CS246 Fall2023 Group Assignment- Chess

Justin Yu J489yu

Germain Zhang gzhangh

Spike Wang y3864wan

Introduction:

A game of Chess consists of one board, a 8x8 matrix, and 32 pieces. In a chess game, you will choose a side from black or white, where white goes first. For each side, it has 16 pieces with 6 types---8 pawns, 1 queen, 1 king, 2 bishops, 2 knights, and 2 rooks. Different type of pieces has different ways of movement, including some special cases. In your turn, you can move your piece in a valid way, and you can capture the piece from your enemy in this move. Once you capture your enemy's king, you win the game. If there is no way for a player to capture his/her enemy's king for both sides, then it is a draw. Once a player wins, it will add one scores on the scoreboard for his/her side.

Overview:

we use MVC(Model View Controller Architectural Pattern) for this game.

Model:

- **Class Piece:**
  - ✧ This is an abstract class with fields for coordinates, status flags, and a string to store the current position and status of the piece.
  - ✧ It has seven inheritors represent different types of chess pieces. Special movements like En-passant, Promotion, and Castling are handled in King, Rook, and Pawn inheritors.
- **Class Board:**
  - ✧ This class contains a vector of vectors, holding pointers to Piece objects.
  - ✧ The operations are included for setting up and clearing the board and moving pieces within the board.
- **Class Player:**
  - ✧ This is an abstract class holding information about players.
  - ✧ It contains operations for player actions include moving pieces, offering a draw, and resigning from the game.
  - ✧ Players are categorized into two subclasses: Human and Computer. Computer players are further stratified into four levels, ranging from random movement to sophisticated strategies.

View:

- **Class Textdisplay:**
  - ✧ This class print the board as text format after a turn reaches its end.
- **Class Graphdisplay**:
  - ✧ This class has the field of a Xming class which store information gotten from the board and showing on the screen, and a flag tells whether to show the graph or not. In the Xming, we will create an 800x800 graph, and a 10x10 square to represent

each piece include empty pieces.

Controller:

- **Class Scoreboard:**
  - ✧ This class has the fields that stores times of a side as a winner. We create operations to add the score for a winner side and print the scoreboard out.
- **Class GameController**:
  - ✧ The central class of the game, containing pointers to Textdisplay and Graphdisplay, with operations to update these views.
  - ✧ Holds a pointer to the Board and pointers to two Player objects, with operations to manage the board post-player actions.
  - ✧ Manages a tree structure for storing an opening book and operations to activate it.
  - ✧ Contains a stack for movements to facilitate game undo and redo functions in the cheat mode.

Extra Credits and Resilience to Change:

Smart Pointer:

There will be no delete command in our project, instead, we will use unique pointer and shared pointer to avoid a memory. In this case, it can reduce our time to debug for the memory leak check.

Redo and Undo:

Those are extra functions that will be add to the core program. We want the chess game to be a real video game so that we add this function; By these two functions, we can practice our skills by undoing and redoing when playing with computer with a adjust difficulty. (see question part)

Opening books:

We add a tree structure to store the data of an opening book so that we can a chess manual of master player. (see question part)

More piece type or more players by Single responsibility principle (SRP) and Factory Pattern:

Since we implement the chess game in to separated classes, and the class player and piece are abstract classes, we can add extra fields if we want to update the game. For example, we can add a new piece type which has different move rules; Also, we can add some restrictions for the player; or, since the class is litter related, we can add more players by simply change the player list in our gamecontroller class; or, even we can arrange the piece randomly in the board class.

Approach on Questions:

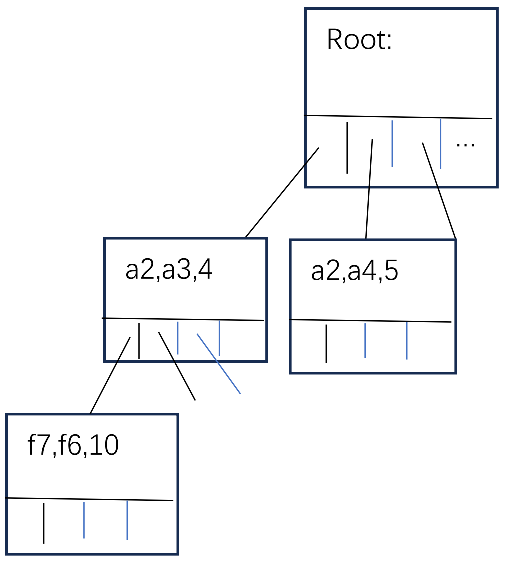
Question: how you would implement a book of standard openings if required.

We can just simply create a txt file or create an SQLite as a database to store the opening move sequences. Then, we need to collect the data from an authoritative website like https://www.chess.com/explorer and transform these data into standard chess format like PGN with possibility of win/draw/loss percentages.

After collecting, we can convert these percentages of move action to weights (an integer) and assign the weights to each piece in each step, where weight values the possibility that white wins. For example, at beginning, we can assign move action of a2 to a4 as weights 5, move action of a2 to a3 as weights 4, and so on.

Now, we can create a class of tree, and each tree structure contains a weight with integer type, a vector<string, string, char> stores a move action, and an array of trees. Now, we need to fill the tree structure with the data we collected before.

For example, in root, there is no weight and vector, just a list of children, Trees. And we can let the first item in Trees be the move action of a2 to a3 with weights 4 and let the second item in Trees be the move action of a2 to a4 with weights 5, and so on. And this will be our first level. Then for the second level, we can let the first child of move action of a2 to a3 be the move action of f7 to f6 with weights 10, and then the second child and so on. By this tree structure, we now create an appropriate container of an opening books.



After we fill all our collect data in such structure, we need to implement some functions to connect it to our chess program. then, a book of standard openings is finished.

Question: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

Strategic Placement of Undo Functionality

The placement of functionalities is critical in the realm of object-oriented programming, particularly in the development of a chess application, for maintaining the integrity of the design structure. In this context, the undo feature belongs ideally to the main game management class, in our case the GameController. This class acts as the chess application's nerve centre, orchestrating the flow of the game, enforcing rules, and maintaining the overall game state. Including undo functionality in this class ensures that all aspects of a move are managed and reversible, including the game state before and after the move.

Implementing an undo feature in a chess game that allows a player to reverse their previous move or perform an unlimited number of undos poses a unique challenge in game state management. To address this, Wepropose a dual-stack approach that not only enables undo but also paves the way for redo functionality. This essay outlines this approach, discussing its implementation, benefits, and how it preserves the game's integrity.

Dual-Stack Approach for Undo and Redo Features

The Core Concept
The fundamental idea revolves around using two stacks: an undo stack and a redo stack. This method leverages the Last In, First Out (LIFO) nature of stacks, making it ideally suited for tracking and reversing moves in a chess game.

Undo Stack
The undo stack functions as the primary record keeper of all moves made during the game. Each move, when executed, is pushed onto this stack. When a player opts to undo a move, the system pops the most recent move from this stack. This move is not discarded; instead, it is transferred to the redo stack. The game state is then reverted to reflect the board's condition before the undone move.

Redo Stack
The redo stack holds all moves that have been undone. It comes into play when a player wishes to redo a move that was previously undone. In such a scenario, the move is popped from the redo stack and reapplied to the game, after which it is pushed back onto the undo stack.

Handling Game States with Dual Stacks

- Making a Move: On making a new move, it is recorded on the undo stack. Concurrently, the redo stack is cleared. This ensures that the redo functionality aligns with the current game state and eliminates potential inconsistencies.
- Undoing a Move: The last move is popped from the undo stack, the game state is reverted accordingly, and the move is pushed onto the redo stack.
- Redoing a Move: If the redo stack is not empty, the last move is retrieved, reapplied to the game, and then returned to the undo stack.

Advantages of the Dual-Stack Approach

Clear Separation of Operations
This method distinctly separates the functionalities of undoing and redoing moves, contributing to a more intuitive and organized implementation. It simplifies the understanding and maintenance of the game's state logic.

Consistent Game State Management
By using this dual-stack system, we ensure that the game state remains consistent with the sequence of moves, undos, and redos. It provides a reliable way to trace back and forth through the moves, preserving the integrity of the game's progression.

Efficient Memory Usage
Stacks are known for their efficiency in both memory usage and performance. This efficiency is especially beneficial in a chess game, where the number of moves can vary significantly, and the game's complexity can escalate quickly.

Conclusion

Finally, the dual-stack approach to implementing undo and redo features in a chess programme provides a strong, efficient, and user-friendly solution. It gracefully handles the complexities of game state management while giving players the freedom to navigate their moves. This method not only improves the player's experience by allowing for strategic rethinking, but it also preserves the structural integrity and game rules. We create a dynamic and engaging chess-playing environment by carefully managing these stacks and integrating them into the game's logic.

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.

Four-handed chess differs to normal chess in the sense that the game is:

1. Played with four players rather than two (We will be referring to these players as Red,Green,Blue and Yellow hereafter)
2. Red is the first player to make a move, followed by Blue, Yellow then Green in **clockwise** order
3. The Grid is extended by 3 cells along each edge
4. Pieces are arranged in the same order with the exception of Blue and Green that have the position of their king and queen swapped
5. The goal is to score the highest amount of points among the four players
    a. Checkmate (+20)
    b. Checking two players simultaneously (+5)
    c. Checking three players simultaneously (+20)
    d. Pieces no longer award points in the case that a player is eliminated
6. Pawns promotions occur on the 8th row from each player's perspective
7. The game ends when three players have been eliminated

<u>(1, 2) Tracking Current Player (4 player variant)</u>
Since four-handed chess is played with four players rather than two players, we can no longer keep track of the current player using a single bool value.

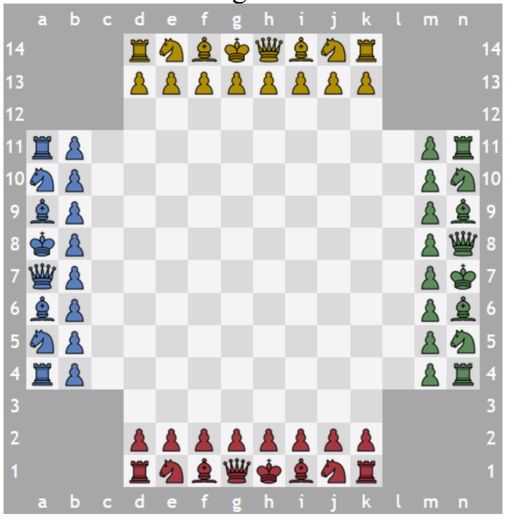
Instead, we could implement our new turn-tracking-functionality using a linked list that behaves similarly to a circular queue(the linked list has tail pointing to the head of the list), where the ordering within the list follows the ordering of players e.g. Red, Green, Yellow followed by Green.

The reason we use a linked list in place of a fixed array is to simplify the tracking process for eliminated players. If the player is eliminated we can simply remove the aforementioned player, and continue along the list to continue keeping track of turns. (Main downside is that removal is O(n), however, as there are only 4 players, removal is extremely quick.)

Now that the tracking of current players is handled with a list, the changeTurn() method within the GameController, along with the move() method in Piece would continue along the list instead of inverting the state of the boolean as done in the two player variation.

We would also need to extend the functionality of our Move class. Our current implementation keeps a log of the coordinates, types and state of any pieces(white and black) affected by a move. Rather than making changes directly to the Move class, we can extend its functionality by creating a wrapper Class that specifies the players(at most two) affected by the move. (Two players is sufficient as it is impossible to capture more than two pieces with a single move) For example, if Blue captures a Green piece then the wrapper Class would bind them to our current implementation accordingly.

<u>(3,4) Changes to the Board</u>
In the four-handed variant of chess the board is extended by three spaces along each side, and the Pieces are arranged in a different order for the Blue and Green pieces.



Since we have decoupled the dependence of our GameController from the initial state of Board, these changes can be implemented by modifying our method setupBoard().

Rather than creating an 8x8 board (represented with nested vectors), we instead create a 14x14 board and initialize the board with the new ordering of pieces. Moreover, as the 3x3 blocks along the corners are not meant to be playable cells (as shown in the image above) we would have to create another Piece subclass to indicate any non-playable areas.

<u>(5) Keeping Track of Scores</u>
In four-handed chess scoring for captured pieces is the same as in normal two-player chess, however, it differs in how ‘checking’ is handled.

This difference would be reflected in our ischeck() and ischeckmate() methods of our GameController. Since we now have to consider 3 other kings, we would have to backtrack from each player's King, to determine the proper scores. We would then repeat our checkmate process for each player's King in a similar fashion.

(6) Pond Promotions
In order to account for the difference in handling pond promotions, we would have to modify the method makeMove() within our GameController. Since pond promotions occur along the 8th row of each player's perspective, we would have to keep track of the current player and whether the move satisfies the constraints for the promotion.

(7) End Condition for the Game
Unlike in two player chess, the four player variation ends when three players are eliminated.

In our program, the end state of the chess game is determined by ischeckmate() – which is not sufficient for the four player variation. We can resolve this by adding a private field to keep track of the number of players eliminated. Our new ischeckmate() function increments the eliminated player count and ends the game if and only if the count is three.

We can alternatively determine the end condition for the game by checking the size of the linked-list keeping track of the current player. Since any player that is eliminated from the chess game is also removed from the list, the game ends if the list has size equal to one.

Final Questions

What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?
From our past projects and personal experiences, our group members decided that in order to proactively address any challenges we may face, it was absolutely crucial to implement an effective strategy before jumping into programming right away. We spent a lot of time in the pre-planning phase to clearly define the structure of the overall program, dividing up each group member's roles and responsibilities, and setting realistic and manageable timelines.

As we spent a lot of time in the pre-planning phase, we were able to do a lot of our work asynchronously. However, although we conducted frequent check-ins on each other's progress and had a clear communication channel, we underestimated the impact of VCSs like "git" and how easy it makes collaboration.

Part of our early struggles while working on the project was that while we were keeping track of each other's progress, we didn't have any tangible means apart from our UML to see how the entire project was coming together. However, once we set up our git repository this was no longer an issue -- we were able to regularly pull changes to stay-up-to-date, and code reviews(e.g. reviewing changes, leaving comments and suggesting improvements) became a more straightforward process.

What would you have done differently if you had the chance to start over?

We felt that we did a tremendous job when it came to planning out the project and assigning tasks. However, if we had the chance to start over, we would have placed a larger emphasis on holding discussions about the specific implementation of certain key components before diving into the project. This includes providing runtime information, considering various factors that may enhance the extensibility of our program, and weighing the trade offs between efficiency and practicality.

While the overall structure of our code did not deviate from our UML all that much, we eventually found that as we broke tasks into smaller chunks there was some unexpected overlaps, which with more **detailed** pre-planning, could have been identified and addressed in a more efficient manner.