CS 246 Final Project: Chess

David Gurevich, Frank Jin, George Liu

Intro

Our final project features a fully functional standard two way Chess game with displays (graphics and command line based) and three levels of AI move generators.

Overview

The project folder consists of the include, src, and asset directories. Graphical elements such as pictures of pieces (png/svg) are stored in the asset directory. The src directory consists of the source code (.cc), and the pieces and players subdirectories are further used to organize game elements by category. The pieces subdirectory contains the 6 types of standard Chess piece implementations (King, Queen, Pawn, Bishop, Knight, Rook) and the players subdirectory contains the 3 levels of AI and human player implementations. The include directory contains the corresponding header files to the implementations in src. Finally, we have a directory of tests that we use to check some of the more complicated features of the program.

Design

The main function of the chess game is to manage a vector of Games. Each Game owns a chessboard, two players, and two observers. It also keeps track of its state. For example, starting a game after running 'setup' will start a game on the board that you just set up. This is thanks to the fact that you are dealing with a single Game object. Once a game starts, all control is handed over to the Game. It checks for win conditions and probes the players for their next move. Once a game is over, control is handed back to the main function, which updates the player's scores, and is ready to make new Games. This flexible design allows for significant opportunities for growth. For example, this architecture can be extended to run multiple games concurrently. It may even act as a back-end for an online chess website.

The Board, the central unit, handles actions which depend on the position of the Pieces on the board. For example, check_valid_move handles checking whether or not a move was valid, which depends on if the move placed the king in check or if the move is blocked by some piece for example. The Board handles execute_move which applies a move and modifies the board position based on the move - this also depends on the current board position to determine if the move was a capture or simply a piece placement. Finally, the Board class handles notifying the observers in case of any positional changes (eg. end of each move).

The Board class also owns 64 Spots, which represent each square on the 8x8 standard Chess board. The Spot class has its board location as fields as well as a pointer to a Piece, so that the type of Piece on the board is known and whether or not there is a Piece (nullptr for no Piece). To prevent high coupling, the Piece class does not have access to the Board class. Instead, the Pieces handle whether they are white or black, and the Piece class acts as a parent class to the six types of pieces. Using valid_path, a virtual function in Piece, the individual subclasses determined whether or not a move (only considering start and end Spots) was a valid path, irrespective of the actual Board position.

An important question we had to address was "what if a move I make puts me in check?". Here, we made use of our vector of Moves in the board class. Within a function, we performed the move in question, and if the player's king was put into check as a result of that move, then we would pop this move off of the vector and revert the changes that the move created (different cases for standard moves, captures, en passant, promotions, and castling). Another related question we had to address was "my king is under check and cannot move, but I am not in checkmate". Under this circumstance, we would search for and identify all of the player's pieces and evaluate whether they had any valid moves in a similar fashion (determining whether a subsequent move would maintain the king in check, or protect the king from check). For all cases other than en passant, we could set the "piece captured" to the ending Spot resulting from the move. However, with en passant, we had to make the appropriate transformations along the *y*-axis based on the color of the attacking piece.

We made good use of the Observer design pattern. There are many events that may occur during a chess game that require the user interface to be updated. The Observer design pattern allowed us to focus on the game logic when it mattered most, rather than having to work around when or where we needed to redraw the text and graphical interfaces. Thanks to the well thought-out design of the graphical interface, we only have to redraw what is absolutely necessary. The board determines if a spot, a move, or the whole board needs to be redrawn, and the GUI reacts accordingly. As a result, the user interface is quick, and responsive enough to flash through a full AI vs AI game.

This chess program has three AI levels which meet the requirements of the project. The first AI level simply chooses possible moves at random (with the help of the Player class). The second and third levels of AI use a system of evaluating the best possible move. The second AI level values captures. The third AI level values captures in the same way as the second AI, but it also has a modifier to ensure that it will not be captured, and is hesitant to move the King too much.

Resilience to change

During our initial drafting period, we collectively recognized the importance of resilient design and how we could utilize object-oriented principles to create this resilience. For example, the Piece base-class is designed such that additional piece types could be easily added if changes were made for a variation of chess. Functions determining validity of piece movements were stored in each subclass and were independent of board size. The use of the observer design pattern made the implementation of a graphical interface relatively easy, and constants stored within the observer.h files aid in allowing board size modifications if implemented.

At the beginning of the project, we wrote a Makefile that easily adapts to different project architectures. Throughout the development process, we made many design changes that fundamentally changed the architecture of the project. The time spent at the beginning, ensuring that our development toolchain was adaptable to our changing demands paid dividends when we had to fundamentally rethink our project.

Answers to Questions

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. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

We have decided that each move made in the game corresponds to a Move object. As a result, we can conveniently store move sequences, such as openings, as vectors of Move objects. We can program a chess AI to check the moves made by the player against the "book" of opening sequences. As far as actually implementing these opening sequences, we can download an existing book of opening sequences using standard chess notation from a third-party source. With this "book," we can write a script that converts chess notation to a hard-coded C++ vector of Move objects. We can store these hard-coded vectors in a header file, and a chess AI can then utilize them.

Opening sequence libraries can be enormous, so an effective search and evaluation algorithm is necessary. In the first few moves, a particular sequence may correspond to many different known opening sequences. An effective search algorithm would have to identify which opening sequences are possible candidates. An example of how this can be accomplished is using standard chess notation. We can convert a given chess Move (and any sequence of Moves) into a string. As a result, the problem of identifying possible move sequences reduces to a string matching problem. Once a set of opening sequences is determined, a chess AI would then have to determine which move to make. This information would come from known statistics about opening sequences.

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

To implement a feature that would allow a player to undo their last move, we would pop off the last move from our vector of moves stored in the Board class as described in the UML. Once this removal has taken place, the corresponding GraphicsObserver and TextObservers will be notified based on the Observer design pattern. Upon this alert, our board will be "redrawn" in the two coordinates on the board of which the undo effects (starting and end pieces of the move). When implemented like this, we can drastically increase the efficiency of the undo behavior where only two coordinates must be redrawn rather than the entire 8x8 chess board. As well, since this is an implementation of the undo feature rather than the redo feature, we can simply discard the last move from the vector rather than storing it in a temporary vector. Another aspect of the logic that we must handle with the undo feature would be reviving any pieces that may have been captured within the move. This can be handled within our Move methods in the UML, where we set the Piece that our move points to in "pieceKilled" back to "alive". Since our method works with a vector containing every move that each player made within the game, the unlimited case would work in the same way, where we would remove and process more than one of the most recent moves in the vector. The most convenient way of handling an unlimited number of undos is through a single undo button which a player is allowed to press an unlimited amount of times.

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.

Assuming four-handed chess follows the same rules as regular chess, apart from the number of players, we can slightly modify our design. Some variant rules to consider are allowing for teammates and a score system to determine a final winner (such as if a player is checkmated, does the game continue?).

In the case of solo four-handed chess, obvious design changes are converting all two player fields into four player fields. In the Player and Piece classes, instead of "white: bool" as one its fields, an option is to have at least three boolean fields (4 for readability) to identify the Player/Piece. Further, the Board class also needs to change "whiteMove: bool" into four similar boolean fields like the aforementioned in the Player and Piece classes. The Game class also needs four player fields to determine how many human and AI players are playing the game.

Another obvious change is the board size. Instead of the traditional 8x8 board, four-handed chess uses the 8x8 board but extended on each side with a 3x8 board. To adjust to this size change, our Board class can take a position field of a 14x14 matrix of Spots in which the four corner 3x3 Spots are out of bounds. The notify functions in GraphicObserver and TextObserver should be adjusted accordingly, to ignore the four 3x3 Spots.

In the team variant, we can have two teams of two players, Team 1 and Team 2. To achieve this, we can add a Team field to the Piece class. This way in the validMove method, we can prevent pieces from capturing other pieces on the same team. To implement a score system for the players, we can add a score field in the player class tracking their current game score and add a method in Game to reset the scores after a game is finished. Further, the Board class can also have a method which takes in a move and updates a player's game score (depending on check, capture, promotion, castle etc).

Extra Credit Features

X11 Graphics

We decided that a user will enjoy the chess program much more if they can view the board with aesthetically pleasing graphics. This is why we set out to draw the board using X11. This was a significant challenge, as X11 has no native way of loading images. We had to find a third-party header-only library to help us load PNGs.

Many hours were spent trying to figure out how to display a chess piece with the right background. In the end, we had to create two images for each piece, one for a white background, and one for a black background. Of course, X11 would not allow us to get by that easily. It turns out that PNGs default to black when they see RGB (0, 0, 0), whereas X11 defaults to white. This has the effect that our images had inverted colors when loaded.

We are happy with certain optimizations that we made, which allow us to only load each image once. After the image is loaded, it is stored in a hashmap, which allows for very fast redrawing. Overall, this was a valuable experience in working with third-party libraries and has led us to the realization that, no matter how adept a C++ programmer you may be, you are ultimately at the mercy of 30+ year old libraries and their documentation.

Smart Pointers only

We set out to not use any `new` or `delete` calls, so as to minimize the chance of memory leaks. We were successful in our goal, as valgrind returns no memory leaks. Using smart pointers forced us to carefully consider who owns what, and this was valuable throughout the development process.

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?

We learnt the importance of organization, task allocation, time management, and use of debugging programs and version control from the development of this software as a team of three. Before writing any code at all, it is important to have a design down on paper, which this course taught well with the expectations of the first due date. Having an initial design allows us to view the large program in smaller, manageable chunks. With the help of object-oriented programming principles, we quickly separated out different aspects of the game and how they were related. This allowed us to begin writing the header files of the program as well as allocating tasks that were isolated and non-interfering with each other.

Similarly to a hackathon or a sprint at work, we recognized the importance of proper coordination and communication throughout the weeks working together on this project. Accountability and time management played a huge role in whether or not we were on track to completing the project in time. Admittedly, our group found ourselves scrambling to make everything fit together at the last minute. Reasons for this included the inability to identify the cause of numerous bugs during run-time. During these situations, we had to step through numerous functions with a debugger (lldb/gdb) to identify the root cause and where our logic or understanding failed. To fix these problems, we isolated the cases in which the bugs occurred and tested repeatedly until we had resolved them. However, as taught in earlier courses, as well as the assignments in this course, writing tests and testing individual functions plays a huge role in expediting the process of combining our code and forming a resultantly bug-free program.

Our group used GitHub for version control. Whenever we were attempting to build a new feature, we could create a separate branch off of our main branch. As a result, we never had problems with losing our code or worrying about the *what-if*'s. Furthermore, using Git meant that any newer versions of our code that performed in unexpected ways in relation to older versions meant we could back-trace with ease to identify where and how something went wrong. Continuously committing and pushing our work also meant our group could work simultaneously on the latest versions of the code.

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

There was definitely room to improve with our work cycle and project management. Rather than having the work pile up in the last couple days leading up to the due date, using a project management tool such as Jira may have been a smart choice. Setting up concrete mini deadlines throughout the two weeks would have allowed us to distribute the load more evenly, and a clear task board identifying what is and isn't done would have made for a more concrete idea of how far along we were with our work.

Conclusion

Our Chess project was not only a valuable experience to apply all the concepts learned in CS246, but also to work with others on a larger scale project. Due to such a successful project, we will be extending our plans to work on four way Chess and implement the additional features such as the opening sequence identifier described in the course outlines.