\* TP2/3 update at the bottom

Project Description:

Name: Online Multiplayer Chess with AI

Description: A chess game where users can play with other users who are on the same local network. They can also to choose play with an AI if they don’t have a partner to play with.

Competitive Analysis:

Similar projects I’ve seen online includes multiplayer rock-paper-scissors or local Chess games. My project is different from all these firstly because it is far more complex than the typical multiplayer game that’s written with the socket module (Chess vs a game like rock-paper-scissors). Secondly, it takes a game like chess (something that lots of projects achieve locally) and enables multiple players to connect wirelessly.

Additionally, my project also includes an AI that players can choose to play against, which is something not normally included in similar projects online. A chess AI typically appears as its own standalone project.

Structural Plan:

The main game/local client will be in a file:

- chess.py:

-Where the chess board/pieces are drawn, the players make moves, and the rules are enforced.

-The positions of pieces are represented as a 2D 8\*8 list of tuples, each containing the color of the piece and the type of the piece. Stored as app.pieces.

-The sprite of the pieces are loaded in with cmu\_112\_graphics/pillow. They are then stored in two separate dictionaries (app.blackPieces/app.whitePieces) that maps the type of each piece to its corresponding cropped sprite.

-Valid moves that a user can make after selecting a chess piece is represented through a 2D 8\*8 list of Boolean values stored as app.validMoves.

The graphical interface will be achieved through:

-cmu\_112\_graphics.py, which will be imported in chess.py. A more detailed description of graphics is included with the above section

The local client and the server will communicate with the file:

-game.py:

-A file that contains the Game class, which stores vital game data accessible by both the server and the client.

-Every two players would share one game object

-The object stores which user has made a move in a variable self.went

-The object stores the game ready state as a Boolean value in self.ready (True when both players are connected

-The object stores the latest move made by either players in a tuple self.move, which is then accessed by the other player to update their local board

-The object stores a game over state as a Boolean value to indicate whether the game is finished (self.over)

The multiplayer functionality will be achieved through two files:

-network.py:

-A file that contains the Network class, which allows the user to connect to the server and send/receive data.

-Upon connecting, immediately receives the assigned game ID from the server

-Allows the user to use the send method to both send and receive data

-server.py:

-The script for running the server and sending/receiving data from local clients.

-This must be running the entire time anyone is playing the game

-Every time a new odd-number client connects, the server creates a new game object, which is stores in a “games” dictionary object. The games dictionary maps game IDs to game objects. The game ID is then passed in as a parameter when the server creates a new client thread. This is so that the server always knows which game object to send to which client.

-Every time a new even-number client connects, the server assigns it an existing game ID.

-The client thread would continuously try to receive data from the client. If no data is being received, connection would be broken. The server then uses the information received in the client thread to update the other local client in the same game.

Algorithmic Plan:

Complex parts of my project:

-Networking:

-Every time the user makes a move on their local board, the move is sent to the server in a tuple object that includes the current position (the piece they are about to move) and the destination.

-The move is then updated in the game object by the server. At the same time, the server updates the self. went variable in the game object so that the local client would know whether they should update this move on their local boards.

-The server sends the game object to everyone. The local client would read the self.went variable to decipher whether the move is made by an opponent and update the local board accordingly.

-All at the same time, the local client constantly tries to get a new game object from the server in timerFired so that the local board always stays up to date.

-More details are included in the structural plan part of this proposal

-Chess AI:

-The AI would be achieved through the minimax algorithm, which computes the value of the chess board based on the pieces that are on it and tries to either minimize or maximize that value.

-The AI would assume that the opponent plays optimally and try to search through to bottom states where it is able to achieve a checkmate

-Once a checkmate is reached, the AI would then go back to the surface most level and execute the move that could lead to such an outcome.

-Checkmate detection

-Achieved by implementing a “killzone” notion.

-The killzone for white pieces is stored in a 2D 8\*8 list of Boolean values app.blackKZ, while the killzone for black pieces is stored in a 2D 8\*8 list of Boolean values app.whiteKZ.

-When the user clicks on a piece, the killzone is outlined on the board. The killzone outlines are stored in a 2D 8\*8 list of Boolean values that is updated and cleared with every mouse click by doing a deepcopy of either the white or the black killzones, depending on which side the player is on.

-The king cannot move into a killzone

-If the king is in a killzone, the check condition would be set to True. (Probably stored in a Boolean variable app.check

-The client would then loop through every single possible move that can be made by the side whose king is in a killzone to see if any move can remove the king from the killzone.

-If no such move is possible, a checkmate is detected.

Timeline Plan:

By tp1 (11/18):

-Finish implementing a basic multiplayer chess game where users can connect and move their pieces

By 11/20:

-Finish implementing all chess features

By tp2 (11/23):

-Finish implementing a fully functional multiplayer chess game with an AI

After tp2:

-Implement any additional features as time allows

Version Control Plan:

I plan to use git and github to back up my project.

A screenshot of a computer screen

Description automatically generated with medium confidence

Module List:

socket

pickle

\_thread

TP2 Update:

-Networking algorithm changes:

-Special moves (En Passant, Castling, and Pawn Promotion) are achieved first on the local client, which then sends a message to the server notifying that a special move was completed. The server would then update the corresponding variable to represent the completion of a special move. Then, the server would send the game object to the other client, who would then recognize that a special move was completed and make adjustments on the local board accordingly.

-The local client now displays a landing page before connecting to the server, allowing the user to choose which side to play as.

-Upon first player’s connection, a new game object is created on the server. The game’s vacancy is represented by a list of two Booleans, where the first element represents the presence of a white player, and the second element represents the presence of a black player. The game only enters a ready state if both Booleans evaluate to True.

-For the second connecting player and everyone after that, the server scans through all the existing games to try to find a vacancy. If such vacancy exists, the new player would join an existing game. If not, the server would create a new game object for the player to wait in.

-Chess.py structural changes:

-Added different app modes to display the landing page, victory page, defeat page, pawn promotion page, waiting page, and disconnected page.

-The pieces are now loaded onto the board according to the side the player chooses, making it so that the player’s side is always on the bottom. The opponent’s moves are mirrored across the row so that they are reflected accurately on the local board.

TP3 Update:

-Checkmate detection changes:

-Added Zobrist hashing to significantly improve checkmate detection speed

-The randomly generated Zobrist table is generated once and stored in a pickle file and loaded into the program in the form of a 3D list at program start up (Line 876)

-Every time check mate detection runs, the hash function (Line 1359) reduces the current board into a hash code. The result from check mate detection is then stored inside a dictionary app.checkDict alone with the hash code. Every time check mate detection runs, app.checkDict is updated with the latest results and also saved permanently into a local file check.pkl.

-Upon app start, the saved check.pkl is loaded into the program in app.checkDict. Every time check mate detection runs, it first checks if the current board’s hash code is already in app.checkDict. If true, then it would simply return the saved result. If not, it would run check mate detection and update app.checkDict as well as check.pkl accordingly.

-Resizable window changes:

-Updated all size parameters to be relative to app.height and app.width so that users can resize their clients according to individual needs