CS246 - Final Project Plan

David Gurevich, Frank Jin, George Liu

Part A: Project breakdown and estimated timeline

Task	Group member(s)	Estimated completion
Move validity tester: King, Rook, Queen	George Liu	Few days
Move validity tester: Pawn, Bishop, Knight	Frank Jin	Few days
Check tester	George Liu, Frank Jin	3 hours
Game Manager	David Gurevich	3 hours
XWindow Front-End	David Gurevich	5 hours
Text Front-End	Frank Jin	3 hours
Observer pattern for FE	David Gurevich	1 hour
Setup mode	Frank Jin	5 hours
Playser infrastructure	David Gurevich	1 hour
AI Level 1, 3	George Liu	3 hours
AI Level 2, 4	David Gurevich	4 hours
Board infrastructure	David Gurevich	2 hours
Command handling	Frank Jin	1 hour

In order to get everyone working as quickly as possible, the barebones infrastructure must be built first. This includes some graphical interface (either text or GUI), a game manager, some player infrastructure, some board infrastructure, and basic command handling.

The first milestone will be to get a setup mode working, such that we can make arbitrary game configurations to test essential features. This requires the aforementioned “barebones” infrastructure. After that, we can begin working on chess logic. This includes checking the validity of moves, checking for checks and mates, capture logic, castling logic, en-passant logic, etc.

Once the game logic is set up, we can begin work on developing chess computers, as well as handling multiple chess games.

Throughout the entire process, we will implement unit tests where appropriate. We will decide which unit testing framework is appropriate for our case in the future.

We will attempt to follow the Google C++ style guide as closely as possible.

Part B: Project specification 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 behaviour 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).