Multiplayer Grid Based Dexterity Training Game

NEA: Georgiy Tnimov

0.1. Abstract

0.2. Client

0.2.1. Client Synopsys

The Client is Alexander Tahiri, a software developer at Studio Squared and the developer of Tapp, a game based on a 4x4 grid, which consists of 12 inactive tiles, and 4 active tiles. Players use the mouse cursor to click on an active tile, which then deactivates that tile and actives a new, currently non-active tile.the objective of Tapp is to achieve as high a score as possible, without making any mistakes. The Client requires a derivative of this game, which tests simultaneous dexterity of both hands, additionally The Client wants to incorporate a competitive aspect to the game, which consists of a leaderboard section, allowing players to see their position whithin the rankings and a Tetris-99-esque game mechanic, where players compete to either achieve the highest score, or last the longest in a mass multiplayer format. The Client has specifically asked for the Catppuccin colour scheme to be used, The Client has sufficient computing power to host both the client,server and database, which will be provided free of charge.

0.2.2. Interview Notes

all interview notes are paraphrased

“for doubletapp, what features are most important to you?”

AT: my main requirement is that the new game tests both hands simultaneously, and has replayability. features such as users and leaderboards, along with a competitive aspect would be awesome

“how many users do you expect to scale to?”

AT: I am estimating up to 50 concurrent users, and aim for small latencies

“any specific UI/GUI choices, and what platform should doubletapp support”

AT: doubletapp should be a website, like the original tapp, and it should use the catppuccin colour scheme.

“any specific technologies you would like implemented?”

AT: I am a fan of Svelte, and would like to use rust as the backend due to its fast speeds and growing technology base, Tapp doesn’t have a database but SQL would be acceptable.

“doubletapp might have a cheating proble, would you like an anticheat?”

AT: an anticheat would be desirable, due to Svelte being unobfuscated a server side anticheat might be best

0.3. Research

https://rust-lang.github.io/async-book/

0.4. Prototyping

A rudimentary prototype has been made, which tested out multiple different input methods for simultaneous inputs, which has finalized in a “cursor”-based system, where you have two cursors controlled by Wasd-like movement, with each set of controls representing their respective cursor, additionally it has been decided that both cursors need to be on individual Tiles, to prevent copying movements on each hand. this prototype also implemented server-side move verification, making it more difficult to cheat. Finally, the UI design of the prototype will be used in later iterations of the project.

0.5. Documented Design

0.5.1. Algorithms

0.5.1.1. Xoshiro256+

after considering many PRNG’s (pseudorandomnumber generators), for example ARC4, seedrandom, ChaCha20, and discounting them due to performance issues / hardware dependent randomization, I decided on using the Xoshiro/Xoroshiro family of algorithms, which are based on the Linear Congruential Generators, which are a (now-obsolete) family of PRNG’s, which use a linear multiplication combined wit hmodulus operations, to create quite large non-repeating sequences, although quite slow and needing very large state. xoshiro generators use a much smaller state (between 128-512) bits, while still maintaining a large periodicity,

```
        // output is generated before the "next" cycle
        let result = self.seed[0].wrapping_add(self.seed[3]);
        // shifting prevents guessing from linearity
        let t = self.seed[1] << 17;

        // these 4 xor operations simulate a matrix transformation
        self.seed[2] ^= self.seed[0];
        self.seed[3] ^= self.seed[1];
        self.seed[1] ^= self.seed[2];
        self.seed[0] ^= self.seed[3];

        // last xor is just a xor
        self.seed[2] ^= t;
        // the rotation ensures that all bits in the seed eventually interact, allowing for much higher periodicity (cycles before you get an identical number, which in the case of xoshiro256+ is 2^256 - 1)
```

```
        self.seed[3] = Xoshiro256plus::rol64(self.seed[3], 45);

            // gets the first 53 bits of the result, as only the first 53 bits are guaranteed to be unpredictable for xoshiro256+, for the other variations i.e ++,*,** they are optimized for all the bits to be randomized, but as xoshiro256+ is optimized for floating points, which we require
        (result >> 11) as f64 * (1.0 / (1u64 << 53) as f64)
```

0.5.1.2. Sigmoid Function

the sigmoid function is a function, that maps any real input onto a S shaped curve, which is bound between values, in my case i am bounding the output of the Xoshiro256+ float to be between 0..11, which allows me to easily use it to generate the “next” state of the game, allowing for a more natural distribution of numbers, as well as a more consistent distribution of numbers, which allows for a more consistent game experience.

```
// simple function, but incredibly useful
fn sigmoid(x: f64) -> f64 {
    1.0 / (1.0 + (-x).exp())
}
```

0.5.1.3. Djikstras Algorithm

0.5.1.4. MergeSort

mergesort is a sorting algorithm, which works by the divide and conquer principle, where it breaks down the array into smaller and smaller arrays, till it gets to arrays of length 2, which it then subsequently sorts from the ground up, returning a sorted array in O(nlog(n)) time complexity & O(n) space complexity

0.5.1.5. Std Dev + Variance

0.5.2. Data Structures

0.5.2.1. Circular Queue

A queue is a data structure following the FIFO (first in first out) principle, where you use a sized array, along with variables to store the capacity, front & back of the array, when a file is queued, the file is put onto the index of the back of the array, and then the back index is added to % capacity unless the back becomes equal to the front, in which the queue returns an error instead, this allows for a non resizable array, which allows a set amount of elements to be

queued, but not more than the size of the array, allowing for efficient memory management

0.5.2.2. HashMap

A hash table (colloquially called a hashmap) is an array that is abstracted over by a “hashing” function, which outputs an index based on an output, usually the hash function aims to be as diverse as possible, but you can also write special hash functions that are more efficient for your given data types.

0.5.2.3. Option/Result Types

an Optional type, is a simple data structure that allows for beautiful error handling, an Option type wraps the output data, allowing for the error to be handled before trying to manipulate data, i.e in a Some(data) or None, where None means that the data was nonexistent, or we can use a result type to handle errors down the stack, where we can pass the error with Err(e) and Ok(d), so if one part of the function layer breaks we can know exactly where it errored and softly handle the error if needed

0.5.2.4.

0.5.3. Function Map

0.5.4. Class Diagrams

0.5.5. Database Design

0.5.6. Mockups & etc..

0.6. Objectives

0.6.1. User Interface

1 user can interact with the grid
  1.1 user can move both cursors using keyboard on the grid
  1.2 user can “submit” moves using a keybind
  1.3 user can reset game (in single player) via a keybind
2 user can change gamemode (singleplayer,multiplayer) on the main page
  2.1 user can change grid size (4x4,5x5,6x6) in singleplayer
  2.2 in singleplayer, user can change time limit (30,45,60)
3 user can access settings
  3.1 user can modify keybinds for each action in the game
  3.2 user can change DAS
  3.3 user can change ARR

- 3.4 user can log out of account
- 3.5 user can reset all keybinds to a sane default

4 user can play the game

- 4.1 on game start, user sees cursors are positioned on opposing sides of the board
- 4.2 on game start, user sees the starting active tiles
- 4.3 user can view current game score
- 4.4 in singleplayer, user sees time remaining
- 4.5 in multiplayer, user can see time remaining for current quota, players remaining and current score
- 4.6 user is notified of their position in the multiplayer game
- 4.7 user can “submit” their move
  - 4.7.1 user can interactively see if the move was valid via a colour interaction which flashes green or red depending on if the move was valid, a valid move is when the two cursors are on two active grid tiles within the grid boundary and they are distinct active tiles
  - 4.7.2 on successfull submit, user sees two new tiles become active, which were previously inactive and are not on current cursor location
- 4.8 cursors are rendered via two different colours, with the two cursors being visually distinct but symmetrically consistent

5 user can see statistics post singleplayer game end

- 5.1 user views their score
- 5.2 user views if their score was validated by the server
- 5.3 user views their leaderboard position
- 5.4 user can copy their game statistics to the clipboard for sharing
- 5.5 if user is logged in and not marked as a cheater, user can view their game in the statistics page
- 5.6 user has the option to start a new game from the results menu

6 user can view leaderboard

- 6.1 user can view leaderboards, in a paginated format

7 user can play the multiplayer gamemode

- 7.1 user can see the other players movements on other grids in the game
- 7.2 user can see their remaining score quota for each 5 second interval period
- 7.3 after a user has been eliminated by not reaching the quota,the user can view their position in the game

8 user can log in to the application

- 8.1 user can login or signup depending on their requirements

8.2 user is shown error codes depending on if account already exists or their login details are incorrect

0.6.2. Server Side

1 User CRUD
  1.1 simple user authentication
    1.1.1 simple verification of authenticity, i.e password hashing & username uniqueness check
2 Database Schema
  2.1 contains user table
  2.2 contains game table, which stores all real authenticated games (not including moves)
  2.3 contains linked user statistics table
3 Game Verification
  3.1 server verifies all moves are valid
  3.2 server verifies that move positioning is within human bounds, i.e ratio of “optimal moves” and timing distribution
  3.3 server verifies that game was submitted within the time limit (with a grace period)
4 Multiplayer implementation
  4.1 server can communicate actions bidirectionally with client
  4.2 each move is verified by the server
  4.3 low latency communication between server and client
  4.4 client can distinguish between types of messages recieved