CS 3110 Final Project Design: Stratego

Team Members:

- Katherine Gioioso (krg48)
- Christopher Sciavolino (cds253)
- Daniel Laine (DLL88)
- Ryan Feldman (rpf53)

Meeting Times: Tuesdays, 2:45 pm, Thursdays, 4:45 pm, Saturdays 11:00 am

Summary:

We are creating the board game *Stratego* along with a text-based interface and an AI that a single user can play against. The AI will be as strong as possible. A description of the general game and rules can be found at the following link:

https://en.wikipedia.org/wiki/Stratego

Key Features:

- A game engine that runs a full game of *Stratego*
- An AI implemented to be as strong as possible
- A text-based, terminal interface for gameplay (Possibly a GUI display too)

Description:

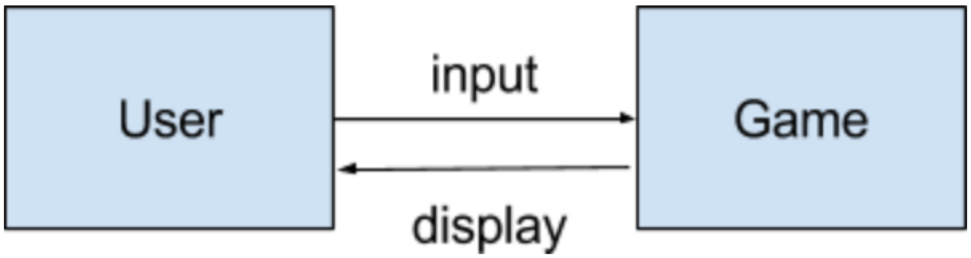
Our plan is to implement a game that contains three phases: setup, gameplay, and victory (or defeat). The setup phase will begin with a prompt for the user to either begin play or read the rules. Information will print out a write-up of how to play followed by the option to “start”. In both cases, selecting start will prompt the player to set up the board (i.e. decide where to place each piece), or have it automatically set up. The user will set up the board by typing in coordinate values for each piece he/she would like to place, moving pieces onto the user’s side of the grid on the board. The game will display each piece, starting with more valued pieces, and the player will choose each position. If the user chooses to have the board automatically created for him/her, then the board will be randomized, only constraining the flag to the last row surrounded by bombs.

The gameplay commands will be input in the terminal. If the user types in a command, say ‘captured,’ the game will display the captured pieces of each player. If the user types in ‘table,’ a table will be displayed, indicating ranks and names for each piece on the board. To make a move, the user will type in two coordinate values; the first represents the piece they would like to move and the second represents the position they want to move it to. Coordinate values will be just like a geometric coordinate plane in the first quadrant. At any time, the user can quit using the ‘quit’ command, or reread the rules by entering the ‘rules’ command. During the AI’s turn, it will move a piece as it sees fit.

The victory phase will display which player won (if any). If the user would like the play again, there will be a command that takes the user back to the very beginning to start over.

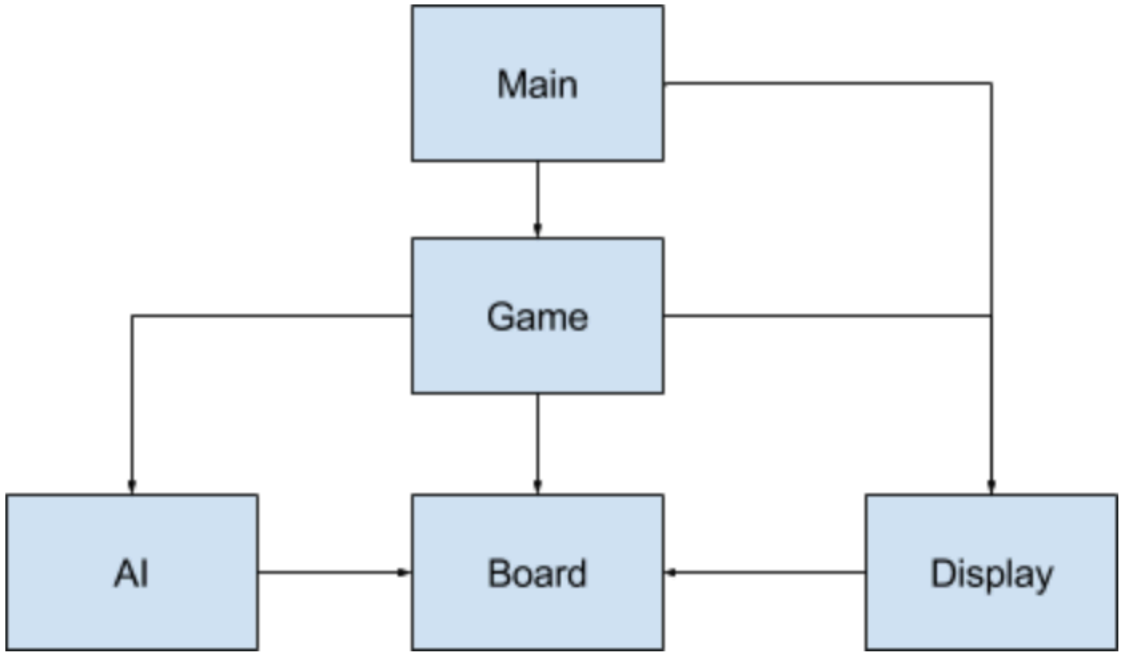
Architecture

C&C diagram:



Description: Our Stratego system is very simple, only involving a User and the Game. The user interacts with the game by giving input, the game processes the input and then displays it to the user. All computations are done in the Game system.

System Design



CS 3110 Final Project Design: Stratego

1. Main

Summary: The Main module welcomes the user, allows them see the rules if they wish, allows them to choose between setting their pieces up automatically or manually, and calls Game to execute a full game of Stratego

Description: The Main module receives the user's input (as described above), handles it, and accordingly calls functions in game, and then calls Game.play to start a full game of stratego with the pieces set up. When the game is finished, the Main module checks to see if the user want to play again. If so, the method is run again; otherwise, the program ends.

2. Game

***Summary*:** The Game module contains the primary methods for running a game of stratego, the main one, [play], being a REPL loop that controls the input and output of the game.

***Details**:* The Game module controls most aspects of game play. The main method in the Game module is a REPL loop that does most of the heavy-lifting behind the game. First, the method will take in the user's input, which will either be a command to move one of the user's pieces or a command to display logistical information (Eg. captured pieces). If the user wants logistical information, the Game module will tell the display module to print out the information desired by the user and take the user's next input. If the user gives a desired move, the Game module first uses Board.is_valid_move to confirm that the user gave a valid move. If the move is invalid, the REPL rejects the input and asks for another input. If the move is valid, the method creates a new board with the move executed and has the AI make a move on the new board. After the AI makes its move, the new board is passed to the display module and displayed for the user. If a piece is captured or something significant happens that the user needs to know but may or may not have noticed, a message will be printed to the display using Display's "print_message" method. This REPL loop executes until the program is terminated or the game is finished. Once the game is finished, the REPL is finished.

3. Board

Summary: The Board module contains all relevant information regarding a singular board object including the locations of all the pieces on tiles and which tiles have nothing on them. They also contain methods that require the information stored in the board data structure commonly called by Game

Description: The Board module contains all functionality and data related to a representation of a board. A board contains a map that maps a (int * int) tuple (representing a position on the board) to a piece option. A piece will be represented a record containing a rank, a player (the player that owns that piece), and a boolean dictating whether that piece has been seen by the AI thus far. The data structure maps to piece options so that we can map empty tiles to *None* as opposed to creating a data structure that would throw errors every time we

searched an empty tile. The Board module also contains methods that use the data stored in the map, such as a method that decides if a given movement is valid or not.

4. AI

Summary: The AI module contains the workings of the AI, aka everything needed to determine the next move. The only function in the .mli file is [choose_best_board], which takes in a board and returns the board resulting from the AI's move.

Description: The AI module will take the current board and get all valid boards that can occur after the AI's turn. Each board will be scored accordingly relative to how desirable it is for the AI. The AI will then choose the board that increases the AI's chances of winning the most and returns it as the AI's move. AI relies on Board to make a move and get a list of all the possible moves a piece can make. Good modularity was used to ensure the representation of board was hidden, including positions (through make_position and get_tuple) and the actual board (by abstracting the basic map functions). AI would work with any Board module.

5. Display

Summary: The display module prints or displays all relevant information to the user in the console

Description: The Display module manages the display of the game to the player through a text interface in the terminal. Each valid turn, Display takes information from Game about the current game state and displays it for the user. In the ASCII, text-based implementation of Display, the user must type a specified command to get information about ranks of pieces, piece descriptions, and captured pieces. Display also handles messages that the game wishes to communicate to the user about the past turn (Eg. piece captures / confrontations). Its GUI implementation needs to handle three things: giving the player the status of the board after a move has been made (during setup and game play), giving a representation of pieces that have been captured when requested, and giving a brief reference guide (again when requested).

We initially stated that if time permits, we will make a GUI for display only while still taking input from the command line. We did not complete a GUI for display.

Data

Board: The most significant data structure is the board, which is a **map** (using the Map.Make module of the standard library) **from an (int * int) tuple** (the tuple represents a coordinate of a tile on the board) **to a record option** containing the information about which piece, if any, is located at that tile.

Pieces: Pieces will be represented with **records** storing their **rank**, which **player**'s piece it is, and whether the **AI has seen the piece** or not.

Positions: Positions will be represented with **(int * int) tuples** (pairs), just like a normal (x, y) coordinate system starting at (0, 0) until (9, 9).

Captured Lists: In Game, there will be **two OCaml arrays** containing the **pieces** that have been captured by each player. These lists will be updated any time there is a capture.

External Dependencies

We did not use anything other than the standard OCaml library.

Testing Plan

Each team member tested his/her code adequately before pushing it to the repository. When meeting, each team member explained their work to the rest of the team. The rest of the team looked through the test cases and ensured that they were sufficient.

In the beginning of implementation, before we had a working game or interface, we tested individual functions in utop and an OUnit test suite. In many cases, Utop was used because our modularity prevented us from directly accessing the helper functions we wanted to test. Initial testing for both ai and board followed a similar pattern:

- 2 by 2 boards were used to check that both make_move and ai_move were working correctly and two pieces battled correctly. For the ai, this also allowed us to test at depth 1; the ai could either decide to attack a piece or dodge an attack.
- 3 by 3 and 5 by 5 boards were used to test slightly more complex movements. For Board, that meant testing a scout moving multiple squares at once. For ai, with these size boards were could test deeper depths. Tests at this size were particularly useful because the game trees were small enough that we could both reason through the tree ourselves and verify with print statements. On a larger scale this would have been impractical.
- The tests above verified that the minimax algorithm was correct and would scale to larger boards. They did not, however, effectively test whether get_valid_boards was working correctly. Larger (7 by 7) boards were then used to test get_valid_boards and its helper functions.

As the game started to come together, we began to “play-test.” By this we do **not** mean that we wrote code until the game worked and then just played normal games. We had a methodical and rigorous approach to ensuring that our code worked as expected. As new functionality was added, we used utop to create the environment we wanted to test and then used functions to do so. Here are some of the environments/situations we tested:

1. Creating AI pieces and player pieces using make_piece and then:
    a. Moving in each direction
    b. Attempting to move off the board

    c. Attempting to attack friendly pieced
    d. Attempting to move flags and bombs
2. Moving scouts multiple tiles at once
3. Winning/losing a game
4. Battling special pieces (i.e. miners defeat bombs, spies defeat marshals, etc)

Division of Labor

Dan Laine: Estimated hours spent: 30

Most of the time I spent was on AI. I wrote a lot of the helper functions is the GameAI module (has_move, score, etc). Wrote all of the code responsible for AI setup and the player's auto setup (do_setup, fill, etc.) Helped refine, neaten and document many other functions throughout the code, especially in Board and AI.

Katie Gioioso: Estimated hours spent: 30

I mostly worked with Dan on the AI. I wrote some of the helper functions (get_valid_boards, ai_move, get_moves_piece) and the actual minimax algorithm including its helper functions (get_max, get_min, break_tie). I did a lot of the initial testing of ai because most of it was testing minimax. I also helped debug while play testing.

Chris Sciavolino: Estimated hours spent: 30

I spent my time implementing board.ml, game.ml, and their respective .mli files. Once those two files were completed, most my time was dedicated to play-testing and fixing bugs in game that came about from the files being implemented independently. The greatest challenge that I came across was a good balance between encapsulation and usability when it came to board and its functions. For the most part, I think that our implementation is well-encapsulated with the right amount of functionality.

Ryan Feldman: Estimated hours spent: 30

I implemented display.ml and helped with game.ml, along with making sure everything had clean, organized code. I also extensively play-tested and fixed bugs that came up, which included having friends play the game and give me comments. I focused on making code more clear, and making sure the game was simple to understand for the user.