Team

Eric Johnson (edj36), Pehuén Moure (ppm44), Kenta Takatsu (kt426), Eric Zhang (ez79)

System Description

Idea

Multi-player implementation of Scrabble

Key Features

- AI implementation that evaluates possible words and corresponding placements while calculating the score each combination generates, used to play against a person.
- Implementation of connection from one front-end client to back-end server.
- Implementation of connection from the back-end server to multiple front-end client instances at once.
- Server side logic to implement game-play and keeping score. The server side will take one user's input at a time and then process and display their input on the game board for all other players. If there is only one player then the server side logic will use the use the AI as the other player.
- Terminal based interface that displays the game board to the user. The front-end will accept inputs from the user when they are taking their turn.
- Word checking dictionary with random letter generator. This feature will evaluate player's moves and determine if they are valid. Valid moves will be placed on the game-board and all other player's boards will be updated. After each turn the game will randomly add a letter to the player's list.

Description

We intend to build a multi-player Scrabble game, that can be played with another person on the same computer or against an AI. The game will be played through either a terminal window, or inside of a simple GUI, built with an Ocaml graphics library. Eventually (after the game is implemented) players of scrabble will each be able to play on their own computer, with central server storing the game state and constantly evaluating the game logic with each player's moves. The AI component implementation will then be moved server-side, and can take the place of a second player if someone is playing by themselves. Finally we will have a robust implementation of a data structure that allows us to effectively query a dictionary of all the words in the English language, for validating players moves and checking that the words the enter are valid in the English language.

Game play will be similar to as if you are playing Scrabble in person, but the application will automate some of the process for a player to move. When it is a player's turn to move, they will enter letters one by one, with coordinates, from their rack of possible tiles. Once they have entered their word, the game will check that the word is valid in the Scrabble dictionary, and also that the word is possible to form given the current game state. For example, if the player has letters {`b, a, s, k, e, t`} in their tray, and they enter `base`, the game would allow that, but if they enter `ball` or `bsk`, the game would let the player know that neither word was valid, the first because the player does not have the letters to form `ball`, and the second because the word `bsk` is not a word in the Scrabble dictionary. After choosing a valid move, the player will submit the move to the game, and the game will update with that move. If the player cannot form a word, they will submit "Draw" and be given a letter from the "letter bag" of letters left in the game. Game play continues in this fashion until someone wins the game, someone quits the game, or the game cannot be continued.

Architecture

- **Server** This component will be the ”back-end” of the game. Whenever a valid request is made it will update the game-state (including game board).
- **Dictionary** This component will serve as a filter for the user inputs. It will be used to validate moves before passing them onto the server.
- **AI** This component will simulate a human playing the game. It will produce a move given a set of tiles, a game state, and a dictionary of valid words.
- **Game-board** This component is the physical interface displaying the game state (with scores, tiles available, etc.) to the players of the game.

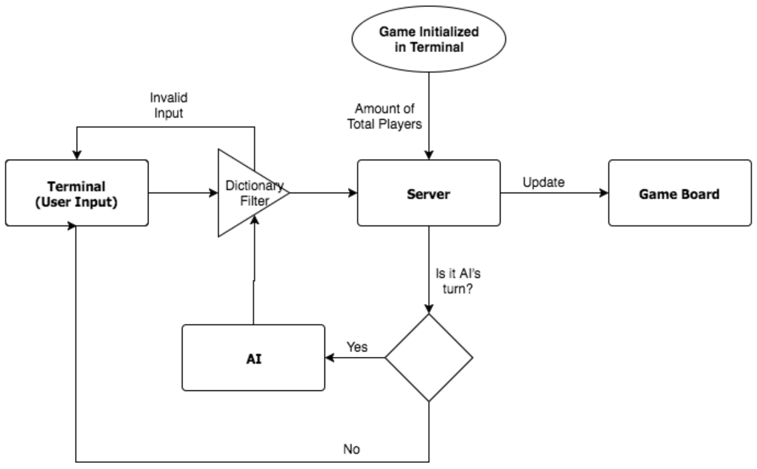

Figure 1: Components and Connectors

System Design

Important modules that will be implemented:

- **AI** This module is where the AI component of the game will be implemented. The module needs to be able to evaluate possible moves and corresponding scores, and then pick a move to play. After choosing a move, AI needs to be able to ”play” the move in the game, and then await their next turn.
- **GUI** This module will be the medium by which the user can see their progress in the game. It will display the game board, the user's score, and the tiles they have available for an upcoming move.
- **Player** This module represents a player playing the game. This is similar to AI, in that the rest of the game should not know whether a move was submitted by a human or by the computer, however, different from AI, the human using the game inputs the word to use in their next move.
- **Server** This module serves as the ”back-end” of the Scrabble game. This is where updating of the state will occur. The module will be given a valid move from Filter and will update the current game state, producing a new game state with correctly updated scores, etc. We are choosing to call this module ”Server” because eventually we hope that we will be able to make our game web-based, and as we said above, this module would be what serves as the back-end of the web application.
- **State** This module holds information about the state of the game at any given time. This includes a data structure representing the game, a mechanism to keep score for multiple players, and some functionality for updating the state as the game progresses.

- **Filter** This module will take input (from a user or from the AI) and evaluate it against the game rules, if the input passes, it will pass the move onto Server.

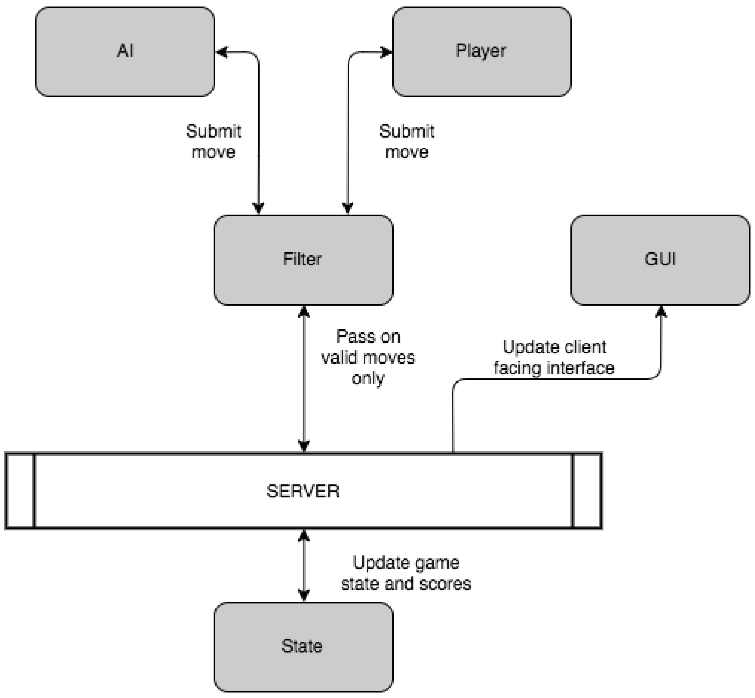

Figure 2: Module Dependency

Data

There is a large amount of data that needs to be kept tracked of as the Scrabble game progresses. Below are the structures and the proposed implementations for the components of our game.

- **Game State** We envisioned all the data associated with the current state of the game to be in a record, game state. This record will include:
  - Current Scrabble Board
  - Scoreboard
  - Rack of tiles for each players
  - Turn (whose turn it is)
  - Letter bag of tiles left for players to receive
- **Scrabble Board** is a 15 x 15 matrix of tiles. This will be implemented as a List of Lists of Tiles.
- **Tile** Each tile will have information about its location and if the tile offers bonus points. It will be implemented as a record with three fields: coordinates, bonus status, and isOccupied.
- **Bonus Status** is a variant that can be double letter, triple letter, double word, and triple word, the conditions for bonus points in the scrabble game.
- **Scrabble Dictionary** Dictionary of which potential moves can be compared to. If the word in the move is in the Scrabble dictionary, then it is a valid move. Since this dictionary will have a large amount of words, we need an efficient data structure to search through. We are going to use the map module to implement this.

- **Scoreboard** an association list that keeps track of the amount of points each player has. We chose to implement it this way because it is easy to update and it is simple to change the number of players.
- **Letterbag** will represent that letters that are left in play, and will be where the players draw letters into their racks. It will be represented as a list of letters. Each **Letter** is a record with two fields: character and points.
- **Player's Rack** Each player's rack will be represented as a list of letters that cannot exceed seven.

External Dependencies

- We will be using the Graphics Module to help render and update the game board. It will be used exclusively in gui.mli.
- We will be using the Map module provided by OCaml to implement the Scrabble dictionary described above.

Testing Plan

We see testing as an instrumental part of the success of our project. We plan to make use of both black-box testing as well as glass box testing because they both have their pros and cons, and using both methods will make our project more rigorous. We will implement tests on a constant basis for each function we write, ensuring that each previous function is correct before writing the subsequent one.

For each module, we will have different roles for the members of the team. There will be implementers and testers. The implementers will write the code for the module and make sure to include very well-defined specifications for all the functions. The tester will use the black-box method and will not know how each function is implemented, ensuring that the tests are not biased. These tests will focus on simple inputs as well as boundary cases. For example, in the case of testing the implementation of the Scrabble Board, the tester will first have simple letter input unit tests to see if it works, but also test to see if the proper error is raised if a number or other invalid input is used. One complication we might have with black-box testing is that, given each member has a general idea of how each component is implemented, it might be hard to keep unbiased; the tester will give his best effort to not let these biases influence the tests.

The glass-box testing will be done by the implementer of the code. Since the test has knowledge of the implementation, this way of testing can address errors that do not show up in the specification. With functions that use pattern matching, we will test using the path coverage method, creating test for each possible outcome of the function to test for correctness. For example, to test the Bonus Status variant, we will have test of each of its cases (double letter, double word, triple letter, triple word).

To make sure our testing plan does not fall through, we will have to hold each other accountable for creating quality tests. One way that we already hold each the team accountable is splitting roles during testing (tester vs. implementer), so the tester can check for correct code and specifications, while the implementer can check for quality test cases. Another way we will make sure that each member is committed to testing is having frequent meetings, where we will discuss the status of testing and constantly checking if each person is writing correct code.