Closer pieces may overlap pieces further away.	Test passed.
32	Press and hold the left mouse button while moving the cursor vertically and horizontally.	The viewing angle is adjusted proportionally to the movement of the cursor.	Test passed.
33	Press and hold the "W" key.	The camera moves upwards at a constant rate.	Test passed.
34	Press and hold the "S" key.	The camera moves downwards at a constant rate.	Test passed.
35	Press and hold the "A" key.	The camera moves leftwards at a constant rate.	Test passed.
36	Press and hold the "D" key.	The camera moves rightwards at a constant rate.	Test passed.
37	Scroll up.	The camera zooms out at a constant rate.	Test passed.
38	Scroll down.	The camera zooms in at a constant rate.	Test passed.

4.2.6 Objective 6

The interface should communicate relevant information to the user.

39	Hover over a position where a piece can be placed.	A white circle highlighting that position appears.	Test passed.
40	Hover over a piece that can be retrieved.	A white circle highlighting that piece appears.	Test passed.
41	There exist pieces that can be promoted.	These pieces pulse when it is the player's turn again.	Test passed.
42	There exist pieces that cannot be promoted.	These pieces remain dim when it is the player's turn again.	Test passed.
43	There exist positions at which pieces may be placed.	There are translucent grey pieces at these positions.	Test passed.
44	Press the "New Game" button on the Start screen.	A stopwatch for each player appears. These increase when it is the player's turn.	Test passed.
45	Press the "New Game" button on the Start screen.	The number of pieces each player has is displayed.	Test passed.
46	Press the "New Game" button on the Start screen.	The current player is shown.	Test passed.
47	Press the "New Game" button on the Start screen.	The player is prompted to take an action by displayed text.	Test passed.
48	Press the "New Game" button on the Start screen.	Text displayed tells the player if an error occurs such as undoing when there are no moves to undo.	Test passed.

4.3 Video

This video is evidence of the final tests and also the product itself. I go through the objectives and demonstrate how the product satisfies each.

Link: https://youtu.be/dRIYs1yDXsM

Chapter 5

Evaluation

5.1 Objective 1

"The player should be able to play a game of Pylos."

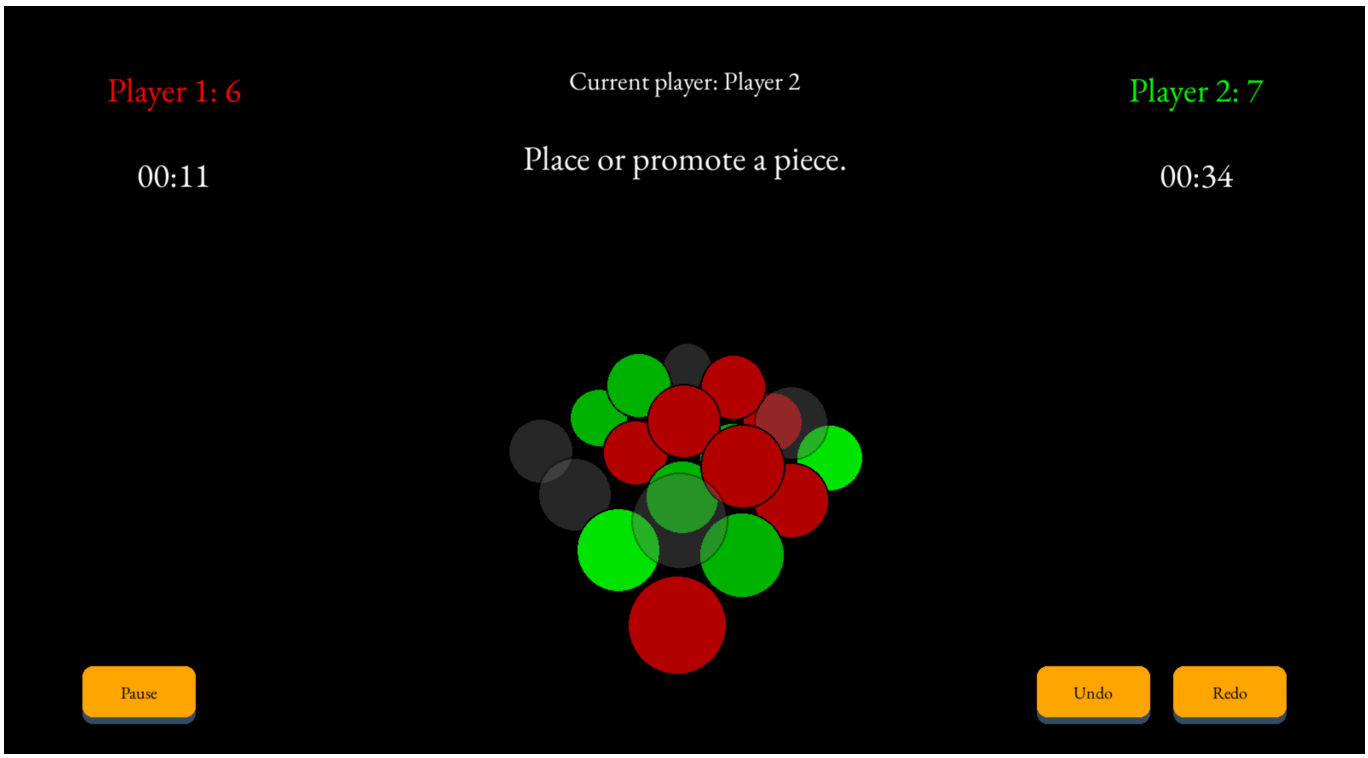


Figure 5.1: Playing a game of Pylos.

In the game screen, the player may place pieces and retrieve pieces according to the rules (parts (a), (b), (c); tests 1 - 5); they can win the game (part (d); test 6); they may look through the previous moves made (part (e); tests 7-9). The player cannot interact with the game board when moves have been undone (part (f); tests 9-10). The player can play a game of Pylos. This objective has been met. There is not much to be improved upon for this objective.

5.2 Objective 2

"The player should be able to customise the settings for each game."

The Start screen (part (a); test 11) allows the players to change the colours of the pieces and the background (part (b); tests 12-13). They may also change the rules (part (c); tests 15-18), and change the size of the board (part (d); tests 19-20). I have improved upon the initial objective by allowing the players to input their names. This makes the record of the games

more clear and may be helpful when the user is reviewing their recent games. This objective has been met and even improved upon.

One possible improvement is more freedom in colour selection. Currently, there are 8 colours to choose from. Perhaps allowing the players to choose from a colour wheel would be better. Also, instead of making the player cycle through board sizes, a drop-down menu could have been used for more intuitive use.

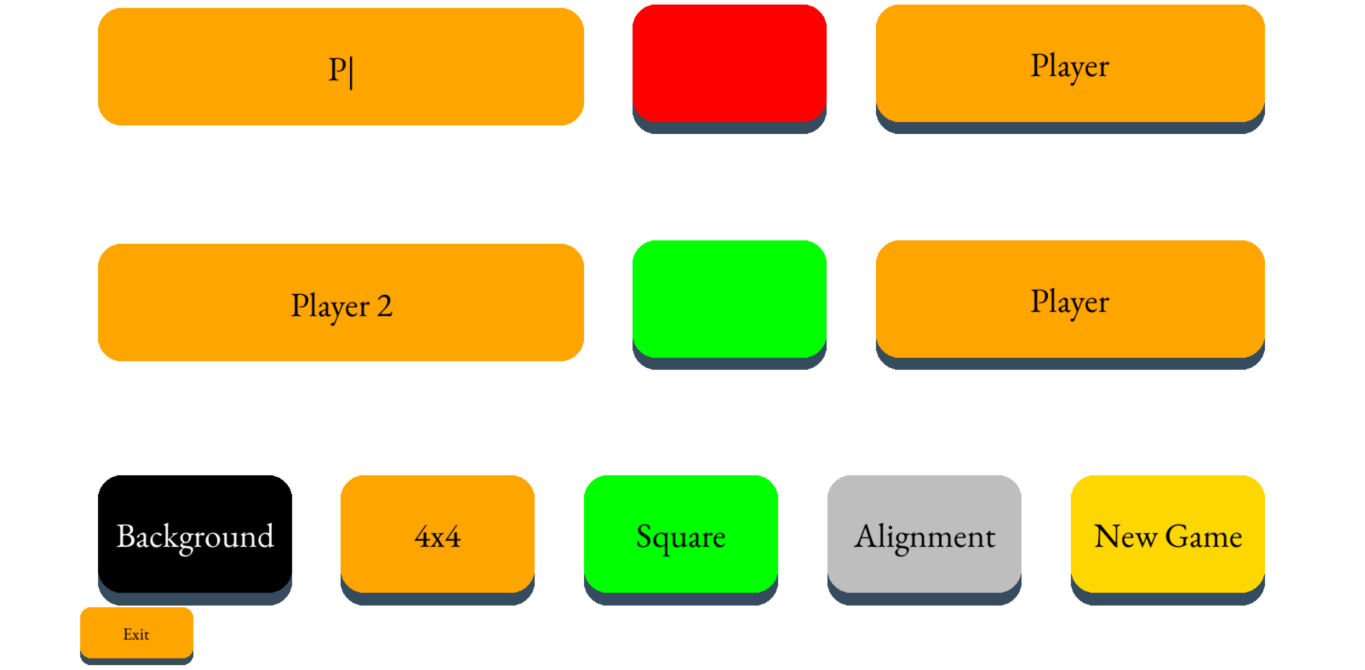


Figure 5.2: Configuring settings for a new game.

5.3 Objective 3

"The player should be able to play against a computer player."

The user can play against a computer player that moves first or second by their choosing (parts (a), (b); tests 21-22). The computer player moves within 5 seconds of the human player making a move every time (part (c); test 23).

The computer player has no difficulty setting, so the user cannot choose a different difficulty. This could be improved but within the time frame of development, it was not very realistic. The minimax algorithm's playstyle is also very straightforward, so it is not too difficult to play against.

5.4 Objective 4

"The player should be able to replay recent games played."

The Watch screen is shown in Figure 5.3. Relevant information such as the duration of the game, the winner and the loser, the size of the board, and the rules enabled are shown (part (a); test 24). Once a game is selected (clicked), the replay screen (Figure 5.4) appears, and the user can press the "Next" button and the "Previous" button to go forwards and backwards through the game (part (b); tests 25-28).

Currently, the program deletes games played more than eight games ago. More games could have been stored, perhaps using multiple pages for the user to go through to find older games.

Game No.	Duration	Winner/ Loser	Board Size	Square Made	Alignment	Moves Made
16	01:22	CPU/ Player 1	4x4	Yes	No	54
15	00:39	Player 2/ Player 1	4x4	Yes	No	36
14	01:00	Player 2/ Player 1	4x4	Yes	No	42
13	00:06	Player 2/ Player 1	3x3	No	Yes	14
12	00:05	Player 2/ Player 1	3x3	No	Yes	14
11	01:16	Player 2/ Player 1	4x4	Yes	No	38
10	00:01	Player 1/ Player 2	4x4	Yes	No	34
9	00:48	Player 1/ Player 2	4x4	Yes	No	34

Exit

Figure 5.3: Looking at recent games.

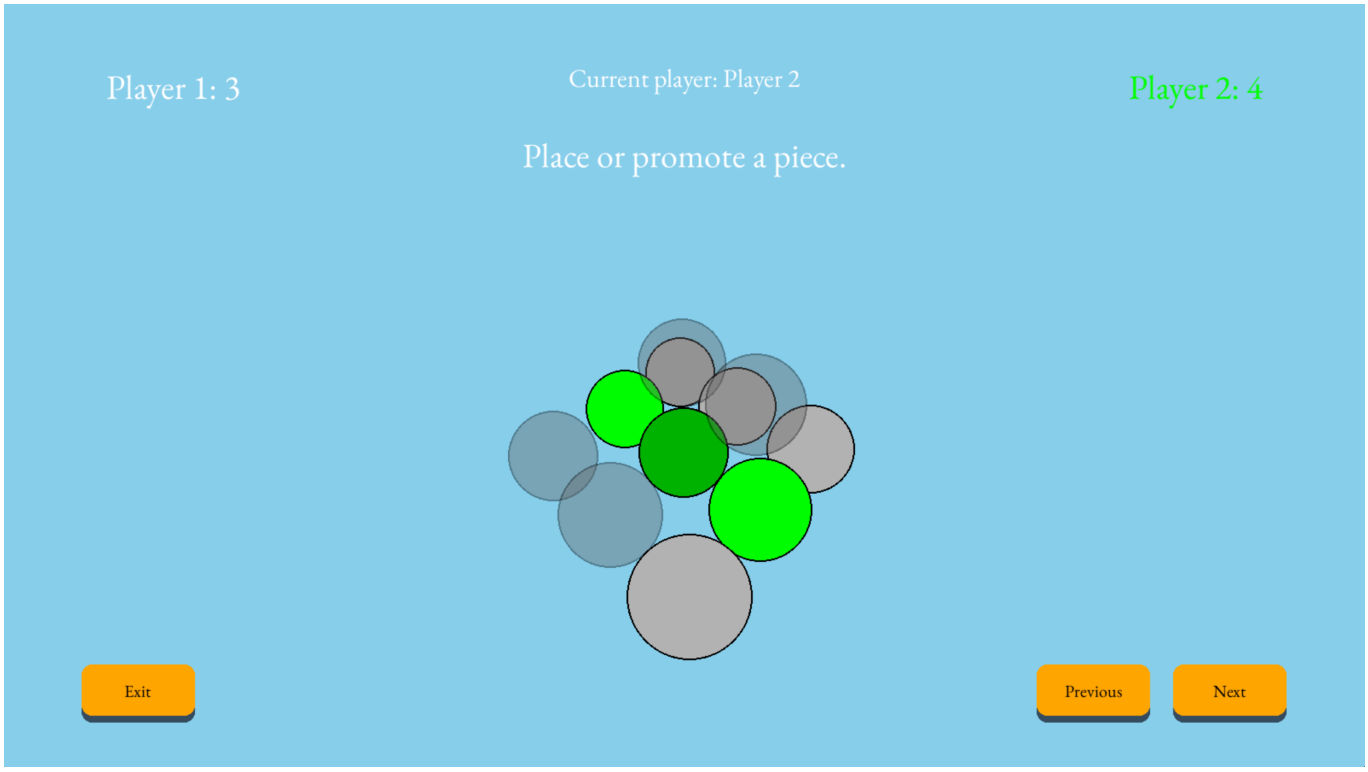


Figure 5.4: Watching a replay.

5.5 Objective 5

"When in-game or watching a replay, the player should be able to see a three-dimensional rendering of the game board and manoeuvre the camera around it." The user can see placed pieces as coloured spheres (Figure 5.1) (part (a); tests 29-30). Perspective exists (part(b); test 31). By holding the left mouse button and moving the cursor, the viewing angle can be adjusted (parts (c), (d); test 32). Similarly, using the "W", "A", "S", and "D" keys also move the camera (parts (c), (d); tests 33-36). The camera can zoom in and out using the scroll up and scroll down respectively (part (e); tests 37-38). Refer to the video provided in the testing chapter for a demonstration of the camera controls.

This objective was met well, with intuitive controls and smooth camera control. There is not much to be improved upon.

5.6 Objective 6

"The interface should communicate relevant information to the player." In Figure 5.1, you can see that the promotable pieces are pulsing and the non-retrievable pieces remain dim (parts (b), (c); tests 41-42). There are translucent grey pieces at these positions (part (d); test 43). The time elapsed and the number of pieces left for each player, the player whose turn it is, and the status message are shown (parts (e), (f), (g), (h); tests 43-47). In the video, the white circle highlighting pieces being hovered over is shown (part (a); tests 39-40). The error message is also shown at the end (part (i); test 48).

5.7 Client Feedback

Q: Do you feel that the final product met all the requirements?
A: Yes, the product has definitely met my expectations, with a good representation of the game board, a computer player to play against, and a replay system. This product took all the good parts of the existing solutions and added even more to them. I am very satisfied with this product.

Q: Is there anything that you would improve on this product?
A: The performance demand on my computer is still very much a problem, it drains the battery quickly. The colour options are limited, I would have preferred a wider range of colours for both the pieces and the background. The computer player only has one difficulty, I would have liked to play against different playstyles and difficulties as well. I can even watch two computer players with different strategies play against each other, that would be educational and interesting.

Q: What about things you may have liked to add?
A: Multi-player support as I don't have friends (that enjoy Pylos as much as I do), so it would be fun to play against real people instead of a computer player. Also, I wish I could download or import games from other sources to watch a replay of. However, I understand that would require some format for Pylos games which does not currently exist.

My client was satisfied with the product that I made. There are still things that the client would have liked to include in the project but these were raised after seeing the product in its completed form. Perfection is an illusion and considering the short time frame in which the product was developed, I think this is a good result.

5.8 Summary

This project was a success. Each objective was entirely met. I even made some improvements.

I learnt a lot throughout development, including implementing minimax, proper coding conventions, and linear algebra techniques used in a graphics rendering context. I feel that this has benefited my coding skills greatly. I am now better at analysing problems and breaking them down to make them easier to solve. I am also more familiar with the process of designing a project like this and the things to pay attention to when analysing.