Name: Connor Hanson
Candidate Number: 5216

St Aidan's and St John Fisher
Associated Sixth Form

Centre Number: 48239

A level Computer Science (7517)

Connor's Chess

CONTENTS

- 1 Analysis – 3
  - 1.1 Background – 3
    - 1.1.1 Chess – 3
    - 1.1.2 AI – 4
    - 1.1.3 End User/Purpose – 4
    - 1.1.4 Third Party/Client – 4
  - 1.2 Research – 5
    - 1.2.1 Minimax – 5
    - 1.2.2 Client Server Networking – 8
    - 1.2.3 Multi-Threading – 8
  - 1.3 Objectives – 9
  - 1.4 Modelling and Prototyping – 10
    - 1.4.1 Client Server – 10
    - 1.4.2 Multi-Threading – 11
    - 1.4.3 Minimax – 12
  - 1.5 UI Modelling – 14
    - 1.5.1 Chess Club Survey – 15
- 2 Design – 15
  - 2.1 Gameplay – 16
    - 2.1.1 Menu Classes – 16
    - 2.1.2 Piece Classes - 17
    - 2.1.3 Board Dictionary – 17
    - 2.1.4 Top-Level Flowchart – 18
  - 2.2 UI Design – 19
    - 2.2.1 Assets – 19
    - 2.2.2 Menus – 20
  - 2.3 File Structure – 29
  - 2.4 Internal Data Storage – 32
  - 2.5 Main Algorithms – 47
    - 2.5.1 Minimax – 47
    - 2.5.2 Quiescent Search – 51
  - 2.6 Client-Server – 54
    - 2.6.1 Client Pseudocode – 54
    - 2.6.2 Server Pseudocode – 55
    - 2.6.3 Client-Server Diagram – 58
  - 2.7 Stack – 58
    - 2.7.1 Stack Pseudocode – 58
    - 2.7.2 Stack Diagram – 60
  - 2.8 Move Check Methods – 60
    - 2.8.1 Check Move Method Pseudocode – 60
    - 2.8.2 Legal Move Method Pseudocode – 65
- 3 Commented Solution – 66
- 4 Testing – 186
- 5 Evaluation – 195
  - 5.1 Objectives – 195
  - 5.2 User Feedback – 199
  - 5.3 Analysis of User Feedback – 200
  - 5.4 Next Steps – 200

1.0 Analysis Section

1.1 Background

1.1.1 Chess

Chess is a 2-player strategy board game which originated in the 6th century consisting of 16 black pieces and 16 white pieces. On both sides, there are 6 types of pieces: Pawn, Bishop, Knight, Rook, Queen, and King. Each piece has different movement capabilities and has different values: 1, 3, 3, 5, 9 and infinity respectively.

Today, chess is one of the most popular strategy games that people play in person or online across the globe. There are even bots which you can play against if there is no one to have as your opponent - which is great practice for real games against people. In most chess games online, there is a points system, where when you beat other players, you take their points and move up as they move down, this drastically increases the "stakes" of the game as each player doesn't want to lose their points.

Chess has become more popular in recent years due to the widespread influence of online personalities such as Magnus Carlsen and Hikaru Nakamura, who are both chess grandmasters. Another element in chess is the timer, where games can last from 10 seconds to 3 days depending on the situation and how intense you want the game to be. This also largely increases the challenge of the game. Most games are 10 minutes long, which gives each player a 10-minute timer that ticks down while they are making their move. When the move is made the timer flips onto the other player and their time starts to tick down. This repeats until one player wins by either their opponent timing out or checkmate. There are also 2 other game-ending situations: stalemate (where the king is not in check but cannot move) or insufficient material (where there aren't enough pieces on the board for either player to checkmate).

Each piece in chess has a different set of moves, both for attacking and moving around the board. These are as follows:

Pawn: move one square up or 2 if on starting square, can attack only diagonally upwards.

Bishop: can move anywhere in a diagonal line from its position. This also goes for attacking.

Knight: can move in an "L" shape, one to the side and 2 up or down and vice versa. This also goes for attacking, one other unique ability of the knight is that it can "jump" over other pieces, completely ignoring them in its moves.

Rook: similarly, to the bishop, it can move anywhere but only: up, down, left, or right, as opposed to diagonally. The rook also uses this movement to attack.

Queen: is a combination of both the bishop and the rook, it can move in any direction: up, down, left, and right and then anywhere diagonal from its current position.

King: being the most important piece, you would think that it would be the most mobile. However, the king can only move one square in any direction so long as it doesn't move into check, this also goes for attacking.

Special Moves:

En passant: if a pawn has moved 2 spaces from its starting position in one move, then it can be taken on the square one up from its starting position as if it has only moved one square.

Castle: if neither the king nor a rook has moved yet, you can swap their positions and move them to the two central squares on the same rank.

Online Chess interests me, as I love problem-solving, and the challenge aspect is very appealing. Climbing the worldwide rankings and increasing my points can become addictive. I also think chess is great for all ages, and anyone who wants to develop their mental skills.

1.1.2 AI

AI is used in many games. It can be seen in Chess, Checkers, tic tac toe and many more games with a multiple-choice aspect. One of the first uses of AI in games goes back to the 1970s when space invaders used stored patterns to simulate random movement. Famously, AI was also used in the game GO, which at the time beat the world champion. This gave AI a large boost in popularity until today when AI is everywhere. In chess, there is a huge following for AI, and there's even an AI championship, where each engine will play against others to find the strongest. One very famous AI is Stockfish which is known as the best Chess AI in the world and has won the AI championship 19 times. In the last few years, AI in general has had a massive increase in popularity, with the likes of ChatGPT and DALL-E. This is mainly because of how capable these AIs have become.

1.1.3 End User/Purpose

Simon is a teacher who runs a chess club for his students but doesn't have enough chess boards or an even number of students for everyone to play at the same time. Simon would like a chess program allowing people to play against each other and to play by themselves. For this, I will have to create an application which allows members of the club to play locally against each other. This program must also allow the students to play with an AI opponent when there is no one to play against. For chess games, there are many options online, but Simon's school doesn't allow him to access them because they are blocked on the network by a proxy server. Therefore, this program will have to be accessible via downloadable software.

1.1.4 Third Party/Client

The third party will be my client Simon, who is very familiar with chess and online applications and understands what can be created. Simon has also played many games with the ability to play locally with other players. He will tell me exactly what he wants in the app and will give a clear reference point for my end goal. Simon wants the game to be simple to use, but he also wants the ability to customise certain aspects. In addition, the game must be flexible enough to accommodate different skill levels.

- 12/5/2023 – I researched some chess algorithms to use for the bot and presented them to Simon. Simon agrees that the minimax algorithm will be best, with its possibility for increased efficiency and how well it works with chess and similar turn-based games. We discussed the possibility of changing the difficulty of the minimax by adjusting the maximum depth of the search, which Simon quite liked. He agreed that this would work perfectly with any new chess players joining the club and for others who are more advanced.

- 19/5/2023 – I had another conversation with Simon about how he would like the game to work for new or inexperienced players. He mentioned that the menus must be easy to navigate and use. I showed Simon some online chess games, to understand if there were any elements of these games he particularly liked. We will have another conversation about these games next week.

- 26/5/2023 – Simon has used all the online chess games I recommended and has given me a list of features he enjoyed. He liked being able to change how long the timer was in each game, which would allow him to play quick matches (3 minutes), mid-length (5 minutes) and longer, more thought-out games (10 minutes). He also liked, being able to change the difficulty of each bot as we discussed previously. For this, he would like 3 difficulties: "Beginner", "Intermediate" and "Master". To further increase the difficulty of the AI opponent, we decided that the use of the quiescence search would be best. This allows for a greater depth, while keeping the computation time low on "quiet" positions. This will only be enabled on the "Master" difficulty, to be used by the more advanced players in the club.

- 2/6/2023 – Simon and I further spoke about the algorithm and how, for timed games, it needs to be as efficient as possible. He has noticed that each turn is only a few seconds at most when playing online chess games. To do this I will have to implement alpha-beta pruning and transposition tables to reduce computation time.

1.2 Research - 1.2.1 Minimax[1]

The minimax algorithm is one of many used as an opponent in games. It works by having a maximising and a minimising player. It runs through all possible moves in a game and chooses the best situation based on which has the highest value. This value is dictated by whether following that path helps to win the game. The objective is to gain the highest number of points for the maximising player and the lowest number of points for the minimising player. The path with the highest number of points is then selected.

Below is a simple pseudocode implementation of the minimax algorithm.

MINIMAX

```
SUBROUTINE minimax(state, depth, maximise)
    IF depth = 0 or game_end(state) THEN
        RETURN value
    ENDIF
    IF maximise THEN
        Value ← -infinity
        FOR moves IN state
            Make_move(state)
            Value ← max(value, minimax(state,depth-1, False))
            Remove_move(state)
        END FOR
        RETURN value
    ELSE THEN
        Value ← infinity
        FOR moves IN state
            Make_move(state)
            Value ← min(value, minimax(state,depth-1,True))
            Remove_move(state)
        END FOR
        RETURN value
    ENDIF
ENSUBROUTINE
```

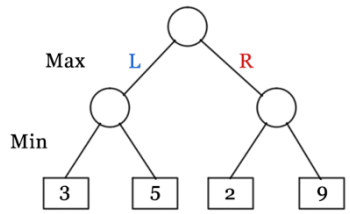
EXPLANATION

1. The function checks if it's at the end of the minimax or if the game is over at which point it returns the value of the current node as the base case.

2. If the player is the maximising player, the value is first set to -infinity because, therefore any value returned will be larger, a move is then made (which could be any turn-based game in this case), and the function is called again with the depth being 1 less than before and the player switching to minimise. The recursive function then runs until the base case is met, at which point the greatest value is returned.

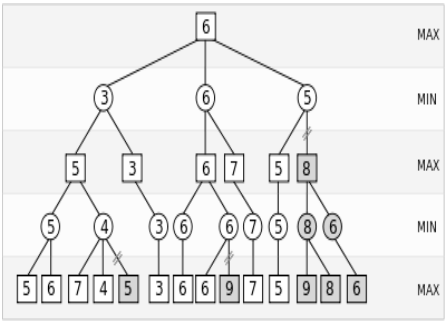
3. The minimising player works the same as the maximising player except, it looks for the minimum between the returned value of the recursive function and the current highest value instead of the maximum.

Because the minimax is a recursive algorithm, visualising its steps can be challenging. However, representing it as a tree diagram can be useful. Here is a simple example of how each turn looks. The algorithm traverses the tree and finds the maximum value down each path. This is then compared to the current largest value at each parent node before a path for the maximising player is chosen. Therefore, if the maximising player starts at the first node it will choose the node on the right because further down the tree there is the current highest value. My implementation of the minimax won't create a tree as a data type, but will instead treat the call stack as a tree and traverse it.



Alpha-Beta Pruning[2]

Commonly, the minimax algorithm is perceived as slow due to the sheer number of possible positions that the algorithm needs to evaluate. To combat this, there are many methods to increase efficiency, such as alpha-beta pruning. This technique holds an alpha and beta value throughout the execution of the recursive function. The alpha and beta values represent the best moves for either player, with the alpha value representing the best move for the maximising player and the beta value representing the best move for the minimising player. These values are updated during each iteration by comparing them with the current node's value. If the beta value becomes less than or equal to the alpha value, it indicates that the minimising player has the superior move and therefore the tree will be cut.



[1] https://en.wikipedia.org/wiki/Minimax
[2] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Fail-soft and fail-hard alpha-beta pruning

Fail-hard and fail-soft are terms used to determine when the tree is pruned in the minimax algorithm. Fail-hard refers to the alpha and beta comparison occurring before the alpha or beta value is updated, while fail-soft refers to updating the alpha or beta value before comparison. Fail-hard is often used for the fastest processing time, whereas fail-soft allows the minimax to explore the sibling nodes to be sure that the pruned path is guaranteed to result in a loss. For my game, I will use fail-soft because, for games like chess, I think that it is important for the AI to be as strong as possible. However, if fail-soft is too slow, I may switch to using fail-hard to maximise efficiency.

Here is the above minimax pseudocode, adapted for alpha-beta-pruning.

MINIMAX Fail-Soft

```
SUBROUTINE minimax(state, depth, maximise, alpha, beta)
    IF depth = 0 or game_end(state) THEN
        RETURN value
    ENDIF
    IF maximise THEN
        Value ← -infinity
        FOR moves IN state
            Make_move(state)
            Value ← max(value, minimax(state,depth-1, False, alpha, beta))
            Remove_move(state)
            alpha ← max(alpha, value)
            IF beta<=alpha THEN
                Break
            ENDIF
        END FOR
        RETURN value
    ELSE THEN
        Value ← infinity
        FOR moves IN state
            Make_move(state)
            Value ← min(value, minimax(state,depth-1,True))
            Remove_move(state)
            Beta ← min(beta, value)
            IF beta<=alpha THEN
                Break
            ENDIF
        END FOR
        RETURN value
    ENDIF
ENDSUBROUTINE
```

MINIMAX Fail-Hard

```
SUBROUTINE minimax(state, depth, maximise, alpha, beta)
    IF depth = 0 or game_end(state) THEN
        RETURN value
    ENDIF
    IF maximise THEN
        Value ← -infinity
        FOR moves IN state
            Make_move(state)
            Value ← max(value, minimax(state,depth-1, False, alpha, beta))
            Remove_move(state)
            IF beta<=alpha THEN
                Break
            ENDIF
            Alpha ← max(alpha, value)
        END FOR
        RETURN value
    ELSE THEN
        Value ← infinity
        FOR moves IN state
            Make_move(state)
            Value ← min(value, minimax(state,depth-1,True))
            Remove_move(state)
            IF beta<=alpha THEN
                Break
            ENDIF
            Beta ← min(beta, value)
        END FOR
        RETURN value
    ENDIF
ENSUBROUTINE
```

Quiescence Search[3]

While a human player can often know when to abandon a bad move or look further into a promising move, the minimax algorithm lacks this ability. To solve this, the quiescence search aims to mimic this by increasing the depth of a search in an "active" position with a second recursive function. Often, the quiescent search will only look at attacking or checking moves, as they are most likely to lead to a "quieter" position. This is to make sure there are no hidden traps behind seemingly good moves and avoid "the horizon effect"[4] or, where a terrible move is only revealed in a future position. There are many ways to decide whether the position is "active" or "quiet". The most common approach is to see if the position has a large amount of attacking moves or moves where pieces can be captured. Similarly to the minimax, this will traverse the call stack and treat it as a tree oppose to creating a tree data type.

Below is a pseudocode implementation of the quiescent search algorithm and Fail-Soft alpha beta pruning.

```
SUBROUTINE quiescent_search(state, depth, maximise, alpha, beta)
    IF depth = 0 or quiescent_end(state, depth) THEN
        RETURN value
    ENDIF
    IF maximise THEN
        Value ← -infinity
        FOR moves IN attacking_moves(state)
            Make_move(state)
            Value ← max(value, quiescent_search(state,depth-1, False, alpha, beta))
            Remove_move(state)
            alpha ← max(alpha, value)
            IF beta<=alpha THEN
                Break
            ENDIF
        END FOR
        RETURN value
    ELSE THEN
        Value ← infinity
        FOR moves IN attacking_moves(state)
            Make_move(state)
            Value ← min(value, quiescent_search(state,depth-1,True))
            Remove_move(state)
            Beta ← min(beta, value)
            IF beta<=alpha THEN
                Break
            ENDIF
        END FOR
        RETURN value
    ENDIF
ENDSUBROUTINE
```

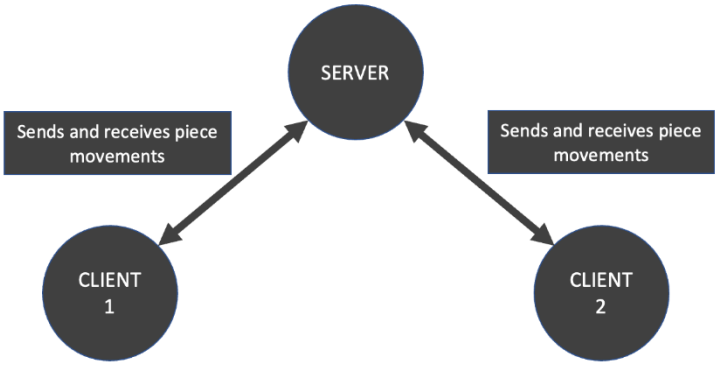
Transposition Tables[5]

Transposition tables are another way to increase the efficiency of the minimax algorithm. Although there are millions of possible positions in a chess game, some of these positions may repeat throughout different games. And so, a transposition table can be used to store previously encountered positions along with their best move. Therefore, when a known position is seen again, it doesn't require reprocessing but can instead refer to the transposition table to find the best move. These tables will be stored and accessed from a JSON[6] file.

[3] https://en.wikipedia.org/wiki/Quiescence_search
[4] https://en.wikipedia.org/wiki/Horizon_effect
[5] https://en.wikipedia.org/wiki/Transposition_table
[6] https://docs.python.org/3/library/json.html

1.2.2 Client Server Networking

The diagram on the right illustrates the connection between two clients and a server. Every tick, a "None" signal is sent from each client to the server. In the context of chess, each client will make a move which will be sent to the server, instead of the "None", for the server to send to the other client. This data will then be checked to see if it is a move and not any other data (like text chat, or just a None). After the move has been validated, it is executed on the recipient client's chessboard. This setup will also enable local text chat functionality.



To run my client-server network, I will use the Python Socket[7] library, which will allow me to implement both the client and server. This library allows for plenty of customisability when configuring my server. When making decisions for my network setup, I have considered factors such as IPV4 versus IPV6 and UDP versus TCP protocols. Firstly, I have chosen IPV4 because of its widespread standardisation and simplicity over IPV6. I have also decided upon TCP as the protocol of choice for its high reliability. Since chess is a turn-based game that doesn't require fast communication, the speed advantages of UDP are unnecessary, especially with TCP's speed being more than sufficient.

For my client-server, I can only send bits and bytes rather than strings, integers, or arrays. To overcome this limitation, I will serialise the data using either Pickle[8] or JSON. I have chosen to use Pickle due to its flexibility and that it is ready to send after serialisation, whereas JSON will require additional ASCII to binary translation. To run the server, I will have to find its local IP address so that each client knows where to connect. One thing to keep in mind is that the clients must be on the same local network as the server to connect to it without requiring port forwarding.

To further increase the capability of my client-server networking model, I will add a "received" message to ensure that any data initially lost in transmission can easily be resent. This reinforces the client-server on a whole, eliminating any communication issues that may occur.

1.2.3 Multi-Threading

The server can only establish one connection at a time. To fix this issue and connect to both clients, I will have to use multithreading with the threading[9] library in Python. This allows me to establish a single connection on each instance of the running server code. Using this, I can isolate and manage each client individually, ensuring that the data sent is exactly where it needs to be. Furthermore, this will allow players to seamlessly switch between playing against the minimax and a local opponent while any network processing can be carried out in the background. I will also use threading with the single-player functionality when playing against the minimax. This will eliminate any freezing or stuttering while the opponents next move is being processed.

[7] https://docs.python.org/3/library/socket.html
[8] https://python.readthedocs.io/en/stable/library/pickle.html#module-pickle
[9] https://docs.python.org/3/library/threading.html

1.3 Objectives

- The general backbone of the chess game must have the following features:
    - Check all possible legal moves when a piece is selected and display these moves on the board.
    - In timed games, a countdown clock must show how much time each player has left.
        - The countdown should have a toggle button to turn it on or off before the game starts.
        - This countdown length should be able to be chosen with the options:
            - 3 minutes.
            - 5 minutes.
            - 10 minutes.
    - After each piece is taken, it must be shown in order of value, on the opponent's side.
    - The difference in the total score of each side's pieces should be displayed on the winning side.
    - It should check if either player is in check, and if so, don't allow that player to move unless it gets them out of a checked position.
    - An ADT stack should be used for the menus and should:
        - Store all menus in a layered format with the most recently accessed menus nearest the top.
        - Display the menu at the top of the stack.
        - Remove and return the top menu, or access settings if in a game, when "escape" is pressed.
        - Add a menu to the top of the stack if the relating button is pressed.

- The program must include a computer-controlled opponent using the recursive minimax and quiescence search algorithms. This should:
    - Allow the user to change the difficulty level by altering the maximum depth of the search.
        - For beginner difficulty, this should use a minimax depth of 2.
        - For intermediate difficulty, this should use a minimax depth of 3.
        - For master difficulty, this should use a minimax depth of 2 and a quiescent search depth of 2.
    - Use the recursive quiescence search to further increase the capability of the minimax. Which should:
        - Use alpha-beta pruning to decrease processing time.
        - Search all attacking and checking moves.
        - Return the static evaluation of the end node in the call stack.
    - Alpha-beta pruning should be used to remove unnecessary branches of the minimax tree and further decrease processing time between turns.
    - Hold previous positions and their best moves to increase efficiency early on in a game.
    - The core game while playing against the minimax algorithm should include the following features:
        - A draw button which only causes a draw if the minimax is losing.
        - A resign button that ends the game with a resignation message.
        - A rewind button to allow players to take back moves reverting to a previous frame in a stack.
            - The ADT stack should hold board frames.
            - It should also allow me to pop to take the last board/move off and return it.
            - It should add the position to the stack after each move is made.
            - It should have methods to check if its full or empty.
            - It should have a method to look at the top item in the stack.
            - The stack should have a method to display the stack.

- The program must include complex client-server networking, which should:
    - Allow users to play games locally against others using multi-threading. with the following features:
        - A draw button that sends a request to draw with the opponent.
            - The draw request should last 3 turns before resetting unless against the minimax, in which case, it will only accept the draw if the player is losing.
        - A resign button that sends a resign message to the opponent.
        - A text chat where both players can send messages to each other.
            - This should display "opponent: message" or "you: message" depending on who sent the message and who views it.
        - It will tell the opponents game when a move is made.
    - Use a list acting as a data queue to hold the data that has been received and is ready to be processed.
    - Use an error checking send and receive model that sends a 'received' message when a new piece of data has been received, so that there are no errors during transmission.
    - Players should also be able to play while using different operating systems without problems.

1.4 Modelling and Prototyping

1.4.1 Client Server

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(("8.8.8.8", 12345))
print(“Local IP:”, s.getsockname()[0])
s.close()
```

Here is a basic Python implementation of how the server finds its local IP address. It does this by creating an arbitrary connection, in this case, Google’s main domain name system’s IP address (“8.8.8.8”). This connection uses “socket.SOCK_DGRAM” which indicates the use of UDP. I have used UDP in this case over TCP because there is no need for efficiency but the faster the connection communicates, the better. Indicated on line 4, is the use of the port “12345”. There is no real reason to use this port, but the higher the better because most low ports are reserved. I then print “s.getsockname()[0]” which returns the local IP address and port of the server’s connection. And so, the use of “[0]” prints only the local IP address from “(IP, port)”. Finally, the connection is closed. Below is an example of what this prints.

```
“Local IP: 192.0.2.146”
```

Below is a full implementation of the server code.

```
import pickle
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(("8.8.8.8", 12345))
print("Local IP:", s.getsockname()[0])
s.close()

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
port = 12345
s.bind((“”, port))
s.listen(5)
c, addr = s.accept()
print("connected to ",addr)

while True:
   print(count)
   try:
      c.send(pickle.dumps(None))
      print(pickle.loads(c.recv(1024)))
   except ConnectionResetError:
      c.close()
      break
```

After finding the server's local IP, I can establish a socket, this time using “socket.SOCK_STREAM” indicating the use of TCP which, as I mentioned previously, allows for a more reliable connection at the cost of unnecessary speed for chess. The address and port are then bound to the socket with “s.bind(“”, port)” where (“”) represents the local IP address. “s.listen(5)” allows me to set the maximum number of failed connections before the server just refuses all new connections. This then waits for a connection at “c, addr = s.accept()” with “c” corresponding to the connection itself and “addr” representing the address of the connection. A loop is then started to send and receive data between itself and the client, in which, I use pickle to serialise and de-serialise the data. The “try” and “except” statements allow the code to continue running in the event of a connection loss.

Below is an implementation of the client code.

```
import pickle
import socket
s = socket.socket()
s.connect((“192.0.2.146”, 12345))
while True:
   print(s.getsockname())
   try:
      print(pickle.loads(s.recv(607)))
      s.send(pickle.dumps(None))
   except ConnectionResetError:
      break
s.close()
```

Here, when I establish the socket, I connect to the server given the port and IP that the server is bound to. You will notice that the order of sending and receiving data is flipped, this is so that when the server sends, the client receives and vice versa.

1.4.2 Multithreading

```
import time
import pickle
import socket
import threading

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(("8.8.8.8", 80))
print(s.getsockname()[0])
s.close()

def send_data(p):
   global active_connections
   while True:
      try:
         p.send(pickle.dumps(counts))
         print(pickle.loads(p.recv(1025)))
      except ConnectionResetError:
         c.close()
         active_connections-=1
         print(f"active connections: {active_connections}")
         break

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
active_connections = 0
port = 9422
s.bind(('', port))
print("socket binded to", port)
s.listen(5)

while True:
   try:
      print(f"active connections: {active_connections}")
      print("establishing a connection")
      c, addr = s.accept()
      print(“Connected to”, addr)
      thread = threading.Thread(target = send_data, args = (c,), daemon = True)
      thread.start()
      active_connections+=1
   except:
      break
print("server closed")
```

In the code above, I have used multithreading to allow for multiple client connections on the same server. To do this, I have used the "threading" library which allows me to start a new instance of a subroutine with the given arguments. For this, I have created the "send_data" procedure that can be used in threads, sending and receiving data in a connection with a client. This will then break out of the thread if there is an error with the connection. After a new thread has started, the server starts listening for another connection so that a second client can join.

1.4.3 Minimax

Below is a simple Python implementation for the minimax algorithm for a noughts and crosses game.

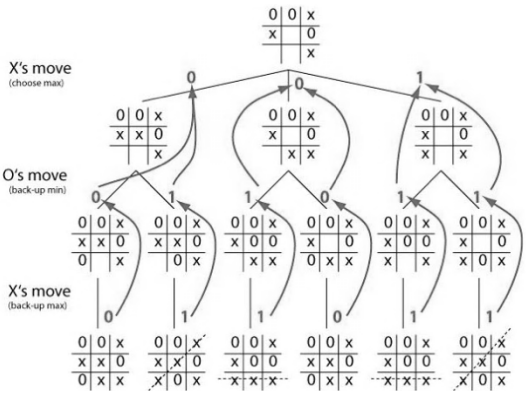
```
def minimax(state, depth, player):
    if depth == 0 or game_over(state):
        score = evaluate_board(state)
        return [-1, -1, score]

    if player == comp:
        score = [-1, -1, -999]
        for i in find_empty_cells(state):
            x, y = i[0], i[1]
            state[x][y] = player
            if (j := minimax(state, depth - 1, -player)[2]) > score[2]:
                score[2] = j
                score[0], score[1] = x, y
            state[x][y] = 0
        return score

    else:
        score = [-1, -1, 999]
        for i in find_empty_cells(state):
            x, y = i[0], i[1]
            state[x][y] = player
            if (j := minimax(state, depth - 1, -player)[2]) < score[2]:
                score[2] = j
                score[0], score[1] = x, y
            state[x][y] = 0
        return score
```

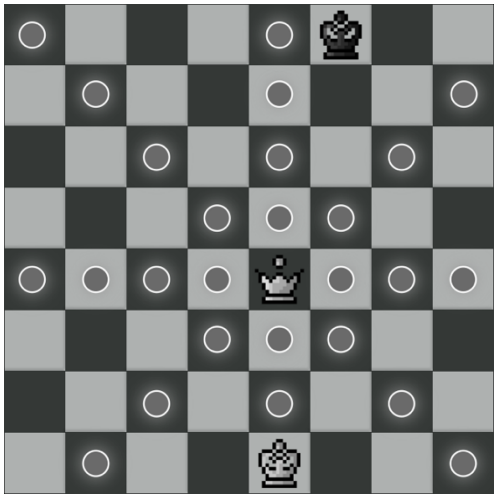
Here, I use parameters: "state" for the current board state, "depth" for the current depth in the minimax and "player" to define whether it is the maximising or minimising player's turn. This also uses the following functions: "find_empty_cells()" to return all empty cells and therefore all possible positions to place a nought or cross, "game_over()" to return whether the game has finished either by win or draw, and "evaluate_board()" to return the current value of the board for the maximising player. This runs through each position from the starting state alternating the player after each move. Once the end of the game has been reached or the depth = 0 the function returns "[-1, -1, score]" to its previous call where the first 2 indexes can be changed to the x and y of the best move. This is then returned up the call stack until it reaches the original call where the move can be made.

Here is an example of the minimax tree in the context of noughts and crosses from ResearchGate[10]. The example shows how, when the end of the game is reached, the position is evaluated. This value is then passed up the tree where, on the minimising turn the worst move is returned which is either 1 or 0 depending on whether a win is guaranteed from the current position. The largest of these values is then chosen on the initial maximising player turn and the respective move is made. If all move options for the maximising player result in the same value, the first move will always be chosen no matter whether it's a win or a loss. So, in the case of the diagram, the move chosen will be one on the far right, this is because it has the highest value.



[10] https://www.researchgate.net/figure/Game-tree-for-Tic-Tac-Toe-game-using-MiniMax-algorithm_fig1_262672371

The complexity of the minimax algorithm increases dramatically when moving from noughts and crosses to chess.



Here is a diagram showing all the possible moves for the white queen piece in the middle of the board. Based solely on this model, the game has over 2000 different possible positions for the next 3 moves. which, when compared to noughts and crosses, is astounding. After just 6 moves from the start of a chess game there are 120,921,506 possible positions, vs just 8640 possible positions for the first 6 moves a in noughts and crosses game.

Due to this colossal increase in complexity, the time it takes the minimax to make each move also increases, from around 0.005 seconds with a depth of 9 on noughts and crosses to 2 seconds with a depth of 3 in chess (both using alpha-beta pruning). This is also caused by the new addition to the minimax, of storing captured pieces values on each node of the tree.

Below is an implementation of fail-soft alpha-beta pruning in the minimax shown above.

```
def minimax(state, depth, player, alpha, beta):

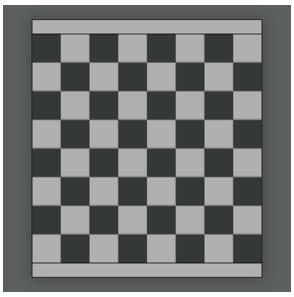
    if depth == 0 or game_over(state):
        score = evaluate_board(state)
        return [-1, -1, score]

    if player == comp:
        score = [-1, -1, -999]
        for i in find_empty_cells(state):
            x, y = i[0], i[1]
            state[x][y] = player
            if (j := minimax(state, depth - 1, -player, alpha, beta)[2]) > score[2]:
                score[2] = j
                score[0], score[1] = x, y
            state[x][y] = 0
            alpha = max(score[2],alpha)
            if beta <= alpha:
                break
        return score

    else:
        score = [-1, -1, 999]
        for i in find_empty_cells(state):
            x, y = i[0], i[1]
            state[x][y] = player
            if (j := minimax(state, depth - 1, -player, alpha, beta)[2]) < score[2]:
                score[2] = j
                score[0], score[1] = x, y
            state[x][y] = 0
            beta = min(score[2],min)
            if beta <= alpha:
                break
        return score
```

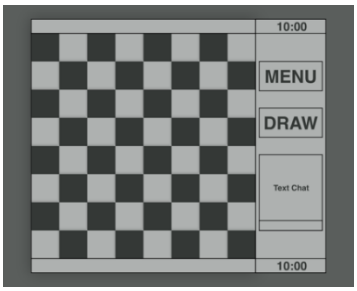
This sets the alpha and beta values as the best moves for each player. Therefore, If the beta value is less than the alpha value then the branch is cut. This means that the opponent's best move is better than the minimax's best move. This decreases the processing time for the first move from 0.6s to 0.02s, which makes a dramatic difference when playing against the minimax.

1.5 UI Modelling



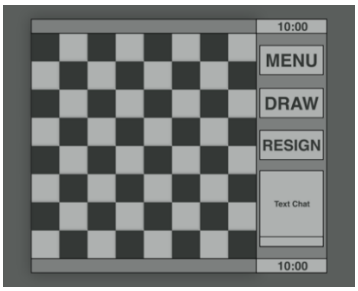
Here is my first iteration of the GUI for the chess game. After showing this to Simon, he suggested that i implement the following in the next draft:

- A button to allow players to offer a draw.
- Timers to display how many minutes seconds each player has.
- A text chat for players to communicate.



Here is a revision of the first draft. I have added, a menu button, a draw button, a text chat box, and a timer for each player. After showing this to Simon, he suggested these improvements:

- Add a “resign” button to allow players to concede
- Separate the sections so it is easier to differentiate between each



Here is the final main UI design. Simon is now happy with how everything is laid out. All functions will remain the same whether the user is against another player or the minimax.



After further conversations with Simon, we have decided that a simple stacked button layout is ideal for the main menu and other menus to follow. However, he would like the buttons to have rounded corners as to not be so harsh.



Simon likes the new revision of the main menu; however, he would like the text to contrast more against the background. Therefore, I will a colour for the text that contrasts much more than the 2 shades of grey in this example.



Here is the final revision of the main menu design, with a contrasting text colour and rounded buttons. The buttons will also become a darker shade when hovered over and will move down when pressed.

1.5.1 Chess Club Survey

The main users of my chess program will be the student members of Simon's chess club. To ensure that they are content with the design and playability of the game, I have conducted a survey as follows. The questions were used to gain a greater understanding of what a real end-user expects from a chess application.

Questions	Student 1	Student 2	Student 3	Response
What are some key design aspects you will expect from the chess program?	"I think that the ability to change the main colours of the program will allow better visibility and will allow some personalisation for each game."	"It is very important to be able to navigate through the menu using the keyboard, for example, pressing escape to return to the previous menu."	"All text should be easily visible and should have a consistent style throughout the program for consistency."	I will make sure to include a Theme menu, escape keyboard navigation, and a clear text and asset style throughout.
For the main UI style, would you prefer a pixel-art style or a rounded realism style?	"Pixel-art will help text be bold and readable, while realism will allow the program to feel closer to playing a game over the board. so, both pixel-art and realism will fit the game effectively."	"Pixel-art is much better as it feels much more refined and finished with everything being made up of squares."	"Realism will be very difficult to capture accurately and so may feel clunky. I would much prefer a fast interface that is pixel art, over one that is clunky and rounded."	Based on this feedback, I will use a pixel-art theme to give a consistent, readable, and refined feel.
How do you feel about my current UI mock-ups which I discussed with Simon? (above in 1.5.1)	"There is definitely still some refining to do before the final product is reached. Overall, the general layout is great for easy access to anything that you might need."	"Once the menus are altered to fit a more consistent style, it will bring the program together as a whole and, bring it to life a bit more than the mock-ups."	"I like them but, I feel like there needs to be more substance to the menus. For example, subtle animations or moving parts would add lots of value."	The consensus seems to be that the mock-ups are good, but I need to add some subtle movement to give them some more life and to create a greater user experience.

2.0 Documented Design

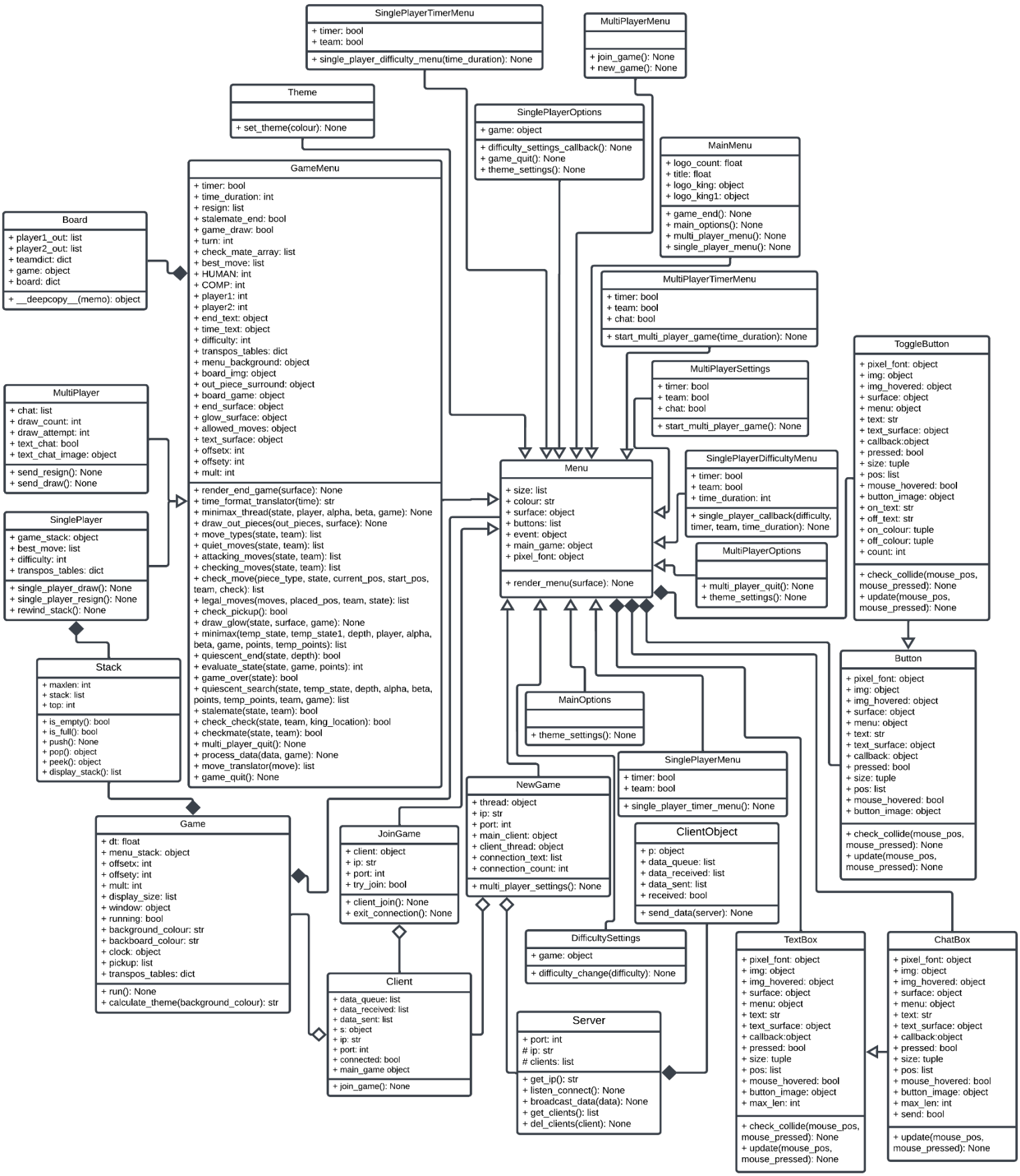
I am creating a Chess game for Simon, so that he will be able to have more members in his chess club without running out of boards. It will also act as a practice game using the minimax algorithm with varying difficulties.

For my game to meet my objectives, I will need to design:

- A user interface.
- OOP classes for each piece type.
- OOP class for the board.
- OOP classes for buttons and menus.
- A recursive minimax algorithm for call stack traversal with:
    - Alpha-Beta pruning.
    - Recursive quiescence search extension.
    - JSON transposition tables.
- A complex client-server network model for the transmission of moves, text chat and game events.

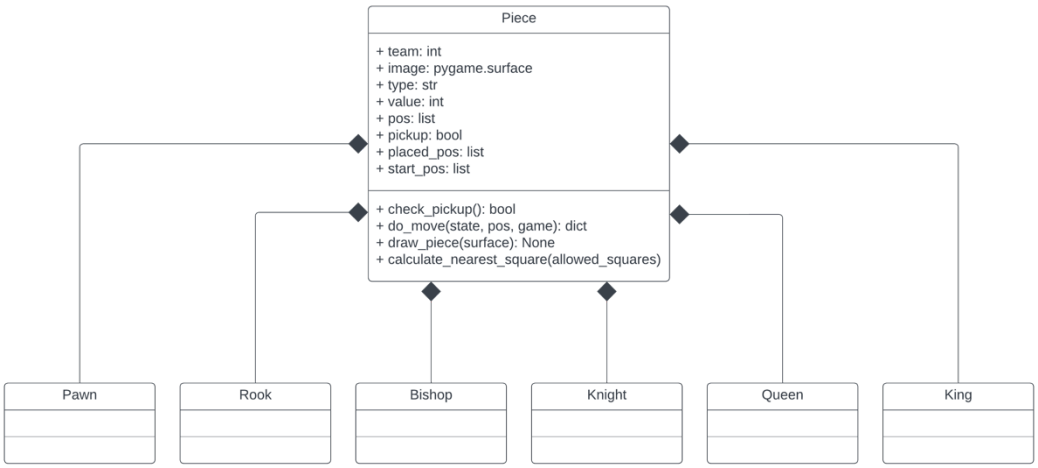
2.1 Gameplay

2.1.1 Menu Classes



2.1.2 Piece Classes

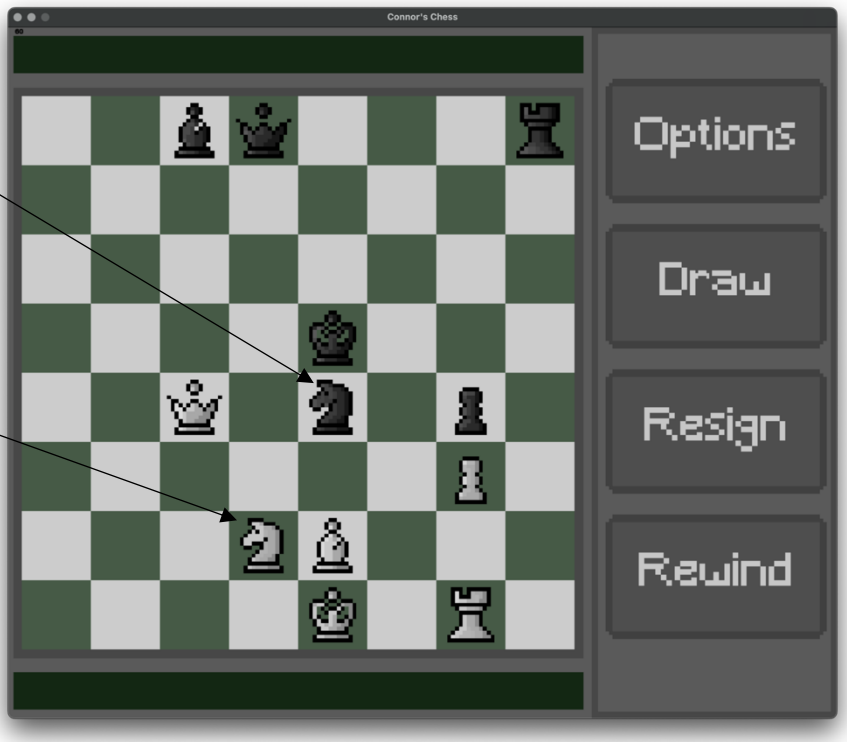
Below is a class diagram, showing the relationships of each piece child classes with the parent Piece class. None of the pieces have different attributes, only different values for these attributes, this is because all pieces are mostly treated the same. The only exceptions are handled by these differing attribute values.



2.1.3 Board Dictionary

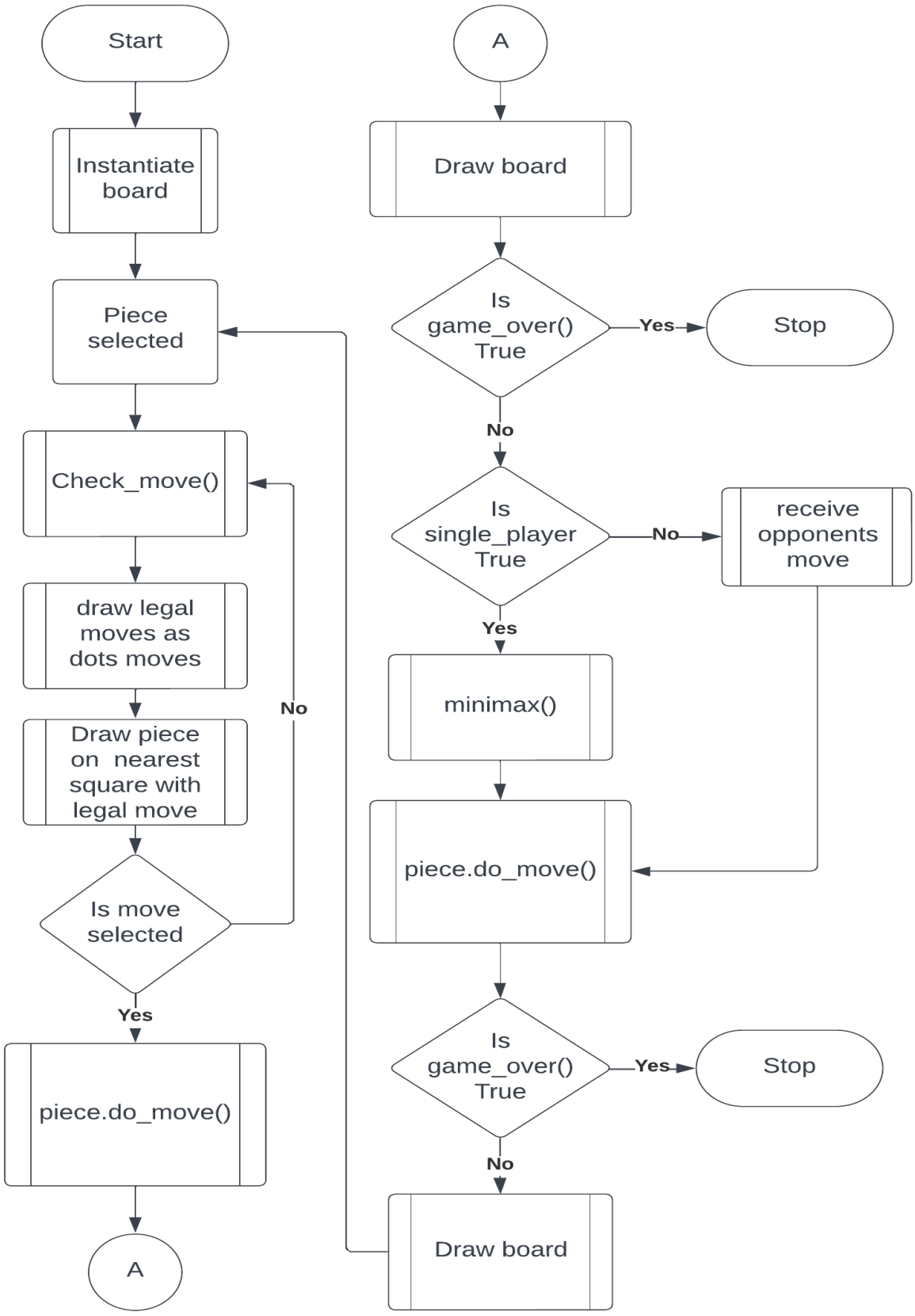
On the left, is an example of how the board dictionary could look and, on the right, is the board that it represents. The main reason that I have used a dictionary to represent the board, and not an array, is that when iterating through the board, I only need to look at pieces which are still in play. This saves wasted processing power that would have been spent looking at empty spaces which have no use. Inside the dictionary, the key is the position of the piece in the form (x, y), where (0, 0) is in the top left, and the value is an instance of a piece class (e.g. “Knight”). The arrows below show the relation between the instance in the dictionary and the physical piece on the board.

```
self.board = {
        (4, 4): pieces.Knight(self.game.player1, (4, 4), self.game),
        (2, 0): pieces.Bishop(self.game.player1, (2, 0), self.game),
        (6, 4): pieces.Pawn(self.game.player1, (6, 4), self.game),
        (4, 3): pieces.King(self.game.player1, (4, 3), self.game),
        (3, 0): pieces.Queen(self.game.player1, (3, 0), self.game),
        (7, 0): pieces.Rook(self.game.player1, (7, 0), self.game),
        (3, 6): pieces.Knight(self.game.player2, (3, 6), self.game),
        (6, 5): pieces.Pawn(self.game.player2, (6, 5), self.game),
        (4, 7): pieces.King(self.game.player2, (4, 7), self.game),
        (2, 4): pieces.Queen(self.game.player2, (2, 4), self.game),
        (4, 6): pieces.Bishop(self.game.player2, (4, 6), self.game),
        (6, 7): pieces.Rook(self.game.player2, (6, 7), self.game),
    }
```



2.1.4 Top-level flowchart

Below, is a top-level flow chart describing how the user interacts with the main parts of the system, and how some sections interact with each other.



2.2 UI Design

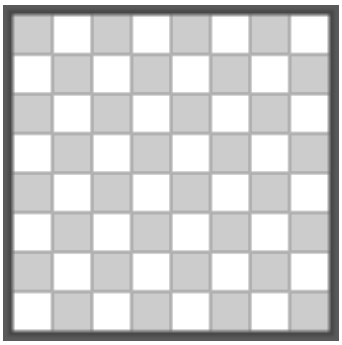
2.2.1 Assets

After several conversations with Simon, my third party, I have chosen to use these pixel-art piece assets for both single-player and multi-player games, which I found on https://opengameart.org/content/pixel-chess-pieces

2.2.1.1 Pieces



2.2.1.2 Board



Here is my main board image, which I drew after my previous conversation with Simon about what he would like the UI to look like. Using a transparent background on a PNG has allowed me to use a theme colour which is standardised across the whole program



This is the surround that I am using for the out pieces (pieces that have been taken). Using only a surround, has also allowed me to use a consistent colour theme



This is a sidebar for the main game, which I will draw buttons onto next to the main board

2.2.1.3 Buttons



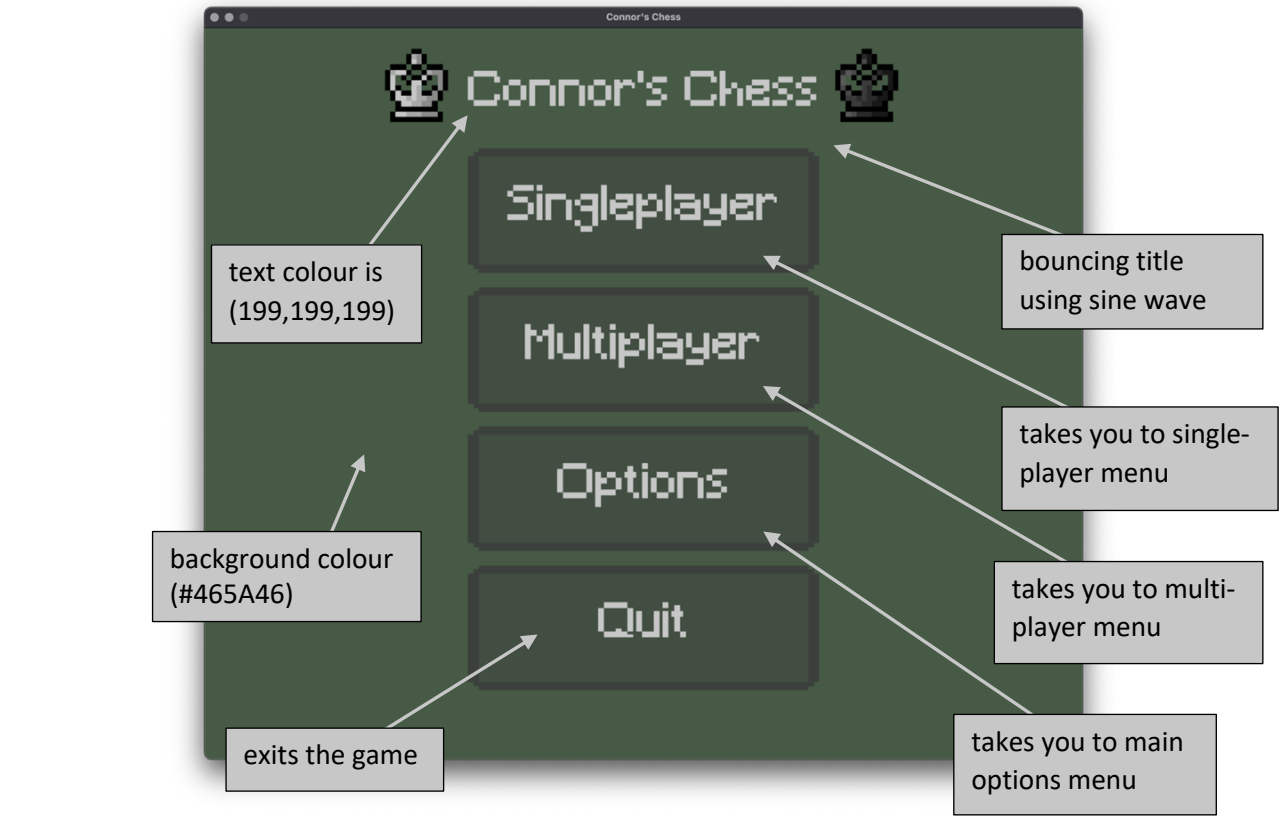
This is the standard button background for all buttons. This is also semi-transparent, which allows the theme colour of the background to be seen through it



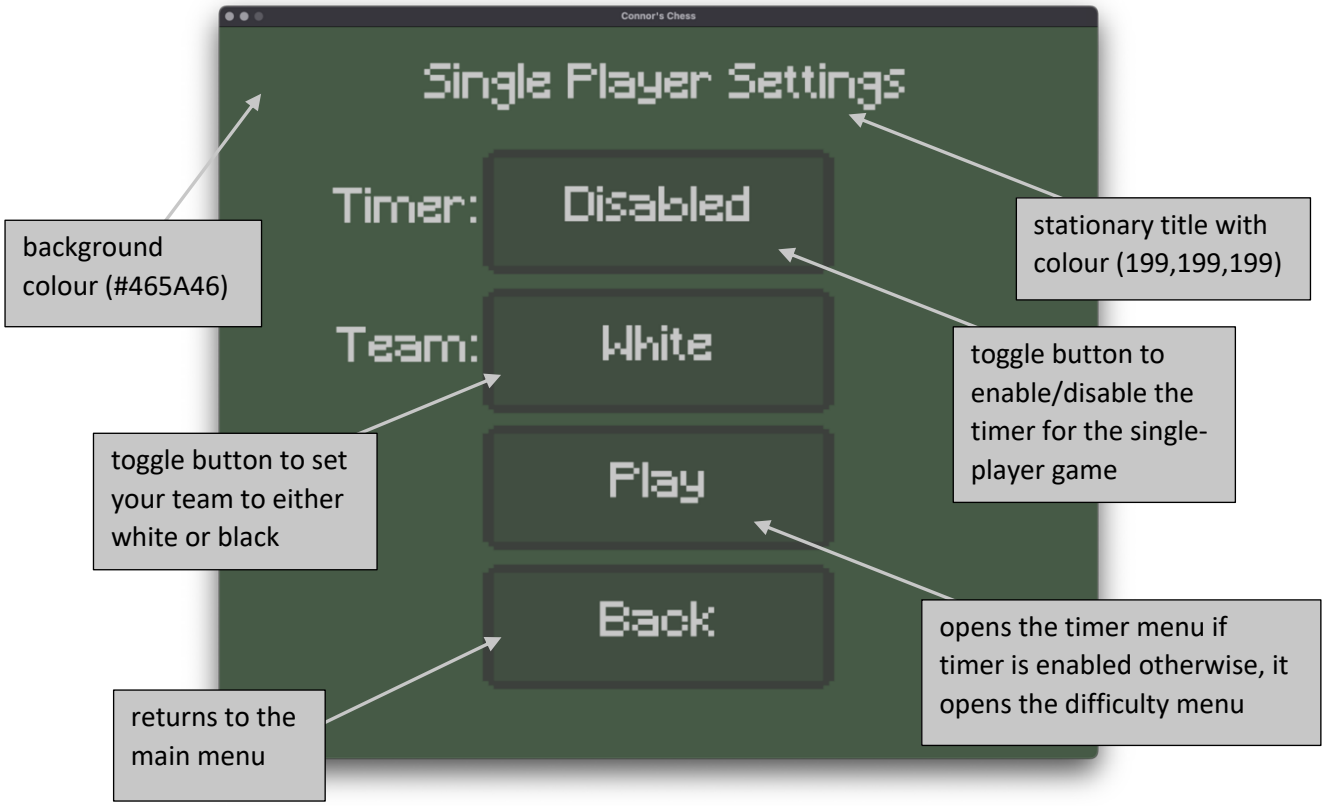
This button image is the same as the one above however, it is slightly darker to signify that the button is being hovered over by the mouse. the button will also move down by 5 pixels when pressed to give a “pressed down“ feel

2.2.2 Menus

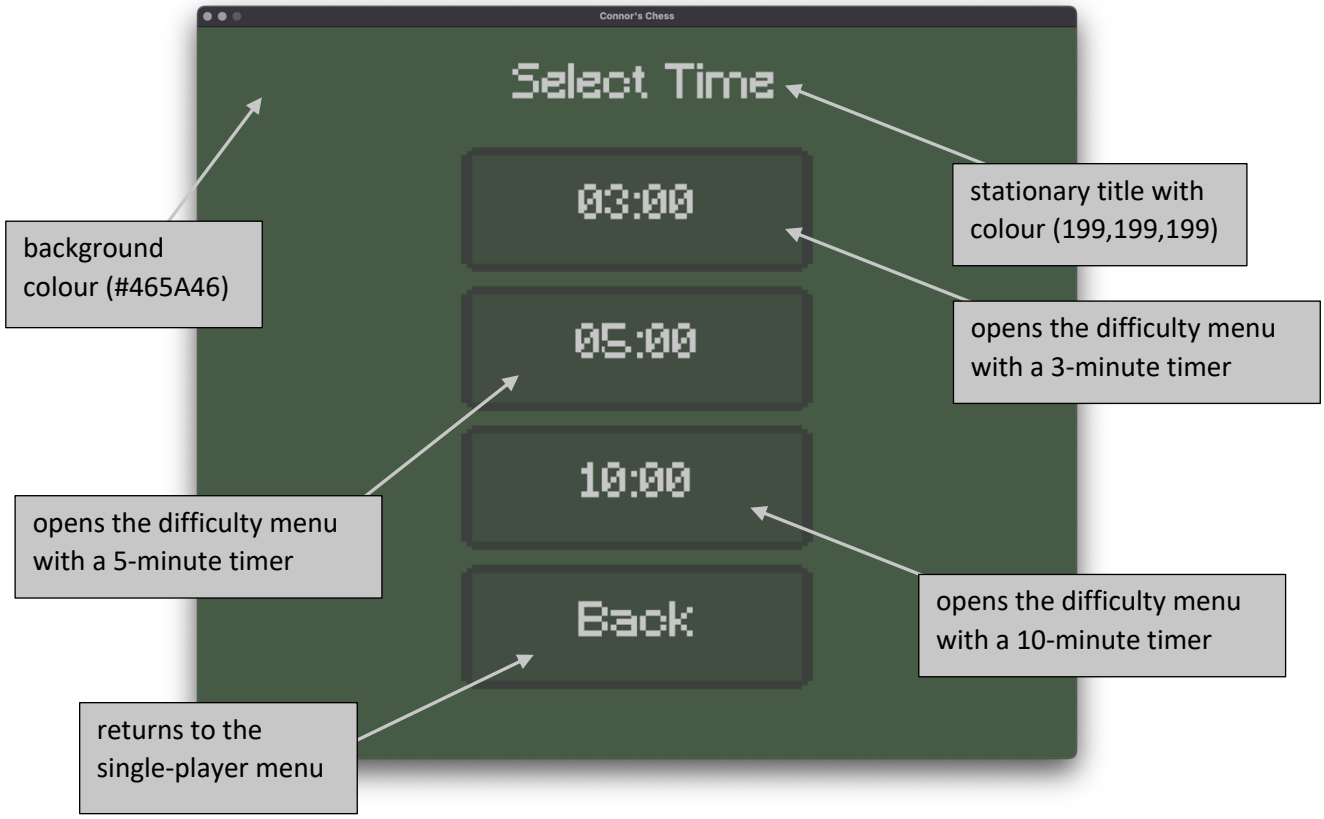
2.2.2.1 Main menu



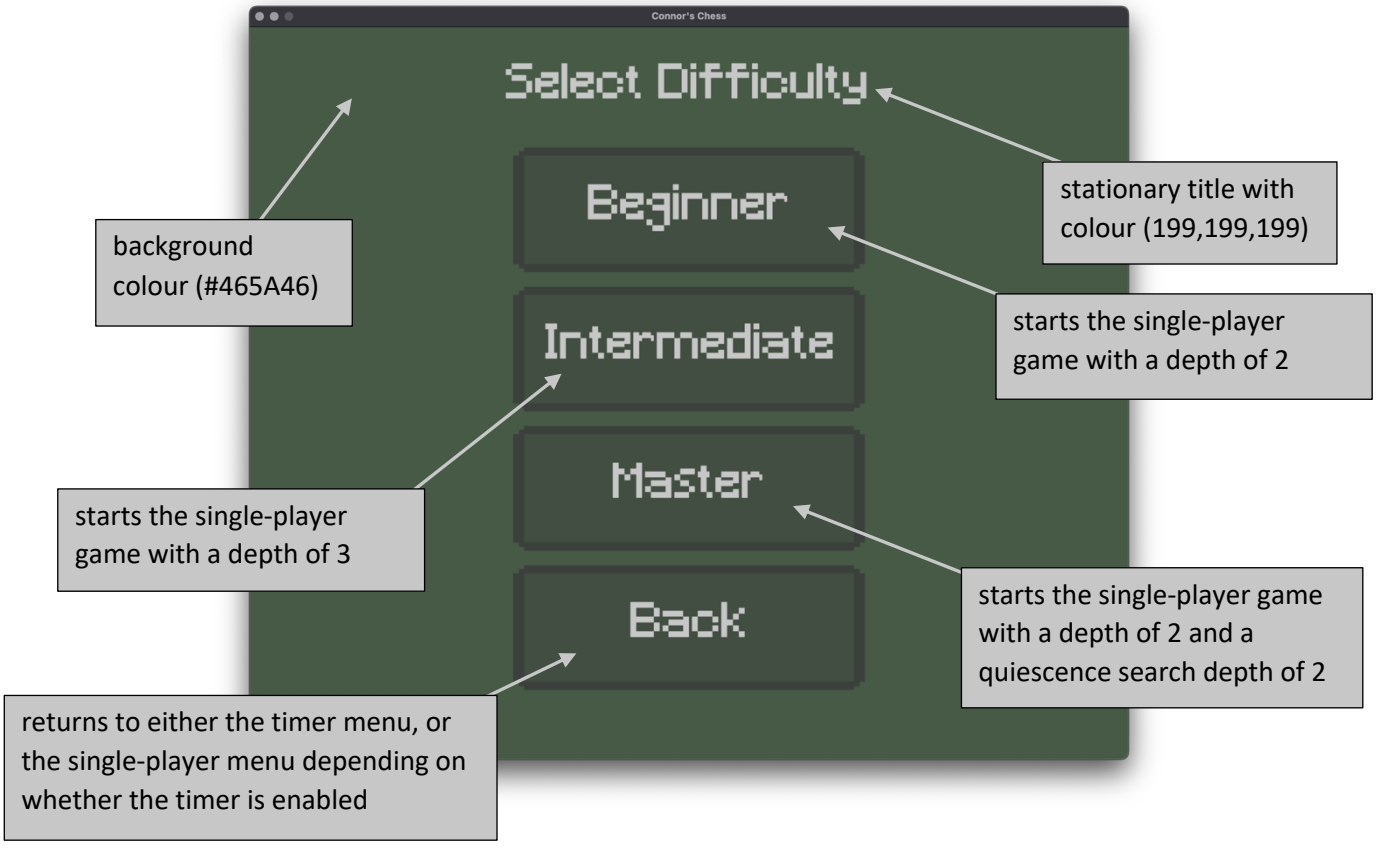
2.2.2.2 Single-player menu



2.2.2.3 Single-player timer menu



2.2.2.4 Single-player difficulty menu



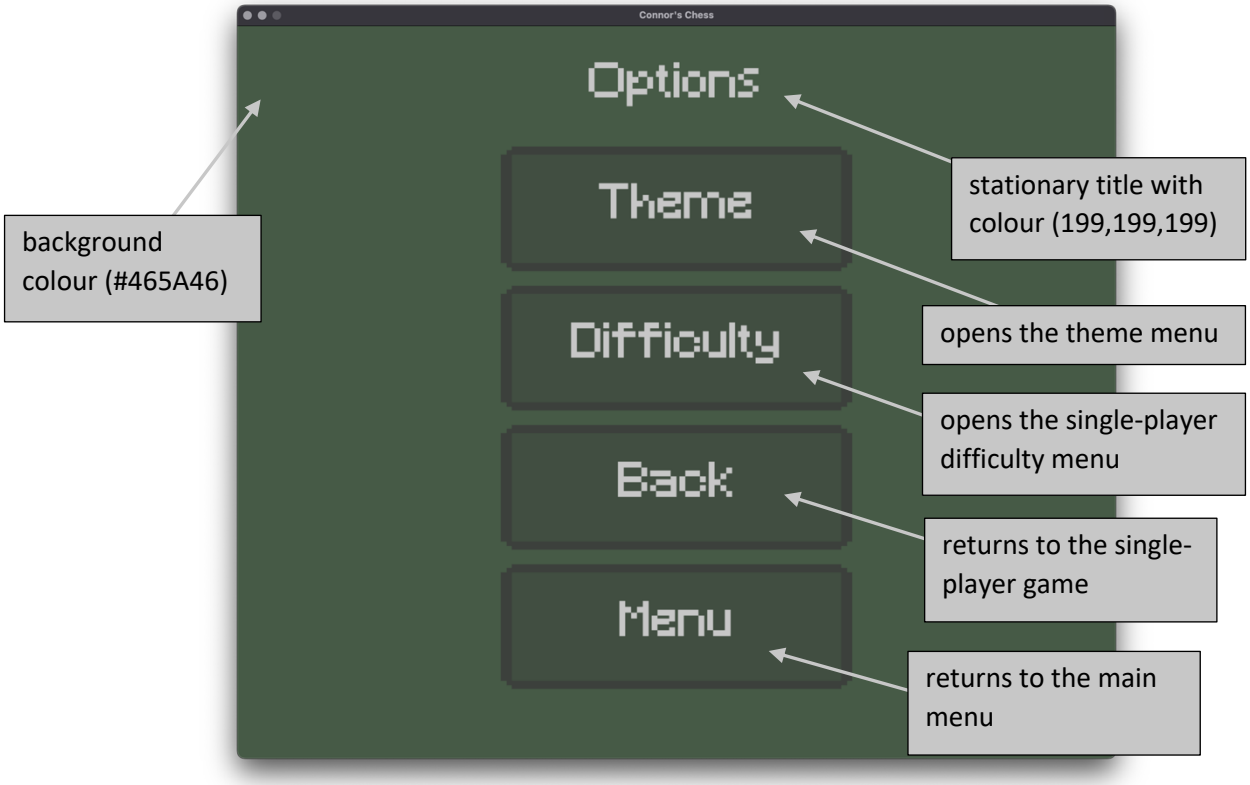
2.2.2.5 Single-player with timer enabled



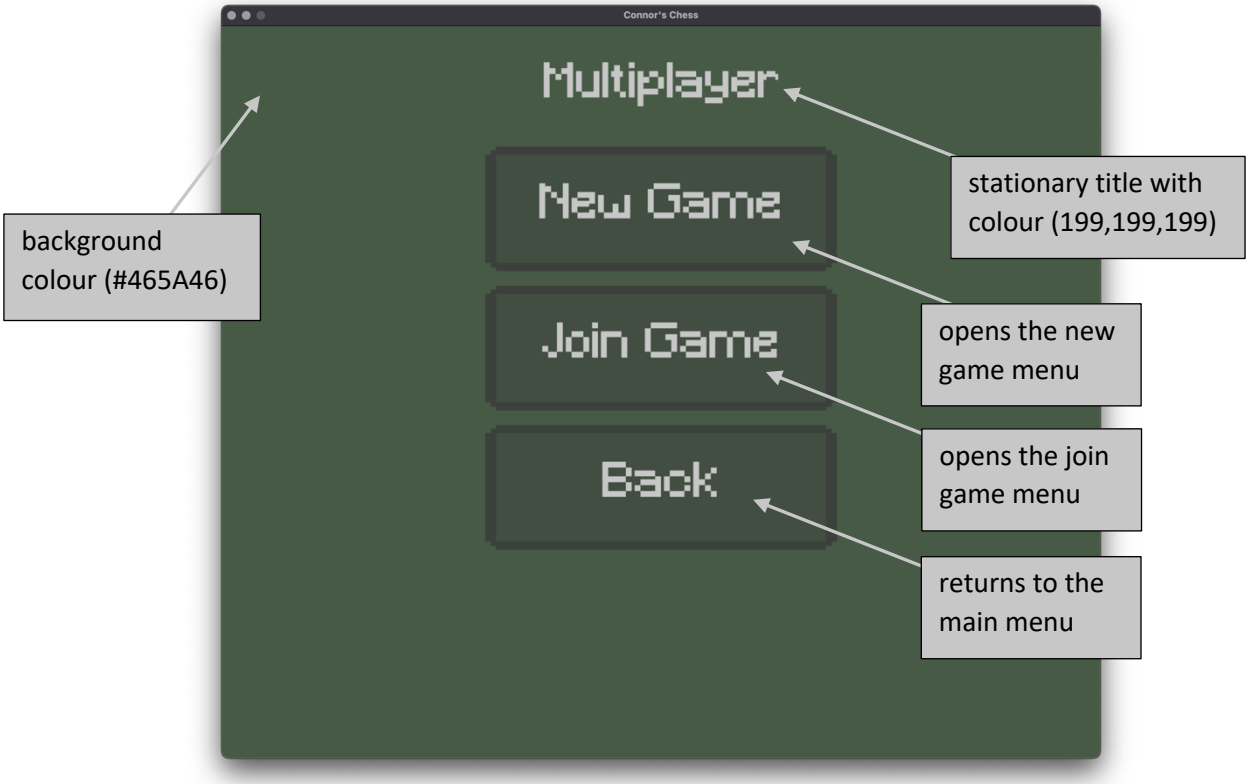
2.2.2.6 Single-player with timer disabled



2.2.2.7 Single-player options menu



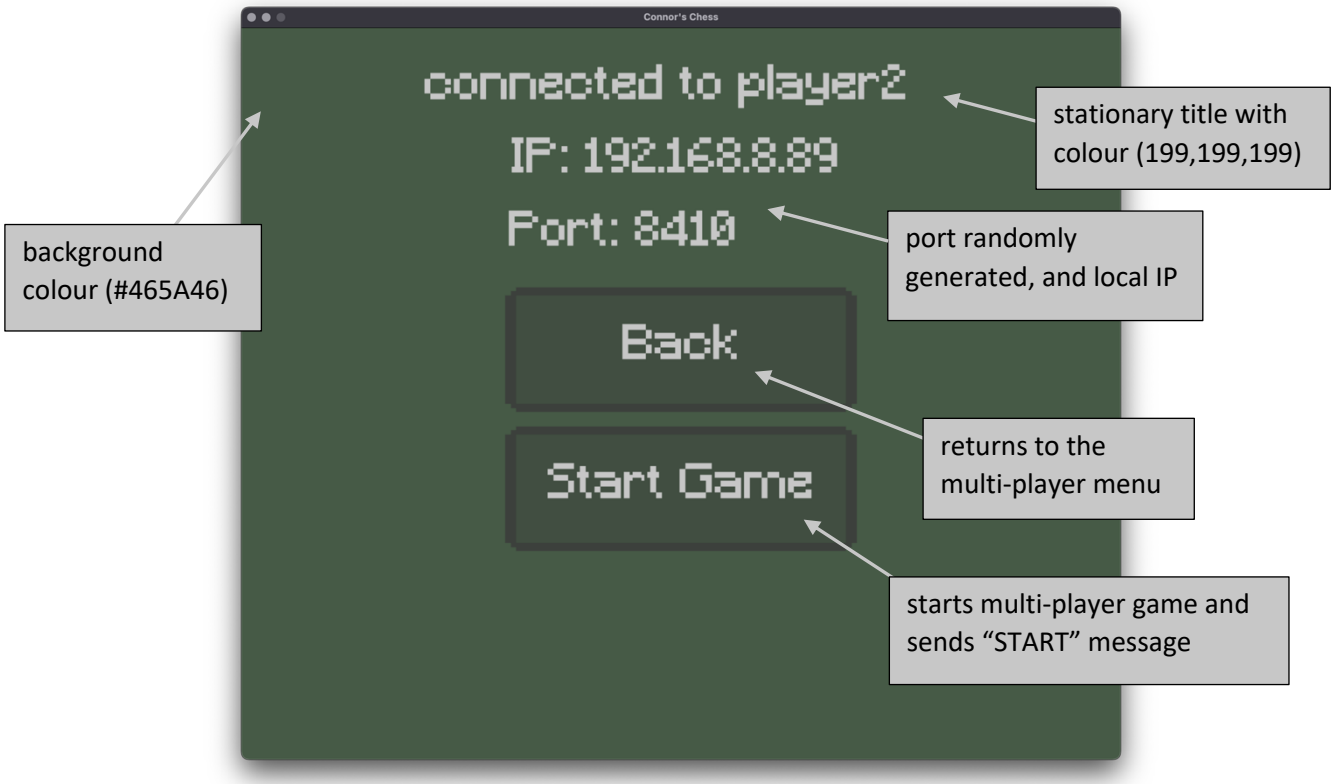
2.2.2.8 Multi-player menu



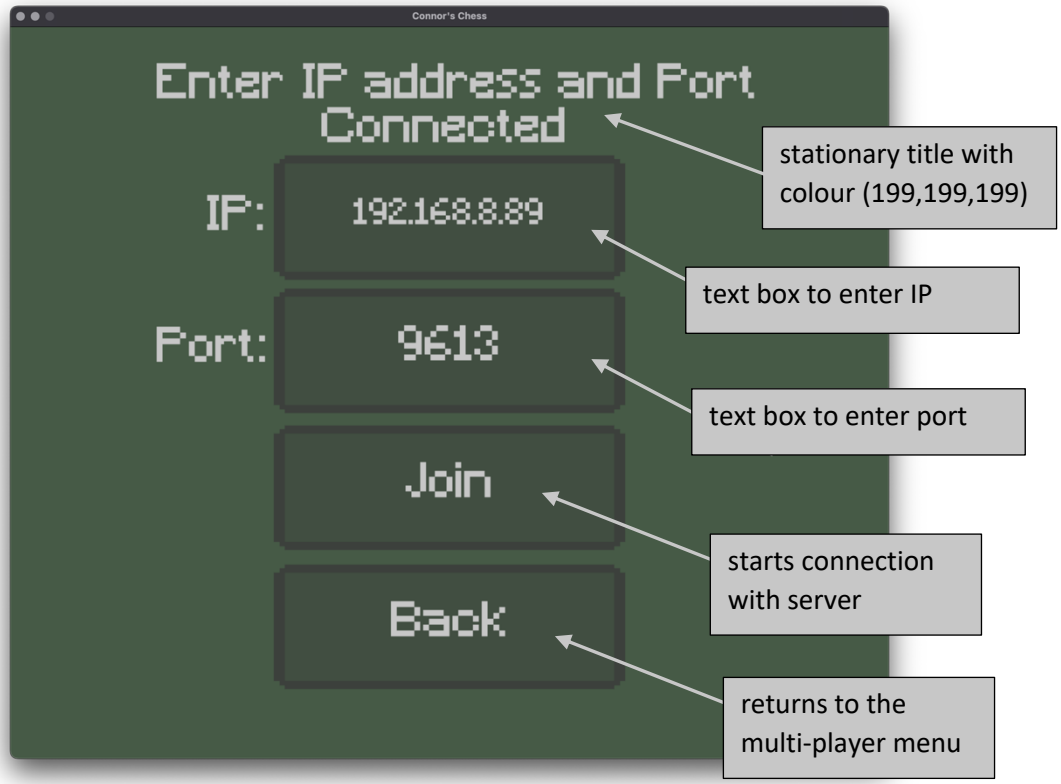
2.2.2.9.1 New game menu waiting for connection



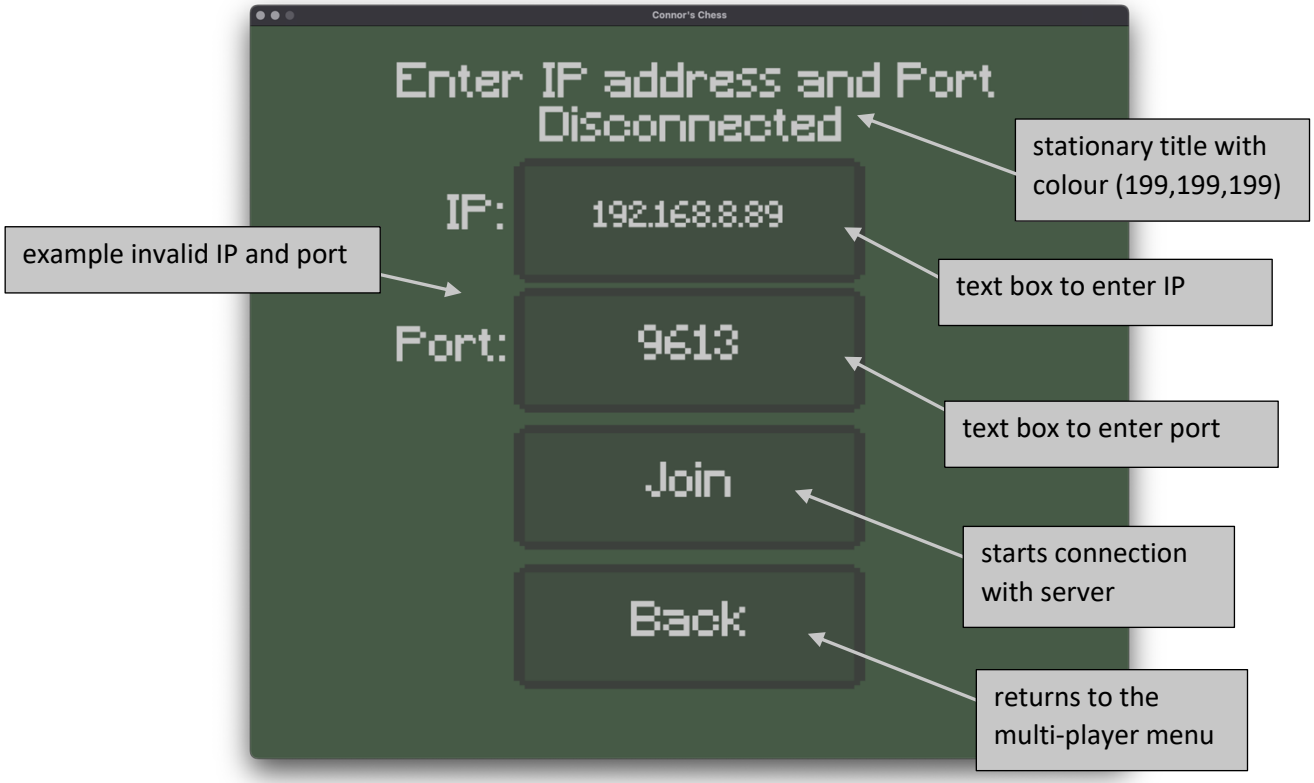
2.2.2.9.2 New game menu connected



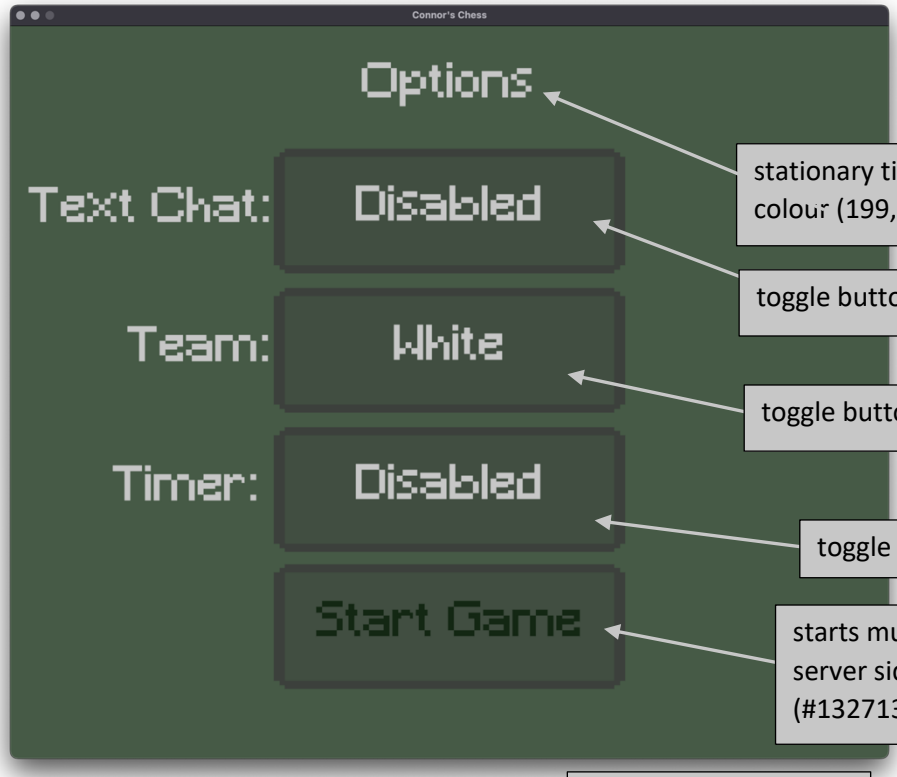
2.2.2.10.1 Join game menu - connected



2.2.2.10.2 Join game menu - disconnected



2.2.2.11 Multi-player settings menu



stationary title with colour (199,199,199)

toggle button for text chat

toggle button to change team

toggle button for timer

starts multi-player game on server side using text colour (#132713)

2.2.2.12.1 Multi-player menu with timer and text chat enabled



timer displayed in white when it is not this players turn

displays pieces that have been taken by this player with background colour (#132713)

opens the multi-player options menu

allows player to offer a draw where, in this example, this player has been offered a draw and has 3 turns to accept before it disappears

piece increases in size when picked up

resign button allows player to give up and admit defeat

black circles show legal moves for the piece being picked up

text chat allows players to send each other messages if enabled

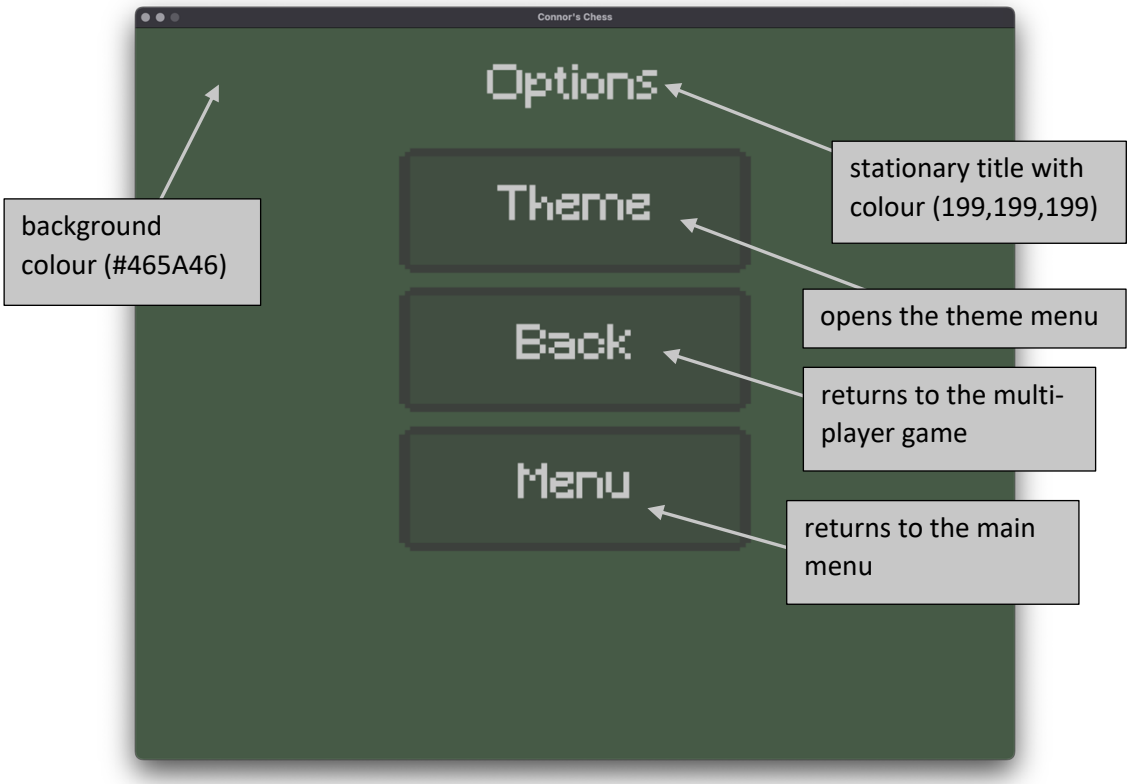
displays pieces that have been taken by this player

timer text displayed in red (199,55,55) on the side of the player whose turn it is

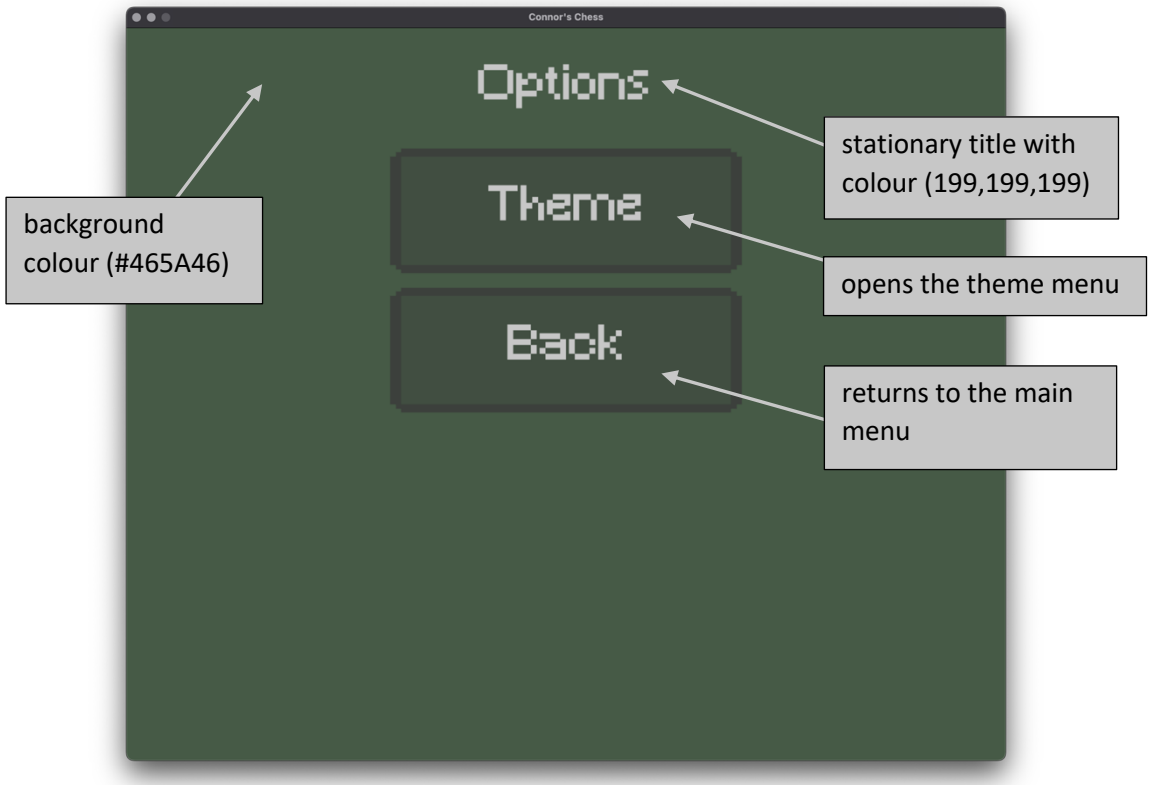
2.2.2.12.2 Multi-player with timer and text chat disabled



2.2.2.12.3 Multi-player options menu

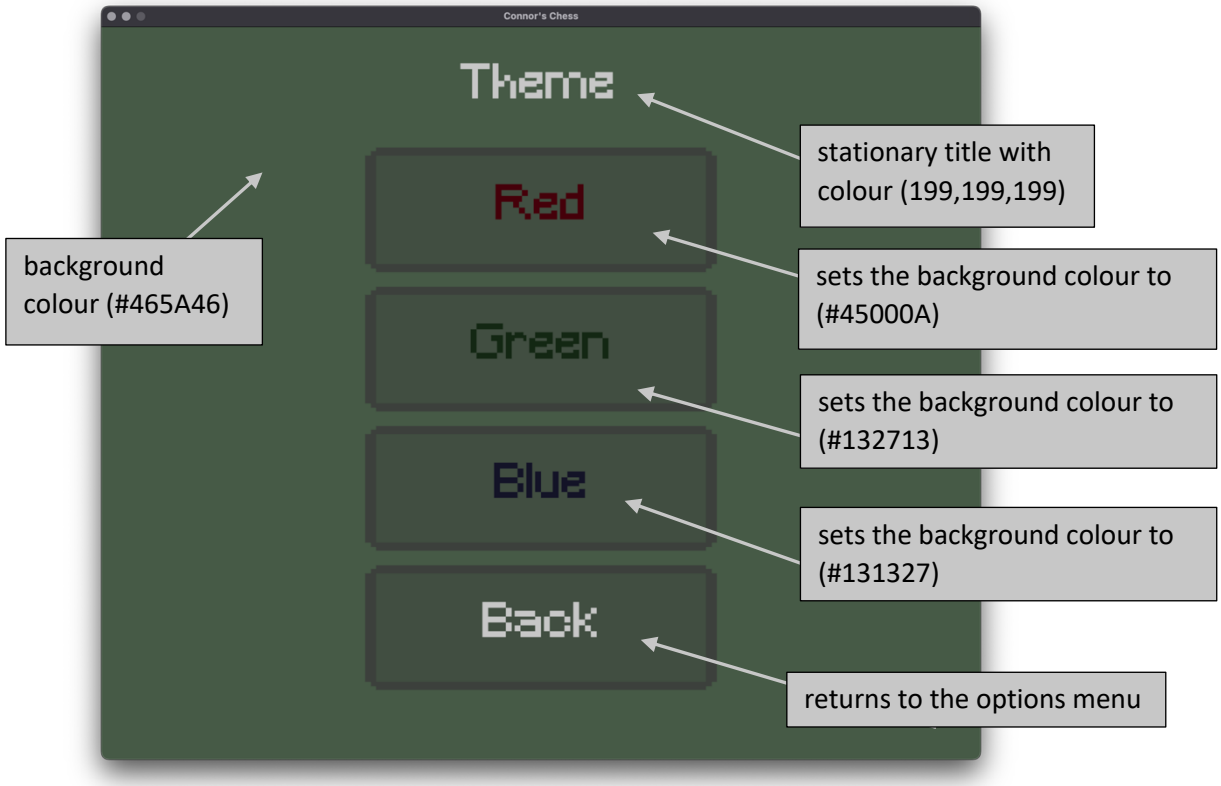


2.2.2.13 Options menu



2.2.2.14.1 Theme menu

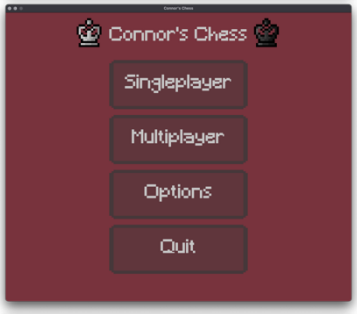
the theme set here, is consistent throughout the program



2.2.2.14.2 Main menu in all themes



Main menu in (#131327)



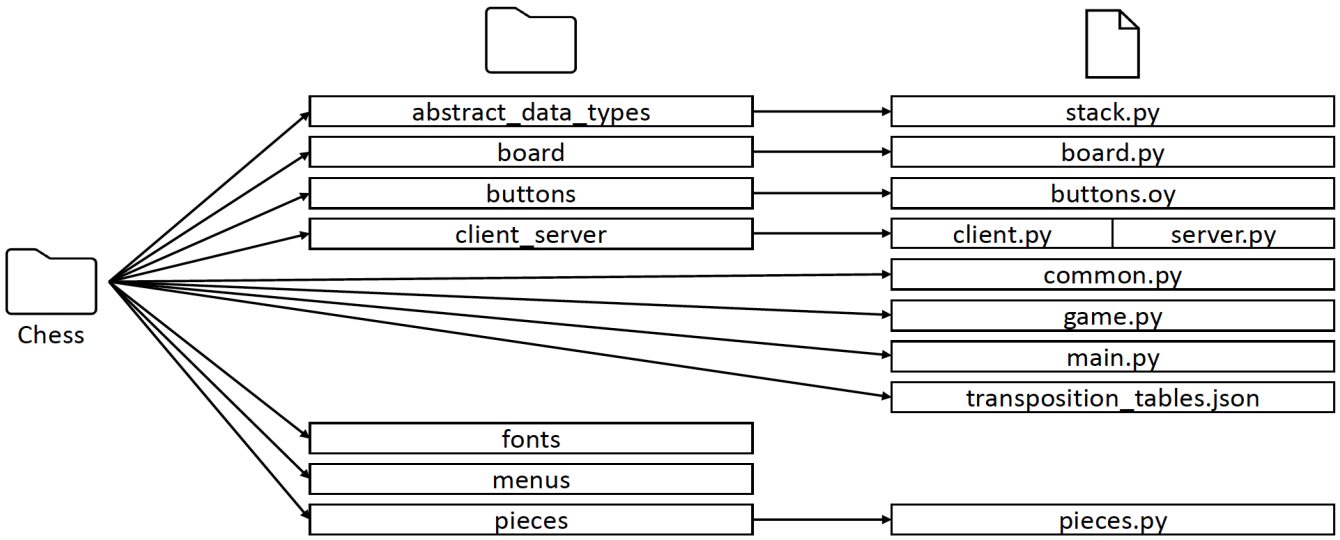
Main menu in (#45000A)



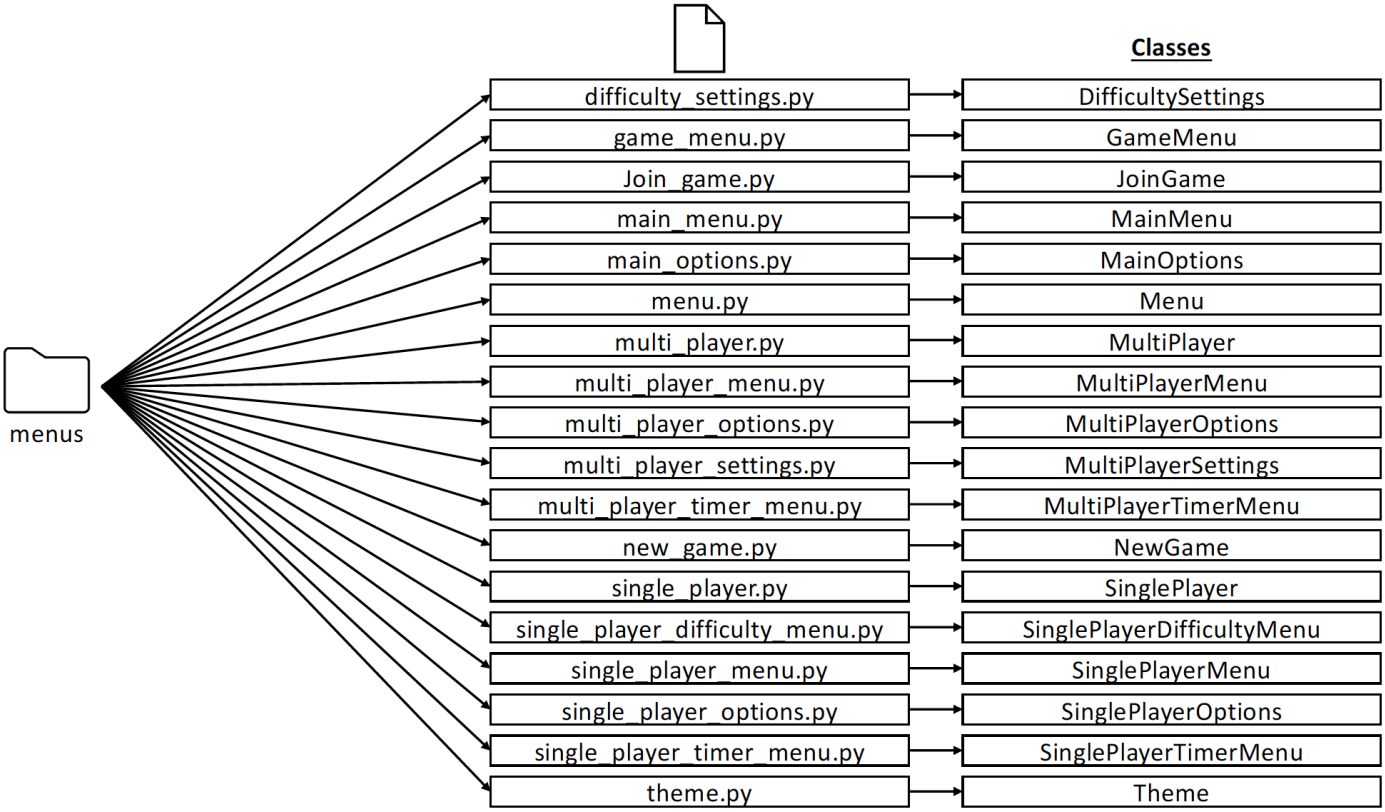
Main menu in (#132713)

2.3 File Structure

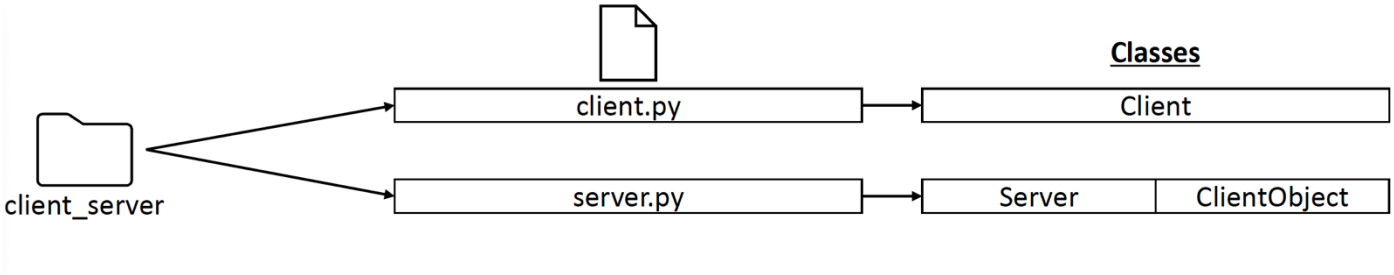
2.3.1 top-level root structure



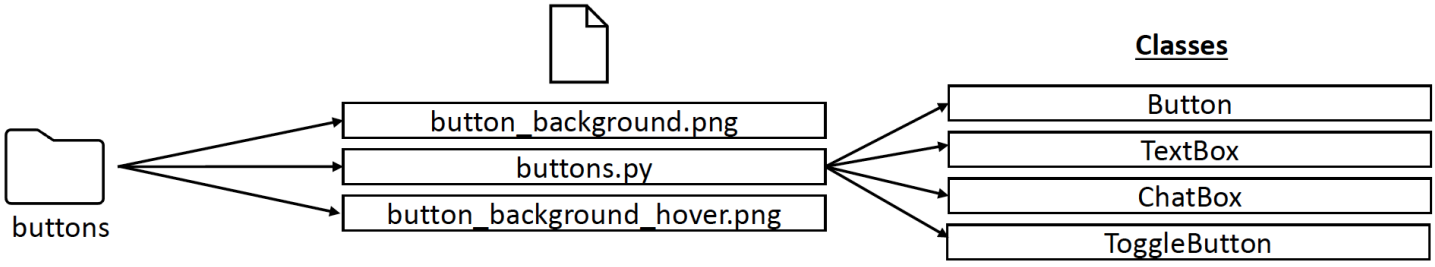
2.3.2 menus



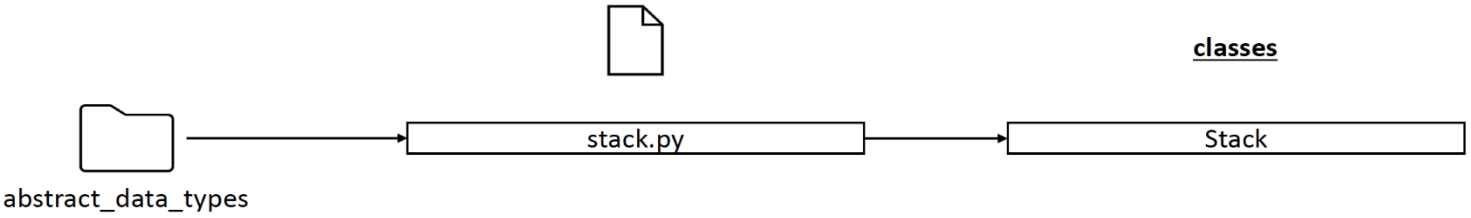
2.3.3 client-server



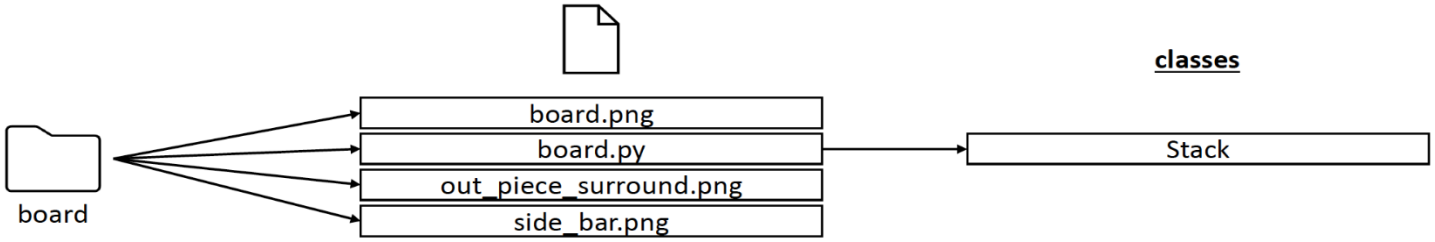
2.3.4 buttons



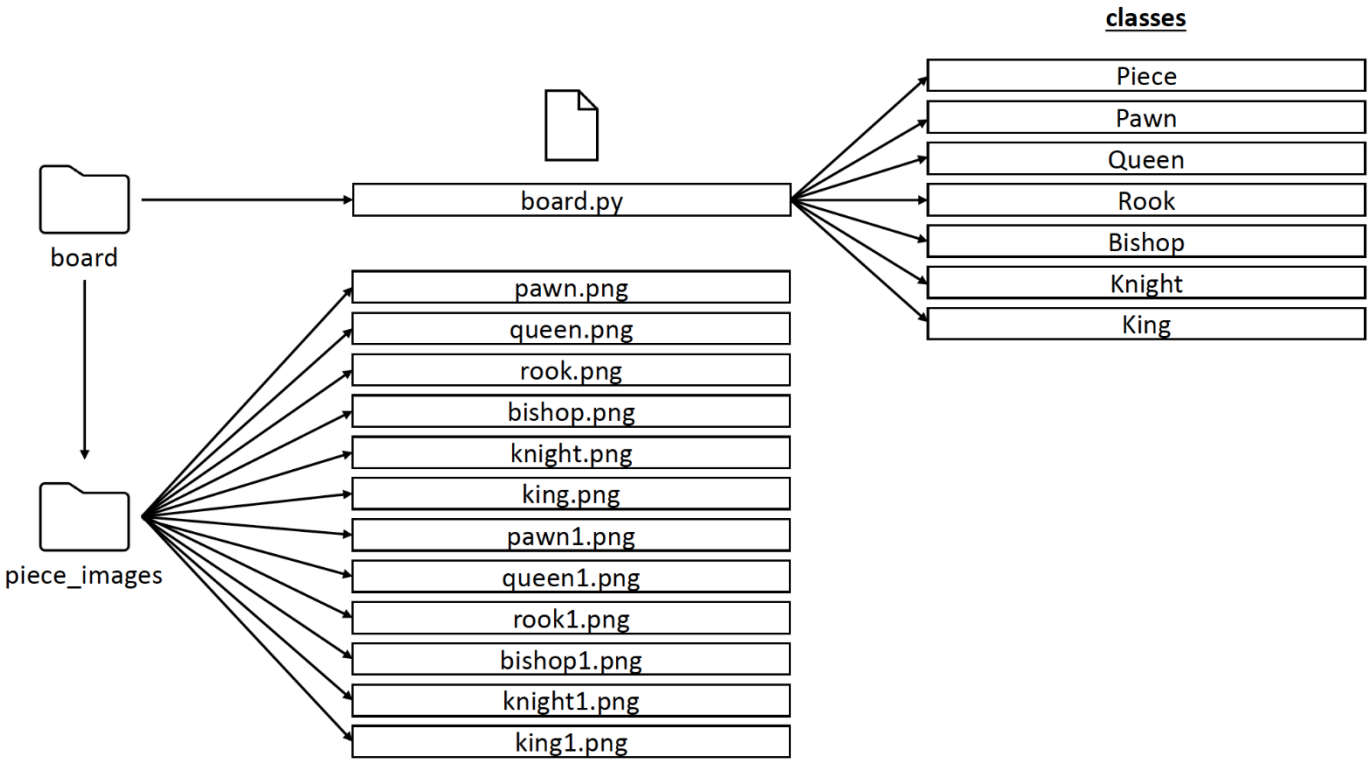
2.3.5 abstract data types



2.3.6 board



2.3.7 pieces



2.4 Internal data storage

Classes

Piece class – allows instantiation of pieces		
Attribute	Data type	Purpose
team	int	holds the team of the piece (1,0)
image	pygame.Surface	holds image for piece
type	str	holds the piece type e.g. (“Q” for queen)
value	int	holds the point value of the piece
pos	list	holds the current drawn position of the piece
pickup	bool	holds whether the piece is being picked up or not
placed_pos	list	holds the current placed position
start_pos	list	holds the original placed pos of the piece
Method	Purpose	
check_pickup()	checks if the piece is being picked up	
do_move()	makes the move on the board	
draw_piece()	draws the piece on the given surface	
calculate_nearest_square()	finds the closest square to the mouse position which is a legal move for the piece	

Child piece classes

These classes hold the data required to create an instance of each piece, with each inheriting attributes and methods from the Piece class

Class name	Purpose
Queen	creates instance of a queen piece
Pawn	creates instance of a pawn piece
Knight	creates instance of a knight piece
Bishop	creates instance of a bishop piece
King	creates instance of a king piece
Rook	creates instance of a rook piece

Stack class – allows instantiation of a stack		
Attribute	Data type	Purpose
maxlen	int	sets the maximum length of the stack
stack	list	acts as the main stack array
top	int	holds the position of the top of the stack
Method	Purpose	
is_empty()	checks if the stack is empty	
is_full()	checks if the stack is full	
push()	adds a new item to the top of the stack	
pop()	removes the top item from the stack and returns it	
peek()	returns the top item of the stack	
display_stack()	returns the stack as a list, so that it can be processed	

Game class – allows instantiation of the game		
Attribute	Data type	Purpose
dt	float	the number of seconds since the last frame was rendered
menu_stack	object	holds a stack of all currently used menus
offset	int	holds the main offset for the x axis
offset	int	holds the main offset for the y axis
mult	int	holds the main multiplier for the GUI
display_size	list	holds the display size in a list
window	pygame.Surface	holds the main window surface
running	bool	holds a Boolean that determines if the game is running
background_colour	str	holds the main background colour in hex
backboard_colour	str	holds the main backboard colour in hex
clock	object	holds the pygame clock object
pickup	list	holds a list containing pieces that are currently being picked up
transpos_tables	dict	holds a dictionary of relating the position to the best move in the transposition tables
Method	Purpose	
run()	holds the main game loop	
calculate_theme()	calculates the lighter background colour	

Button class – allows instantiation of the button		
Attribute	Data type	Purpose
pixel_font	pygame font	holds the font for the text on top of the button
img	pygame.Surface	holds the button image
img_hovered	pygame.Surface	holds the image for the button when a mouse is being hovered
surface	pygame.Surface	holds the surface which everything is drawn onto
menu	object	holds the reference to the menu that the button was called in
text	str	holds the text displayed on the button
text_surface	pygame.Surface	holds a surface which has the text drawn on it
callback	object	holds the function to be called back when the button is pressed
pressed	bool	holds a boolean which determines whether the mouse is being pressed
size	tuple	holds the size in the x and y direction of the button
pos	list	holds the x and y coordinates of the button
mouse_hovered	bool	holds a Boolean which determines whether the mouse is over the button
button_image	pygame.Surface	holds the overwrite image if entered
Method	Purpose	
check_collide()	checks various states of the button and mouse collision setting 'self.pressed' accordingly	
update()	draws the button to the display and calls 'check_collide'	

TextBox class – allows instantiation of the text box		
Attribute	Data type	Purpose
pixel_font	pygame font	holds the font for the text on top of the button
img	pygame.Surface	holds the button image
img_hovered	pygame.Surface	holds the image for the button when a mouse is being hovered
surface	pygame.Surface	holds the surface which everything is drawn onto
menu	object	holds the reference to the menu that the button was called in
text	str	holds the text displayed on the button
text_surface	pygame.Surface	holds a surface which has the text drawn on it
callback	object	holds the function to be called back when the button is pressed
pressed	bool	holds a boolean which determines whether the mouse is being pressed
size	tuple	holds the size in the x and y direction of the button
pos	list	holds the x and y coordinates of the button
mouse_hovered	bool	holds a Boolean which determines whether the mouse is over the button
button_image	pygame.Surface	holds the overwrite image if entered
max_len	int	holds the maximum length for the text to be entered

Method	Purpose
check_collide()	checks various states of the button and mouse collision setting 'self.pressed' accordingly
update()	draws the button to the display and calls 'check_collide'

ChatBox class : inherits from TextBox – allows instantiation of the text chat box		
Attribute	Data type	Purpose
pixel_font	pygame font	holds the font for the text on top of the button
img	pygame.Surface	holds the button image
img_hovered	pygame.Surface	holds the image for the button when a mouse is being hovered
surface	pygame.Surface	holds the surface which everything is drawn onto
menu	object	holds the reference to the menu that the button was called in
text	str	holds the text displayed on the button
text_surface	pygame.Surface	holds a surface which has the text drawn on it
callback	object	holds the function to be called back when the button is pressed
pressed	bool	holds a boolean which determines whether the mouse is being pressed
size	tuple	holds the size in the x and y direction of the button
pos	list	holds the x and y coordinates of the button
mouse_hovered	bool	holds a Boolean which determines whether the mouse is over the button
button_image	pygame.Surface	holds the overwrite image if entered
max_len	int	holds the maximum length for the text to be entered
send	bool	defines whether the enter button has been pressed
Method	Purpose	
check_collide()	checks various states of the button and mouse collision setting 'self.pressed' accordingly	
update()	draws the button to the display and calls 'check_collide'	

ToggleButton class : inherits from Button – allows instantiation of the toggle button		
Attribute	Data type	Purpose
pixel_font	pygame font	holds the font for the text on top of the button
img	pygame.Surface	holds the button image
img_hovered	pygame.Surface	holds the image for the button when a mouse is being hovered
surface	pygame.Surface	holds the surface which everything is drawn onto
menu	object	holds the reference to the menu that the button was called in
text	str	holds the text displayed on the button
text_surface	pygame.Surface	holds a surface which has the text drawn on it
callback	object	holds the function to be called back when the button is pressed
pressed	bool	holds a boolean which determines whether the mouse is being pressed
size	tuple	holds the size in the x and y direction of the button
pos	list	holds the x and y coordinates of the button
mouse_hovered	bool	holds a Boolean which determines whether the mouse is over the button
button_image	pygame.Surface	holds the overwrite image if entered
on_text	str	holds the text that is displayed when the button is pressed
off_text	str	holds the text that is displayed when the button isn't being pressed
on_colour	tuple	holds the colour of the text when the button is pressed
off_colour	tuple	holds the colour of the text when the button isn't being pressed
count	int	acts as a cooldown for the button, so that it stays pressed when the mouse button is held
Method	Purpose	
check_collide()	checks various states of the button and mouse collision setting 'self.pressed' accordingly	
update()	draws the button to the display and calls 'check_collide'	

Board class – allows instantiation of the board		
Attribute	Data type	Purpose
player1_out	list	holds the pieces that have been taken by player 2
player2_out	list	holds the pieces that have been taken by player 1
teamdict	dict	holds the score for both players in a dictionary
game	object	holds a reference to the main game menu
board	dict	holds the pieces and positions of the pieces in a dictionary
Method	Purpose	
__deepcopy__	overrides the default deepcopy method to not copy the game attribute	

Client class – allows instantiation of a client		
Attribute	Data type	Purpose
data_queue	list	acts as a queue to store data ready to be processed
data_received	list	holds the data that has been received
data_sent	list	holds the data that has previously been sent
s	object	stores the connection object between the server and client
ip	str	holds the local ip address of the server to be connected to
port	int	holds the port of the connection s between the server and client
connected	bool	holds a bool referring to the current state of the connection
main_game	object	holds a reference to an instantiation of the Game class
Method	Purpose	
join_game	establishes the connection between itself and the server and runs the connection loop	

Server class – allows instantiation of a server		
Attribute	Data type	Purpose
port	int	holds the port through which the connection is established
_ip	str	holds the local ip address of the server
_clients	list	holds a list of references to the clients connected to the server
Method	Purpose	
get_ip	makes a connection to 8.8.8.8 and finds its own local IP address	
listen_connect	listens for a connection	
broadcast_data	sends data given to all clients in "_clients"	
get_clients	returns list of clients	
del_clients	allows removal of given client from the list of clients	

ClientObject class – allows instantiation of a client object		
Attribute	Data type	Purpose
p	object	stores the connection object between the server and client
data_queue	list	holds the data ready to be processed
data_received	list	holds the data that has been received
data_sent	list	holds list of data that has been sent
received	bool	defines whether data has been received by client
Method	Purpose	
send_data	runs the connection loop between itself and the client	

Menu class – allows instantiation of a menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	

Theme class : inherits from Menu – allows instantiation of the theme menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
set_theme	changes the background and backboard colour to the given theme colour	

SinglePlayerTimerMenu class : inherits from Menu – allows instantiation of the single player timer menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
timer	bool	defines whether the timer is being used
team	bool	defines what team the player is
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
single_player_difficulty_menu	creates an instance of the SinglePlayerDifficultyMenu class in the menu stack	

SinglePlayerOptions class : inherits from Menu – allows instantiation of the single player options menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
game	object	holds a reference to the instantiated SinglePlayer object
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
difficulty_settings_callback	creates an instance of the DifficultySettings class in the menu stack	
game_quit	pops off the menu stack and returns to the main menu	
theme_settings	creates an instance of the Theme class in the menu stack	

MultiPlayerMenu class : inherits from Menu – allows instantiation of the multi-player menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
join_game	creates an instance of the JoinGame class in the menu stack	
new_game	creates an instance of the NewGame class in the menu stack	

MainMenu class : inherits from Menu – allows instantiation of the main menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
logo_count	float	holds a count that increases each tick
title	float	holds the sin of “logo_count” to make the title move up and down
logo_king	pygame.Surface	holds the pygame surface for the black king image
logo_king1	pygame.Surface	holds the pygame surface for the white king image
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
game_end	stops the main game loop	
main_options	creates an instance of the MainOptions class in the menu stack	
multi_player_menu	creates an instance of the MultiPlayerMenu class in the menu stack	
single_player_menu	creates an instance of the SinglePlayerMenu class in the menu stack	

MultiPlayerTimerMenu class : inherits from Menu – allows instantiation of the multi-player timer menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
timer	bool	defines whether the timer is enabled on the multi-player game
team	bool	defines what team this player is on
chat	bool	defines whether the chat is to be used in the multi-player game
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
start_multi_player_game	creates an instance of the MultiPlayer class in the menu stack	

MultiPlayerSettings class : inherits from Menu – allows instantiation of the multi-player settings menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
timer	bool	defines whether the timer is enabled on the multi-player game
team	bool	defines what team this player is on
chat	bool	defines whether the chat is to be used in the multi-player game
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
start_multi_player_game	creates an instance of the MultiPlayer or MutliPlayerTimerMenu class in the menu stack defined by the “timer” attribute	

SinglePlayerDifficultyMenu class : inherits from Menu – allows instantiation of the single-player timer menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
timer	bool	defines whether the timer is enabled on the multi-player game
team	bool	defines what team this player is on
time_duration	int	defines the game time duration in seconds
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
single_player_callback	creates an instance of the SinglePlayer class in the menu stack	

MultiPlayerOptions class : inherits from Menu – allows instantiation of the multi-player options menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
multi_player_quit	closes both client server connections and pops off the menu stack	
theme_settings	creates an instance of the Theme class in the menu stack	

SinglePlayerMenu class : inherits from Menu – allows instantiation of the single-player menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
timer	bool	defines whether the timer is enabled
team	bool	defines what team this player is on
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
single_player_timer_menu	creates an instance of the SinglePlayer or SinglePlayerTimerMenu class in the menu stack defined by the “timer” attribute	

MainOptions class : inherits from Menu – allows instantiation of the main options menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
theme_settings	creates an instance of the Theme class in the menu stack	

DifficultySettings class : inherits from Menu – allows instantiation of the difficulty settings menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
game	object	holds a reference to the instantiated SinglePlayer object
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
difficulty_change	changes the difficulty to the difficulty given	

NewGame class : inherits from Menu – allows instantiation of the new game menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
thread	threading.Thread	holds the thread in which the server runs
ip	str	stores the local ip address of the server
port	int	stores the port in which the connection is established
main_client	object	stores the instantiation of the client on the server side
client_thread	threading.Thread	holds the thread in which the client runs
connection_text	list	holds a list of connection text options to animate “waiting for connection”
conncetion_count	int	the count used for animating the “connection_text”
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
multi_player_settings	creates an instance of the MultiPlayerSettings class in the menu stack	

JoinGame class : inherits from Menu – allows instantiation of the join game menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
client	object	stores an instantiation of the client object
ip	str	stores the ip address entered by the client into the text box
port	int	stores the port entered by the client into the text box
try_join	bool	determines whether the client has tried to join the server connection
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
client_join	instantiates the client object and starts the connection between itself and the server	
exit_connection	closes the client server connection	

GameMenu class : inherits from Menu – allows instantiation of the game menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
timer	bool	defines whether the timer is enabled for the game
time_duration	int	defines the game duration in seconds
resign	list	holds a list of bools defining if either player has resigned
stalemate_end	bool	determines if the game has ended in stalemate
game_draw	bool	determines if the game has ended in a draw
turn	int	determines which players turn it is
check_mate_array	list	holds a list of bools which determine if either player has won by checkmate
best_move	list	defines if the minimax has evaluated the position and holds the best move
HUMAN	int	used to define what team the human is playing as
COMP	int	used to define what team the minimax is playing as
player1	int	used to define what team player 1 is on
player2	int	used to define what team player 2 is on
end_text	pygame.Font	stores the font and size used for the end screen text
time_text	pygame.Font	stores the font and size used for the timer text
difficulty	int	holds the difficulty that the single player game is currently set to
transpos_tables	dict	holds a dictionary of all previously played positions and the best move in each
menu_background	pygame.Surface	stores the image used for the sidebar
board_img	pygame.Surface	stores the image used for the backboard
out_piece_surround	pygame.Surface	stores the image used as a surround for both sets of out pieces
board_game	object	holds the instance of the board class on which the game is played
end_surface	pygame.Surface	stores the surface on which the end screen is drawn
glow_surface	pygame.Surface	holds the surface on which the glow of each piece Is drawn
allowed_moves	pygame.Surface	holds the surface on which the allowed moves for each piece are drawn
text_surface	pygame.Surface	holds the surface on which most text is drawn e.g. the scores of each player
offsetx	int	stores the main board offset along the x axis
offsety	int	stores the main board offset along the y axis
mult	int	stores the main board offset multiplier
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
render_end_game	draws the “end_surface” for all game ending instances	
time_format_translator	reformats the given time from seconds to minutes in the layout: “00:00”	
minimax_thread	calls the minimax function in a thread with calculated difficulty and transposition tables	
draw_out_pieces	draws the pieces that have been taken on both sides	
move_types	returns an array of all attacking, quiet and checking moves	
quiet_moves	returns quiet moves	
attacking_moves	returns attacking moves	
checking_moves	returns checking moves	
check_move	returns all legal moves for given piece	
legal_moves	checks if moves are legal, and doesn’t cause itself to be in check	
check_pickup	checks if a piece is being picked up	
draw_glow	draws the glow behind each piece when the mouse is hovered over them	
minimax	calculates the best move in the current position based on the depth and the balance of points for each player after each turn	
quiescent_end	dictates the base case for the quiescent search	
evaluate_state	returns the static evaluation for the current state in favour of the maximising player	
game_over	checks if the game is over	
quiescent_search	performs a secondary search after the minimax search, to extend the depth until a quiet position	
stalemate	checks if there is a stalemate	
check_check	checks if the given player is in check	
checkmate	checks if the given player is in checkmate	
multi_player_quit	pops off the menus stack to the main menu and closes the client server connection	
process_data	processes the given data received from the client server connection and executes the command	
move_translator	translates the given move for the opponent	
game_quit	pops off the menu stack to the main menu	

Main GameMenu methods

check_move()		
Variables	Data type	Purpose
piece_type	str	holds the piece type
state	dict	holds a dictionary mapping co-ordinates to piece instances
current_pos	tuple	holds the current placed position of the piece
start_pos	list	holds start position of the piece
team	int	holds team of the piece (1,0)
check	int	used as bool to do determine whether the moves should be checked to see if they are legal
moves	list	holds pieces possible moves
pos	list	holds vectors for positions around current_pos
used_pos	list	holds temporary position of piece to find allowed moves
i	tuple	holds a term in pos during the iteration of pos

check_check()		
Variables	Data type	Purpose
state	dict	holds a dictionary mapping co-ordinates to piece instances
team	int	holds team value (1,0)
king_location	tuple	holds position of the king given the team
i	object	holds piece object through the iteration of state.values()

legal_moves()		
Variables	Data type	Purpose
moves	list	holds moves to be processed
placed_pos	tuple	holds the current position of the piece which's moves are being checked
team	int	holds the team of the piece which's moves are being checked
state	dict	holds a dictionary which maps the piece's position to their object
moves_copy	list	holds a copy of moves so that some positions can be deleted without affecting the iteration
i	list	holds a move during the iteration of moves

check_pickup()		
Variables	Data type	Purpose
i	object	holds object of piece in board

draw_glow()		
Variables	Data type	Purpose
i	object	holds a piece object in the board values
mouse_pos	tuple	holds the mouse position altered for display offset
mult	int	used to turn pixel co-ordinates into grid co-ordinates
offsetx	int	used to offset the grid on the x axis to fit to the board
offsety	int	used to offset the grid on the y axis to fit to the board

stalemate()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
team	int	holds the team of the piece which's moves are being checked
i	object	holds a piece object in the board values

checkmate()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
team	int	holds the team of the piece which's moves are being checked
i	object	holds a piece object in the board values

game_over()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object

evaluate_state()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
points	int	holds the point value of pieces taken for the position to be evaluated

draw_out_pieces()		
Variables	Data type	Purpose
out_pieces	list	holds the objects of pieces that have been taken
surface	pygame.surface	holds the pygame surface that the pieces should be drawn to
white	list	holds list of piece objects where the team is white
black	list	holds list of piece objects where the team is black
i	object	holds piece objects during the iteration of out_pieces
x	object	holds the piece object during the iteration of white and black
i	int	holds the position of piece objects in during the iteration of white and black

minimax()		
Variables	Data type	Purpose
temp_state	dict	is a dictionary which maps the piece's position to their object
temp_state1	dict	is a dictionary which maps the piece's position to their object
depth	int	holds the depth of the minimax search
player	int	denotes whether the player is maximising or minimising
alpha	int	holds the minimum score for the maximising player
beta	int	holds the maximum score for the minimising player
points	int	holds the current point balance for the branch
temp_points	int	holds the temporary point balance for the current branch
transpos	list	holds 2D array representation of temp_state to refer to the transposition table
score	list	holds the position of the piece, where it should move, and the value of the move for the current best move
shuffled_pieces	list	holds a list of the piece objects on the board in a random order so if moves are equal in score then a random move is chosen between them

minimax_thread()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
depth	int	holds the depth of the minimax search
player	int	denotes whether the player is maximising or minimising
alpha	int	holds the minimum score for the maximising player
beta	int	holds the maximum score for the minimising player
transpos	list	holds 2D array representation of temp_state to refer to the transposition table

move_types()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
team	int	defines which team the moves are being found for
temp_state	dict	holds a copy of state to determine whether moves are checking moves without affecting state
attack	list	a list holding all attacking moves
quiet	list	a list holding all moves which don't attack
check	list	a list holding checking moves

quiet_moves()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
team	int	defines which team the moves are being found for

attacking_moves()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
team	int	defines which team the moves are being found for

checking_moves()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
team	int	defines which team the moves are being found for

quiescent_end()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
depth	int	holds the current depth of the quiescent search algorithm

quiescent_search()		
Variables	Data type	Purpose
state	dict	is a dictionary which maps the piece's position to their object
temp_state	dict	is a dictionary which maps the piece's position to their object
depth	int	holds the depth of the minimax search
alpha	int	holds the minimum score for the maximising player
beta	int	holds the maximum score for the minimising player
points	int	holds the current point balance for the branch
temp_points	int	holds the temporary point balance for the current branch
team	int	defines whether it is the minimising or maximising players turn to search
score	int	holds the current move's value

MultiPlayer and SinglePlayer classes

MultiPlayer class : inherits from GameMenu – allows instantiation of the multi-player game menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
timer	bool	defines whether the timer is enabled for the game
time_duration	int	defines the game duration in seconds
resign	list	holds a list of bools defining if either player has resigned
stalemate_end	bool	determines if the game has ended in stalemate
game_draw	bool	determines if the game has ended in a draw
turn	int	determines which players turn it is
check_mate_array	list	holds a list of bools which determine if either player has won by checkmate
best_move	list	defines if the minimax has evaluated the position and holds the best move
HUMAN	int	used to define what team the human is playing as
COMP	int	used to define what team the minimax is playing as
player1	int	used to define what team player 1 is on
player2	int	used to define what team player 2 is on
end_text	pygame.Font	stores the font and size used for the end screen text
time_text	pygame.Font	stores the font and size used for the timer text
difficulty	int	holds the difficulty that the single player game is currently set to
transpos_tables	dict	holds a dictionary of all previously played positions and the best move in each
menu_background	pygame.Surface	stores the image used for the sidebar
board_img	pygame.Surface	stores the image used for the backboard
out_piece_surround	pygame.Surface	stores the image used as a surround for both sets of out pieces
board_game	object	holds the instance of the board class on which the game is played
end_surface	pygame.Surface	stores the surface on which the end screen is drawn
glow_surface	pygame.Surface	holds the surface on which the glow of each piece Is drawn
allowed_moves	pygame.Surface	holds the surface on which the allowed moves for each piece are drawn
text_surface	pygame.Surface	holds the surface on which most text is drawn e.g. the scores of each player
offsetx	int	stores the main board offset along the x axis
offsety	int	stores the main board offset along the y axis
mult	int	stores the main board offset multiplier
chat	list	holds a list of the messages in the text chat
draw_count	int	counts the number of turns after a draw is offered
draw_attempt	int	counts the number of attempted draws so an offer can be sent multiple times
text_chat	bool	determines whether the text chat is enabled
text_chat_image	pygame.Surface	holds the image for the text chat background
Method	Purpose	
render_menu	draws to the screen and processes everything necessary for this menu	
render_end_game	draws the “end_surface” for all game ending instances	
time_format_translator	reformats the given time from seconds to minutes in the layout: “00:00”	
minimax_thread	calls the minimax function in a thread with calculated difficulty and transposition tables	
draw_out_pieces	draws the pieces that have been taken on both sides	
move_types	returns an array of all attacking, quiet and checking moves	
quiet_moves	returns quiet moves	
attacking_moves	returns attacking moves	
checking_moves	returns checking moves	
check_move	returns all legal moves for given piece	

legal_moves	checks if moves are legal, and doesn't cause itself to be in check
check_pickup	checks if a piece is being picked up
draw_glow	draws the glow behind each piece when the mouse is hovered over them
minimax	calculates the best move in the current position based on the depth and the balance of points for each player after each turn
quiescent_end	dictates the base case for the quiescent search
evaluate_state	returns the static evaluation for the current state in favour of the maximising player
game_over	checks if the game is over
quiescent_search	performs a secondary search after the minimax search, to extend the depth until a quiet position
stalemate	checks if there is a stalemate
check_check	checks if the given player is in check
checkmate	checks if the given player is in checkmate
multi_player_quit	pops off the menus stack to the main menu and closes the client server connection
process_data	processes the given data received from the client server connection and executes the command
move_translator	translates the given move for the opponent
game_quit	pops off the menu stack to the main menu
send_resign	sends a "resign" message to the opponent
send_draw	sends a "draw" message to the opponent

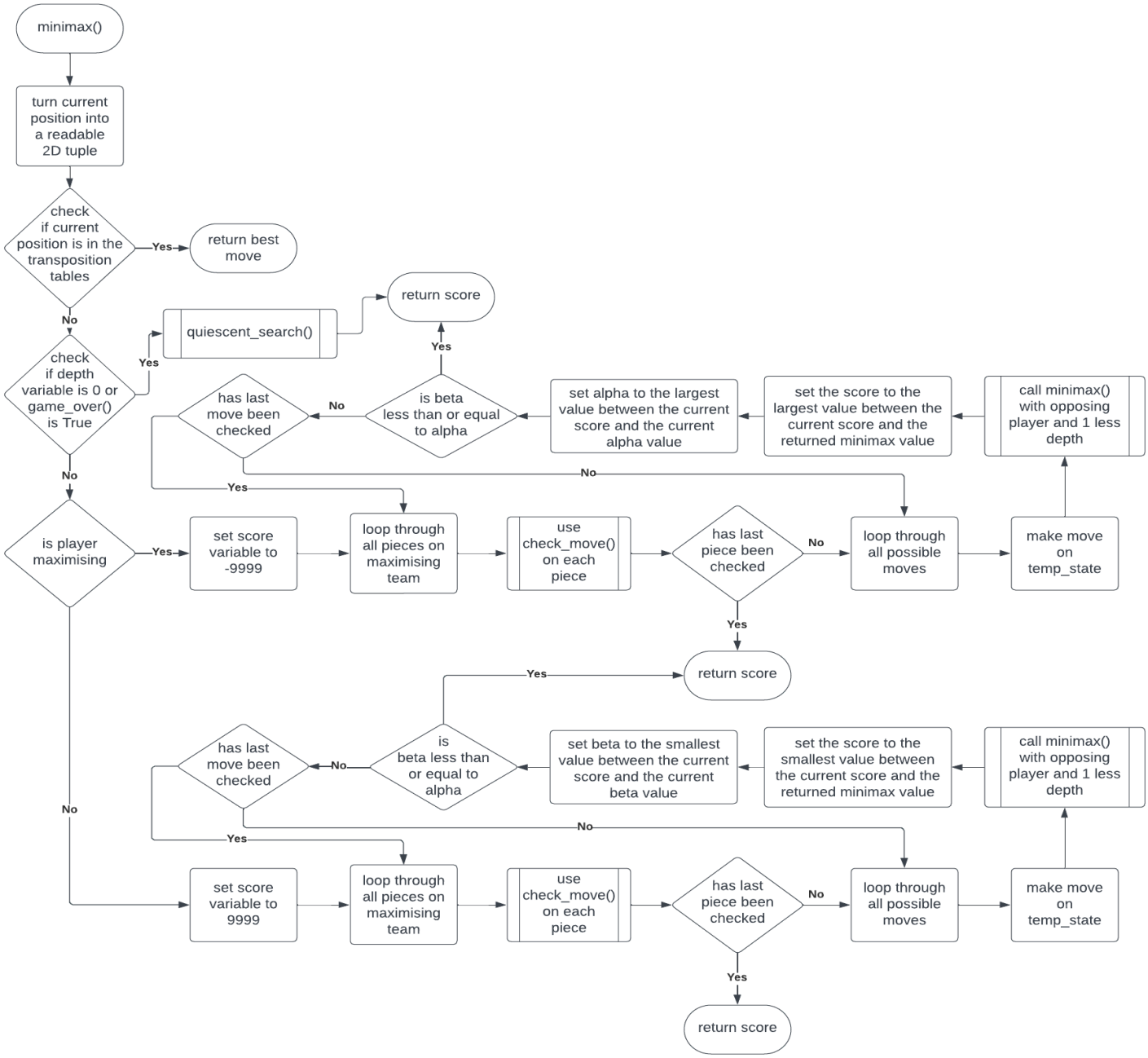
SinglePlayer class : inherits from GameMenu – allows instantiation of the single-player game menu		
Attribute	Data type	Purpose
size	list	holds the size of the menu displayed on the screen
colour	str	holds the hexadecimal code for the background colour
surface	pygame.Surface	holds its own surface object
buttons	list	holds a list of buttons
event	pygame.Event	holds the pygame event for keyboard inputs
main_game	object	holds a reference to an instantiation of the Game class
pixel_font	pygame.font	holds the default font used on all menus
timer	bool	defines whether the timer is enabled for the game
time_duration	int	defines the game duration in seconds
resign	list	holds a list of bools defining if either player has resigned
stalemate_end	bool	determines if the game has ended in stalemate
game_draw	bool	determines if the game has ended in a draw
turn	int	determines which players turn it is
check_mate_array	list	holds a list of bools which determine if either player has won by checkmate
best_move	list	defines if the minimax has evaluated the position and holds the best move
HUMAN	int	used to define what team the human is playing as
COMP	int	used to define what team the minimax is playing as
player1	int	used to define what team player 1 is on
player2	int	used to define what team player 2 is on
end_text	pygame.Font	stores the font and size used for the end screen text
time_text	pygame.Font	stores the font and size used for the timer text
difficulty	int	holds the difficulty that the single player game is currently set to
transpos_tables	dict	holds a dictionary of all previously played positions and the best move in each
menu_background	pygame.Surface	stores the image used for the sidebar
board_img	pygame.Surface	stores the image used for the backboard
out_piece_surround	pygame.Surface	stores the image used as a surround for both sets of out pieces
board_game	object	holds the instance of the board class on which the game is played
end_surface	pygame.Surface	stores the surface on which the end screen is drawn
glow_surface	pygame.Surface	holds the surface on which the glow of each piece Is drawn
allowed_moves	pygame.Surface	holds the surface on which the allowed moves for each piece are drawn

text_surface	pygame.Surface	holds the surface on which most text is drawn e.g. the scores of each player
offsetx	int	stores the main board offset along the x axis
offsety	int	stores the main board offset along the y axis
mult	int	stores the main board offset multiplier
game_stack	object	stores the stack which holds the a frame of the game after each turn
best_move	list	defines if the minimax has evaluated the position and holds the best move
difficulty	int	holds the difficulty that the single player game is currently set to
transpos_tables	dict	holds a dictionary of all previously played positions and the best move in each

Method	Purpose
render_menu	draws to the screen and processes everything necessary for this menu
render_end_game	draws the “end_surface” for all game ending instances
time_format_translator	reformats the given time from seconds to minutes in the layout: “00:00”
minimax_thread	calls the minimax function in a thread with calculated difficulty and transposition tables
draw_out_pieces	draws the pieces that have been taken on both sides
move_types	returns an array of all attacking, quiet and checking moves
quiet_moves	returns quiet moves
attacking_moves	returns attacking moves
checking_moves	returns checking moves
check_move	returns all legal moves for given piece
legal_moves	checks if moves are legal, and doesn’t cause itself to be in check
check_pickup	checks if a piece is being picked up
draw_glow	draws the glow behind each piece when the mouse is hovered over them
minimax	calculates the best move in the current position based on the depth and the balance of points for each player after each turn
quiescent_end	dictates the base case for the quiescent search
evaluate_state	returns the static evaluation for the current state in favour of the maximising player
game_over	checks if the game is over
quiescent_search	performs a secondary search after the minimax search, to extend the depth until a quiet position
stalemate	checks if there is a stalemate
check_check	checks if the given player is in check
checkmate	checks if the given player is in checkmate
multi_player_quit	pops off the menus stack to the main menu and closes the client server connection
process_data	processes the given data received from the client server connection and executes the command
move_translator	translates the given move for the opponent
game_quit	pops off the menu stack to the main menu
single_player_draw	checks whether the minimax is winning and if so, the draw is declined, if not, the draw is accepted
single_player_resign	used as a call-back to end the game in favour of the minimax
rewind_stack	sets the game board to the previous item in the game stack object to take back a turn

2.5 Main Algorithms - 2.5.1 Minimax

Below is a flow chart representing the recursive minimax algorithm and how it functions in my chess game.



This flowchart shows the procedural steps of the minimax algorithm and how it interacts with other parts of the game. When the function is called, it first checks whether the position is in the transposition table and if it has been seen before, if it has, the best move has already been calculated so is returned directly. If the position hasn't been seen before, then it checks whether the depth of the search is 0 (the end of the search has been reached) or game_over() is true (the game has ended by: checkmate, stalemate, draw or resignation). If either of these are true, it calls and returns the quiescent search function so that the position can be further evaluated rather than just taking the static positions game value. If the function continues, it checks whether the current player is maximising (is the computer). If it is, all pieces are looped through with each of their moves also being looped through, where each is played on a temporary board to understand the outcome of that move. After each move, the minimax is called again, where the returned value is compared with the current best score. Alpha-Beta pruning is then used to see whether the position will ever be reached, if it won't, the tree will be cut and the current score will be returned. This process goes the same for the minimising player with some parts of the system being reversed, such as: the initial score value, the comparison between the minimax returned value and the current score, and that beta is changed before Alpha-Beta pruning rather than alpha.

Below is a pseudocode implementation for my minimax algorithm.

```
SUBROUTINE minimax(
                temp_state,
                temp_state1,
                depth,
                player,
                alpha,
                beta,
                points,
                temp_points,
                )
        transpos ← tuple(tempstate)
        IF transpos IN transpos_tables THEN
                RETURN transpos_tables[transpos]
        ENDIF
        IF depth = 0 OR game_over() = TRUE THEN
                RETURN [
                        (-1, -1),
                        (-1, -1),
                        quiescent_search(
                                temp_state,
                                temp_state,
                                2,
                                -evaluate_state(temp_state),
                                evaluate_state(temp_state),
                                temp_points,
                                temp_points,
                                COMP,
                        ),
                ]
        ENDIF
        IF player = COMP THEN
                Score ← [(-1, -1), (-1, -1), -9999]
```

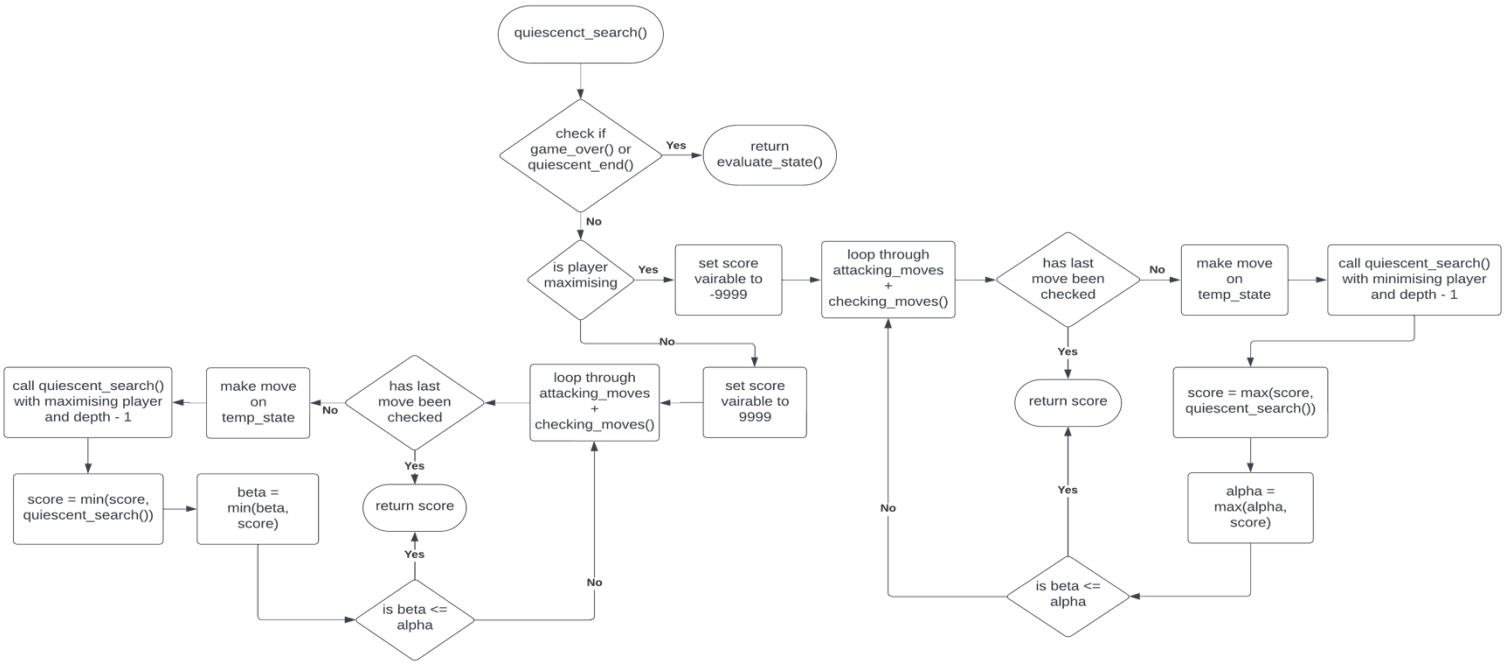
```
FOR piece IN temp_state
        IF piece.team = COMP THEN
                FOR move IN check_move(
                                Piece.type,
                                temp_state,
                                piece.pos,
                                piece.start_pos,
                                piece.team,
                                )
                        temp_points ← temp_points + temp_state1[move].value
                        temp_state1[move] ← piece
                        DEL temp_state1[piece.pos]
                        minimax ← minimax(
                                        temp_state1,
                                        temp_state1,
                                        depth - 1,
                                        NOT player,
                                        alpha,
                                        beta,
                                        temp_points,
                                        temp_points,
                                        )
                        IF minimax[2] > score[2] THEN
                                        score[0] ← piece.pos
                                        score[1] ← move
                                        score[2] ← minimax
                        ENDIF
                        temp_points←points
                        temp_state1←temp_state
                        alpha←max(alpha, score[2])
                        IF beta <= alpha THEN
                                RETURN score
                        ENDIF
```

```
                    ENDFOR
        ENDFOR
        RETURN score
ENDIF
IF player = HUMAN THEN
        Score ← [(-1, -1), (-1, -1), 9999]
        FOR piece IN temp_state
                IF piece.team = HUMAN THEN
                        FOR move IN check_move(
                                Piece.type,
                                temp_state,
                                piece.pos,
                                piece.start_pos,
                                piece.team,
                                )
                            temp_points ← temp_points - temp_state1[move].value
                            temp_state1[move] ← piece
                            DEL temp_state1[piece.pos]
                            minimax ← minimax(
                                        temp_state1,
                                        temp_state1,
                                        depth - 1,
                                        NOT player,
                                        alpha,
                                        beta,
                                        temp_points,
                                        temp_points,
                                        )
                            IF minimax[2] < score[2] THEN
                                        score[0] ← piece.pos
                                        score[1] ← move
                                        score[2] ← minimax
                            ENDIF
```

```
                    temp_points←points
                    temp_state1←temp_state
                    beta←min(beta, score[2])
                    IF beta <= alpha THEN
                        RETURN score
                    ENDIF
                ENDFOR
            ENDFOR
            RETURN score
        ENDIF
ENDSUBROUTINE
```

2.5.2 Quiescent search

Below is a flowchart outlining the process of a quiescent search and how it extends the search of the minimax for active positions



Similarly to the minimax, the quiescent search both creates and traverses a tree. When the function is called, it checks whether game_over() is true or if quiescent_end() is true, if either is, the static evaluation of the position from evaluate_state() is returned. It then checks if the player if maximising, in which case, it loops through attacking_moves() + checking_moves() as that is the fastest way to find a quiet position. After making each move on temp_state, it makes a recursive call of quiescent_search() with the opposite player and depth - 1. Score is then set to the maximum between the current score and the current alpha value before it checks if beta is less than or equal to alpha. If it is, the current score is returned, if not, it continues to the start of the loop. The minimising side of the function is largely the same, with a few changes: the initial score value, the comparison between the minimax returned value and the current score, and that beta is changed before Alpha-Beta pruning rather than alpha.

Below is pseudocode for this process.

```
SUBROUTINE quiescent_search(
                    state,
                    temp_state,
                    depth,
                    alpha,
                    beta,
                    points,
                    temp_points,
                    team
                    )
        IF game_over(state) OR quiescent_end(state, depth) THEN
                    RETURN evaluate_state(state, points)
        ENDIF
        IF team = COMP THEN
                    score ← -9999
                    FOR move IN attacking_moves(state, team) + checking_moves(state, team)
                                IF move IN temp_state THEN
                                            temp_points ← temp_points + temp_state[move[1]].value
                                ENDIF
                                temp_state[move[1]] ← temp_state[move[0]]
                                DEL temp_state[move[0]]
                                score ← max(quiescent_search(
                                                        temp_state,
                                                        temp_state,
                                                        depth - 1,
                                                        alpha,
                                                        beta,
                                                        temp_points,
                                                        temp_points,
                                                        NOT team
                                                        ), score)
                                temp_state ← state
```

```
                temp_points ← points
                alpha ← max(alpha, score)
                IF beta <= alpha THEN
                        RETURN score
                ENDIF
        ENDFOR
        RETURN score
ENDIF

IF team = HUMAN THEN
        score ← 9999
        FOR move IN attacking_moves(state, team) + checking_moves(state, team)
                IF move IN temp_state THEN
                        temp_points ← temp_points - temp_state[move[1]].value
                ENDIF
                temp_state[move[1]] ← temp_state[move[0]]
                DEL temp_state[move[0]]
                score ← min(quiescent_search(
                                        temp_state,
                                        temp_state,
                                        depth - 1,
                                        alpha,
                                        beta,
                                        temp_points,
                                        temp_points,
                                        NOT team
                                        ), score)
                temp_state ← state
                temp_points ← points
                beta ← min(beta, score)
                IF beta <= alpha THEN
                        RETURN score
                ENDIF
```

```
                ENDFOR
                RETURN score
        ENDIF
```

2.6 Client-Server

Below is a pseudocode implementation of my client-server model

2.6.1 Client Pseudocode

```
class Client
        public data_queue
        public data_received
        public data_sent
        public s
        public ip
        public port
        public connected
        public main_game
        public data_to_send
        public SUBROUTINE join_game()
                s.connect((ip, port))
                connected ←TRUE
                WHILE TRUE
                        IF close_join_connection = TRUE THEN
                                close_join_connection ← FALSE
                                BREAK
                        ENDIF
                        data ← s.recv(1024)
                        IF data = “START” THEN
                                start_multi_player_game ← TRUE
                        ENDIF
                        IF data = “END” THEN
                                main_game.menu_stack.pop()
                                BREAK
                        ENDIF
```

```
                IF data = “received” and data_queue[0] IN data_sent THEN
                        DEL data_queue[0]
                        data ←[]
                ENDIF
                IF data NOT IN data_received and data != [] THEN
                        Additem(data_received, data)
                ENDIF
                IF data_queue != [] THEN
                        data_to_send ← data_queue[0]
                ELSE
                        IF data != [] THEN
                                data_to_send ← “received”
                        ELSE
                                data_to_send ← []
                        ENDIF
                ENDIF
                s.send(data_to_send)
                IF data_queue != [] THEN
                        Additem(data_sent, data_queue[0])
                ENDIF
        ENDWHILE
        connected ←False
        s.close()
    ENDSUBROUTINE
ENDCLASS
```

2.6.2 Server Pseudocode

```
class Server
    protected ip
    public port
    protected clients
    public SUBROUTINE get_IP()
        WITH socket(socket.AF_INET, socket.SOCK_DGRAM) AS s
            s.connect((“8.8.8.8”,  80))
```

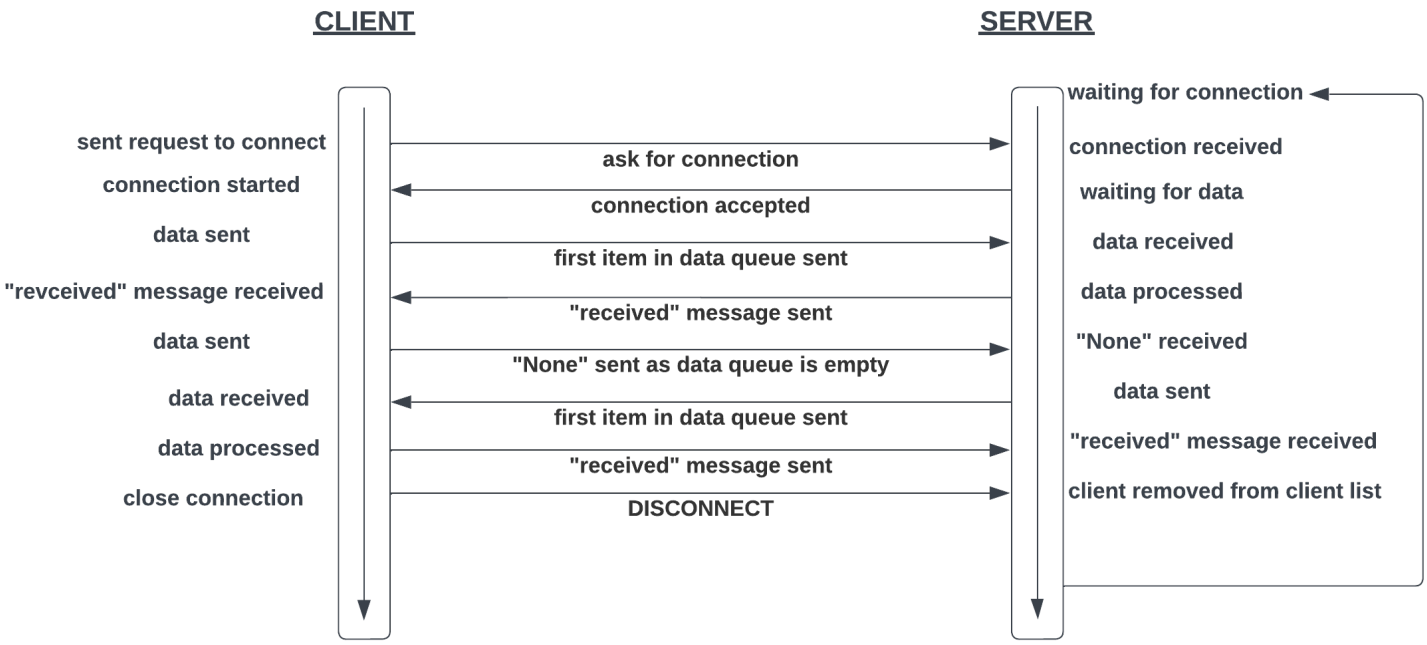
```
                RETURN s.getsockname()[0]
            ENDWITH
        ENDSUBROUTINE
        public SUBROUTINE listen_connect()
            WITH socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s
                s.bind((“”, port))
                s.listen(5)
                WHILE TRUE
                    c, addr ← s.accept()
                    client ←ClientObject(c)
                    thread = Thread(client.send_data)
                    thread.start()
                    _clients.append(client)
                ENDWHILE
            ENDWITH
        ENDSUBROUTINE
        public SUBROUTINE broadcast_data(data)
            FOR i in _clients
                Additem(i.data_queue, data)
            ENDFOR
        ENDSUBROUTINE
        public SUBROUTINE get_clients()
            RETURN _clients
        ENDSUBROUTINE
        public SUBROUTINE del_client(client)
            DEL _clients[THIS]
        ENDSUBROUTINE

class ClientObject
        public p
        public data_queue
        public data_received
        public data_sent
```

```
public received
public SUBROUTINE send_data(server)
        WHILE TRUE
                IF received = TRUE THEN
                        p.send("received")
                        received ← FALSE
                        CONTINUE
                ENDIF
                IF close_server_connection = TRUE THEN
                        p.send(END")
                        server.del_client(THIS)
                        BREAK
                ENDIF
                p.send(data_queue[0])
                IF data_queue != [] and received = False THEN
                        Additem(data_sent, data_queue[0])
                ENDIF
                data = p.recv(1024)
                IF data != [] and data NOT IN data_received THEN
                        received ← TRUE
                        Additem(data_received, data)
                        Additem(server.get_clients()[NOT THIS], data)
                ENDIF
                IF data = "received" AND data_queue[0] in data_sent THEN
                        DEL data_queue[0]
                ENDIF
        ENDWHILE
        server.del_client(THIS)
        p.close()
        BREAK
ENDSUBROUTINE
```

2.6.3 Client Server Diagram

Below is a diagram representing the connection between a client and the server. This shows the main steps of: waiting for connection, data received and data sent.



2.7 Stack

2.7.1 Stack Pseudocode

```
class Stack
        public maxlen
        public stack
        public top
        public SUBROUTINE is_empty()
                RETURN top
        ENDSUBROUTINE

        public SUBROUTINE is_full()
                if top = maxlen THEN
                        RETURN TRUE
                ELSE
                        RETURN FALSE
```

```
        ENDIF
ENDSUBROUTINE

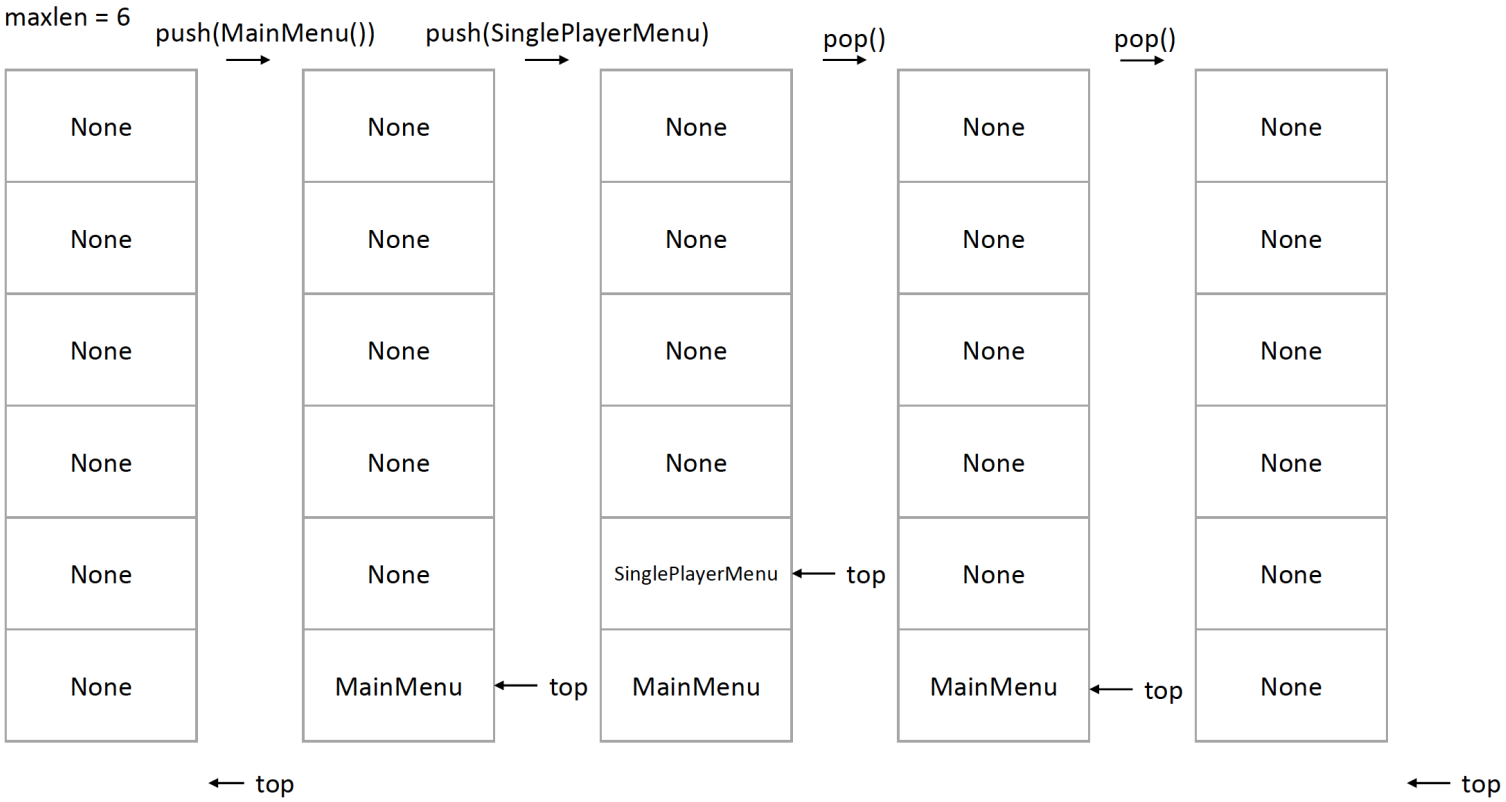
public SUBROUTINE push(item)
        IF is_full() = FALSE THEN
                top ← top+1
                stack[top] ← item
        ENDIF
ENDSUBROUTINE

public SUBROUTINE pop()
        IF is_empty() = False THEN
                item ← stack[top]
                stack[top] ← NULL
                top ← top-1
                RETURN item
        ENDIF
ENDSUBROUTINE

public SUBROUTINE peek()
        IF is_empty = FALSE THEN
                RETURN stack[top]
        ELSE
                “empty”
        ENDIF
ENDSUBROUTINE

public SUBROUTINE display_stack()
        RETURN stack
ENDSUBROUTINE
```

2.7.2 Stack Diagram



2.8 Move Check Methods

The main process of these functions is to try each move and, check which of its moves are allowed. For a move to be allowed it must: not have another piece on the target square unless the piece is attacking, not put itself in check or keep itself in check and, must only move in a set way based on the piece type.

2.8.1 Check Move Method Pseudocode

```
SUBROUTINE check_move(piece_type, state, current_pos, start_pos, team, check)
        moves ← []
        # IF PIECE IS A QUEEN
        IF piece_type = "Q" THEN
                pos ← [(0, -1), (1, -1), (1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1)]
                FOR I IN pos
                        WHILE TRUE
                                used_pos[0] ← used_pos[0] + i[0]
                                used_pos[1] ← used_pos[1] + i[1]
                                IF (
                                        used_pos[0] > 7
                                        OR used_pos[0] < 0
```

```
                                OR used_pos[1] > 7
                                OR used_pos[1] < 0
                                OR used_pos IN state THEN

                                IF (
                                        used_pos IN state
                                        AND state[used_pos].team != team THEN
                                                Additem(moves, used_pos)
                                ENDIF
                                used_pos ← current_pos
                                BREAK
                        ENDIF
                        IF used_pos NOT IN state AND used_pos NOT IN moves THEN
                                Additem(moves, used_pos)
                        ENDIF
                ENDWHILE
        ENDFOR
ENDIF
# IF PIECE IS A PAWN
IF piece_type = "P" THEN
        IF current_pos = start_pos THEN
                IF start_pos[1] = 1 THEN
                        IF (current_pos[0], current_pos[1] + 2) NOT IN state
                                AND (current_pos[0], current_pos[1] + 1) NOT IN state THEN
                                        Additem(moves, [current[0], current_pos[1] + 2])
                        ENDIF
                ELSE THEN
                        IF (current_pos[0], current_pos[1] - 2) NOT IN state
                                AND (current_pos[0], current_pos[1] - 1) NOT IN state THEN
                                        Additem(moves, [current[0], current_pos[1] - 2])
                        ENDIF
                ENDIF
        IF start_pos[1] = 1 THEN
```

```
                IF (7 >= current_pos[0] >=0
                        AND 7>= current_pos[1] + 1 >= 0
                        AND (current_pos[0], current_pos[1] + 1) NOT IN state THEN
                                Additem(moves,[current_pos[0], current_pos[1] + 1])

                IF ((current_pos[0] + 1, current[1] + 1) IN state
                        AND state[(current_pos[0] + 1, current_pos[1] + 1)].team != team) THEN
                                Additem(moves,[current_pos[0] + 1, current_pos[1] + 1])
                IF ((current_pos[0] - 1, current[1] + 1) IN state
                        AND state[(current_pos[0] - 1, current_pos[1] + 1)].team != team) THEN
                                Additem(moves,[current_pos[0] - 1, current_pos[1] + 1])
                ENDIF
        ELSE THEN
                IF (7>=current_pos[0] >=0
                        AND 7>=current_pos[1] – 1 >=0
                        AND [current_pos[0],current_pos[1] -1] NOT IN state} THEN
                                Additem(moves,[current_pos[0]], current_pos[1]-1)
                IF (current_pos[0]+1, current_pos[1]-1) IN state
                        AND state[[current_pos[0] +1, current_pos[1]-1]].team !=team THEN
                                Additem(moves, [current_pos[0]+1, current_pos[1]-1])
                IF (current_pos[0]-1, current_pos[1]-1) IN state
                        AND state[[current_pos[0]-1, current_pos[1]-1]].team != team THEN
                                Additem(moves, [current_pos[0]-1, current_pos[1]-1])
                ENDIF
        ENDIF
ENDIF
# IF PIECE IS A BISHOP
IF piece_type = “B” THEN
        pos ← [(1, -1), (1, 1), (-1, 1), (-1, -1)]
        FOR I IN pos
                WHILE TRUE
                        used_pos[0] ← used_pos[0] + i[0]
                        used_pos[1] ← used_pos[1] + i[1]
```

```
                IF (
                    used_pos[0] > 7
                    OR used_pos[0] < 0
                    OR used_pos[1] > 7
                    OR used_pos[1] < 0
                    OR used_pos IN state THEN

                    IF (
                        used_pos IN state
                        AND state[used_pos].team != team THEN
                            Additem(moves, used_pos)
                    ENDIF
                    used_pos ← current_pos
                    BREAK
                ENDIF
                IF used_pos NOT IN state AND used_pos NOT IN moves THEN
                    Additem(moves, used_pos)
                ENDIF
            ENDWHILE
        ENDFOR
ENDIF
# IF PIECE IS A ROOK
IF piece_type = "R" THEN
        pos ← [(0, -1), (1, 0), (0, 1), (-1, 0)]
        FOR I IN pos
            WHILE TRUE
                used_pos[0] ← used_pos[0] + i[0]
                used_pos[1] ← used_pos[1] + i[1]
                IF (
                    used_pos[0] > 7
                    OR used_pos[0] < 0
                    OR used_pos[1] > 7
                    OR used_pos[1] < 0
```

```
                                OR used_pos IN state THEN


                                IF (
                                        used_pos IN state
                                        AND state[used_pos].team != team THEN
                                                Additem(moves, used_pos)
                                ENDIF
                                used_pos ← current_pos
                                BREAK
                        ENDIF
                        IF used_pos NOT IN state AND used_pos NOT IN moves THEN
                                Additem(moves, used_pos)
                        ENDIF
                ENDWHILE
        ENDFOR
ENDIF
# IF PIECE IS A KING
IF piece_type = “K” THEN
        pos ← [(0, -1), (1, -1), (1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1)]
        FOR i IN pos
                IF ((current_pos[0] +i[0], current_pos[0] + i[1]) NOT IN state
                        AND 7 >= current_pos[0] + i[0] >=0
                        AND7 >= current_pos[1] + i[1] >=0 THEN
                                Additem(moves, [current_pos[0] + i[0], current[1] + i[1]])
                IF (current_pos[0] + i[0], current_pos[1] + i[1]) IN state
                        AND state[(current_[ps[0] + i[0], current_pos[1] + I[1])].team != team THEN
                                Additem(moves, [current_pos[0] + i[0], current_pos[1] + i[1]])
                ENDIF
        ENFOR
ENDIF
# IF PIECE IS A KNIGHT
IF piece_type = “N” THEN
        pos ← [(2, -1),(2, 1),(-2, 1),(-2, -1),(1, 2),(-1, 2),(-1, -2),(1, -2)]
```

```
            FOR i IN pos
                IF ((current_pos[0] +i[0], current_pos[0] + i[1]) NOT IN state
                    AND 7 >= current_pos[0] + i[0] >=0
                    AND7 >= current_pos[1] + i[1] >=0 THEN
                        Additem(moves, [current_pos[0] + i[0], current[1] + i[1]])
                IF (current_pos[0] + i[0], current_pos[1] + i[1]) IN state
                    AND state[(current_[ps[0] + i[0], current_pos[1] + I[1])].team != team THEN
                        Additem(moves, [current_pos[0] + i[0], current_pos[1] + i[1]])
                ENDIF
            ENFOR
        ENDIF
        IF NOT check THEN
            RETURN legal_moves(moves, current_pos, team, state)
        RETURN moves
ENDSUBROUTINE
```

2.8.2 Legal Moves Method Pseudocode

```
SUBROUTINE legal_moves(moves, placed_pos, team, state)
    FOR pos IN moves
        temp_state = state
        temp_state[pos] ← temp_state[placed_pos]
        DEL temp_state[placed_pos]
        IF check_check(temp_state, team) THEN
            DEL moves[pos]
        ENDIF
    ENDFOR
    RETURN moves
ENDSUBROUTINE
```

3 Commented Solution

3.1 main.py file

```
import game
game = game.Game()
game.run()
```

3.2 game.py file

```
import common
import abstract_data_types.stack as stack
import menus.join_game
import menus.main_menu
import menus
import sys
import pygame
import menus.multi_player
import menus.single_player_options
import menus.multi_player_options
import menus.single_player
pygame.init()


class Game:
    def __init__(self):
        self.dt = 1 / 60
        self.menu_stack = stack.Stack(10)
        self.offsetx = 30
        self.offsety = 110
        self.mult = 100
        self.display_size = [1200, 1000]
        self.window = pygame.display.set_mode(self.display_size)
        pygame.display.set_caption("Connor's Chess")
        pygame.display.set_icon(
            pygame.image.load(r"pieces/piece_images/king1.png").convert_alpha()
        )
        glow_surface = pygame.surface.Surface(self.display_size, flags=pygame.SRCALPHA)
```

```
    glow_surface.set_alpha(40)
    self.running = True
    allowed_moves = pygame.surface.Surface(self.display_size, flags=pygame.SRCALPHA)
    allowed_moves.set_alpha(100)
    self.background_colour = "#132713"
    self.backboard_colour = self.calculate_theme(self.background_colour)
    self.clock = pygame.time.Clock()
    self.menu_stack.push(
        menus.main_menu.MainMenu(
            self.display_size, self.backboard_colour, self.window
        )
    )
    self.pickup = []
    self.transpos_tables = {}

def run(self):
    """
    runs the main game loop, processing keyboard inputs and drawing the last item in the menu stack
    """
    while self.running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            if event.type == pygame.KEYDOWN:
                self.menu_stack.peek().event = event
                if (
                    event.key == pygame.K_ESCAPE
                    and self.menu_stack.peek() != self.menu_stack.display_stack()[0]
                ):
                    if isinstance(
                        self.menu_stack.peek(), menus.single_player.SinglePlayer
                    ):
                        self.menu_stack.push(
```

```
                    menus.single_player_options.SinglePlayerOptions(
                        self.display_size,
                        self.backboard_colour,
                        self.menu_stack.peek(),
                    )
                )
                break
            if isinstance(
                self.menu_stack.peek(), menus.multi_player.MultiPlayer
            ):
                self.menu_stack.push(
                    menus.multi_player_options.MultiPlayerOptions(
                        self.display_size,
                        self.backboard_colour,
                    )
                )
                break
            if isinstance(self.menu_stack.peek(), menus.join_game.JoinGame):
                if self.menu_stack.peek().client is not None:
                    self.menu_stack.peek().client.connected = False
                self.menu_stack.pop()
                break

            self.menu_stack.pop()
        self.pickup = []
        if isinstance(self.menu_stack.peek(), menus.single_player.SinglePlayer):
            self.transpos_tables = self.menu_stack.peek().transpos_tables
        self.menu_stack.peek().render_menu(self.window)
        self.dt = self.clock.tick(60) / 1000
        pygame.display.flip()
    common.save_json(self.transpos_tables)
    pygame.quit()
    sys.exit()
```

```
def calculate_theme(self, background_colour):
    """
    calculates the lighter background colour, based on the given background colour
    """
    hexdict = {
        "0": "0",
        "1": "1",
        "2": "2",
        "3": "3",
        "4": "4",
        "5": "5",
        "6": "6",
        "7": "7",
        "8": "8",
        "9": "9",
        "A": "10",
        "B": "11",
        "C": "12",
        "D": "13",
        "E": "14",
        "F": "15",
        "10": "A",
        "11": "B",
        "12": "C",
        "13": "D",
        "14": "E",
        "15": "F",
        "16": "F",
        "17": "F",
        "18": "F",
    }
    return "#" + "".join(
```

```
        [hexdict[str(int(hexdict[i]) + 3)] for i in (background_colour[1:]).upper()]
    )
```

3.3 pieces.py file

```
import threading
import pygame
import common
import copy
import menus.multi_player
import menus.single_player as single_player


class Piece:
    def __init__(self, team, image, piece_type, pos, value, game):
        self.team = team
        self.image = image
        self.type = piece_type
        self.value = value
        self.pos = pos
        self.pickup = False
        self.placed_pos = self.pos
        self.start_pos = self.pos
        self.game = game
        self.offsetx = 30
        self.offsety = 110
        self.mult = 100

    def __deepcopy__(self, memo):
        """
        overwrites the default deepcopy function so that the main game reference isn't copied
        """
        cls = self.__class__
        result = cls.__new__(cls)
        memo[id(self)] = result
```

```
    for k, v in self.__dict__.items():
        if k == "game" or k == "image":
            setattr(result, k, v)
            continue
        setattr(result, k, copy.deepcopy(v, memo))
    return result


def check_pickup(self):
    """
    checks if itself is being picked up and, if dropped on a legal move square, it makes the move
    """
    import client_server.client as client

    if self.team == self.game.turn and self.team == self.game.player2:
        if self.pickup or not self.game.check_pickup():
            if not self.pickup:
                if (
                    (pygame.mouse.get_pos()[0] - self.offsetx + 10) // self.mult
                    == self.pos[0]
                    and (pygame.mouse.get_pos()[1] - self.offsety + 10) // self.mult
                    == self.pos[1]
                    and pygame.mouse.get_pressed()[0]
                ):
                    self.pickup = True
                    self.game.main_game.pickup.append(self.placed_pos)
                    return True
                else:
                    self.pickup = False
                    return False
            else:
                if pygame.mouse.get_pressed()[2]:
                    self.pickup = False
                    return False
```

```
elif pygame.mouse.get_pressed()[0]:
    self.game.main_game.pickup.append(self.placed_pos)
    return True
elif not pygame.mouse.get_pressed()[0]:
    self.pickup = False
    if list(
        self.calculate_nearest_square(
            self.game.check_move(
                self.type,
                self.game.main_game.menu_stack.peek().board_game.board,
                self.placed_pos,
                self.start_pos,
                self.team,
            )
        )[0]
    ) in self.game.check_move(
        self.type,
        self.game.main_game.menu_stack.peek().board_game.board,
        self.placed_pos,
        self.start_pos,
        self.team,
    ):
        if not isinstance(self.game, single_player.SinglePlayer):
            client.client.data_queue.append(
                [
                    "move",
                    list(self.game.board_game.board.keys())[
                        list(
                            self.game.board_game.board.values()
                        ).index(self)
                    ],
                    (
                        self.calculate_nearest_square(
```

```
                        self.game.check_move(
                            self.type,
                            self.game.main_game.menu_stack.peek().board_game.board,
                            self.placed_pos,
                            self.start_pos,
                            self.team,
                        )
                    )[0]
                ),
            ]
        )
        self.game.main_game.menu_stack.peek().board_game.board = self.do_move(
            self.game.main_game.menu_stack.peek().board_game.board,
            (
                self.calculate_nearest_square(
                    self.game.check_move(
                        self.type,
                        self.game.main_game.menu_stack.peek().board_game.board,
                        self.placed_pos,
                        self.start_pos,
                        self.team,
                    )
                )[0]
            ),
            self.game,
        )

        return False

def do_move(self, state, pos, game):
    """
    does the given move onto the given state by adding a new instance of the piece and removing the old one
    """
```

```
import menus
import random


if isinstance(game, menus.multi_player.MultiPlayer):
    game.draw_count -= 1
if (
    isinstance(game, menus.single_player.SinglePlayer)
    and self.team == game.player2
):
    game.game_stack.push(copy.deepcopy(game.board_game))
if game.turn == self.team:
    game.turn = int(not game.turn)
else:
    return state
if game.game_over(state):
    return state
temp_state = {}
pos = tuple(pos)
shuffle_state = list(state.keys())
random.shuffle(shuffle_state)
for i in shuffle_state:
    temp_state[i] = state[i]
if pos in state.keys():
    game.board_game.teamdict[self.team] += state[pos].value
    if state[pos].team == game.player1:
        game.board_game.player1_out.append(state[pos])
    else:
        game.board_game.player2_out.append(state[pos])
state[pos] = state[tuple(self.pos)]
del state[tuple(self.pos)]
self.pos = pos
self.placed_pos = pos
```

```
    if self.team == game.HUMAN and isinstance(game, single_player.SinglePlayer):
        thread = threading.Thread(
            target=game.minimax_thread,
            daemon=True,
            args=(
                state,
                game.COMP,
                -9999,
                9999,
                game,
            ),
        )
        thread.start()

    self.game.check_mate_array[int(not self.team)] = self.game.checkmate(
        state, int(not self.team)
    )
    self.game.stalemate_end = self.game.stalemate(state, int(not self.team))
    return state

def draw_piece(self, surface):
    """
    draws the piece object onto the given surface
    """
    if self.check_pickup():
        newpos = self.calculate_nearest_square(
            self.game.check_move(
                self.type,
                self.game.main_game.menu_stack.peek().board_game.board,
                self.placed_pos,
                self.start_pos,
                self.team,
            )
```

```
        )[0]
        surface.blit(
            self.image,
            (
                newpos[0] * self.mult + self.offsetx - 2.2,
                newpos[1] * self.mult + self.offsety - 2,
            ),
        )
    else:
        surface.blit(
            pygame.transform.scale(self.image, (self.mult - 20, self.mult - 20)),
            (
                self.placed_pos[0] * self.mult + self.offsetx,
                self.pos[1] * self.mult + self.offsety,
            ),
        )


    def calculate_nearest_square(self, allowed_squares):
        """
        calculates the nearest legal move square to the mouse position so that the piece can snap to its allowed
positions
        """
        allowed_squares.append(self.placed_pos)
        mouse_pos = pygame.mouse.get_pos()
        nearest = [(99, 99), 99999]
        for i in allowed_squares:
            if (
                (((mouse_pos[1] - self.offsety - 35) / self.mult - i[1]) ** 2)
                + ((mouse_pos[0] - self.offsetx - 35) / self.mult - i[0]) ** 2
            ) ** 0.5 < nearest[1]:
                nearest = [
                    tuple(i),
                    (
```

```
                (((mouse_pos[1] - self.offsety - 45) / self.mult - i[1]) ** 2)
                + ((mouse_pos[0] - self.offsetx - 45) / self.mult - i[0]) ** 2
            )
            ** 0.5,
        ]
    return nearest


class Queen(Piece):
    def __init__(self, team, pos, game):
        super().__init__(
            team,
            pygame.transform.scale(
                pygame.image.load(
                    r"pieces/piece_images/queen.png"
                    if team
                    else r"pieces/piece_images/queen1.png"
                ).convert_alpha(),
                (common.mult - 15, common.mult - 15),
            ),
            "Q",
            pos,
            9,
            game,
        )
        self.starting_pos = pos


class Pawn(Piece):
    def __init__(self, team, pos, game):
        super().__init__(
            team,
            pygame.transform.scale(
```

```
            pygame.image.load(
                r"pieces/piece_images/pawn.png"
                if team
                else r"pieces/piece_images/pawn1.png"
            ).convert_alpha(),
            (common.mult - 15, common.mult - 15),
        ),
        "P",
        pos,
        1,
        game,
    )
    self.starting_pos = pos


class Knight(Piece):
    def __init__(self, team, pos, game):
        super().__init__(
            team,
            pygame.transform.scale(
                pygame.image.load(
                    r"pieces/piece_images/knight.png"
                    if team
                    else r"pieces/piece_images/knight1.png"
                ).convert_alpha(),
                (common.mult - 15, common.mult - 15),
            ),
            "N",
            pos,
            3,
            game,
        )
        self.starting_pos = pos
```

```
class Rook(Piece):
    def __init__(self, team, pos, game):
        super().__init__(
            team,
            pygame.transform.scale(
                pygame.image.load(
                    r"pieces/piece_images/rook.png"
                    if team
                    else r"pieces/piece_images/rook1.png"
                ).convert_alpha(),
                (common.mult - 15, common.mult - 15),
            ),
            "R",
            pos,
            5,
            game,
        )
        self.starting_pos = pos


class Bishop(Piece):
    def __init__(self, team, pos, game):
        super().__init__(
            team,
            pygame.transform.scale(
                pygame.image.load(
                    r"pieces/piece_images/bishop.png"
                    if team
                    else r"pieces/piece_images/bishop1.png"
                ).convert_alpha(),
                (common.mult - 15, common.mult - 15),
```

```
            ),
            "B",
            pos,
            3,
            game,
        )
        self.starting_pos = pos


class King(Piece):
    def __init__(self, team, pos, game):
        super().__init__(
            team,
            pygame.transform.scale(
                pygame.image.load(
                    r"pieces/piece_images/king.png"
                    if team
                    else r"pieces/piece_images/king1.png"
                ).convert_alpha(),
                (common.mult - 15, common.mult - 15),
            ),
            "K",
            pos,
            999,
            game,
        )
        self.starting_pos = pos
```

3.4 board.py file

```
import pygame
import common
import copy
```

```
def draw_board(pos, size, img, surface, colour, img2, img3):
    """
    draws the backboard and surround to the given surface
    """
    backboard = pygame.surface.Surface(size)
    backboard.fill(colour)
    surface.blit(backboard, pos)
    surface.blit(pygame.transform.scale(img, size), pos)
    surface.blit(
        pygame.transform.scale(img2, (850, 80)),
        (common.offsetx - 35, common.offsety + size[1] - 40),
    )
    surface.blit(
        pygame.transform.scale(img2, (850, 80)), (common.offsetx - 35, size[1] - 850)
    )
    surface.blit(pygame.transform.scale(img3, (355, 1000)), (845, 0))



class Board:
    def __init__(self, game):
        import pieces.pieces as pieces

        self.player1_out = []
        self.player2_out = []
        self.teamdict = {
            game.player1: 0,
            game.player2: 0,
        }
        self.game = game
        self.board = (
            {
```

```
    (4, 1): pieces.Pawn(self.game.player1, (4, 1), self.game),
    (3, 1): pieces.Pawn(self.game.player1, (3, 1), self.game),
    (1, 0): pieces.Knight(self.game.player1, (1, 0), self.game),
    (2, 0): pieces.Bishop(self.game.player1, (2, 0), self.game),
    (6, 0): pieces.Knight(self.game.player1, (6, 0), self.game),
    (5, 0): pieces.Bishop(self.game.player1, (5, 0), self.game),
    (0, 1): pieces.Pawn(self.game.player1, (0, 1), self.game),
    (1, 1): pieces.Pawn(self.game.player1, (1, 1), self.game),
    (2, 1): pieces.Pawn(self.game.player1, (2, 1), self.game),
    (5, 1): pieces.Pawn(self.game.player1, (5, 1), self.game),
    (6, 1): pieces.Pawn(self.game.player1, (6, 1), self.game),
    (7, 1): pieces.Pawn(self.game.player1, (7, 1), self.game),
    (0, 0): pieces.Rook(self.game.player1, (0, 0), self.game),
    (4, 0): pieces.King(self.game.player1, (4, 0), self.game),
    (3, 0): pieces.Queen(self.game.player1, (3, 0), self.game),
    (7, 0): pieces.Rook(self.game.player1, (7, 0), self.game),
    (3, 6): pieces.Pawn(self.game.player2, (3, 6), self.game),
    (1, 7): pieces.Knight(self.game.player2, (1, 7), self.game),
    (4, 6): pieces.Pawn(self.game.player2, (4, 6), self.game),
    (6, 7): pieces.Knight(self.game.player2, (6, 7), self.game),
    (2, 7): pieces.Bishop(self.game.player2, (2, 7), self.game),
    (0, 6): pieces.Pawn(self.game.player2, (0, 6), self.game),
    (1, 6): pieces.Pawn(self.game.player2, (1, 6), self.game),
    (2, 6): pieces.Pawn(self.game.player2, (2, 6), self.game),
    (5, 6): pieces.Pawn(self.game.player2, (5, 6), self.game),
    (6, 6): pieces.Pawn(self.game.player2, (6, 6), self.game),
    (7, 6): pieces.Pawn(self.game.player2, (7, 6), self.game),
    (0, 7): pieces.Rook(self.game.player2, (0, 7), self.game),
    (4, 7): pieces.King(self.game.player2, (4, 7), self.game),
    (3, 7): pieces.Queen(self.game.player2, (3, 7), self.game),
    (5, 7): pieces.Bishop(self.game.player2, (5, 7), self.game),
    (7, 7): pieces.Rook(self.game.player2, (7, 7), self.game),
}
```

```
if not game.player1
else {
    (4, 1): pieces.Pawn(self.game.player1, (4, 1), self.game),
    (3, 1): pieces.Pawn(self.game.player1, (3, 1), self.game),
    (1, 0): pieces.Knight(self.game.player1, (1, 0), self.game),
    (2, 0): pieces.Bishop(self.game.player1, (2, 0), self.game),
    (6, 0): pieces.Knight(self.game.player1, (6, 0), self.game),
    (5, 0): pieces.Bishop(self.game.player1, (5, 0), self.game),
    (0, 1): pieces.Pawn(self.game.player1, (0, 1), self.game),
    (1, 1): pieces.Pawn(self.game.player1, (1, 1), self.game),
    (2, 1): pieces.Pawn(self.game.player1, (2, 1), self.game),
    (5, 1): pieces.Pawn(self.game.player1, (5, 1), self.game),
    (6, 1): pieces.Pawn(self.game.player1, (6, 1), self.game),
    (7, 1): pieces.Pawn(self.game.player1, (7, 1), self.game),
    (0, 0): pieces.Rook(self.game.player1, (0, 0), self.game),
    (3, 0): pieces.King(self.game.player1, (3, 0), self.game),
    (4, 0): pieces.Queen(self.game.player1, (4, 0), self.game),
    (7, 0): pieces.Rook(self.game.player1, (7, 0), self.game),
    (3, 6): pieces.Pawn(self.game.player2, (3, 6), self.game),
    (1, 7): pieces.Knight(self.game.player2, (1, 7), self.game),
    (4, 6): pieces.Pawn(self.game.player2, (4, 6), self.game),
    (6, 7): pieces.Knight(self.game.player2, (6, 7), self.game),
    (2, 7): pieces.Bishop(self.game.player2, (2, 7), self.game),
    (0, 6): pieces.Pawn(self.game.player2, (0, 6), self.game),
    (1, 6): pieces.Pawn(self.game.player2, (1, 6), self.game),
    (2, 6): pieces.Pawn(self.game.player2, (2, 6), self.game),
    (5, 6): pieces.Pawn(self.game.player2, (5, 6), self.game),
    (6, 6): pieces.Pawn(self.game.player2, (6, 6), self.game),
    (7, 6): pieces.Pawn(self.game.player2, (7, 6), self.game),
    (0, 7): pieces.Rook(self.game.player2, (0, 7), self.game),
    (3, 7): pieces.King(self.game.player2, (3, 7), self.game),
    (4, 7): pieces.Queen(self.game.player2, (4, 7), self.game),
    (5, 7): pieces.Bishop(self.game.player2, (5, 7), self.game),
```

```
            (7, 7): pieces.Rook(self.game.player2, (7, 7), self.game),
        }
    )

  def __deepcopy__(self, memo):
    """
    overwrites the default deepcopy function so that the main game reference isn't copied
    """
    cls = self.__class__
    result = cls.__new__(cls)
    memo[id(self)] = result
    for k, v in self.__dict__.items():
      if k == "game":
        setattr(result, k, v)
        continue
      setattr(result, k, copy.deepcopy(v, memo))

    return result
```

3.5 menu.py file

```
import pygame

pygame.init()



class Menu:
  def __init__(self, size, colour):
    import main

    self.size = size
    self.surface = pygame.Surface(self.size)
    self.colour = colour
    self.buttons = []
    self.event = ""
```

```
        self.main_game = main.game
        self.pixel_font = pygame.font.Font("buttons/pixel_font.ttf", 64)

    def render_menu(self, surface):
        """
        displays the menu, processing everything in each game tick
        """
        surface.blit(self.surface, (0, 0))
        self.surface.fill(self.main_game.backboard_colour)
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])


def end_connection(menu):
    """
    closes the connection between the client and server
    """
    import client_server.server as server

    server.close_server_connection = True
    menu.main_game.menu_stack.pop()
```

3.6 main_menu.py file

```
import math
import pygame
from menus.menu import Menu
import menus.main_options as main_options
import menus.multi_player_menu as multi_player_menu
import menus.single_player_menu as single_player_menu


class MainMenu(Menu):
    def __init__(self, size, colour, surface):
        import buttons.buttons as button
```

```
super().__init__(size, colour)
self.logo_count = 0
self.title = math.sin(self.logo_count)
self.logo_king = pygame.image.load(
   "pieces/piece_images/king1.png"
).convert_alpha()
self.logo_king1 = pygame.image.load(
   "pieces/piece_images/king.png"
).convert_alpha()
self.buttons = [
   button.Button(
      [self.size[0] // 2, self.size[1] // 4 - 50],
      (480, 170),
      surface,
      self,
      "Singleplayer",
      self.single_player_menu,
   ),
   button.Button(
      [self.size[0] // 2, self.size[1] // 4 + 140],
      (480, 170),
      surface,
      self,
      "Multiplayer",
      self.multi_player_menu,
   ),
   button.Button(
      [self.size[0] // 2, self.size[1] // 4 + 330],
      (480, 170),
      surface,
      self,
      "Options",
```

```
            self.main_options,
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 520],
            (480, 170),
            surface,
            self,
            "Quit",
            self.game_end,
        ),
    ]

def render_menu(self, surface):
    """
    displays the menu, processing everything in each game tick
    """
    import main

    self.logo_count += 0.1
    self.title = math.sin(self.logo_count)
    self.surface.fill(main.game.backboard_colour)

    self.surface.blit(
        self.pixel_font.render("Connor's Chess", False, (199, 199, 199)),
        (360, self.title * 5 + 55),
    )
    self.surface.blit(
        pygame.transform.scale(self.logo_king, (100, 100)),
        (855, self.title * 5 + 25),
    )
    self.surface.blit(
        pygame.transform.scale(self.logo_king1, (100, 100)),
        (240, self.title * 5 + 25),
```

```
        )
        surface.blit(self.surface, (0, 0))
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

    def game_end(self):
        """
        stops the main game loop
        """
        import main

        main.game.running = False

    def main_options(self):
        """
        adds the main options menu to the menu stack
        """
        import main

        main.game.menu_stack.push(
            main_options.MainOptions(main.game.display_size, main.game.backboard_colour)
        )

    def multi_player_menu(self):
        """
        adds the multi-player menu to the menu stack
        """
        import main

        main.game.menu_stack.push(
            multi_player_menu.MultiPlayerMenu(
                main.game.display_size, main.game.backboard_colour
            )
```

```
        )

    def single_player_menu(self):
        """
        adds the single-player menu to the menu stack
        """
        import main

        main.game.menu_stack.push(
            single_player_menu.SinglePlayerMenu(
                main.game.display_size, main.game.backboard_colour
            )
        )
```

3.7 difficulty_settings.py file

```
import functools
import pygame
from menus.menu import Menu


class DifficultySettings(Menu):
    def __init__(self, size, colour, game):
        import buttons.buttons as button
        super().__init__(size, colour)
        self.game = game
        self.buttons = [
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 - 50],
                (480, 170),
                self.main_game.window,
                self,
                "Beginner",
                functools.partial(self.difficulty_change, 1),
            ),
```

```
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 140],
                (480, 170),
                self.main_game.window,
                self,
                "Intermediate",
                functools.partial(self.difficulty_change, 2),
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 330],
                (480, 170),
                self.main_game.window,
                self,
                "Master",
                functools.partial(self.difficulty_change, 3),
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 520],
                (480, 170),
                self.main_game.window,
                self,
                "Back",
                self.main_game.menu_stack.pop,
            ),
        ]

    def render_menu(self, surface):
        """
        displays the menu, processing everything in each game tick
        """
        self.surface.fill(self.main_game.backboard_colour)
        self.surface.blit(
            self.pixel_font.render("Select Difficulty", False, (199, 199, 199)),
```

```
            (350, 50),
        )
        surface.blit(self.surface, (0, 0))
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

    def difficulty_change(self, difficulty):
        """
        changes the difficulty of the game menu and pops itself off the top of the stack
        """
        self.game.difficulty = difficulty
        self.main_game.menu_stack.pop()
```

3.8 join_game.py file

```
import functools
import pygame
from menus.menu import Menu
import menus.multi_player as multi_player


class JoinGame(Menu):
    def __init__(self, size, colour):
        import buttons.buttons as button
        import client_server.server as server

        server.close_join_connection = False
        self.client = None
        self.ip = ""
        self.port = 0
        self.try_join = False
        self.points_font = pygame.font.Font(r"buttons/pixel_font.ttf", 20)
        super().__init__(size, colour)
        self.buttons = [
            button.TextBox(
```

```
        [self.size[0] // 2, self.size[1] // 4 - 50],
        (480, 170),
        self.main_game.window,
        self,
        text_size=48,
        max_len=15,
    ),
    button.TextBox(
        [self.size[0] // 2, self.size[1] // 4 + 140],
        (480, 170),
        self.main_game.window,
        self,
        text_size=64,
        max_len=5,
    ),
    button.Button(
        [self.size[0] // 2, self.size[1] // 4 + 330],
        (480, 170),
        self.main_game.window,
        self,
        "Join",
        self.client_join,
    ),
    button.Button(
        [self.size[0] // 2, self.size[1] // 4 + 520],
        (480, 170),
        self.main_game.window,
        self,
        "Back",
        functools.partial(
            self.exit_connection,
        ),
    ),
```

```
        ]

    def client_join(self):
        """
        instantiates a client object and starts a thread for the client join the server connection
        """
        import client_server.client as client
        import threading

        try:
            join_client = client.Client(self.ip, int(self.port), self.main_game)
            client.client = join_client
            client_thread = threading.Thread(target=join_client.join_game, daemon=True)
            client_thread.start()
            self.client = join_client
        except (ConnectionRefusedError, ValueError):
            pass
        self.try_join = True

    def exit_connection(self):
        """
        closes the client server connection
        """
        import client_server.client as client

        client.close_join_connection = True
        self.main_game.menu_stack.pop()

    def render_menu(self, surface):
        """
        displays the menu, processing everything in each game tick
        """
        import client_server.client as client
```

```
if client.start_multi_player_game[0]:
    self.main_game.menu_stack.push(
        multi_player.MultiPlayer(
            self.main_game.display_size,
            self.main_game.backboard_colour,
            client.start_multi_player_game[1],
            int(not client.start_multi_player_game[2]),
            client.start_multi_player_game[3],
            client.start_multi_player_game[4],
        )
    )

self.ip = self.buttons[0].text
self.port = self.buttons[1].text
self.surface.fill(self.main_game.backboard_colour)
self.surface.blit(
    self.pixel_font.render("Enter IP address and Port", False, (199, 199, 199)),
    (200, 50),
)
self.surface.blit(
    self.pixel_font.render("IP:", False, (199, 199, 199)),
    (265, 230),
)
self.surface.blit(
    self.pixel_font.render("Port:", False, (199, 199, 199)),
    (200, 410),
)
for i in list(self.ip) + list(self.port):
    if i not in list("1234567890."):
        self.surface.blit(
            self.points_font.render(
                "Invalid data type entered!",
```

```
                False,
                (199, 199, 199),
            ),
            (470, 150),
        )
    try:
        if self.client.connected:
            self.surface.blit(
                self.pixel_font.render("Connected", False, (199, 199, 199)),
                (425, 110),
            )
        if self.try_join and not self.client.connected:
            self.surface.blit(
                self.pixel_font.render("Disconnected", False, (199, 199, 199)),
                (395, 110),
            )
    except AttributeError:
        pass

    surface.blit(self.surface, (0, 0))
    for i in self.buttons:
        i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])
```

3.9 main_options.py file

```
import pygame
import menus.theme as theme
from menus.menu import Menu


class MainOptions(Menu):
    def __init__(self, size, colour):
        import buttons.buttons as button

        super().__init__(size, colour)
```

```
        self.buttons = [
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 - 50],
                (480, 170),
                self.main_game.window,
                self,
                "Theme",
                self.theme_settings,
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 140],
                (480, 170),
                self.main_game.window,
                self,
                "Back",
                self.main_game.menu_stack.pop,
            ),
        ]

    def render_menu(self, surface):
        """
        displays the menu, processing everything in each game tick
        """
        self.surface.fill(self.main_game.backboard_colour)
        self.surface.blit(
            self.pixel_font.render("Options", False, (199, 199, 199)),
            (480, 50),
        )
        surface.blit(self.surface, (0, 0))
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

    def theme_settings(self):
```

```
        """
        pushes the theme settings menu to the menu stack
        """
        self.main_game.menu_stack.push(
            theme.Theme(self.main_game.display_size, self.main_game.backboard_colour)
        )
```

3.10 multi_player_menu.py file

```
import pygame
import menus.join_game as join_game
from menus.menu import Menu
import menus.new_game as new_game



class MultiPlayerMenu(Menu):
    def __init__(self, size, colour):
        import buttons.buttons as button

        super().__init__(size, colour)
        self.buttons = [
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 - 50],
                (480, 170),
                self.main_game.window,
                self,
                "New Game",
                self.new_game,
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 140],
                (480, 170),
                self.main_game.window,
                self,
                "Join Game",
```

```
            self.join_game,
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 330],
            (480, 170),
            self.main_game.window,
            self,
            "Back",
            self.main_game.menu_stack.pop,
        ),
    ]

def render_menu(self, surface):
    """
    displays the menu, processing everything in each game tick
    """
    self.surface.fill(self.main_game.backboard_colour)
    self.surface.blit(
        self.pixel_font.render("Multiplayer", False, (199, 199, 199)),
        (440, 50),
    )
    surface.blit(self.surface, (0, 0))
    for i in self.buttons:
        i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

def join_game(self):
    """
    adds the join game menu to the menu stack
    """
    self.main_game.menu_stack.push(
        join_game.JoinGame(
            self.main_game.display_size, self.main_game.backboard_colour
        )
```

```
        )

    def new_game(self):
        """
        adds the new game menu to the menu stack
        """
        self.main_game.menu_stack.push(
            new_game.NewGame(
                self.main_game.display_size, self.main_game.backboard_colour
            )
        )
```

3.11 multi_player_options.py file

```
import pygame
import menus.theme as theme
from menus.menu import Menu


class MultiPlayerOptions(Menu):
    def __init__(self, size, colour):
        import buttons.buttons as button

        super().__init__(size, colour)
        self.buttons = [
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 - 50],
                (480, 170),
                self.main_game.window,
                self,
                "Theme",
                self.theme_settings,
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 140],
```

```
            (480, 170),
            self.main_game.window,
            self,
            "Back",
            self.main_game.menu_stack.pop,
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 330],
            (480, 170),
            self.main_game.window,
            self,
            "Menu",
            self.multi_player_quit,
        ),
    ]

def multi_player_quit(self):
    """
    pops off the menu stack and disconnects the client server connection
    """
    import client_server.client as client

    self.main_game.menu_stack.pop()
    self.main_game.menu_stack.pop()
    self.main_game.menu_stack.pop()
    if (
        self.main_game.menu_stack.peek()
        != self.main_game.menu_stack.display_stack()[0]
    ):
        self.main_game.menu_stack.pop()
    client.close_join_connection = True

def render_menu(self, surface):
```

```
        """
        displays the menu, processing everything in each game tick
        """
        self.surface.fill(self.main_game.backboard_colour)
        self.surface.blit(
            self.pixel_font.render("Options", False, (199, 199, 199)),
            (480, 50),
        )
        surface.blit(self.surface, (0, 0))
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

    def theme_settings(self):
        """
        pushes the theme settings menu to the menu stack
        """
        self.main_game.menu_stack.push(
            theme.Theme(self.main_game.display_size, self.main_game.backboard_colour)
        )
```

3.12 multi_player_settings.py file

```
import pygame
from menus.multi_player_timer_menu import MultiPlayerTimerMenu
from menus.menu import Menu
from menus.multi_player import MultiPlayer



class MultiPlayerSettings(Menu):
    def __init__(self, size, colour):
        import buttons.buttons as button

        super().__init__(size, colour)
        self.team = 1
        self.chat = False
```

```
self.timer = False
self.buttons = [
    button.ToggleButton(
        [self.size[0] // 2, self.size[1] // 4 + -50],
        (480, 170),
        self.main_game.window,
        self,
        on_text="Enabled",
        off_text="Disabled",
        on_colour=(0, 0, 0),
        off_colour=(199, 199, 199),
    ),
    button.ToggleButton(
        [self.size[0] // 2, self.size[1] // 4 + 140],
        (480, 170),
        self.main_game.window,
        self,
        on_text="Black",
        off_text="White",
        on_colour=(0, 0, 0),
        off_colour=(199, 199, 199),
    ),
    button.ToggleButton(
        [self.size[0] // 2, self.size[1] // 4 + 330],
        (480, 170),
        self.main_game.window,
        self,
        on_text="Enabled",
        off_text="Disabled",
        on_colour=(0, 0, 0),
        off_colour=(199, 199, 199),
    ),
    button.Button(
```

```
            [self.size[0] // 2, self.size[1] // 4 + 520],
            (480, 170),
            self.main_game.window,
            self,
            "Start Game",
            self.start_multi_player_game,
            text_colour=self.main_game.background_colour,
        ),
    ]

def start_multi_player_game(self):
    """
    starts the game and sends a 'START' message to the other client, so they start the game aswell
    """
    import client_server.server as server

    if self.timer:
        self.main_game.menu_stack.push(
            MultiPlayerTimerMenu(
                self.size, self.colour, self.timer, self.team, self.chat
            )
        )

    else:
        server.server.broadcast_data(["START", self.chat, self.team, self.timer, 0])
        self.main_game.menu_stack.push(
            MultiPlayer(
                self.main_game.display_size,
                self.main_game.backboard_colour,
                self.chat,
                self.team,
                self.timer,
                0,
```

```
            )
        )


    def render_menu(self, surface):
        """
        displays the menu, processing everything in each game tick
        """
        self.chat = self.buttons[0].pressed
        self.team = int(not self.buttons[1].pressed)
        self.timer = self.buttons[2].pressed
        self.surface.fill(self.main_game.backboard_colour)
        self.surface.blit(
            self.pixel_font.render("Options", False, (199, 199, 199)),
            (480, 50),
        )
        self.surface.blit(
            self.pixel_font.render("Text Chat:", False, (199, 199, 199)),
            (25, 220),
        )
        self.surface.blit(
            self.pixel_font.render("Team:", False, (199, 199, 199)),
            (160, 410),
        )
        self.surface.blit(
            self.pixel_font.render("Timer:", False, (199, 199, 199)),
            (140, 600),
        )
        surface.blit(self.surface, (0, 0))
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])
```

3.13 Multi_player_timer_menu.py file

```
import functools
import pygame
```

```
from menus.menu import Menu
from menus.multi_player import MultiPlayer


class MultiPlayerTimerMenu(Menu):
    def __init__(self, size, colour, timer, team, chat):
        import buttons.buttons as button

        super().__init__(size, colour)
        self.timer = timer
        self.team = team
        self.chat = chat
        self.buttons = [
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 - 50],
                (480, 170),
                self.main_game.window,
                self,
                "03:00",
                functools.partial(self.start_multi_player_game, 180),
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 140],
                (480, 170),
                self.main_game.window,
                self,
                "05:00",
                functools.partial(self.start_multi_player_game, 300),
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 330],
                (480, 170),
                self.main_game.window,
```

```
            self,
            "10:00",
            functools.partial(self.start_multi_player_game, 600),
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 520],
            (480, 170),
            self.main_game.window,
            self,
            "Back",
            self.main_game.menu_stack.pop,
        ),
    ]

def render_menu(self, surface):
    """
    displays the menu, processing everything in each game tick
    """
    self.surface.fill(self.main_game.backboard_colour)
    self.surface.blit(
        self.pixel_font.render("Select Time", False, (199, 199, 199)),
        (350, 50),
    )
    surface.blit(self.surface, (0, 0))
    for i in self.buttons:
        i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

def start_multi_player_game(self, time_duration):
    """
    starts the game after receiving the 'START' message
    """
    import client_server.server as server
```

```
        server.server.broadcast_data(
            ["START", self.chat, self.team, self.timer, time_duration]
        )
        self.main_game.menu_stack.push(
            MultiPlayer(
                self.main_game.display_size,
                self.main_game.backboard_colour,
                self.chat,
                self.team,
                self.timer,
                time_duration,
            )
        )
```

3.14 new_game.py file

```
import functools
import threading
import menus.multi_player_settings as multi_player_settings
import pygame

from menus.menu import Menu, end_connection



class NewGame(Menu):
    def __init__(self, size, colour):
        import buttons.buttons as button
        import client_server.server as server
        import client_server.client as client

        super().__init__(size, colour)
        server.close_server_connection = False
        server.server = server.Server()
        self.buttons = [
            button.Button(
```

```
        [self.size[0] // 2, self.size[1] // 4 + 140],
        (480, 170),
        self.main_game.window,
        self,
        "Back",
        functools.partial(end_connection, self),
    ),
    button.Button(
        [self.size[0] // 2, self.size[1] // 4 + 330],
        (480, 170),
        self.main_game.window,
        self,
        "Start Game",
        self.multi_player_settings,
    ),
]

self.thread = threading.Thread(target=server.server.listen_connect, daemon=True)
self.thread.start()
self.ip = server.server.get_ip()
self.port = server.server.port
self.main_client = client.Client(self.ip, self.port, self.main_game)
client.client = self.main_client
self.client_thread = threading.Thread(
    target=self.main_client.join_game, daemon=True
)
self.client_thread.start()
self.connection_text = [
    "Waiting for connection.",
    "Waiting for connection .",
    "Waiting for connection  .",
    "connected to player2",
]
```

```
    self.connection_count = 0

def render_menu(self, surface):
    """
    displays the menu, processing everything in each game tick
    """
    import client_server.server as server

    self.connection_count += 0.05

    self.surface.fill(self.main_game.backboard_colour)
    self.surface.blit(
        self.pixel_font.render(
            self.connection_text[
                3
                if len(server.server.get_clients()) == 2
                else round(self.connection_count) % 3
            ],
            False,
            (199, 199, 199),
        ),
        (250, 50),
    )
    self.surface.blit(
        self.pixel_font.render(f"IP: {self.ip}", False, (199, 199, 199)),
        (365, 150),
    )
    self.surface.blit(
        self.pixel_font.render(f"Port: {self.port}", False, (199, 199, 199)),
        (365, 250),
    )
    surface.blit(self.surface, (0, 0))
    if len(server.server.get_clients()) == 2:
```

```
            self.buttons[1].update(
                pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0]
            )
        self.buttons[0].update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

    def multi_player_settings(self):
        """
        pushes the multi-player settings menu to the menu stack
        """
        self.main_game.menu_stack.push(
            multi_player_settings.MultiPlayerSettings(
                self.main_game.display_size, self.main_game.backboard_colour
            )
        )
```

3.15 single_player_difficulty_menu.py file

```
import functools

import pygame

from menus.menu import Menu
from menus.single_player import SinglePlayer



class SinglePlayerDifficultyMenu(Menu):
    def __init__(self, size, colour, timer, team, time_duration):
        import buttons.buttons as button

        super().__init__(size, colour)
        self.timer = timer
        self.team = team
        self.time_duration = time_duration
        self.buttons = [
            button.Button(
```

```
            [self.size[0] // 2, self.size[1] // 4 - 50],
            (480, 170),
            self.main_game.window,
            self,
            "Beginner",
            functools.partial(
                self.single_player_callback,
                1,
                self.timer,
                self.team,
                self.time_duration,
            ),
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 140],
            (480, 170),
            self.main_game.window,
            self,
            "Intermediate",
            functools.partial(
                self.single_player_callback,
                2,
                self.timer,
                self.team,
                self.time_duration,
            ),
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 330],
            (480, 170),
            self.main_game.window,
            self,
            "Master",
```

```
            functools.partial(
                self.single_player_callback, 3, self.timer, self.team, time_duration
            ),
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 520],
            (480, 170),
            self.main_game.window,
            self,
            "Back",
            self.main_game.menu_stack.pop,
        ),
    ]

def render_menu(self, surface):
    """
    displays the menu, processing everything in each game tick
    """
    self.surface.fill(self.main_game.backboard_colour)
    self.surface.blit(
        self.pixel_font.render("Select Difficulty", False, (199, 199, 199)),
        (350, 50),
    )
    surface.blit(self.surface, (0, 0))
    for i in self.buttons:
        i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

def single_player_callback(self, difficulty, timer, team, time_duration):
    """
    pushes the single-player game menu to the menu stack
    """
    self.main_game.menu_stack.push(
        SinglePlayer(
```

```
            self.main_game.display_size,
            self.main_game.backboard_colour,
            difficulty,
            timer,
            int(not team),
            time_duration,
        )
    )
```

3.16 single_player_menu.py file

```
import pygame
from menus.menu import Menu
from menus.single_player_timer_menu import SinglePlayerTimerMenu
from menus.single_player_difficulty_menu import SinglePlayerDifficultyMenu



class SinglePlayerMenu(Menu):
    def __init__(self, size, colour):
        import buttons.buttons as button
        super().__init__(size, colour)
        self.timer = False
        self.team = False
        self.buttons = [
            button.ToggleButton(
                [self.size[0] // 2, self.size[1] // 4 - 50],
                (480, 170),
                self.main_game.window,
                self,
                on_text="Enabled",
                off_text="Disabled",
                on_colour=(0, 0, 0),
            ),
            button.ToggleButton(
                [self.size[0] // 2, self.size[1] // 4 + 140],
```

```
            (480, 170),
            self.main_game.window,
            self,
            on_text="Black",
            off_text="White",
            on_colour=(0, 0, 0),
            off_colour=(199, 199, 199),
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 330],
            (480, 170),
            self.main_game.window,
            self,
            "Play",
            self.single_player_timer_menu,
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 520],
            (480, 170),
            self.main_game.window,
            self,
            "Back",
            self.main_game.menu_stack.pop,
        ),
    ]

def single_player_timer_menu(self):
    """
    checks if the timer has been enabled. If it has, the timer menu will be added to the menu stack.
    If not, the difficulty menu will be added instead.
    """
    if self.timer:
        self.main_game.menu_stack.push(
```

```
            SinglePlayerTimerMenu(
                self.main_game.display_size,
                self.main_game.backboard_colour,
                self.timer,
                self.team,
            )
        )
    else:
        self.main_game.menu_stack.push(
            SinglePlayerDifficultyMenu(
                self.size, self.colour, self.timer, self.team, 0
            )
        )

def render_menu(self, surface):
    """
    displays the menu, processing everything in each game tick
    """
    self.timer = self.buttons[0].pressed
    self.team = int(self.buttons[1].pressed)
    self.surface.fill(self.main_game.backboard_colour)
    self.surface.blit(
        self.pixel_font.render("Single Player Settings", False, (199, 199, 199)),
        (280, 50),
    )
    self.surface.blit(
        self.pixel_font.render("Timer:", False, (199, 199, 199)),
        (155, 220),
    )
    self.surface.blit(
        self.pixel_font.render("Team:", False, (199, 199, 199)),
        (160, 410),
    )
```

```
        surface.blit(self.surface, (0, 0))
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])
```

3.17 single_player_options.py file

```
import pygame
from menus.menu import Menu
import menus.difficulty_settings as difficulty_settings
import menus.theme as theme



class SinglePlayerOptions(Menu):
    def __init__(self, size, colour, game):
        import buttons.buttons as button

        super().__init__(size, colour)
        self.game = game
        self.buttons = [
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 - 50],
                (480, 170),
                self.main_game.window,
                self,
                "Theme",
                self.theme_settings,
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 140],
                (480, 170),
                self.main_game.window,
                self,
                "Difficulty",
                self.difficulty_settings_callback,
            ),
```

```
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 330],
                (480, 170),
                self.main_game.window,
                self,
                "Back",
                self.main_game.menu_stack.pop,
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 520],
                (480, 170),
                self.main_game.window,
                self,
                "Menu",
                self.game_quit,
            ),
        ]

    def render_menu(self, surface):
        """
        displays the menu, processing everything in each game tick
        """

        self.surface.fill(self.main_game.backboard_colour)
        self.surface.blit(
            self.pixel_font.render("Options", False, (199, 199, 199)),
            (480, 50),
        )
        surface.blit(self.surface, (0, 0))
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

    def difficulty_settings_callback(self):
```

```
        """
        adds the difficulty settings menu to the menu stack
        """
        self.main_game.menu_stack.push(
            difficulty_settings.DifficultySettings(
                self.main_game.display_size, self.main_game.backboard_colour, self.game
            )
        )

    def game_quit(self):
        """
        pops off the menu stack to return to the main menu
        """
        for i in range(4):
            if (
                self.main_game.menu_stack.peek()
                != self.main_game.menu_stack.display_stack()[0]
            ):
                self.main_game.menu_stack.pop()

    def theme_settings(self):
        """
        pushes the theme settings menu to the menu stack
        """
        self.main_game.menu_stack.push(
            theme.Theme(self.main_game.display_size, self.main_game.backboard_colour)
        )
```

3.18 single_player_timer_menu.py file

```
import functools
import pygame
from menus.menu import Menu
from menus.single_player_difficulty_menu import SinglePlayerDifficultyMenu
```

```
class SinglePlayerTimerMenu(Menu):
    def __init__(self, size, colour, timer, team):
        import buttons.buttons as button

        super().__init__(size, colour)
        self.timer = timer
        self.team = team
        self.buttons = [
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 - 50],
                (480, 170),
                self.main_game.window,
                self,
                "03:00",
                functools.partial(self.single_player_difficulty_menu, 180),
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 140],
                (480, 170),
                self.main_game.window,
                self,
                "05:00",
                functools.partial(self.single_player_difficulty_menu, 300),
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 330],
                (480, 170),
                self.main_game.window,
                self,
                "10:00",
                functools.partial(self.single_player_difficulty_menu, 600),
            ),
```

```
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 520],
                (480, 170),
                self.main_game.window,
                self,
                "Back",
                self.main_game.menu_stack.pop,
            ),
        ]

    def render_menu(self, surface):
        """
        displays the menu, processing everything in each game tick
        """
        self.surface.fill(self.main_game.backboard_colour)
        self.surface.blit(
            self.pixel_font.render("Select Time", False, (199, 199, 199)),
            (430, 50),
        )
        surface.blit(self.surface, (0, 0))
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

    def single_player_difficulty_menu(self, time_duration):
        """
        pushes the single-player difficulty menu to the menu stack
        """
        self.main_game.menu_stack.push(
            SinglePlayerDifficultyMenu(
                self.main_game.display_size,
                self.main_game.backboard_colour,
                self.timer,
                self.team,
```

```
            time_duration,
        )
    )
```

3.19 theme.py file

```
import functools
import pygame
from menus.menu import Menu


class Theme(Menu):
    def __init__(self, size, colour):
        import buttons.buttons as button

        super().__init__(size, colour)
        self.buttons = [
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 - 50],
                (480, 170),
                self.main_game.window,
                self,
                "Red",
                functools.partial(self.set_theme, "#45000a"),
                text_colour="#45000a",
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 140],
                (480, 170),
                self.main_game.window,
                self,
                "Green",
                functools.partial(self.set_theme, "#132713"),
                text_colour="#132713",
            ),
```

```
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 330],
                (480, 170),
                self.main_game.window,
                self,
                "Blue",
                functools.partial(self.set_theme, "#131327"),
                text_colour="#131327",
            ),
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 520],
                (480, 170),
                self.main_game.window,
                self,
                "Back",
                self.main_game.menu_stack.pop,
            ),
        ]

    def render_menu(self, surface):
        """
        displays the menu, processing everything in each game tick
        """
        self.surface.fill(self.main_game.backboard_colour)
        self.surface.blit(
            self.pixel_font.render("Theme", False, (199, 199, 199)),
            (490, 50),
        )
        surface.blit(self.surface, (0, 0))
        for i in self.buttons:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

    def set_theme(self, colour):
```

```
        """
        sets the main colour theme to the given colour
        """
        self.main_game.backboard_colour = self.main_game.calculate_theme(colour)
        self.main_game.background_colour = colour
```

3.20 game_menu.py file

```
import copy
import random
import pygame
import common
from board import board
from menus.menu import Menu


class GameMenu(Menu):
    def __init__(self, size, colour, team, time_duration, timer):
        super().__init__(size, colour)
        self.timer = timer
        self.time_duration = [time_duration, time_duration]
        self.resign = [False, False]
        self.stalemate_end = False
        self.game_draw = False
        self.turn = 1
        self.check_mate_array = [False, False]
        self.best_move = [False, [(-1, -1), (-1, -1), 0]]
        self.HUMAN = team
        self.COMP = int(not team)
        self.player1 = self.COMP
        self.player2 = self.HUMAN
        self.end_text = pygame.font.Font(r"buttons/pixel_font.ttf", 128)
        self.time_text = pygame.font.Font(r"buttons/pixel_font.ttf", 64)
        self.difficulty = None
        self.transpos_tables = common.open_json()
```

```
self.menu_background = pygame.image.load(
    r"board/board_images/side_bar.png"
).convert_alpha()
self.board_img = pygame.image.load(
    r"board/board_images/board.png"
).convert_alpha()
self.out_piece_surround = pygame.image.load(
    r"board/board_images/out_piece_surround.png"
).convert_alpha()
self.board_game = board.Board(self)
self.end_surface = pygame.surface.Surface(
    self.main_game.display_size, flags=pygame.SRCALPHA
)
self.glow_surface = pygame.surface.Surface(
    self.main_game.display_size, flags=pygame.SRCALPHA
)
self.glow_surface.set_alpha(40)
self.allowed_moves = pygame.surface.Surface(
    self.main_game.display_size, flags=pygame.SRCALPHA
)
self.end_surface.set_alpha(230)
self.end_surface.fill(self.colour)
self.allowed_moves.set_alpha(100)
self.allowed_moves.fill((0, 0, 0, 0))
self.glow_surface.fill((0, 0, 0, 0))
self.text_surface = pygame.surface.Surface(size, flags=pygame.SRCALPHA)
self.font = pygame.font.SysFont("arial", 10, True, False)
self.points_font = pygame.font.Font(r"buttons/pixel_font.ttf", 20)
self.text_surface.fill((0, 0, 0, 0))
self.buttons = []
self.offsetx = 30
self.offsety = 110
self.mult = 100
```

```
def render_menu(self, surface):
    """
    renders the game menu and
    """
    try:
        if not int(self.time_duration[self.turn]):
            self.time_duration[self.turn] = 0
        elif self.timer:
            self.time_duration[self.turn] -= self.main_game.dt
    except IndexError:
        pass
    surface.fill(self.main_game.background_colour)
    self.text_surface.fill((0, 0, 0, 0))
    try:
        if self.best_move[0]:
            self.board_game.board[self.best_move[1][0]].do_move(
                self.board_game.board, self.best_move[1][1], self
            )
            self.best_move[0] = False
    except KeyError:
        pass
    self.allowed_moves.fill((0, 0, 0, 0))
    self.glow_surface.fill((0, 0, 0, 0))
    board.draw_board(
        (self.offsetx - 35, self.offsety - 35),
        (8.5 * self.mult, 8.5 * self.mult),
        self.board_img,
        surface,
        self.main_game.backboard_colour,
        self.out_piece_surround,
        self.menu_background,
    )
```

```
self.draw_glow(self.board_game.board.copy(), surface, self)
surface.blit(self.glow_surface, (0, 0), special_flags=pygame.BLEND_RGB_SUB)
for i in self.board_game.board.values():
    if not i.pickup:
        i.draw_piece(surface)
try:
    self.board_game.board[tuple(self.main_game.pickup[0])].draw_piece(surface)
except IndexError:
    pass
self.draw_out_pieces(
    self.board_game.player1_out + self.board_game.player2_out, surface
)

for i in self.board_game.board.values():
    if i.pickup:
        for j in self.check_move(
            i.type, self.board_game.board, i.placed_pos, i.start_pos, i.team
        ):
            pygame.draw.circle(
                self.allowed_moves,
                (50, 50, 50),
                (
                    j[0] * self.mult + self.offsetx + 40,
                    j[1] * self.mult + self.offsety + 40,
                ),
                self.mult // 10,
            )
surface.blit(self.allowed_moves, (0, 0), special_flags=pygame.BLEND_RGB_SUB)
self.text_surface.blit(
    self.font.render(f"{int(self.main_game.clock.get_fps())}", True, (0, 0, 0)),
    (10, 0),
)
"""
```

```
draws the score for each player
"""
self.text_surface.blit(
    self.points_font.render(
        f"+{abs(self.board_game.teamdict[self.player1] - self.board_game.teamdict[self.player2])}"
        if self.board_game.teamdict[self.player1]
        - self.board_game.teamdict[self.player2]
        > 0
        else "",
        True,
        (0, 0, 0),
    ),
    (20, 70),
)
self.text_surface.blit(
    self.points_font.render(
        f"+{abs(self.board_game.teamdict[self.player2] - self.board_game.teamdict[self.player1])}"
        if self.board_game.teamdict[self.player2]
        - self.board_game.teamdict[self.player1]
        > 0
        else "",
        True,
        (0, 0, 0),
    ),
    (20, 915),
)
"""
draws timers for each player
"""
if self.timer:
    self.text_surface.blit(
        self.time_text.render(
            self.time_format_translator(int(self.time_duration[self.player2])),
```

```
                False,
                (199, 199, 199) if self.turn == self.player1 else (199, 55, 55),
            ),
            (940, 920),
        )
        self.text_surface.blit(
            self.time_text.render(
                self.time_format_translator(int(self.time_duration[self.player1])),
                False,
                (199, 199, 199) if self.turn == self.player2 else (199, 55, 55),
            ),
            (940, 20),
        )
    """
    updates all the buttons and draws the surfaces onto the window
    """
    for i in self.buttons[:-1]:
        i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])
    surface.blit(self.text_surface, (0, 0))
    self.render_end_game(surface)

def render_end_game(self, surface):
    """
    draws the surface for all end game instances
    """
    if self.game_draw:
        self.turn = 2
        surface.blit(self.end_surface, (0, 0))
        self.main_game.window.blit(
            self.end_text.render("Draw", False, (199, 199, 199)), (450, 350)
        )
        self.buttons[-1].update(
            pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0]
```

```
        )
    if self.check_mate_array[self.player1]:
        self.turn = 2
        surface.blit(self.end_surface, (0, 0))
        self.main_game.window.blit(
            self.end_text.render("Victory", False, (199, 199, 199)), (380, 150)
        )
        self.main_game.window.blit(
            self.end_text.render("Checkmate", False, (199, 199, 199)), (260, 320)
        )
        self.buttons[-1].update(
            pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0]
        )
    if self.check_mate_array[self.player2]:
        self.turn = 2
        surface.blit(self.end_surface, (0, 0))
        self.main_game.window.blit(
            self.end_text.render("Defeat", False, (199, 199, 199)), (400, 150)
        )
        self.main_game.window.blit(
            self.end_text.render("Checkmate", False, (199, 199, 199)), (260, 320)
        )
        self.buttons[-1].update(
            pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0]
        )
    if self.resign[self.player1]:
        self.turn = 2
        surface.blit(self.end_surface, (0, 0))
        self.main_game.window.blit(
            self.end_text.render("Victory", False, (199, 199, 199)), (380, 150)
        )
        self.main_game.window.blit(
            self.end_text.render("Resign", False, (199, 199, 199)), (400, 320)
```

```
            )
            self.buttons[-1].update(
                pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0]
            )
        if self.resign[self.player2]:
            self.turn = 2
            surface.blit(self.end_surface, (0, 0))
            self.main_game.window.blit(
                self.end_text.render("Defeat", False, (199, 199, 199)), (400, 150)
            )
            self.main_game.window.blit(
                self.end_text.render("Resign", False, (199, 199, 199)), (400, 320)
            )
            self.buttons[-1].update(
                pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0]
            )
        if not self.time_duration[self.player1] and self.timer:
            self.turn = 2
            surface.blit(self.end_surface, (0, 0))
            self.main_game.window.blit(
                self.end_text.render("Victory", False, (199, 199, 199)), (380, 150)
            )
            self.main_game.window.blit(
                self.end_text.render("Out Of Time", False, (199, 199, 199)), (250, 320)
            )
            self.buttons[-1].update(
                pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0]
            )
        if not self.time_duration[self.player2] and self.timer:
            self.turn = 2
            surface.blit(self.end_surface, (0, 0))
            self.main_game.window.blit(
                self.end_text.render("Defeat", False, (199, 199, 199)), (400, 150)
```

```
            )
            self.main_game.window.blit(
                self.end_text.render("Out Of Time", False, (199, 199, 199)), (250, 320)
            )
            self.buttons[-1].update(
                pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0]
            )
        if self.stalemate_end:
            self.turn = 2
            surface.blit(self.end_surface, (0, 0))
            self.main_game.window.blit(
                self.end_text.render("Stalemate", False, (199, 199, 199)), (260, 320)
            )
            self.buttons[-1].update(
                pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0]
            )

    def time_format_translator(self, time):
        """
        reformats the given time from seconds to minutes and seconds in the layout: 00:00
        """
        if time // 60 in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
            mins = "0" + str(time // 60)
        else:
            mins = str(time // 60)
        if time % 60 in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
            secs = "0" + str(time % 60)
        else:
            secs = str(time % 60)
        return f"{mins}:{secs}"

    def minimax_thread(self, state, player, alpha, beta, game):
        """
```

```
    calls the minimax function in a thread with calculated difficulty and transposition table relation
    """
    transpos = [["." for i in range(8)] for j in range(8)]
    for i in state.keys():
        transpos[i[1]][i[0]] = state[i].type
    for i in range(len(transpos)):
        transpos[i] = tuple(transpos[i])
    transpos = tuple(transpos)
    if game.difficulty in [1, 3]:
        depth = 2
    else:
        depth = 3
    move = self.minimax(
        state.copy(), state.copy(), depth, player, alpha, beta, game
    )
    self.best_move[0], self.best_move[1] = True, move
    if player == self.player1:
        if transpos not in self.transpos_tables:
            self.transpos_tables[transpos] = tuple(self.best_move[1])


def draw_out_pieces(self, out_pieces, surface):
    """
    draws the pieces that have been taken on both sides
    """
    white, black = [], []
    for i in out_pieces:
        if not i.team:
            black.append(i)
        else:
            white.append(i)
    white, black = sorted(white, key=lambda x: x.value), sorted(
        black, key=lambda x: x.value
    )
```

```
    if self.player1:
        white, black = black, white
    for i in range(len(black)):
        surface.blit(
            pygame.transform.scale(black[i].image, (45, 45)),
            (i * 55 + 8, self.offsety + self.mult * 8.29),
        )
    for i in range(len(white)):
        surface.blit(
            pygame.transform.scale(white[i].image, (45, 45)),
            (i * 55 + 8, self.mult - 80),
        )


def move_types(self, state, team):
    """
    returns an array of all: attacking, quiet and checking moves
    """
    attack, quiet, check = [], [], []
    for i in state.values():
        if i.team == team:
            for j in self.check_move(
                i.type,
                state,
                list(state.keys())[list(state.values()).index(i)],
                i.start_pos,
                i.team,
            ):
                if tuple(j) in list(state.keys()):
                    attack.append(
                        (
                            list(state.keys())[list(state.values()).index(i)],
                            tuple(j),
                        )
```

```
                )
            else:
                quiet.append(
                    (
                        list(state.keys())[list(state.values()).index(i)],
                        tuple(j),
                    )
                )
    temp_state = state.copy()
    for i in attack + quiet:
        temp_state[i[1]] = temp_state[i[0]]
        del temp_state[i[0]]
        self.check_check(temp_state.copy(), temp_state[i[1]].team)
        check.append(i)
        temp_state = state.copy()
    return [quiet, attack, check]

def quiet_moves(self, state, team):
    """
    returns quiet moves
    """
    return self.move_types(state, team)[0]

def attacking_moves(self, state, team):
    """
    returns attacking moves
    """
    return self.move_types(state, team)[1]

def checking_moves(self, state, team):
    """
    returns checking moves
    """
```

```
    return self.move_types(state, team)[2]


def check_move(self, piece_type, state, current_pos, start_pos, team, check=0):
    """
    returns legal moves for each piece type
    """
    state = state.copy()
    moves = []
    # Queen
    if piece_type == "Q":
        pos = [(0, -1), (1, -1), (1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1)]
        used_pos = list(current_pos)
        for i in pos:
            while True:
                used_pos[0], used_pos[1] = used_pos[0] + i[0], used_pos[1] + i[1]
                if (
                    used_pos[0] > 7
                    or used_pos[0] < 0
                    or used_pos[1] > 7
                    or used_pos[1] < 0
                    or tuple(used_pos) in list(state.keys())
                ):
                    if (
                        tuple(used_pos) in list(state.keys())
                        and state[tuple(used_pos)].team != team
                    ):
                        moves.append(copy.deepcopy(used_pos))
                    used_pos = list(copy.deepcopy(current_pos))
                    break
                if (
                    tuple(used_pos) not in list(state.keys())
                    and list(used_pos) not in moves
                ):
```

```
                moves.append(copy.deepcopy(used_pos))
# Pawn
if piece_type == "P":
    if current_pos == start_pos:
        if start_pos[1] == 1:
            if (current_pos[0], current_pos[1] + 2) not in list(
                state.keys()
            ) and (
                current_pos[0],
                current_pos[1] + 1,
            ) not in list(
                state.keys()
            ):
                moves.append([current_pos[0], current_pos[1] + 2])
        else:
            if (current_pos[0], current_pos[1] - 2) not in list(
                state.keys()
            ) and (
                current_pos[0],
                current_pos[1] - 1,
            ) not in list(
                state.keys()
            ):
                moves.append([current_pos[0], current_pos[1] - 2])

    if start_pos[1] == 1:
        if (
            7 >= current_pos[0] >= 0
            and 7 >= current_pos[1] + 1 >= 0
            and tuple((current_pos[0], current_pos[1] + 1))
            not in list(state.keys())
        ):
            moves.append([current_pos[0], current_pos[1] + 1])
```

```
            if (current_pos[0] + 1, current_pos[1] + 1) in list(
                state.keys()
            ) and state[(current_pos[0] + 1, current_pos[1] + 1)].team != team:
                moves.append([current_pos[0] + 1, current_pos[1] + 1])
            if (current_pos[0] - 1, current_pos[1] + 1) in list(
                state.keys()
            ) and state[(current_pos[0] - 1, current_pos[1] + 1)].team != team:
                moves.append([current_pos[0] - 1, current_pos[1] + 1])
        else:
            if (
                7 >= current_pos[0] >= 0
                and 7 >= current_pos[1] - 1 >= 0
                and tuple((current_pos[0], current_pos[1] - 1))
                not in list(state.keys())
            ):
                moves.append([current_pos[0], current_pos[1] - 1])
            if (current_pos[0] + 1, current_pos[1] - 1) in list(
                state.keys()
            ) and state[(current_pos[0] + 1, current_pos[1] - 1)].team != team:
                moves.append([current_pos[0] + 1, current_pos[1] - 1])
            if (current_pos[0] - 1, current_pos[1] - 1) in list(
                state.keys()
            ) and state[(current_pos[0] - 1, current_pos[1] - 1)].team != team:
                moves.append([current_pos[0] - 1, current_pos[1] - 1])
    # Bishop
    if piece_type == "B":
        pos = [(1, -1), (1, 1), (-1, 1), (-1, -1)]
        used_pos = list(current_pos)
        for i in pos:
            while True:
                used_pos[0], used_pos[1] = used_pos[0] + i[0], used_pos[1] + i[1]
                if (
                    used_pos[0] > 7
```

```
                or used_pos[0] < 0
                or used_pos[1] > 7
                or used_pos[1] < 0
                or tuple(used_pos) in list(state.keys())
            ):
                if (
                    tuple(used_pos) in list(state.keys())
                    and state[tuple(used_pos)].team != team
                ):
                    moves.append(copy.deepcopy(used_pos))
                used_pos = list(copy.deepcopy(current_pos))
                break
            if (
                tuple(used_pos) not in list(state.keys())
                and list(used_pos) not in moves
            ):
                moves.append(copy.deepcopy(used_pos))
    # Rook
    if piece_type == "R":
        pos = [(0, -1), (1, 0), (0, 1), (-1, 0)]
        used_pos = list(current_pos)
        for i in pos:
            while True:
                used_pos[0], used_pos[1] = used_pos[0] + i[0], used_pos[1] + i[1]
                if (
                    used_pos[0] > 7
                    or used_pos[0] < 0
                    or used_pos[1] > 7
                    or used_pos[1] < 0
                    or tuple(used_pos) in list(state.keys())
                ):
                    if (
                        tuple(used_pos) in list(state.keys())
```

```
                and state[tuple(used_pos)].team != team
            ):
                moves.append(copy.deepcopy(used_pos))
            used_pos = list(copy.deepcopy(current_pos))
            break
        if (
            tuple(used_pos) not in list(state.keys())
            and list(used_pos) not in moves
        ):
            moves.append(copy.deepcopy(used_pos))
# King
if piece_type == "K":
    pos = [(0, -1), (1, -1), (1, 0), (1, 1), (0, 1), (-1, 1), (-1, 0), (-1, -1)]
    for i in pos:
        if (
            (current_pos[0] + i[0], current_pos[1] + i[1])
            not in list(state.keys())
            and 7 >= current_pos[0] + i[0] >= 0
            and 7 >= current_pos[1] + i[1] >= 0
        ):
            moves.append([current_pos[0] + i[0], current_pos[1] + i[1]])
        if (current_pos[0] + i[0], current_pos[1] + i[1]) in list(
            state.keys()
        ) and state[
            (current_pos[0] + i[0], current_pos[1] + i[1])
        ].team != team:
            moves.append([current_pos[0] + i[0], current_pos[1] + i[1]])
# Knight
if piece_type == "N":
    pos = [
        (2, -1),
        (2, 1),
        (-2, 1),
```

```
            (-2, -1),
            (1, 2),
            (-1, 2),
            (-1, -2),
            (1, -2),
        ]
        for i in pos:
            if (
                (current_pos[0] + i[0], current_pos[1] + i[1])
                not in list(state.keys())
                and 7 >= current_pos[0] + i[0] >= 0
                and 7 >= current_pos[1] + i[1] >= 0
            ):
                moves.append([current_pos[0] + i[0], current_pos[1] + i[1]])
            if (current_pos[0] + i[0], current_pos[1] + i[1]) in list(
                state.keys()
            ) and state[
                (current_pos[0] + i[0], current_pos[1] + i[1])
            ].team != team:
                moves.append([current_pos[0] + i[0], current_pos[1] + i[1]])
    if not check:
        return self.legal_moves(moves, current_pos, team, state)
    return moves

def legal_moves(self, moves, placed_pos, team, state):
    """
    checks if the move legal, and doesn't cause itself to be in check
    """
    moves_copy = moves.copy()
    for i in moves:
        pos = tuple(i)
        temp_state = state.copy()
        temp_state[pos] = temp_state[tuple(placed_pos)]
```

```
            del temp_state[tuple(placed_pos)]
            if self.check_check(temp_state.copy(), team):
                del moves_copy[moves_copy.index(i)]
        return moves_copy

    def check_pickup(self):
        """
        checks if a piece is being picked up
        """
        for i in self.board_game.board.values():
            if i.pickup:
                return True
        return False

    def draw_glow(self, state, surface, game):
        """
        draws the glow behind each piece when the mouse is hovered over them
        """
        for i in state.values():
            if i.check_pickup():
                return
        mouse_pos = (
            (pygame.mouse.get_pos()[0] - self.offsetx + 10) // self.mult,
            (pygame.mouse.get_pos()[1] - self.offsety + 10) // self.mult,
        )

        if mouse_pos in game.board_game.board.keys():
            game.glow_surface.blit(
                pygame.transform.scale(
                    game.board_game.board[mouse_pos].image,
                    (self.mult, self.mult),
                ),
                (
```

```
                mouse_pos[0] * self.mult + self.offsetx - 10,
                mouse_pos[1] * self.mult + self.offsety - 10,
            ),
        )
        surface.blit(game.glow_surface, (0, 0), special_flags=pygame.BLEND_RGB_SUB)

    def minimax(
        self,
        temp_state,
        temp_state1,
        depth,
        player,
        alpha,
        beta,
        game,
        points=0,
        temp_points=0,
    ):
        """
        calculates the best move in the current position based on the depth and the balance of points for each player after each turn
        """
        transpos = [["." for i in range(8)] for j in range(8)]
        for i in temp_state.keys():
            transpos[i[1]][i[0]] = temp_state[i].type
        for i in range(len(transpos)):
            transpos[i] = tuple(transpos[i])
        transpos = tuple(transpos)
        try:
            if transpos in tuple(self.transpos_tables.keys()):
                return list(self.transpos_tables[transpos])
        except IndexError:
            pass
```

```
# Base case
if depth == 0 or self.game_over(temp_state):
    return (
        [
            (-1, -1),
            (-1, -1),
            self.quiescent_search(
                temp_state.copy(),
                temp_state.copy(),
                2,
                -self.evaluate_state(temp_state, game),
                self.evaluate_state(temp_state, game),
                temp_points,
                temp_points,
                game.COMP,
                game,
            ),
        ]
        if game.difficulty == 3
        else [
            (-1, -1),
            (-1, -1),
            self.evaluate_state(temp_state, game, temp_points),
        ]
    )

# General case
# Maximising player
if player == game.COMP:
    score = [(-1, -1), (-1, -1), -9999]
    shuffled_pieces = list(temp_state.values())
    random.shuffle(shuffled_pieces)
```

```
for i in shuffled_pieces:
    if i.team == game.COMP:
        shuffled_moves = self.check_move(
            i.type,
            temp_state,
            list(temp_state.keys())[list(temp_state.values()).index(i)],
            i.start_pos,
            i.team,
        )
        random.shuffle(shuffled_moves)
        for j in shuffled_moves:
            if tuple(j) in temp_state:
                temp_points += temp_state[tuple(j)].value
            temp_state1[tuple(j)] = temp_state1[
                list(temp_state1.keys())[
                    list(temp_state1.values()).index(i)
                ]
            ]
            del temp_state1[
                list(temp_state1.keys())[
                    list(temp_state1.values()).index(i)
                ]
            ]

            # Recursive call
            if (
                x := (
                    self.minimax(
                        temp_state1.copy(),
                        temp_state1.copy(),
                        depth - 1,
                        int(not bool(player)),
                        alpha,
```

```
                    beta,
                    game,
                    temp_points,
                    temp_points,
                )[2]
            )
        ) > score[2]:
            score[0], score[1], score[2] = (
                tuple(
                    list(temp_state.keys())[
                        list(temp_state.values()).index(i)
                    ]
                ),
                tuple(j),
                x,
            )
        temp_points = points
        temp_state1 = temp_state.copy()
        alpha = max(alpha, score[2])
        if beta < alpha and game.difficulty == 1:
            return score
        elif beta <= alpha:
            return score
    return score

# Minimising player
if player == game.HUMAN:
    score = [(-1, -1), (-1, -1), 9999]
    shuffled_pieces = list(temp_state.values())
    random.shuffle(shuffled_pieces)
    for i in shuffled_pieces:
        if i.team == game.HUMAN:
            shuffled_moves = self.check_move(
```

```
    i.type,
    temp_state,
    list(temp_state.keys())[list(temp_state.values()).index(i)],
    i.start_pos,
    i.team,
)
random.shuffle(shuffled_moves)
for j in shuffled_moves:
    if tuple(j) in temp_state:
        temp_points -= temp_state[tuple(j)].value
    temp_state1[tuple(j)] = temp_state1[
        list(temp_state1.keys())[
            list(temp_state1.values()).index(i)
        ]
    ]
    del temp_state1[
        list(temp_state1.keys())[
            list(temp_state1.values()).index(i)
        ]
    ]

    # Recursive call
    if (
        x := (
            self.minimax(
                temp_state1.copy(),
                temp_state1.copy(),
                depth - 1,
                int(not bool(player)),
                alpha,
                beta,
                game,
                temp_points,
```

```
                        temp_points,
                    )[2]
                )
            ) < score[2]:
                score[0], score[1], score[2] = (
                    tuple(
                        list(temp_state1.keys())[
                            list(temp_state1.values()).index(i)
                        ]
                    ),
                    tuple(j),
                    x,
                )
            temp_points = points
            temp_state1 = temp_state.copy()

            # Alpha-Beta pruning
            beta = min(beta, score[2])
            if beta < alpha and game.difficulty == 1:
                return score
            elif beta <= alpha:
                return score
    return score

def quiescent_end(self, state, depth):
    """
    dictates the base case for the quiescent search
    """
    if not depth:
        return True
    if (
        len((self.attacking_moves(state, 1) + self.attacking_moves(state, 0)))
        <= len(list(state.values())) // 6
```

```
        ):
            return True
        if not self.checking_moves(state, 1) + self.checking_moves(state, 0):
            return True
        return False


    def evaluate_state(self, state, game, points=0):
        """
        returns the evaluation for the current state in favour of the maximising player, given: checkmates,
stalemates and piece balance
        """
        if self.checkmate(state, game.HUMAN):
            return 999
        elif self.checkmate(state, game.COMP):
            return -999
        elif (x := (self.stalemate(state, 1) or self.stalemate(state, 0))) and (
            (game.board_game.teamdict[0] - game.board_game.teamdict[1])
        ) < 0:
            return 5
        elif x and (game.board_game.teamdict[0] - game.board_game.teamdict[1]) > 0:
            return -5
        else:
            return (game.board_game.teamdict[0] - game.board_game.teamdict[1]) + points


    def game_over(self, state):
        """
        checks if the game is over
        """
        if (
            self.stalemate(state, 1)
            or self.stalemate(state, 0)
            or self.checkmate(state, 1)
            or self.checkmate(state, 0)
```

```
    ):
        return True

def quiescent_search(
    self, state, temp_state, depth, alpha, beta, points, temp_points, team, game
):
    """
    performs a secondary search after the minimax search, to extend the depth until a quiet position
    and looks at only checking and attacking moves for the fastest path to a quiet position
    """

    # Base case
    if self.game_over(state) or self.quiescent_end(state.copy(), depth):
        return self.evaluate_state(state, game, points)

    # General case
    # Maximising player
    if team == game.COMP:
        score = -9999
        attack_check = self.attacking_moves(
            state.copy(), team
        ) + self.checking_moves(state.copy(), team)
        random.shuffle(attack_check)
        for i in attack_check:
            if tuple(i[1]) in temp_state.keys():
                temp_points += temp_state[i[1]].value
            temp_state[i[1]] = temp_state[i[0]]
            del temp_state[i[0]]

            # Recursive call
            score = max(
                self.quiescent_search(
                    temp_state.copy(),
```

```
                temp_state.copy(),
                depth - 1,
                alpha,
                beta,
                temp_points,
                temp_points,
                int(not team),
                game,
            ),
            score,
        )
        temp_state = state.copy()
        temp_points = points
        alpha = max(alpha, score)
        if beta <= alpha:
            return score

# Minimising player
else:
    score = 9999
    attack_check = self.attacking_moves(
        state.copy(), team
    ) + self.checking_moves(state.copy(), team)
    random.shuffle(attack_check)
    for i in attack_check:
        if tuple(i[1]) in temp_state.keys():
            temp_points -= temp_state[i[1]].value
        temp_state[i[1]] = temp_state[i[0]]
        del temp_state[i[0]]

        # Recursive call
        score = min(
            self.quiescent_search(
```

```
                temp_state.copy(),
                temp_state.copy(),
                depth - 1,
                alpha,
                beta,
                temp_points,
                temp_points,
                int(not team),
                game,
            ),
            score,
        )
        temp_state = state.copy()
        temp_points = points
        beta = min(beta, score)
        if beta <= alpha:
            return score
    return score


def stalemate(self, state, team):
    """
    checks if there is a stale mate
    """
    if self.check_check(state, team):
        return False
    for i in state.values():
        if i.team == team:
            if self.check_move(
                i.type,
                state,
                list(state.keys())[list(state.values()).index(i)],
                i.start_pos,
                team,
```

```
            ):
                return False
        return True


    def check_check(self, state, team, king_location=(-1, -1)):
        """
        checks if the given player is in check
        """
        for i in state.values():
            if i.team == team and i.type == "K":
                king_location = list(state.keys())[list(state.values()).index(i)]
        for i in state.values():
            if i.team != team and list(king_location) in self.check_move(
                i.type,
                state,
                list(state.keys())[list(state.values()).index(i)],
                i.start_pos,
                i.team,
                1,
            ):
                return True
        return False


    def checkmate(self, state, team):
        """
        checks if the given player on the given board is in checkmate
        """
        if not self.check_check(state, team):
            return False
        for i in state.values():
            if i.team == team:
                if self.check_move(
                    i.type,
```

```
                state,
                list(state.keys())[list(state.values()).index(i)],
                i.start_pos,
                team,
            ):
                return False
    return True


def multi_player_quit(self):
    """
    pops off the menu stack to the main menu and closes the connection
    """
    import client_server.client as client

    self.main_game.menu_stack.pop()
    self.main_game.menu_stack.pop()
    self.main_game.menu_stack.pop()
    if (
        self.main_game.menu_stack.peek()
        != self.main_game.menu_stack.display_stack()[0]
    ):
        self.main_game.menu_stack.pop()
    client.close_join_connection = True


def process_data(self, data, game):
    """
    processes the given data
    """
    if data[0] == "move":
        try:
            game.board_game.board[tuple(self.move_translator(data[1:])[0])].do_move(
                game.board_game.board, (self.move_translator(data[1:])[1]), game
            )
```

```
            except KeyError:
                pass
        if data[0] == "chat":
            game.chat = [f"OPPONENT: {data[1]}"] + game.chat
            if len(game.chat) > 5:
                del game.chat[-1]
        if data[0] == "draw":
            game.draw_count = 3
        if data[0] == "draw_accepted":
            game.game_draw = True
        if data[0] == "resign":
            game.resign[game.player1] = True


    def move_translator(self, move):
        """
        translates the given move for the opponent
        """
        return [(7 - move[0][0], 7 - move[0][1]), (7 - move[1][0], 7 - move[1][1])]


    def game_quit(self):
        """
        pops off the menu stack to the main menu
        """
        for i in range(4):
            if (
                self.main_game.menu_stack.peek()
                != self.main_game.menu_stack.display_stack()[0]
            ):
                self.main_game.menu_stack.pop()
```

3.21 mutli_player.py file

```
import pygame
from board import board as board
from menus.game_menu import GameMenu
```

```
from menus.menu import end_connection
from menus.multi_player_options import MultiPlayerOptions
import functools


class MultiPlayer(GameMenu):
    def __init__(self, size, colour, text_chat, team, timer, time_duration):
        import buttons.buttons as button

        super().__init__(size, colour, team, time_duration, timer)
        self.chat = []
        self.draw_count = 0
        self.draw_attempt = 0
        self.text_chat = text_chat
        self.text_chat_image = pygame.transform.scale(
            pygame.image.load("buttons/button_background.png"), (320, 140)
        )

        self.buttons = (
            [
                button.Button(
                    [self.size[0] // 2 + 425, self.size[1] // 4 - 140],
                    (320, 180),
                    self.main_game.window,
                    self,
                    "Menu",
                    functools.partial(
                        self.main_game.menu_stack.push,
                        MultiPlayerOptions(self.size, self.colour),
                    ),
                ),
                button.Button(
                    [self.size[0] // 2 + 425, self.size[1] // 4 + 70],
```

```
            (320, 180),
            self.main_game.window,
            self,
            "Draw",
            self.send_draw,
            text_colour=(199, 199, 199)
            if not self.draw_count
            else (199, 55, 55),
        ),
        button.Button(
            [self.size[0] // 2 + 425, self.size[1] // 4 + 280],
            (320, 180),
            self.main_game.window,
            self,
            "Resign",
            self.send_resign,
        ),
        button.ChatBox(
            [self.size[0] // 2 + 425, self.size[1] // 4 + 610],
            (320, 70),
            self.main_game.window,
            self,
            max_len=14,
            text_size=24,
        ),
        button.Button(
            [self.size[0] // 2, self.size[1] // 4 + 280],
            (320, 180),
            self.main_game.window,
            self,
            "Quit",
            self.multi_player_quit,
        ),
```

```
]
if self.text_chat
else [
    button.Button(
        [self.size[0] // 2 + 425, self.size[1] // 4 - 40],
        (320, 180),
        self.main_game.window,
        self,
        "Menu",
        functools.partial(
            self.main_game.menu_stack.push,
            MultiPlayerOptions(self.size, self.colour),
        ),
    ),
    button.Button(
        [self.size[0] // 2 + 425, self.size[1] // 4 + 170],
        (320, 180),
        self.main_game.window,
        self,
        "Draw",
        self.send_draw,
        text_colour=(199, 199, 199)
        if not self.draw_count
        else (199, 55, 55),
    ),
    button.Button(
        [self.size[0] // 2 + 425, self.size[1] // 4 + 380],
        (320, 180),
        self.main_game.window,
        self,
        "Resign",
        self.send_resign,
    ),
```

```
            button.Button(
                [self.size[0] // 2, self.size[1] // 4 + 280],
                (320, 180),
                self.main_game.window,
                self,
                "Quit",
                self.multi_player_quit,
            ),
        ]
    )

def render_menu(self, surface):
    """
    displays the menu, processing everything in each game tick
    """
    import client_server.server as server
    import client_server.client as client

    try:
        if not int(self.time_duration[self.turn]):
            self.time_duration[self.turn] = 0
        elif self.timer:
            self.time_duration[self.turn] -= self.main_game.dt
    except IndexError:
        pass
    if client.client.data_received:
        self.process_data(client.client.data_received[0], self)
        del client.client.data_received[0]

    try:
        if len(server.server.get_clients()) != 2:
            end_connection(self)
            self.main_game.menu_stack.pop()
```

```
except AttributeError:
    pass
try:
    if not client.client.connected:
        client.start_multi_player_game = [False]
        self.multi_player_quit()
except AttributeError:
    pass
surface.fill(self.main_game.background_colour)
self.text_surface.fill((0, 0, 0, 0))
self.allowed_moves.fill((0, 0, 0, 0))
self.glow_surface.fill((0, 0, 0, 0))
board.draw_board(
    (self.offsetx - 35, self.offsety - 35),
    (8.5 * self.mult, 8.5 * self.mult),
    self.board_img,
    surface,
    self.main_game.backboard_colour,
    self.out_piece_surround,
    self.menu_background,
)
self.draw_glow(self.board_game.board.copy(), surface, self)
surface.blit(self.glow_surface, (0, 0), special_flags=pygame.BLEND_RGB_SUB)
try:
    for i in self.board_game.board.values():
        if not i.pickup:
            i.draw_piece(surface)
    try:
        self.board_game.board[tuple(self.main_game.pickup[0])].draw_piece(
            surface
        )
    except AttributeError:
        pass
```

```
except IndexError:
    pass
self.draw_out_pieces(
    self.board_game.player1_out + self.board_game.player2_out, surface
)

for i in self.board_game.board.values():
    if i.pickup:
        for j in self.check_move(
            i.type, self.board_game.board, i.placed_pos, i.start_pos, i.team
        ):
            pygame.draw.circle(
                self.allowed_moves,
                (50, 50, 50),
                (
                    j[0] * self.mult + self.offsetx + 40,
                    j[1] * self.mult + self.offsety + 40,
                ),
                self.mult // 10,
            )
surface.blit(self.allowed_moves, (0, 0), special_flags=pygame.BLEND_RGB_SUB)
self.text_surface.blit(
    self.font.render(f"{int(self.main_game.clock.get_fps())}", True, (0, 0, 0)),
    (10, 0),
)
self.text_surface.blit(
    self.points_font.render(
        f"+{abs(self.board_game.teamdict[self.player1] - self.board_game.teamdict[self.player2])}"
        if self.board_game.teamdict[self.player1]
        - self.board_game.teamdict[self.player2]
        > 0
        else "",
        True,
```

```
            (0, 0, 0),
        ),
        (20, 70),
    )
    self.text_surface.blit(
        self.points_font.render(
            f"+{abs(self.board_game.teamdict[self.player2] - self.board_game.teamdict[self.player1])}"
            if self.board_game.teamdict[self.player2]
            - self.board_game.teamdict[self.player1]
            > 0
            else "",
            True,
            (0, 0, 0),
        ),
        (20, 915),
    )
    if self.text_chat:
        surface.blit(
            self.text_chat_image, (self.size[0] // 2 + 265, self.size[1] // 4 + 445)
        )
    if self.text_chat:
        for i in range(len(self.chat)):
            self.text_surface.blit(
                self.points_font.render(self.chat[i], False, (199, 199, 199)),
                (880, 800 - i * 20),
            )
        if self.buttons[3].send:
            client.client.data_queue.append(["chat", self.buttons[3].text])
            self.chat = [f"YOU: {self.buttons[3].text}"] + self.chat
            if len(self.chat) > 5:
                del self.chat[-1]
            self.buttons[3].text = ""
            self.buttons[3].send = False
```

```
if self.timer:
    self.text_surface.blit(
        self.time_text.render(
            self.time_format_translator(int(self.time_duration[self.player1])),
            False,
            (199, 199, 199) if self.turn == self.player2 else (199, 55, 55),
        ),
        (940, 20),
    )
    self.text_surface.blit(
        self.time_text.render(
            self.time_format_translator(int(self.time_duration[self.player2])),
            False,
            (199, 199, 199) if self.turn == self.player1 else (199, 55, 55),
        ),
        (940, 930),
    )
if self.draw_count > 0:
    if self.text_chat:
        self.text_surface.blit(
            self.points_font.render(
                f"accept draw? turns left: {self.draw_count}",
                False,
                (199, 55, 55),
            ),
            (900, 430),
        )
    else:
        self.text_surface.blit(
            self.points_font.render(
                f"accept draw? turns left: {self.draw_count}",
                False,
                (199, 55, 55),
```

```
                ),
                (900, 530),
            )
        for i in self.buttons[:-1]:
            i.update(pygame.mouse.get_pos(), pygame.mouse.get_pressed()[0])

        surface.blit(self.text_surface, (0, 0))
        self.render_end_game(surface)

    def send_resign(self):
        import client_server.client as client

        client.client.data_queue.append(["resign"])
        self.resign[self.player2] = True

    def send_draw(self):
        import client_server.client as client

        if self.draw_count > 0:
            self.game_draw = True
            client.client.data_queue.append(["draw_accepted", self.draw_attempt])
            return
        self.draw_attempt += 1
        client.client.data_queue.append(["draw", self.draw_attempt])
```

3.22 single_player.py file

```
import copy
import functools
import threading
from menus.single_player_options import SinglePlayerOptions
import common
from abstract_data_types import stack as stack
from menus.game_menu import GameMenu
```

```
class SinglePlayer(GameMenu):
    def __init__(self, size, colour, difficulty, timer, team, time_duration):
        import buttons.buttons as button

        super().__init__(size, colour, team, time_duration, timer)
        self.game_stack = stack.Stack(200)
        self.best_move = [False, [(-1, -1), (-1, -1), 0]]
        self.difficulty = difficulty
        self.transpos_tables = common.open_json()
        if self.player1:
            thread = threading.Thread(
                target=self.minimax_thread,
                daemon=True,
                args=(
                    self.board_game.board,
                    self.COMP,
                    -9999,
                    9999,
                    self,
                ),
            )
            thread.start()
        self.buttons = [
            button.Button(
                [self.size[0] // 2 + 425, self.size[1] // 4 - 140],
                (320, 180),
                self.main_game.window,
                self,
                "Options",
                functools.partial(
                    self.main_game.menu_stack.push,
                    SinglePlayerOptions(self.size, self.colour, self),
```

```
        ),
    ),
    button.Button(
        [self.size[0] // 2 + 425, self.size[1] // 4 + 70],
        (320, 180),
        self.main_game.window,
        self,
        "Draw",
        self.single_player_draw,
    ),
    button.Button(
        [self.size[0] // 2 + 425, self.size[1] // 4 + 280],
        (320, 180),
        self.main_game.window,
        self,
        "Resign",
        self.single_player_resign,
    ),
    button.Button(
        [self.size[0] // 2 + 425, self.size[1] // 4 + 490],
        (320, 180),
        self.main_game.window,
        self,
        "Rewind",
        self.rewind_stack,
    ),
    button.Button(
        [self.size[0] // 2, self.size[1] // 4 + 280],
        (320, 180),
        self.main_game.window,
        self,
        "Quit",
        self.game_quit,
```

```
            ),
        ]

    def single_player_draw(self):
        """
        checks whether the minimax is winning and if so, the draw is declined, if not, the draw is accepted
        """
        if (
            self.board_game.teamdict[self.player1]
            - self.board_game.teamdict[self.player2]
            < 0
        ):
            self.game_draw = True

    def single_player_resign(self):
        """
        used as a callback to end the game in favour of the minimax
        """
        self.resign[self.player2] = True

    def rewind_stack(self):
        """
        sets the game board to the previous item in the game stack object to take back a turn
        """
        if self.turn == self.player2:
            previous_frame = self.game_stack.pop()
            if previous_frame is not None:
                self.board_game = copy.deepcopy(previous_frame)
```

3.23 client.py file

```
import pickle
import socket


close_join_connection = False
```

```
start_multi_player_game = [False]



class Client:
    def __init__(self, ip, port, main_game):
        self.data_queue = []
        self.data_received = []
        self.data_sent = []
        self.s = socket.socket()
        self.ip = ip
        self.port = port
        self.connected = False
        self.main_game = main_game


    def join_game(self):
        """
        connects the client to the requested IP address and port.
        Sends the first item in 'self.data_queue' then waits for a 'received' message until it deletes the item from
'self.data_queue'.
        If the data received is not None, then it send a 'received' message to the client.
        """
        global close_join_connection
        global start_multi_player_game
        self.s.connect((self.ip, self.port))
        self.connected = True
        while True:
            try:
                data = pickle.loads(self.s.recv(1024))
                if close_join_connection:
                    close_join_connection = False
                    raise ConnectionResetError


                try:
```

```
            if data[0] == "START":
                start_multi_player_game = [True, *(data[1:])]
        except IndexError:
            pass
        if data == "END":
            self.main_game.menu_stack.pop()
            break
        try:
            if data == "received" and self.data_queue[0] in self.data_sent:
                del self.data_queue[0]
                data = []
        except IndexError:
            pass
        if data not in self.data_received and data:
            self.data_received.append(data)
        try:
            self.s.send(
                pickle.dumps(
                    self.data_queue[0]
                    if self.data_queue
                    else ("received" if data else [])
                )
            )
        except IndexError:
            self.s.send(pickle.dumps("received" if data else []))
        if self.data_queue:
            self.data_sent.append(self.data_queue[0])

    except ConnectionResetError:
        break

self.connected = False
self.s.close()
```

client: Client

3.24 server.py file

```
import threading
import pickle
import socket
import random
import sys


close_server_connection = False



class Server:
    def __init__(self):
        self._ip = self.get_ip()
        self.port = random.randint(3456, 12345)
        self._clients = []


    def get_ip(self):
        """
        connects to google's main dns and returns the local ip address bound to the connection.
        """
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
            s.connect(("8.8.8.8", 80))
            return s.getsockname()[0]


    def listen_connect(self):
        """
        listens for a client connection in a thread.
        If a connection is received, a new thread is started with the client and a 'ClientObject' is instantiated
        """
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
```

```
        s.bind(("", self.port))
        s.listen(5)
        while True:
            # waits for connection
            c, addr = s.accept()
            client = ClientObject(c)
            thread = threading.Thread(
                target=client.send_data, args=(self,), daemon=True
            )
            thread.start()
            self._clients.append(client)

    def broadcast_data(self, data):
        """
        send the given data to all clients
        """
        for i in self._clients:
            i.data_queue.append(data)

    def get_clients(self):
        """
        returns a list of all 'ClientObject' objects
        """
        return self._clients

    def del_client(self, client):
        """
        removes the given client from the client list
        """
        self._clients.remove(client)


server: Server
```

```
class ClientObject:
    def __init__(self, p):
        self.p = p
        self.data_queue = []
        self.data_received = []
        self.data_sent = []
        self.received = False


    def send_data(self, server):
        """
        sends the first item in 'self.data_queue' then waits for a 'received' message until it deletes the item from
'self.data_queue'.
        If the data received is not None, then it send a 'received' message to the client.
        """
        while True:
            if self.received:
                self.p.send(pickle.dumps("received"))
                self.received = False
                continue
            if close_server_connection:
                self.p.send(pickle.dumps("END"))
                server.del_client(self)
                sys.exit()
            try:
                self.p.send(pickle.dumps(self.data_queue[0] if self.data_queue else []))
                if self.data_queue and not self.received:
                    self.data_sent.append(self.data_queue[0])
                data = pickle.loads(self.p.recv(1024))
                if data and data not in self.data_received:
                    self.received = True
                    self.data_received.append(data)
```

```
                server.get_clients()[
                    int(not server.get_clients().index(self))
                ].data_queue.append(data)
            try:
                if data == "received" and self.data_queue[0] in self.data_sent:
                    try:
                        del self.data_queue[0]
                    except IndexError:
                        pass
            except IndexError:
                pass
        except (ConnectionResetError, EOFError):
            server.del_client(self)
            self.p.close()
            break
```

3.25 buttons.py file

```
import pygame


pygame.init()



class Button:
    def __init__(
        self,
        pos,
        size,
        surface,
        menu,
        text,
        func,
        image=None,
        text_colour=None,
        text_size=64,
```

```
):
    self.pixel_font = pygame.font.Font("buttons/pixel_font.ttf", text_size)
    self.img = pygame.image.load("buttons/button_background.png").convert_alpha()
    self.img_hovered = pygame.image.load(
        "buttons/button_background_hover.png"
    ).convert_alpha()
    self.surface = surface
    self.menu = menu
    self.text = text
    self.text_surface = self.pixel_font.render(
        self.text,
        False,
        (199, 199, 199) if text_colour is None else text_colour,
    )
    self.callback = func
    self.pressed = False
    self.size = (size[0], size[1])
    self.pos = [
        pos[0] - self.size[0] / 2,
        pos[1] - self.text_surface.get_height() / 2,
    ]
    self.mouse_hovered = False
    self.button_image = image


def check_collide(self, mouse_pos, mouse_pressed):
    """
    checks various states of the button and mouse collision setting 'self.pressed' accordingly
    """
    if (
        self.pos[0] + self.size[0] > mouse_pos[0] > self.pos[0]
        and self.pos[1] + self.size[1] > mouse_pos[1] > self.pos[1]
    ):
        self.mouse_hovered = True
```

```
        else:
            self.mouse_hovered = False
        if mouse_pressed and self.mouse_hovered and not self.pressed:
            self.pressed = True
            self.pos[1] += 5
        if not self.mouse_hovered and self.pressed:
            self.pressed = False
            self.pos[1] -= 5
        elif self.pressed and not mouse_pressed and self.mouse_hovered:
            self.pressed = False
            self.pos[1] -= 5
            self.callback()

        return

    def update(self, mouse_pos, mouse_pressed):
        """
        draws the button to the display with either the default button background or, a specific background
passed as an argument
        """
        if self.button_image is None:
            self.surface.blit(
                pygame.transform.scale(
                    self.img if not self.mouse_hovered else self.img_hovered, self.size
                ),
                self.pos,
            )
            self.surface.blit(
                self.text_surface,
                (
                    (self.pos[0] + self.size[0] / 2)
                    - self.text_surface.get_width() / 2,
                    (self.pos[1] + self.size[1] / 2)
```

```
                - self.text_surface.get_height() / 2,
            ),
        )
    else:
        self.surface.blit(
            pygame.transform.scale(self.button_image, self.size), self.pos
        )
    self.check_collide(mouse_pos, mouse_pressed)


class TextBox:
    def __init__(
        self,
        pos,
        size,
        surface,
        menu,
        text_colour=None,
        max_len=12,
        text_size=64,
    ):
        self.img = pygame.image.load("buttons/button_background.png").convert_alpha()
        self.img_hovered = pygame.image.load(
            "buttons/button_background_hover.png"
        ).convert_alpha()
        self.pixel_font = pygame.font.Font("buttons/pixel_font.ttf", text_size)
        self.text = ""
        self.size = size
        self.text_colour = text_colour
        self.text_surface = self.pixel_font.render(
            self.text,
            False,
            (199, 199, 199) if text_colour is None else text_colour,
```

```
        )
        self.pos = [
            pos[0] - self.size[0] / 2,
            pos[1] - self.text_surface.get_height() / 2,
        ]
        self.surface = surface
        self.menu = menu
        self.mouse_hovered = False
        self.pressed = False
        self.max_len = max_len

    def check_collide(self, mouse_pos, mouse_pressed):
        """
        checks various combinations of the mouse position over the image and whether LMB is being pressed or
not and sets 'self.pressed' accordingly
        """
        if (
            self.pos[0] + self.size[0] > mouse_pos[0] > self.pos[0]
            and self.pos[1] + self.size[1] > mouse_pos[1] > self.pos[1]
        ):
            self.mouse_hovered = True
        else:
            self.mouse_hovered = False
        if mouse_pressed and self.mouse_hovered and not self.pressed:
            self.pressed = True
            self.pos[1] += 5
        if not self.mouse_hovered and self.pressed and mouse_pressed:
            self.pressed = False
            self.pos[1] -= 5

    def update(self, mouse_pos, mouse_pressed):
        """
        processes the keys entered and transfers them into text
```

```
"""
try:
    if self.pressed:
        if self.menu.event.key == pygame.K_BACKSPACE:
            self.text = self.text[:-1]
            self.menu.event = ""
        elif self.menu.event.key == pygame.K_RETURN:
            self.pressed = False
            self.pos[1] -= 5
            self.menu.event = ""
        elif self.pressed and len(self.text) < self.max_len:
            self.text += self.menu.event.unicode
            self.menu.event = ""
except AttributeError:
    pass
self.text_surface = self.pixel_font.render(
    self.text,
    False,
    (199, 199, 199) if self.text_colour is None else self.text_colour,
)
self.surface.blit(
    pygame.transform.scale(
        self.img if not self.mouse_hovered else self.img_hovered, self.size
    ),
    self.pos,
)
self.surface.blit(
    self.text_surface,
    (
        (self.pos[0] + self.size[0] / 2) - self.text_surface.get_width() / 2,
        (self.pos[1] + self.size[1] / 2) - self.text_surface.get_height() / 2,
    ),
)
```

```
        self.check_collide(mouse_pos, mouse_pressed)



class ChatBox(TextBox):
    def __init__(
        self,
        pos,
        size,
        surface,
        menu,
        text_colour=None,
        max_len=12,
        text_size=64,
    ):
        super().__init__(pos, size, surface, menu, text_colour, max_len, text_size)
        self.send = False

    def update(self, mouse_pos, mouse_pressed):
        """
        processes the keys entered and transfers them into text.
        if enter is pressed, 'self.send' is set to true so the message can be sent
        """
        try:
            if self.pressed:
                if self.menu.event.key == pygame.K_BACKSPACE:
                    self.text = self.text[:-1]
                    self.menu.event = ""
                elif self.menu.event.key == pygame.K_RETURN:
                    self.pressed = False
                    self.pos[1] -= 5
                    self.send = True
                    self.menu.event = ""
                elif self.pressed and len(self.text) < self.max_len:
```

```
                self.text += self.menu.event.unicode
                self.menu.event = ""
        except AttributeError:
            pass
        self.text_surface = self.pixel_font.render(
            self.text,
            False,
            (199, 199, 199) if self.text_colour is None else self.text_colour,
        )
        self.surface.blit(
            pygame.transform.scale(
                self.img if not self.mouse_hovered else self.img_hovered, self.size
            ),
            self.pos,
        )
        self.surface.blit(
            self.text_surface,
            (
                (self.pos[0] + self.size[0] / 2) - self.text_surface.get_width() / 2,
                (self.pos[1] + self.size[1] / 2) - self.text_surface.get_height() / 2,
            ),
        )
        self.check_collide(mouse_pos, mouse_pressed)


class ToggleButton(Button):
    def __init__(
        self,
        pos,
        size,
        surface,
        menu,
        image=None,
```

```
        image_hover=None,
        text_colour=None,
        text_size=64,
        on_text="ON",
        off_text="OFF",
        on_colour=(199, 199, 199),
        off_colour=(199, 199, 199),
    ):
        super().__init__(
            pos,
            size,
            surface,
            menu,
            image,
            image_hover,
            text_colour,
            text_size,
        )
        self.on_text = on_text
        self.off_text = off_text
        self.on_colour = on_colour
        self.off_colour = off_colour
        self.count = 0
        self.text_surface = self.pixel_font.render(
            self.off_text,
            False,
            self.off_colour,
        )

    def check_collide(self, mouse_pos, mouse_pressed):
        """
        checks various combinations of the mouse position over the image and whether LMB is being pressed or
not and sets 'self.pressed' accordingly
```

```
"""
if (
    self.pos[0] + self.size[0] > mouse_pos[0] > self.pos[0]
    and self.pos[1] + self.size[1] > mouse_pos[1] > self.pos[1]
):
    self.mouse_hovered = True
else:
    self.mouse_hovered = False
if (
    mouse_pressed
    and self.mouse_hovered
    and not self.pressed
    and self.count > 10
):
    self.count = 0
    self.pressed = True
    self.text_surface = self.pixel_font.render(
        self.on_text,
        False,
        self.on_colour,
    )
    self.pos[1] += 5
if self.mouse_hovered and self.pressed and mouse_pressed and self.count > 10:
    self.count = 0
    self.pressed = False
    self.text_surface = self.pixel_font.render(
        self.off_text,
        False,
        self.off_colour,
    )
    self.pos[1] -= 5
return
```

```
    def update(self, mouse_pos, mouse_pressed):
        """
        draws the button to the display with either the default button background or, a specific background passed as an argument
        """
        self.count += 1
        if self.button_image is None:
            self.surface.blit(
                pygame.transform.scale(
                    self.img if not self.mouse_hovered else self.img_hovered, self.size
                ),
                self.pos,
            )
            self.surface.blit(
                self.text_surface,
                (
                    (self.pos[0] + self.size[0] / 2)
                    - self.text_surface.get_width() / 2,
                    (self.pos[1] + self.size[1] / 2)
                    - self.text_surface.get_height() / 2,
                ),
            )
        else:
            self.surface.blit(
                pygame.transform.scale(self.button_image, self.size), self.pos
            )
        self.check_collide(mouse_pos, mouse_pressed)
```

3.26 stack.py file

```
class Stack:
    def __init__(self, max_len):
        self.maxlen = max_len
        self.stack = [None] * max_len
        self.top = -1
```

```
def is_empty(self):
    """
    checks if stack is empty
    """
    return self.top == -1


def is_full(self):
    """
    checks if stack is full
    """
    return self.top == self.maxlen


def push(self, item):
    """
    adds a new item to the top of the stack
    """
    if not self.is_full():
        self.top += 1
        self.stack[self.top] = item


def pop(self):
    """
    removes the top item from the stack
    """
    if not self.is_empty():
        item = self.stack[self.top]
        self.stack[self.top] = None
        self.top -= 1
        return item


def peek(self):
    """
```

```
        returns the top item of the stack
        """
        return self.stack[self.top] if not self.is_empty() else "empty"

    def display_stack(self):
        """
        returns the stack as a list, so that it can be processed
        """
        return self.stack
```

3.27 common.py file

```
import json

offsetx = 30
offsety = 110
mult = 100


def open_json():
    """
    opens the transposition tables json file and returns the data it loads
    """
    transpos_tables = {}
    with open("transposition_tables.json", "r") as f:
        json_tables = json.load(f)
    for j, i in enumerate(json_tables):
        for k in range(len(json_tables[j][0])):
            json_tables[j][0][k] = tuple(json_tables[j][0][k])
        for k in range(len(json_tables[j][1])):
            try:
                json_tables[j][1][k] = tuple(json_tables[j][1][k])
            except TypeError:
                pass
        json_tables[j][0] = tuple(json_tables[j][0])
```

```
            json_tables[j][1] = tuple(json_tables[j][1])
            json_tables[j] = tuple(json_tables[j])
            transpos_tables[json_tables[j][0]] = json_tables[j][1]
        return transpos_tables


def save_json(transpos_tables):
    """
    opens the transposition tables json file and dumps the data from the 'transpos_tables' dictionary
    """
    json_tables = []
    for i in transpos_tables.keys():
        json_tables.append([i, transpos_tables[i]])
    for j in range(len(json_tables)):
        try:
            json_tables[j] = list(json_tables[j])
            for k in range(len(json_tables[j][0])):
                json_tables[j][0][k] = list(json_tables[j][0][k])
            for k in range(len(json_tables[j][1])):
                json_tables[j][1][k] = list(json_tables[j][1][k])
        except TypeError:
            pass
    with open("transposition_tables.json", "w") as f:
        json.dump(json_tables, f)
```

4 Testing – VIDEO LINK: tinyurl.com/ye25bj9k

Objective	Expected result	Actual result	Outcome	Timestamp
All button types move as indented.	Standard buttons should move down when pressed and up when not pressed, they should darken when the mouse hovers over them and lighten when it doesn't. Toggle buttons should move down when pressed and stay in that position, until they are pressed again, in which case, they will move back up. These should also darken when hovered over and lighten when not. Text boxes move down when pressed and are ready to type in. typing works correctly, and backspace works to delete previously entered text. These too should darken when hovered over and lighten when not. Text entered should be perfectly centred in the middle of the box.	All buttons move as expected with each moving down when the mouse is being pressed and is hovering over the button however, up when not.	Successful	00:00
"Single player" button Works correctly.	Display switches to show the "single player menu" rather than the "main menu" which allows you to access all single player settings.	The display switched to the "single player menu" where you have buttons for, timer, team, play and back.	Successful	00:33
"Single player menu" toggle buttons work correctly.	Buttons should move up and down with the text changing to indicate that it has been pressed.	When the timer button is enabled, the timer is enabled. and the team button correctly links with the team in the single player game. The button also moves down when activated and up when deactivated	Successful	00:36

“Play” button on “single player menu” works correctly.	The button should switch the display to show either the “single player timer menu” or the “single player difficulty menu” depending on whether the timer button is enabled. the team selected on the “single player menu” should now be seen on the board.	If the timer is enabled, the “single player timer menu” is shown, and if not, the “single player difficulty menu” is shown with the team selected.	Successful	00:43
Time duration buttons work as intended.	Each timer button should switch the display to show the “difficulty settings menu” and carries the time value for the button pressed.	The “difficulty settings menu" opens with the correct time selected.	Successful	00:51
“Difficulty settings menu” buttons work as expected.	Each button should switch the display to show the “single player game menu” using the team indicated previously and the timer if it is enabled.	The single player game started with the given difficulty (1,2 or 3 for “Beginner” “Intermediate” and “Master” respectively) and, if enabled, translates the time into minutes and seconds in the form “00:00”.	Successful	01:11
“Options” button on the “Single player game menu” works as intended.	The display should switch to the “single player options menu” with buttons, “theme”, “difficulty”, “back” and “menu”.	The “single player options menu” opens correctly with all buttons visible.	Successful	01:39
“Theme” button on the “single player options menu” works correctly.	The game display should switch to the “theme” menu and show all 3 theme settings.	The “theme” menu opens as it should, showing the “red”, “green” and “blue” buttons to change the main colour theme of the program.	Successful	01:42
The theme buttons, “red”, “green” and “blue” work as expected.	Each button should change the main “background colour” and “backboard colour” to universally set the colour theme.	The “red”, “green” and “blue” buttons all change the main theme to their colours respectively, which can be seen across the whole program.	Successful	01:42

The “back” button on the “theme” menu works as it should	This should revert the menu back to the “single player options menu”.	This works as expected with buttons, “theme”, “difficulty”, “back” and “menu” now showing.	Successful	01:51
“Difficulty” button on the “single player options menu” works correctly.	The game display should switch to the “Difficulty settings” menu and show all difficulty settings.	All 3 difficulty settings buttons, “beginner”, “intermediate” and “master”, are shown	Successful	02:00
The difficulty settings buttons, “beginner”, “intermediate” and “master” work as expected.	Each button should change the single player difficulty to the respective value.	This works as expected, with “beginner” setting the difficulty to “1”, “intermediate” setting it to “2” and, “master” settings it to “3”.	Successful	02:00
The “back” button on the “difficulty settings” menu works as it should.	This should revert the menu back to the “single player options menu”.	This works as expected with buttons, “theme”, “difficulty”, “back” and “menu” now showing.	Successful	02:27
The “back” button on the “single player options menu” works correctly.	This should revert to the “single player game” menu, where the game should have been paused.	This works as intended. It returns to the “single player game menu” and the timer resumes. If a move is being processed by the minimax, then it will continue to process while on another menu. This removes any possible unnecessary waiting time.	Successful	02:14
The “menu” button on the “single player options menu” works correctly.	This should revert to the “single player menu”, so that a new game can be started.	This works as expected. It reverts to the “single player menu” allowing settings to be chosen to start a new game.	Successful	02:31
The single player “draw” button works as intended.	When pressed, this should check if the minimax is losing, if so, then the draw will be accepted, if not, it will be rejected.	This works as it should, checking the balance of piece values across both sides and accepting the draw is the player is winning.	Successful	02:36

The single player “resign” button works as intended.	When pressed, this should end the game showing a loss screen for the player.	This works as intended, showing a loss screen with the text “defeat” and “resign”.	Successful	02:50
The single player “rewind” button works correctly.	When pressed, this should revert the board to the previous frame in the stack.	This works correctly and reverts the board back to the previous frame in the game stack.	Successful	02:56
The pawn piece moves correctly.	This should be able to move forward one square, but 2 on its starting square. It should also be able to move diagonally when taking an opponent’s piece.	The pawn piece moves as expected.	Successful	3:09
The bishop piece moves as expected.	The bishop should be able to move diagonally in any free spaces, unless disrupted by a friendly piece or an enemy piece, in which case, it can take the enemy piece.	The bishop piece moves as expected.	Successful	03:14
The rook piece moves as intended.	The rook should be able to move vertically and horizontally in any free spaces, unless disrupted by a friendly piece or, an enemy piece, in which case, it can take the enemy piece.	The rook piece moves as expected.	Successful	03:21
The queen piece moves as it should.	The queen should be able to move diagonally, horizontally, and vertically, in any free spaces unless, disrupted by a friendly piece or, an enemy piece, in which case, it can take the enemy piece.	The queen piece moves as expected.	Successful	03:28
The King piece moves as expected.	The king should be able to move diagonally, horizontally, and vertically, in any free spaces by one square unless, it is disrupted by a friendly piece or, an enemy piece, in which case, it can take the enemy piece.	The king piece moves as expected.	Successful	03:36

The knight piece moves as intended.	The knight should be able to move in an "L" shape, with any version of, 2 spaces in one direction and 1 space in another.	The knight piece moves as expected.	Successful	03:43
Taking enemy pieces works as expected.	When a piece is taken, it should be removed from the board and shown in its opponent's "taken pieces area".	Each piece can be taken as expected. The difference in taken pieces value, is found, and shown on the winning player's side of the board.	Successful	03:09
Checkmate works as expected.	When the enemy king is in check (king piece is under attack) and, there are no moves to get out of check, then they have been checkmated.	Checkmate works as expected, showing an ending screen of either, "Victory Checkmate" or "Defeat Checkmate".	Successful	03:51
Stalemate works as expected.	when the enemy king is not in check, but there are no legal moves (moves that don't place or keep the king in check).	Stalemate works as expected, showing an ending screen of "Stalemate"	Successful	03:59
Pieces move as expected.	When a piece is pressed, it should increase in size and draw its legal moves on the board as small dots. The piece that is being picked up, should also "snap to" and be drawn on the nearest square which is a legal move. A piece should also not be able to be picked up if it is not the players turn; the player should also never able to pick up the opponents pieces.	All pieces function as they should, with all movement working correctly.	Successful	03:09
The timer Functions properly.	When it is a players turn, the timer on their side, should change to a red colour and the timer should start ticking down. This indicates that it is this players turn to move. The timer should also not tick down when in another menu.	The timer works as expected, not ticking down when paused and only ticking down on the players turn. If one of the players timers goes to 0, then it will display "Defeat Out Of Time" on their screen and, "Victory Out Of Time" on the opponent's screen.	Successful	04:05

The minimax and quiescent search work correctly.	If the time taken between moves increases with a higher difficulty set, then we know that the minimax is searching to a greater depth and is therefore more difficult to play against. So, we should expect a time per move increase when the difficulty is increased.	This works as expected however, with the “master” difficulty on early moves, the quiescent search has no attacking or checking moves. This means that, the time will be close to that of the “intermediate” difficulty early in the game.	Successful	04:16
The transposition tables work as expected.	After a position and move is stored in the transposition tables, the next time that position is played, the move should be played instantly.	This functions as intended. After a move is played, the previous position and move are stored in the “transposition tables” JSON file which is later searched in each position to play the move much quicker.	Successful	04:37
The “Multiplayer” button on the “main menu” works as intended.	The display should switch from the “main menu” to the “multiplayer” menu.	This works as it should where the display switches menus to show, “new game”, “join game” and “back”.	Successful	04:48
The “new game” button on the “multiplayer” menu functions properly.	this should switch the display from the “multiplayer” menu to the “new game” menu.	This works as intended, showing the “new game” menu. This displays the local IP of the server and the port for the connection, so that the client can connect. This also creates both, a client, and a server instance.	Successful	04:52

“Multiplayer settings” menu works as intended.	This should wait for the second clients connection before creating a “start game” button for switching from the “new game” menu to the “multiplayer settings” menu.	This works as intended, waiting for a connection, and opening the menu. This menu then gives the ability to enable or disable the “text chat” or the “timer”. This also allows the creator of the game to select their team. If the timer is not enabled, the “multiplayer game” menu will open with the team chosen and whether the text chat is enabled. This will then relay that information as well as, a “start” message to the opponent, so that they can set up their game accordingly.	Successful	4:57
“Multiplayer timer menu” works as it should.	Each timer button should switch the display to show the “multiplayer game” menu and carries the time value for the button pressed as well as the team chosen and whether the text chat is enabled.	Each timer button opens the “multiplayer game" menu with the given time, team and whether text chat has been enabled. This then sends a “start” message to the opponent, so that the game starts.	Successful	05:07
Multiplayer “draw” works correctly.	When pressed, the draw button should send a “draw” message to the opponent, at which point, they have 3 turns to accept the draw.	This functions as expected and sends a draw message to the opponent. If the recipient of the draw presses the draw button before they have played 3 turns, then the draw will be accepted, and the game will end.	Successful	05:12

Multiplayer "resign" works as it should.	When pressed, the resign button should send a "resign" message to the opponent, in which case, the game will end.	This works as it should with the person who resigned, having "Defeat Resign" shown on the display and their opponents game showing "Victory Resign".	Successful	05:57
"Text chat" functions as intended.	Text chat should show "YOU: message" for the player's own messages and "OPPONENT: message" for the opponent's messages.	The text chat works as it should and, doesn't allow the input of a message longer than 14 characters, to ensure that the message always fits inside the text box. When the enter key is pressed,	Successful	06:08
Moves on a multiplayer game are sent across successfully.	When a move is played, A message containing, "move", the position of the piece, and the position the piece has moved to.	This works correctly, with each move being translated to fit the flipped board when it reaches the other client.	Successful	06:54
Multiplayer timer stays synced across both clients correctly.	The timer should be consistent across both clients.	The multiplayer timer works as it should, not going out of sync with the opponents device.	Successful	07:47
"Join game" button works as intended.	This should switch the display to the "join game" menu where the IP address and port can be entered as show on the server host's display.	This works as intended allowing a player to join a game given an IP address and port.	Successful	08:30

"Join game" textboxes

objective – handle invalid inputs.

Input	Expected Result	Actual Result	Outcome	Timestamp
"12345678901234567890"	It shouldn't allow me to enter this as it is too long.	The maximum character length on the IP text box is 15 and 5 on the port text box.	Successful	08:35
"qwerty"	When the "Join" button is pressed, this should display "Invalid data type entered!" to indicate that is an invalid IP address or port.	This works as expected, displaying "Invalid data type entered!" and, "disconnected" If "join is pressed. "text if a valid port and invalid IP is entered.	Successful	08:56

“__++123-*/”	When entered, this should display “Invalid data type entered!” as this doesn’t fit in either the IP or port text box.	This works correctly, displaying “Invalid data type entered!”.	Successful	09:14
“ ”	If nothing is entered and the “join” button is pressed, there it should do nothing. This is to indicate that a connection cannot be made.	This works as intended, doing nothing to indicate an invalid IP address and port.	Successful	09:32
“255.255.255.255”	This should do nothing as it is a valid IP address.	This works correctly, displaying nothing other than the entered IP address.	Successful	09:41

Objective	**Expected result**	**Actual result**	**Outcome**	**Timestamp**
“Join” button on “join game” menu works as it should.	When correct IP and port are entered on “join game” menu, and “join” button is pressed, the main game should start.	This works as intended. switching the display to the “multiplayer game” menu when the “start” message is received.	Successful	09:53
Multi player “menu” and “theme” buttons work as they do in single player.	The ”menu” and “theme” buttons should open the “multiplayer options menu” and open the “theme” menu respectively.	This works correctly, opening the “multiplayer options” menu and “theme” menu.	Successful	11:40
Client-server networking functions correctly.	Both clients connect to the server and all moves, draws, and resigns are sent across the network.	This works as intended, functioning fully as a local online multiplayer chess game.	Successful	09:53
Client-server works correctly over different operating systems.	This should work normally allowing both clients to connect to the server, with all messages being sent across the network.	This works as it should with no problems.	Successful	10:17
Tell the other client when the game has been quit.	When both clients are connected and one of the clients closes their game, the other client’s game will quit to the main menu to indicate	This works as it should, quitting when the opponent disconnects.	Successful	11:54

	that their opponent has disconnected.			
"Options" button on "main menu" works as intended.	This should open the "main options" menu and give access to the "theme" and "back" buttons.	This works as intended, with the "theme" button opening the "theme" menu and the "back" button switching the display to show the "main menu".	Successful	12:00
"Quit" button on main menu works as it should.	This should quit the game and close the application.	This works as intended closing the application and quitting the game.	Successful	12:18

5 Evaluation – 5.1 Objectives

Main objective – The program should have a working chess game with all expected features.

Objective	Outcome	Comments
When a piece is selected, all the possible legal moves for that piece should be displayed on the board, so that the player knows where they can move.	Completed.	When the mouse is hovering over a piece and then is pressed, the "check_move" method will find all the possible moves for that piece. This will then call the "legal_moves" method to check if the moves found by "check_move" are legal. This will then draw a dot on all squares that are legal moves. The piece is also drawn on the nearest square, with a legal move dot on it, to the mouse position.
When in a timed game, there should be a countdown clock for each player indicating how much time they have left.	Completed.	If enabled with the toggle button, each player has a countdown clock displaying their remaining time which starts at either 3 minutes, 5 minutes, or 10 minutes. The timer should turn red and tick down on the side of whosevers turn it is. Then, when they make a move, their timer will stop, turn white, and the opponents timer will start to tick down.
After each piece is captured, it should be displayed on the opponents side.	Completed.	When a piece is taken, it is displayed on the opponents side in order of piece value.
The difference in total score of each side, should be displayed on the winning player's side.	Completed.	The difference in point balance is shown on the winning players side to indicate how much they are winning by e.g. "+8".
When the mouse hovers over each piece, there should be a shadow.	Completed.	While the mouse is hovering over a piece, there is a shadow shown behind the piece, indicating which piece will be selected if the mouse is pressed.

The main game should use a stack to hold all the menus, allowing only the top menu to be rendered and all previous menus to be saved.	Completed.	All menus are held on the stack. When a new menu is opened, it will be added to the top of the stack using the “push” method. Similarly, when the user wants to go to a previous menu (escape or back button is pressed), the top item on the stack will be removed and returned using the “pop” method. The menu at the top of the stack will always be rendered, this is accessed using the “peek” method.

Main objective – The program should use a computer-controlled opponent utilising the minimax algorithm.

Objective	Outcome	Comments
The minimax algorithm should be used to return an accurate evaluation of the position based on the possible following moves.	Completed.	In the running of the minimax algorithm, a tree will be created which allows for the searching of every possible move following the current position. With a depth of 3, the minimax will search through all its possible moves and after each of those moves, all the opponents retaliating moves, and then all its own moves after each retaliation move. Once an end node is reached (depth is equal to 0 or the game is over), an evaluation will be carried out with either the static evaluation method ”evaluate_state”, or if the position is unstable (lots of attacking or checking moves), then the “quiescent_search” method will be called to extend the search to a quiet position which will then be statically evaluated. Once the best move is found, it is returned up the call stack and the move is played.
The minimax should use alpha-beta pruning to remove any branches that aren’t going to be reached to reduce processing time.	Completed.	Alpha-beta pruning uses an alpha and beta value. The alpha value is set to the greatest score yet to be found for the maximising player and, the beta value is set to the greatest negative value found for the minimising player (this is negative because it is bad for the maximising player and all moves are evaluated from the maximising player’s perspective). The alpha and beta values are then compared where, if the beta value is less than the alpha value, (the position will never be reached because it appears to be so terrible for the maximising player) the branch is then cut.

The quiescence search algorithm should be used to extend the search of the minimax algorithm past an unstable position until a quiet position can be evaluated statically.	Completed.	When an active/unstable position is reached on an end node of the minimax tree, the "quiescent_search" method is called to extend the search until a quiet position is reached. This search will only look at moves that either capture a piece or put the opponent in check. This is because, these moves are more likely to lead to a quiet position in less moves. Once a quiet position is reached, the static evaluation of the position is returned.
The quiescence search should use alpha-beta pruning to remove any positions that aren't going to be reached to decrease processing time.	Completed.	Similarly to the alpha-beta pruning used on the minimax, the alpha-beta pruning here uses both an alpha and beta value, each corresponding to the best maximising score and, the best minimising score respectively. The branch is then also cut if the beta value is ever less than the alpha value.
Transposition tables are also used to save previously visited positions and, the best move found in each position.	Completed.	After each move is made by the minimax, the position is saved with the best move found in that position, inside a JSON file. When the game is first loaded, the JSON file contents are loaded into a dictionary relating each position to the best move. Then, when the minimax is next processing a position, it checks in the transposition tables to see if the position has been played before, if it has, the move relating to the position is then played, drastically decreasing processing time in early positions.
A draw button should be used, to allow players to draw with the minimax.	Completed.	If the minimax is losing, and the player presses the draw button, the game will end in a draw and a "draw" message will be displayed. If the minimax is winning or, the game is currently a tie, the draw button will do nothing. This is to simulate how a real player would act in most situations.
A resign button should be used, to allow players to resign when playing against the minimax.	Completed.	If the resign button is pressed, the game will end displaying the "defeat by resignation" message.

A rewind button should be used, to allow players to take back turns when playing against the AI.	Completed.	A stack is used to store each board instance, which includes captured pieces, point balance, piece positions and a reference to the game instance. The stack uses a ”pop” method to remove the top item and return it, a “push” method to add an item to the top of the stack as well as “is_full” and “is_emtpy” methods to check if the stack is full or empty respectively. The stack also uses a “peek” method to return the top item in the stack, and a “display_stack” method to list all items in the stack. After each turn, the current position is added to the top of the stack using the “push” method, where the top item is always being displayed. If the rewind button is pressed, the “pop” method is called to revert to the previous frame in the stack.

Main objective – The program should use a client-server network to allow local multiplayer across devices.

Objective	**Outcome**	**Comments**
The program should use a client server allowing local multiplayer across devices.	Completed.	Using the “Socket” python library I can establish and send data across a connection between devices. When data is sent, I use a keyword, so that the receiver knows what to do with the data e.g. “START” or “draw”. Whenever a move is played, a message is sent, a draw is offered, etc, the data indicating the action is sent. This data is then continuously sent until the sender receives a “received” message, at which point, it will stop sending the data. Within this network, I have used an array which acts a queue to store data sent and data received. The multi-threading python library “threading” has also allowed me to run each part of the client-server in separate threads. This means that the client server processing can happen in the background while the game runs as normal for the user.
There should be a draw button, that allows players to offer a draw to the opponent.	Completed.	When the draw button is pressed, the opponent receives a notification telling them how many turns they have left to accept the draw. If the opponent presses the draw button while in the draw accept period, then the game will end on both devices and the draw message will be displayed. If the draw window (3 turns) ends and the draw button has not been pressed, it will disappear and wait for another press on either side.

There should be a resign button, which allows the player to instantly concede to the opponent.	Completed.	When the resign button is pressed, data containing the keyword “resign” is sent to the opponent. At which point, both players games end and the resign message will be displayed.
There should be a text chat, which allows the players to communicate with each other during a game.	Completed.	The text chat allows players to type a message, and then press the enter key to send the message. The message is then sent with the keyword “chat”. The message will show in the chat as “YOU: message” if it is a message that you have sent or “OPPONENT: message” if it is a message that your opponent has sent. Only 5 messages are shown at a time and if any more are sent, the chat rolls up deleting the oldest message. The text chat also has a maximum input length of 14 to avoid messages overlapping the text box.
When a move is made by a user, the move is replicated on the opponents game.	Completed.	After each move, the current position and movement position are sent to the opponent with the keyword “move”. The move is then translated with the “move_translator” method before being played on the opponents board.

5.2 User feedback

How user-friendly is the program?

Users found that the simple, pixel-art layout was very easy to read and navigate. I also received multiple comments about the fluidity of each menu, and how well each flowed together. However, when in a game, if the player was using a trackpad, they often struggled moving the pieces a long distance without slipping. They recommended I use some form of confirmation system to ensure that the piece is being moved to the desired position. Some users also noted that, when in a game, they were worried about quitting by accident as the “back” and “menu” buttons were next to each other. Here, again I could use a confirmation button to quit the game to ensure that it isn’t done unintentionally.

Does the program meet both the objectives, and your expectations?

Simon, the third party, found that all his expectations were exceeded. He was also very pleased that all objectives had been met to a high standard. Simon said, “I love how easy it is to use, and how flawless it is!”. He mentioned that, during his testing of the program, there were no bugs, glitches, or defects, and was especially satisfied with how trouble-free the client-server multiplayer functionality was to use. He liked the use of the minimax, and how the difficulty increases gradually with each level. He also loved how when playing the same opening 2 games in a row, the minimax would respond instantly as it had remembered the best move in that position. Simon liked how well the turn-based draw mechanic on the multiplayer games worked and said that all other users found it very easy to use and understand. Simon also commented on how well both timers stayed in sync across devices, even when both users are on different operating systems. He also appreciated the surprise addition, which we hadn’t previously discussed, of sender names in the online text chat e.g. “YOU: message” and “OPPONENT: message”, so that it was easy to understand which messages were sent by who.

Do you have any criticisms of the program?

The third party felt that the game was missing an instruction set for new players to chess. This can cause the beginners to become confused. He mentioned that this could be added before the game starts and, be accessible while in a game. He also said that, for the best players in his club, the "Master" difficulty wasn't enough of a challenge and that a more challenging difficulty should be added for the very best of players.

Are there any other features that you would like to see?

One suggestion from Simon, was to add more special moves into the game e.g. "promotion" or "En passant". This would help to further enhance the playability of the chess game and create more interesting positions. One other suggestion was to add a slider for the timer rather than buttons for each timer setting. This would allow users to set the time to any value that suits them, oppose to sticking with the default values of 3, 5 and 10 minutes.

5.3 Analysis of user feedback

Overall, the feedback for the program was very positive. All users liked my implementation of the objectives and the overall game design. There were a few small suggestions to improve the difficulty of the minimax and the settings when going into each game. I also received some suggestions about the game featuring more special moves, to create more interesting possibilities. There were also very few criticisms about the general usage, GUI design and the client-server multiplayer. These seem to be favoured by most over other components.

5.4 Next steps

Moving forward, I will integrate all the feedback that I have gained from the third party and other users. This will allow me to better the general game experience and overall ease of use. To give the user more game customisability, I will add a slider to change the length of the timer. This way, they are not stuck to the 3, 5 and 10-minute set timer durations. On top of this, I will increase the difficulty of the minimax, on an extra "Expert" level. For this, I will have to further increase the maximum depth of the search to 4 or 5 including the quiescence search. I also intend on adding more special moves like "En passant" and "Promotion", to give the game a more featureful and polished feel. To overcome users accidently dropping their pieces in the wrong positions, I will add a "confirm move" feature. This will ask for confirmation after the move has been made before it is played and will be added as setting in the main options menu. This will also be used for the "menu" button to return to the main menu and quit the game. Overall, there are some minor improvements to be made to the general game experience, and once completed, I will return the revised game to Simon for him to introduce to the chess club once again.