CS510 Rust Final Project
Project Overview and Whitepaper

Michelle Duer

June 7, 2019

Command-Line Tic-Tac-Toe

I chose to implement a simple Tic-Tac-Toe game because I was familiar with the game implementation having written it in python (and I wanted to improve upon this implementation) and with a react.js tutorial. My first goal was to create a command-line version of the game with many unit tests to ensure it was functionally working. This first version of the game would run quickly in the command-line.

Surprisingly, I found starting the program conceptually challenging to design compared to other languages, which felt more intuitive and quick to implement. It took a couple days of thinking about the **Game** *struct*, its mutable variables, with a lot of research from our textbook, the online Rust book and other resources.

I wanted to avoid over-copying between function calls and mainly accomplished this through the Game struct and the *mut self* parameter to avoid unnecessary copying or even dealing with explicit lifetimes. Next, I wanted to focus on functional language features I was less familiar with, for example, by implementing the **Display** *trait* for the **Game** *struct* along with *Default* functionality for the **AutoPlay** and **WinState** structs.

I took advantage of the Rusty *match* control flow along with first-class functions inside the update() function where a simple Boolean check results in different function calls:

```
let loc: usize = match &self.auto_play.play_type[self.curr_player] {
    true => self.auto_move(),
    false => self.manual_move(),
};
```

One scenario that took me a bit longer to figure out was ensuring that a user in manual play would enter a valid character, which was also done with pattern matching:

```
let mut first_char = user_response.chars().next().unwrap();
match &mut first_char {
    ’0’ ... ’8’ => {
        println!("You entered: {}", first_char);
        first_char.to_digit(10).unwrap() as usize
    },
    _ => {
        println!("\nPlease enter a valid response: ");
        self.get_user_input()
    },
}
```

where the first character of the users input must match a valid range between 0-8 and otherwise prompt the user to enter a valid location on the board.

Rusty Tic-Tac-Toe Compiled to WASM

The goal of my second implementation was to take most of the first command-line game and compile it over to WebAssembly (WASM). I began the process by running through the online Rust-WASM tutorial (implementing Conway's Game of Life) [1], which was quite helpful. However, when trying to begin a *hello world* for my game outside the scope of the tutorial where the environment was already defined, I ran into fundamental issues with the compiling. It was incredibly time-consuming to resolve a simple javascript import that had worked during the tutorial, example:

```
‘import { greet } from './pkg/hello_world';
```

The *wasm-bindgen Guide* [2] had an incredibly easy-to-miss comment stating that the aforementioned import statement should work *in the future*, but since that functionality was obviously broken, the following was recommended to get the javascript file running with the WASM:

```
const rust = import('./pkg/hello_world');

rust
  .then(m => m.greet('World!'))
  .catch(console.error);
```

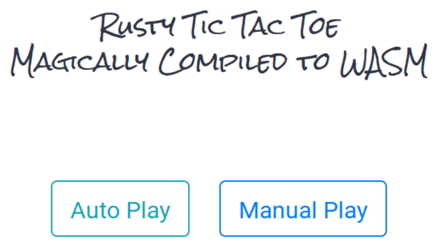
I struggled for days to get this to work properly, and this may also be related to the fact that I had only superficially played with javascript. Even when the code compiled, I could only get the HTML or the javascript to run – not both together (*slightly salty sidenote*: I suspect this may be why the wasm-bindgen guide only uses javascript in their example... but it could also have been my human error or lack of domain knowledge).

My solution was to revert my *wasm-bindgen* dependency back to the version (v. 2.0) that were used in the *Rust-WASM* tutorial, and the compiling process began to work with the first import command (also similar to how imports are declared in the *Rust-WASM* tutorial.

This was a good example of the biggest challenge I ran into when compiling over to WASM. It is still a relatively new language (created in 2015), so there is little documentation from 2019 and some of the examples that are found online do not necessarily build without setting up the correct environment and dependencies. That being said, once the Rust began compiling over to WebAssembly, it was relatively easy and very fun to get it running!

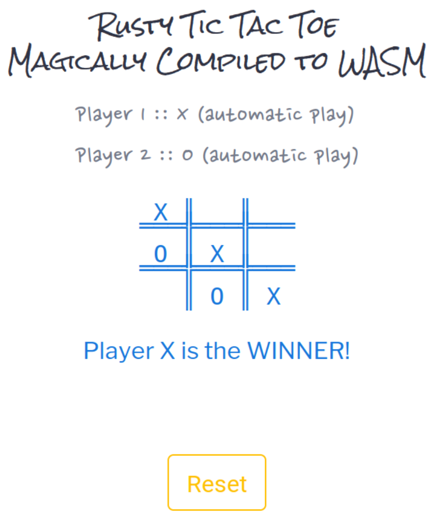
The build directions for both versions of the game can be found by going to the project repository: https://github.com/mkduer/rust-wasm-game.

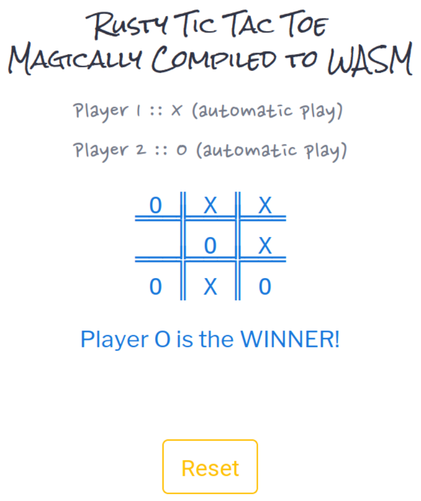
By running the web browser version, the following menu should appear:

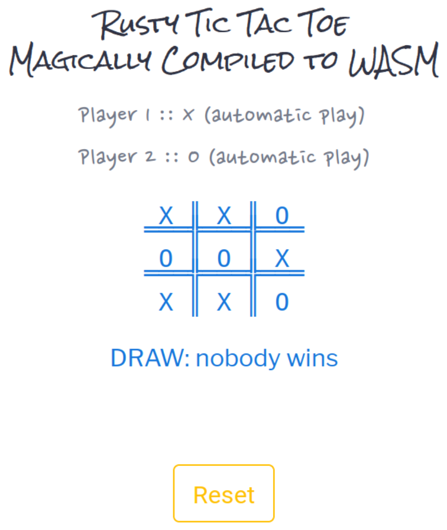


The *Auto Play* will create a game of two players that automatically (and dumbly) play the game. Implementing a reinforcement algorithm down the line could result in more interesting game play.

Screenshots of Auto Play Games

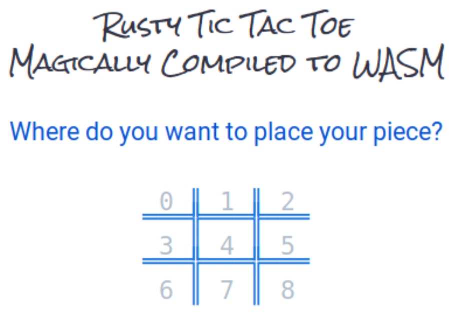

Rusty Tic Tac Toe
Magically Compiled to WASM
Player 1 :: X (automatic play)
Player 2 :: O (automatic play)
X
0
X
0
X
Player X is the WINNER!
Reset


Rusty Tic Tac Toe
Magically Compiled to WASM
Player 1 :: X (automatic play)
Player 2 :: O (automatic play)
0 X X
0 X
0 X 0
Player O is the WINNER!
Reset


Rusty Tic Tac Toe
Magically Compiled to WASM
Player 1 :: X (automatic play)
Player 2 :: O (automatic play)
X X 0
0 0 X
X X 0
DRAW: nobody wins
Reset

The manual-play is still in-progress with an overlay of key values for cells. Current work is focused on listening to a keypress and getting the communication working between the compiled WASM and javascript.

Screenshot of Manual Play


Rusty Tic Tac Toe
Magically Compiled to WASM
Where do you want to place your piece?
0
1
2
3
4
5
6
7
8

Given more time, I would have altered my initial design that is based off of the command-line implementation and represent the board with component cells rather than a single string. This would allow for a better design allowing for mouse-clicks. As of now, the game play requires keyboard presses that will hopefully be fully functional. These key-clicks are represented by the indices, but considering game-play would be better if reversed to reflect a keypad.

Rust-to-WASM with Hindsight

It was relatively straightforward using my existing command-line code and re-using the same functions for the online version. This was particularly useful because the command-line code already had unit-tests so I did not have to spend much time worrying about game-functionality.

As the stdout/stdin from the command-line implementation wouldn't work for the browser. I had to alter some of the display functionality to ensure the output was based on String types that could be compiled over to WASM.

The relatively smooth communication between the compiled-WASM and javascript was pretty cool to watch.

I ran into unexpected compiler errors with the *Rand* dependency and, I suspect it was specifically the *thread_rng* function. I wanted to get the random number generation (for automated play where a cell is randomly chosen from a range) working with *OsRng*, which seemed to be the recommendation – this allows for the specified operating system's random number generator to be used, which sounded great. However, I could not find an equivalent *gen_range* function to choose a number between a specific range.

Talking to Bart, I was told to try the javascript implementation using its Math library. This is on my list of things to do, but I wanted to prioritize the manual play first before the deadline.

Next Steps

1. Altering some the game's design to generate the board as individual cells instead of a string for mouse-click functionality, which seems much more practical than key presses (or allowing for both).
2. I would reverse the key values to correlate to a keypad.
3. I would fix the random number generation.
4. Add more unit tests. In order to save time, I relied heavily on the unit tests that were already created from the first version, but it would be wiser to have unit tests that work specifically for the second version along with tests for correct compiling, and communication with the frontend/backend.
5. If I were to take this project further, I would want to focus on network elements and parallel processes in order to allow multiple versions of the game to be played at the same time.

References

[1] The Rust Team, "Rust and WebAssembly", [Online]. Available: https://rustwasm.github.io/docs/book/introduction.html

[2] The Rust Team, "The 'wasm-bindgen' Guide", [Online]. Available: https://rustwasm.github.io/docs/wasm-bindgen/examples/hello-world.html