Project 3

April 12 , 2023

ECE 421, B1

Group 3

Alfred Lee

Arsh Narkhede

Arvinder Bhullar

Abstract:
This project serves as a great introduction to developing UI/UX designs using Rust, and web assembly. In this project two games namely Connect-4 and TOOT and OTTO were developed leveraging rust and web-assembly replacing the MEAN stack initial code.Added functionalities like making the game accessible to users with color vision deficiency, computerized opponents with varying difficulty levels, and the ability to hold player information and game results were also introduced in this version of the release. Overall the goal of this project was improving the performance of the game, replacing the MEAN stack code and introducing new features to the existing games.

Questions:

1) What can we do on a computer that we can't do on a printed board?
The most important advantage of playing these games on a computer is that we can use the computer as a player to play against human players. This allows some flexibility as now players can practice playing against computerized opponents and get better at the game. Also, a printed board offers limited static representation and it requires some additional assistance to keep track of the scores and view game history.

2) What is a computerized opponent? What are its objectives?
A computerized opponent is an algorithm that can act just as a human opponent in the game.Its objective is to provide a realistic opponent to the player, which involves making decisions based on the game current status following all game rules and objectives. It should have characteristics just like other human opponents. It should prioritize winning initially while at the same time block the player's winning moves as well. We just need a single computerized opponent to play against a player.

3) What design choices exist for the Interface components?
Design choices for the interface components in a web application include things like color of the chips and text, font of the text, dimensions of the board, layout of the buttons and other components, types of user input controls, etc. These parameters depend heavily on factors like the target audience, application type and thus it is important to consider these factors. We also made our application as accessible as possible to varied kinds of users. The UI was designed to be very user friendly.

4) What does exception handling mean in a GUI system?
Exception handling in a GUI system refers to how the program handles errors and unexpected exceptions that can occur while the application is running. A well designed GUI system should handle all these errors and provide individual messages to the user indicating the specific error. This makes the GUI system more robust and allows the system to operate normally.

5) Do we require a command-line interface for debugging purposes?
Yes, we do. This can help us debug the backend game itself without the presence of the frontend code. Also implementing it in a continuous loop can help us understand how the computer is making decisions based on the current state of the board. This was

really helpful to implement, as it can help to isolate faults and get the game ready and make sure it works as expected before implementing the frontend code.

6) What are Model-View-Controller and Model-View-Viewmodel? Are either applicable to your design? Justify your answer.
Model-View-Controller(MVC) and Model-View-Viewmodel(MVVM) are both architectural frameworks that aim to separate the frontend and backend components of a software design.
MVC has three components; Model - handles the backend i.e. algorithm and database; View - handles the frontend i.e. displays the output; Controller - handles the user input and updates both the backend and frontend as required.
MVVM is similar to MVC but instead of a controller it has a layer called ViewModel. This layer serves as an interim layer between Model and View and is responsible for updating status of View and provides data binding between View and Model. **[1]**
The implementation of our project does use a similar architecture like the MVC model. In our approach the system is divided into;
i) Model - The main game algorithm(present in “common” directory) and the database structure(present in “backend” directory);
ii) View - The frontend UI of the application(present in “frontend” directory); and
iii) Controller - In this case this is also handled by the frontend implementation of our code.
But, the MVVM is also an approach that can be leveraged for the implementation of this project.

Rationale for design:
We looked at this example of a full stack application in Rust at: https://blog.logrocket.com/full-stack-rust-a-complete-tutorial-with-examples/

We decided to use a similar model - Model-View-Controller - to implement the Connect4/TootOtto game. This model allows for separation of concerns. We have 3 directories in the project which handle separate responsibilities. The 3 directories are as follows:

1. **Common:** This directory contains the game logic for Connect4 and Toot-Otto. We started from here to have a working model of the game which we tested iteratively.
   For both Connect4 and TootOtto, we use enums and structs to define the player, piece, and difficulty to allow for re-use and expandability.
   For the AI opponent, we decided to use a minimax algorithm for both Connect4 and TootOtto due to its simplicity and its use for board games such as chess. For the AI opponent at easy level, we randomize the piece and column selection to make it an easy opponent. For medium and high difficulty, we specify the depth of the minimax algorithm.

For Connect-4:
The Connect 4 struct stores the game board and the current player. The Piece and Player enums represent the types of pieces that can be placed on the board and the current player respectively. The implementation uses the minimax algorithm for the AI move, and if the difficulty level is set to easy, a random move is chosen instead.
The get_grid() function returns a copy of the current game board. The top_row() function returns the highest empty row in a given column. The user_move() function takes in a column number and places a piece in the top empty row of that column, then switches the current player. If the move is invalid, the function returns false. The ai_move() function takes in a depth parameter to set the difficulty level of the AI move and is used if playing against a computer opponent. If the depth is set to 1, a random move is chosen. Otherwise, the minimax algorithm is used to calculate the best move. The evaluate_board() function assigns a score to the game board and is used by the minimax algorithm to evaluate potential moves.

For Toot and Otto:
The TootOtto struct contains the game state with a board, a current_player, and a difficulty level. The board is represented as a 2D array where each element can either be None or Some(Piece). The Piece enum represents the two different pieces that can be placed, T or O. The Difficulty enum is used to determine the difficulty level of the game.
The struct implements several methods to interact with the game. The new() method initializes a new TootOtto game with an empty board, the current player set to Toot, and the difficulty level set to Easy. The get_current_player() and get_grid() methods return the current player and the current state of the game board, respectively. The set_difficulty() method sets the difficulty level of the game, which is used in the AI's moves, if it is a game versus a computer opponent.
There are three methods to make a move by the different players. The make_move_by_toot() and make_move_by_otto() methods are called when Toot and Otto respectively make a move in a two-player game. The place_piece() method is used by both of these methods to actually place the piece into the grid. The make_move_by_ai() method is used when the AI player makes a move, used only if playing against a computer opponent. This method uses the minimax algorithm to determine the best move for the AI player based on the current state of the game board and the difficulty level.

For Backend:
The Game struct contains the result of a game. The struct has gametype, player1, player2, winner, and date.

2. **Frontend:** Once we had the working models of the 2 games, we added the functionality to display the game on a browser in the frontend directory, which is a separate project. We import the game logic from 'common' into the frontend

project and render the game to html with the yew crate from Rust. We learned from the resources available on the web for how to use yew, how to render html in yew, how to interact with the canvas in Rust, to make a visually appealing UI. The frontend project also contains calls to the rocket client to post game results, get game results and display them to the user.
The frontend project uses the standard layout suggested in yew tutorials to display/serve the html and uses routers to route to different pages when the user clicks on different buttons.
To run the game in the front end, there are different files for TootOttoHuman, TootOttoComputer, Connect4Human, Connect4Computer for different variations of the game. In each file, we define a struct which holds the state of the game, names of the players, name of the winner, status of the game, callback functions and event handlers. We use the enum Msg to handle state changes in the game such as name input, start of game, selection of disc, clicks on the game board. Any time a move is made, render_board() function is used to draw the game on the canvas. The struct is initialized at the beginning when the user lands on the game page. The view() function provided for yew components is used to display the html webpage for each game. Once the game ends, the results are posted to the rocket client and then added to mongodb.

3. **Backend:** For mongodb the backend contains a DB struct which contains the database we want to use. When the struct is created with new it will set the connection URL and set the database to “Project3”. It contains the functions insert_game, get_games, and doc_to_game. insert_game() simply takes the parameters and inserts a game at the current time. get_games() gets the games in the database and returns them in a Result<Vec<Game>..>. doc_to_game() is a helper function that takes the value from the database stream and converts it into a game struct.
For the rocket crate, the backend implements get_games(), add_games(), and Fairing. get_games() is a get method request that returns a Json of a vector of the games. add_games() is a post method request that takes a Json of a game and adds it to the mongodb database. Fairing is a method that allows us to implement a CORS so that our server can specify the type of origin a browser can permit the loading of resources. Basically it allows us to load the resources of our rocket client into our application. All of this allows our rocket client to get results and return results to the application from/to the mongodb database.

Separating the responsibilities this way allowed for easy testing, easy debugging, and better re-use of code.
We also decided to design the implementation of the game as well as the front-end flow to be modular and easily extendable. This way, we were able to modify and use the various functions and Structures throughout the application.

Description of MEAN stack in the Code:

We were successful in transitioning the entire code-base from MEAN stack to a combination of Rust and Web-Assembly. Though MongoDB is still being utilized for the database handling on the backend, it is implemented as a framework in Rust and the entire source code is developed in Rust. All The node, express, angular implementation in the original source code was successfully changed to Rust.

Known Errors:
No known errors were found in this project by the development team.

Extra work:

1. The start game button was disabled to ensure that players would have to enter their names before starting the game, ensuring a good database for keeping track of scores.
2. Both the Connect-4 and Toot and Otto chips were updated to have a letter describing the nature of the chip to assist players with color vision deficiency. In case of Connect-4 chips were updated to have “R” and “Y” to depict Red and Yellow chips respectively.
3. Difficulty levels were added using a drop-down menu to keep the UI as user friendly as possible.
4. A command line interface (CLI) was also provided for both games in the main.rs file located in the “common”directory to test the functionality of the games without the frontend code. It is implemented as an infinite loop that prints the board after each move.

User Manual: Connect-4 with TOOT and OTTO
Introduction:
Connect-4 with TOOT and OTTO is a web application that allows users to play Connect-4 and TOOT and OTTO games either with a human or a computerized opponent. It allows varied difficulty levels when playing against an computerized opponent and has the ability to keep track of scores and scoreboard.

System Requirements:
Rust programming language and all the necessary dependencies should be installed on the system.
Web application frameworks required for this project are:

1. Yew
2. Trunk
3. Rocket

How to Use the Program:

1. Open the command prompt or terminal.
2. Navigate to the directory where the project is saved.
3. Now open three terminals in this directory:
   i) In one terminal navigate to the frontend directory and enter “rustup override set stable” and then enter “trunk serve - - open”.

ii) In another terminal navigate to the backend directory and enter “rustup override set nightly’” and then enter “cargo run”.

iii) In the third terminal, navigate to the backend directory and enter “mongod --dbpath="data"” command. Ensure a data folder exists in the backend directory.

4. The program will install all the necessary dependencies if not already installed.
5. The project would be launched on a local host: http://127.0.0.1:8080.A new browser would pop up to take you to this address or one can use the link that the terminal would provide.
6. Once on the web-page, you would have to maximize the screen. Now you can use the sidebar to learn about both games and play them against a computerized opponent or a second human opponent.
7. For both games you would have to enter the necessary information(player names) for keeping track of scores.
8. While playing against a computerized opponent you would also have to choose a difficulty level using the dropdown menu. By default the difficulty is set to Easy/Beginner.
9. The rules on how to play have been provided in “How to Play Connect 4” and “How to Play Toot” sections of the game.
10. You can now enjoy these games with your friends and family. Carry on playing!!!

Functions of the Program:

1. Home: A welcome screen introducing some help on how to go about using the project.This is the window that pops up when you initially load the game.
2. How to Play Connect 4: A how-to-play tutorial/guide helping beginners to get familiar with the game of Connect-4.
3. Play Connect4 with Computer: Allows users to take on a challenging game of Connect4 against a computerized opponent.
4. Play Connect4 with Another Human: Allows users to take on a challenging game of Connect4 against another human opponent.
5. How to Play Toot: A how-to-play tutorial/guide helping beginners to get familiar with the game of Toot and Otto.
6. Play Toot-Otto with Computer: Allows users to take on a challenging game of Toot and Otto against a computerized opponent.
7. Play Toot-Otto with Another Human: Allows users to take on a challenging game of Toot and Otto against another human opponent.
8. View Game History: Allows players to keep track of all the games they have played in this session.
9. Score Board: Allows players to keep track of their scores in all the games they have played in this session. This can allow players to improve their performance.

Errors and Exceptions:

1. When the column is full, the game UI does not toast (raise) a message informing the user that the column is full and they should try inserting in another column. But, it does not add chips anywhere in the board and still waits for a valid input.

References:
[1] https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm

For structure:
- https://blog.logrocket.com/full-stack-rust-a-complete-tutorial-with-examples/

For frontend:
- https://yew.rs/docs/tutorial
- https://yew.rs/docs/concepts/html/components
- https://yew.rs/docs/concepts/html/elements
- https://github.com/yewstack/yew/tree/master/examples/router
- https://yew.rs/docs/next/concepts/router
- https://github.com/yewstack/yew/blob/yew-v0.20.0/examples/function_router/src/app.rs

Canvas:
- https://github.com/security-union/pixel-rain-with-rust-and-rust
- https://subscription.packtpub.com/book/game-development/9781801070973/2/ch02lvl1sec06/drawing-to-the-canvas
- https://yew.rs/docs/next/concepts/basic-web-technologies/web-sys

Backend:
- https://stackoverflow.com/questions/62412361/how-to-set-up-cors-or-options-for-rocket-rs
- https://dev.to/hackmamba/build-a-rest-api-with-rust-and-mongodb-rocket-version-ah5
- https://theadventuresofaliceandbob.com/posts/rust_rocket_yew_part1.md