CS5610-Web Development | Project 1: Othello

Project Report

Vipul Sharma
Manikantha Dronamraju

Introduction and Game Description

The project is an implementation of the board game Othello. Othello is marketed/popular name of the game Reversi, invented in 1883 by either of two Englishmen (each claiming the other a fraud), Lewis Waterman or John W. Mollett. Othello is a strategy board game for two players, played on an 8x8 unchecked board. Each square in the grid can either be empty or contain a piece. There are sixty-four identical game pieces called disks (often spelled "discs"), which are light on one side and dark on the other. Players take turns placing discs on the board with their assigned color facing up. At the start of the game, four discs are placed in the center of the board, with two dark(black) discs diagonal across and two light discs diagonally across while all of them are adjacent. During a play, any discs of the opponent's color that are in a straight line and bounded by the disc just placed and another disc of the current player's color are turned over to the current player's color. A valid move in the game consists of a move where at least one of the piece is reversed.

The object of the game is to have most discs turned to display your color when the last playable empty square is filled. Some variants of the game exist where the starting position of the pieces differ from the standard order or the objective of the game is reversed, i.e., the one having the least pieces at the end wins the game, are sometimes-but rarely played.

UI Design

The UI of the game has been mainly focused on three technologies: ReactJS, Bootstrap4 and CSS; however, we have also used some elements of react-strap too in it.

A dark background theme (black with a CSS gradient generated image) is used for the game. The board has been given a Forest Green color for the board, which is subtly florescent and soothing to eye at the same time.

The layout has been divided using Bootstrap’s containers, with appropriate padding as and when needed. Reactstrap cards have been used instead of Bootstrap cards in the JSX for better compatibility, and to display the necessary information to the user in an eye-catching manner. react-shapes library has been used to draw discs, and use the auto-close with close b handle toastify element to display the data to the user during the game. We have also used Bootstrap Modal class for user input instead of pop-up windows to make the user experience better and comfortable.

The game is divided into two pages: the home page (or the lobby) where users can see all the games on the server and has the options to create/join/observe a game and the game page, where the game happens. Additionally, two more pages have also been provided to users, view all Games: gives a list of all the Games that are being played or have been completed with a full list of players involved and Observers if any. The page player list: Gives the list of players and the Game they are playing/have played. A navigation bar at the top facilitates for the easy movement of user to switch from one part to another.

The game page is divided into three parts, status boards, the Othello board and the Chat window. The status is displayed on the left corner, consisting of 4 cards which notifies, black & white players info and status, the score of each player at any moment and status of the game: is it in progress or it finished! This has been achieved by using various Reactstrap cards.

The center of the page is occupied by the main game board, the Othello game board, green in color. Here the user whose turn is up can click and chose the position of their disc which is reflected instantaneously if it is legal move. If the user clicks on any square where he/she cannot play, a toast message is displayed to notify the user. All the observers of the game can only view the game, their clicks will result into no change in the state of the game. The board is implemented as a series of Bootstrap buttons, which have been encapsulated into 8 rows and 8 columns.

The rightmost part of the game is provided with a chat window, where the users can chat among themselves. Observers, both registered and guest, can also chat in the chatroom. For each move a player makes, is also updated in the chatroom. The chat is visible to both the players and all the observers of the game. The chatroom is constructed by amalgamating simple html elements, text area, input box and buttons.

UI to Server Protocol

The communication that happens between the server and UI is via web sockets. The games channel is used for the communication purposes. The front-end view start with the options of joining a game, creating a game and spectating a game. To create/join a game a message along with game name and user name is send to the `“join”` function of the channel which then returns the appropriate message whether the game can be joined or not. Once the game is joined the it is broadcasted to all other users and observers via the game channel and UI is updated everywhere. The front-end view consists of the game board, scores of the player and chat window.

For the game board the state contains an array of values “0,1 and 2” representing empty box, black disc and white disc respectively. Along with that the state also contains both black and white player, a player whose turn it is to move the disc, game status with values either waiting, playing or completed, a list of observers for the game and list of the chat messages.

The player whose turn is to move his disc when clicks on the particular box in the board the `handle_in` function is called with the “move” message, user name, row and column of the box clicked. The method called via `handle_in` then sends the response back and updates the UI via react component appropriately. If the user clicks on the box that is invalid a message of “Invalid move” will be shown via the react component. When a valid move is made and the game status does not change to “completed” the player is changed so that the other player can move the disc. Also, the scores are updated after every valid move. In the UI the chat window is available which will show a message whenever a player joins with his name also at any stage if an observer joins the message to all the existing observer and players will be shown. At every move a message is broadcasted that black player or the white player made a move.

When the game status is “Completed” or no other moves are possible, the `handle_in` broadcasts a “finish” message to all the players and observers and prevents any other moves. And the player with the more score is declared as the winner.

Data Structures on Server:

All game states are stored and created by Agent, Agents exist is to hold state and allow us to query and manipulate it. The idea is to start a new Agent with a default state, get that state back, and update it. The Agent module provides a basic server implementation that allows a state that allows state to be retrieved and updated via simple API.

The game application has a state that contains that maintains the following values: they are Game Board, the name of both the players, initially empty list of messages and observers, the player who has his turn and status of the message. Game board has 64 boxes and is initialized as an array with values 0,1 and 2 representing the empty box, black disc and white disc. The Player is either a black player or the white player.

Every state manipulation is handled in different functions, turn of the player are changed after every move using the `change_player()` function that takes the entire game instance as an argument and changes the players accordingly.

The game board changes whenever there is a legal move by the player, and the `move()` function updates the game board after the move has been made. The status of the game is “Playing” and is maintained every time there is a legal move available as soon as the status changes to “Completed” the game board is disabled and a winner is declared.

During course of the game the list `ob` observers is updated every time a observer joins the game and a message in the chat window is made available to all the existing observers and both players.

The chat messages list is also maintained throughout the game and keeps a tap of all the messages shared between the users.

Implementation of Game Rules

We have tried to keep the rule of the game as close to the original rules of the game Reversi, with some minor twists to add some zest. The user who joins the Game is assigned as the white player. The user has to give a unique name for the game, if he/she fails to, they will automatically join any existing game as a spectator of Player 2. These actions are performed by the function `add_new_player().` The observers are distinguished from the players by using the function `who_am_I/2`, which takes the game state and the user under question as arguments and check against the player of the game. If they are not one of them they are inducted as observers and appended to a map, used to display the list of observers in the game.

After a game is created, another player can look up for any games in which another player is waiting and join the game to play the game. The second joining player is assigned the black side of the game. The black assigned player moves first, in the meantime the other (white player) has to wait for the black player to finish his/her move.

The board is constructed by the function `othello/0` which defines the grid position of the players, while initializing the cross position at the start of the game. The function `client_view/1` initializes the board first, which is used by the client to render it to the user.

A point is awarded for each successful flip of the user. The main function for handling this task is `move/4`, which takes the game instance, disc color and the position of click as input and process the input from the user. The function `change_player/1` keeps a track of all the moves, and increment/decrement their points and the state of the game, by calling the function `update_score/2` which take the fame and the cell number (location of the click as input). In the process the move is checked by the server with help of the functions `is_legal_move?/6` & `is_legal_move?/3` and `same_disc?/`6 functions to make absolutely sure if the move was legal or not. If the move is not legal, an auto closing with feature to close toast message displays it to the user. React-toast library has been used for it. If everything is okay, the function `change_disc_count/5` updates the disc counts for the players. No time limit has been imposed for a user to take a turn.

The game continues till all the squares have been covered by both the users or one of the user doesn't have any valid move left. At the end of the game a winner is chosen, the player having the maximum score wins and the game ends. The status is conveyed to the user by a toast message and the change in status in the score board. The task is done by the function `legal_moves_options/2.`

The chat server is controlled by the `start_chat/3` and `handle_in/3` functions.

Challenges and Solutions

1.) Implementing multi-player and unlimited observers:

When joining the game the player being connected will receive the list of current connected players as a response (synchronously, that is). The line that we had to add to the channel is the follows “`send(self, {:after_join, message})`”. What this line helped us doing was to send a message to itself so that at every point in the future when the message arrives (and that can be in fact after the connecting player has received the players list) it will broadcast a "`player:joined`" message to every socket connected to the server, including the one that just connected. While reading through the forums and documentation we realized that it is a pattern very often used.

Another important function that we came across, to solve the problem was “`handle_info({:after_join, _message}, socket`)”, this function does a pattern matching of incoming messages. If it matches the id of the joining player is extracted from the socket assigns and the player is added to the state of the game. Also, the final step is that everyone is notified that a new player is in. The problem made sure that we learn the channels implementation and data passing efficiently.

2.) UI Design:

The design implementation of, a few aspects was a bit challenging. The first problem was to display a pop-up that is soothing to eye when an invalid move is made the solution that we came across was to use “Toastify” for ReactJS, which made colourful toast (a pop-up) with a time lapse on it and providing the user with an option to close if it annoys them.

Another problem was the design that we had in our mind, we were not happy with the output that we got using “Bootstrap” even after many changes to the design something or the other hampered the design. So instead we used “ReactStrap” and that seemed idle to what we wanted also instead of using the normal CSS we used “React Shapes” to come up with the game board and score board as well.

Overall, we did face many minor hiccups and at every step there was a learning lesson that helped us making our understanding more clear and precise, and every time we made changes to our application, we learnt something new.