CS246 Fall 2023 Project - Chess Plan

Plan of Attack

1. Implement the Piece class and subclasses of Piece (each piece class inherits from the base class piece and encapsulates its own behaviour - movement rules and capturing mechanics)
   Since each piece is a separate component on the board and does not have a direct relationship with any other components (each Piece only inherits from the Piece class), it makes more sense to implement each of the 6 pieces first with their own moves, and capturing patterns first. This approach makes debugging more simple, as each piece can be developed and tested without interfering with the game's logic.
   a. Piece () - Nov 25th, 2023
   b. King (Harish) - Nov 25th, 2023
   c. Queen (Harish) - Nov 25th, 2023
   d. Bishop (Brian) - Nov 25th, 2023
   e. Knight (Brian) - Nov 25th, 2023
   f. Rook (Ryan) - Nov 25th, 2023
   g. Pawn (Ryan) - Nov 25th, 2023
2. Implement the board (Ryan) - Nov 26th, 2023
   Since the board owns each piece, it creates an environment where all the actions (move and capture) would be executed. The board provides context for piece interactions, where each movement is affected by the layout and the constraints of the board. With the board implemented, basic game rules and the behaviour of each piece can be tested in a more realistic setting before adding more complex game logics.
3. Implement the Human class (Brian) - Nov 26th, 2023
   After defining the board and the game pieces, the next step is to introduce the entities that will interact with these components. The Human class represents the human players in the game, and implementing it allows for testing how players will interact with the game environment. This can be done by creating a simple test harness that can display the basic grid of the board and taking commands from standard input stream.
4. Implement CPU levels 1-3 (Ryan, Harish, Brian) - Nov 26, 2023
   a. Level1 (Ryan)
   b. Level2 (Harish)
   c. Level3 (Brian)
5. Implement the Game class (Harish) - Nov 27th, 2023
   Now that the players are implemented, it is possible to implement a bare bones Game class without a display, since the main game requirements have been implemented (e.g: Players, Board, Pieces). Can develop alongside TextDisplay and test the control flow alongside TextDisplay as it is being developed.
6. Implement the text display (Ryan) - Nov 27th, 2023

With the Game, Board, all Piece classes, and all Player classes complete, checking that the main chess concepts works correctly can be checked here.

7. Implement the graphics display (Brian) - Nov 28th, 2023
   After implementing the TextDisplay, we can implement GraphicsDisplay and use TextDisplay to test if the graphics are being properly displayed. Having a text display already in place allows for easier testing and verification of the graphics display. The text display can serve as a reference to ensure that the graphical representation accurately reflects the game state.
8. Implement CPU level 4 (Ryan, Harish, Brian) - Dec 3th, 2023
   This task is scheduled after the graphics implementation, as it is more related to the gameplay mechanics rather than the game's presentation. The involvement of all team members suggests this is a significant and complex task.
9. Create final design document (Ryan, Harish, Brian) - Dec 4th, 2023
   The team will work together to create the final design document for the project. This document likely includes details about the game's design, architecture, algorithms used for CPU, and descriptions of the implementation of various features.
   Placing this task towards the end of the timeline ensures that the document reflects the final state of the project, including any last-minute changes or decisions made during development.
10. Extra credit features (Ryan, Harish, Brian) - Dec 4th, 2023
    The extra credit features are to be completed last so that we will have a working chess game first and build the extra credit features upon it.
11. Submit all files to Marmoset (Brian) - Dec 5th, 2023

Question: Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves
and responses to opponents' moves, for the first dozen or so moves of the game. See for example https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

1. Data structure for storing openings
   The first step is to decide how to store the opening sequences. An n-ary tree or graph structure is ideal because chess openings are essentially a branching set of possibilities. Each node in the tree represents a board state, and each edge represents a move. For example:
    - The root node represents the initial board state.

   - The children of the root represent all possible moves for the first turn (each node contains a pointer to a next node, a string representing the next move, and an int representing the win percentage).
   - Each subsequent layer represents the board states after each subsequent move.
2. Integrating with Game Logic
   The opening book needs to be integrated into the game's logic:
   - When a game starts, the program should first check the opening book.
   - As moves are made, the program follows the path down the tree corresponding to those moves.
   - If the current game state matches a node in the tree, the program can use the children of that node to suggest or make the next move.
   - If the game deviates from the book (i.e., reaches a state not in the tree), the program switches to its standard move-making algorithm and creates a new node as a child of the previous node with the new move.
3. Choosing openings
   For choosing the best opening, the program can:
   - Randomly select an opening to add variety.
   - Use statistics (win percentage) to choose the opening with the best historical success rate
4. User Interface
   To accommodate the UI, we can extend the XWindow display and display the list of moves on the right side of the game board. The list of moves will be recalculated after every move through by traversing down the tree and obtaining its list of children nodes in the next level.

Question: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?
To undo one move, a struct could be implemented as a field for a Player, that stores two parallel vectors, one that holds the old state of Pieces that were changed by the move (Piece type, old coordinate), one that holds the new state of Pieces that were changed by the move, as well as integer to represent who's turn it is. A new user input can then be implemented that accepts "undo", and when it is used, the program will check if that player has a previous move defined. If the player does, it will go through the vector of Pieces that changed after the move, and replace them with NullPieces, and then add the pieces in the old state vector to the board.
For unlimited undos, the above idea can be extended to a vector that stores these structures. This vector will work like a stack. Every time undo is called, it will pop from the back of it, and every time a new move is done, it will emplace to the back of it.

Question: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

To extend the game to four players, the Game class would now have 4 Player objects instead of 2. Similarly, when taking input from the user, it will now loop through all four players. Since our Piece class already has a variety of colours for each player, we can choose two colours other than black and white for the new players. To accommodate for the larger board size, the vector of vectors of Pieces will be increased accordingly. The blank corners will be initialized as NullPieces, but to ensure that no real Pieces move there, the calculation of whether a move is valid or not will now consider the additional boundaries (there will be a vector of vectors of Booleans, that is of the same size as the Board. If a Piece can move there, the boolean will be true in that spot). When a round of chess concludes, the Board will now return the Colour of the player who won, so that Game knows which Player to give the point to.

In terms of keeping track of the score, since 4 player chess is based on points earned, the score in the Player class can be used to represent the points the player earned instead. Since our current move function returns a boolean on whether the move was successful, we could modify it in a way such that it returns a state instead (the different states would be in an enumeration class) representing the amount of pointers a player gets for their moves, if any (capture, checkmate, stalemate, etc).