Multiplayer Grid Based Dexterity Training Game

NEA: Georgiy Tnimov

Contents

0.1. Abstract

This project develops a multiplayer grid-based dexterity training game called DoubleTapp, designed to simultaneously test and improve the dexterity of both hands. Building on the existing single-cursor game Tapp, this implementation introduces dual-cursor gameplay requiring coordinated control using different keys for each hand. The system features both singleplayer and multiplayer modes with competitive elements, leaderboards, and server-side anti-cheat mechanisms.

The technical implementation uses Rust for the backend with the Axum framework for websocket connections and PostgreSQL for data persistence. The frontend is built with SvelteKit and Tailwind CSS, featuring customizable controls including Delayed Auto Shift (DAS) functionality. A custom implementation of the Xoshiro256+ PRNG algorithm ensures fairness across game instances.

0.2. Problem Definition

I plan to develop a game, which tests the dexterity of both hands, simultaneously. I believe its important that people can maintain their dexterity of both hands, and this game will help them do that. I also believe the game will be fun, and will be a

good way to pass time. adding a competitive and multiplayer aspect to the game will also help with this.

I plan to develop this game using Rust and Svelte, as well as a websocket server, which will be used to communicate between the client and server.

0.3. Client

0.3.1. Client Synopsys (conclusion)

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.

0.3.2. Interview Notes

(all notes are paraphrased)

Q: What features are most important to you for DoubleTapp?

A: 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.

Q: How many users do you expect to scale to?

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

Q: Any specific UI/GUI choices, and what platform should DoubleTapp support?

A: DoubleTapp should be a website, like the original Tapp, and it should use the Catppuccin color scheme.

Q: Any specific technologies you would like implemented?

A: 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.

Q: DoubleTapp might have a cheating problem, would you like an anticheat?

A: An anticheat would be desirable. Due to Svelte being unobfuscated, a server-side anticheat might be best.

Q: What are your thoughts on monetization for DoubleTapp?

A: I’d prefer to keep it free to play. The focus should be on building a community rather than generating revenue at this stage.

Q: How important is cross-device compatibility?

A: The primary focus should be desktop browsers, but having it work reasonably well on tablets would be a nice bonus. I don’t expect mobile phone support due to the dual-input nature.

Q: Any accessibility considerations you'd like to see implemented?
A: Customizable keybindings would be essential since this is a dexterity game. Also, ensuring the color scheme has sufficient contrast for visibility would be good.

0.4. Success Criteria

- game is completely functional
- server can handle 50 concurrent users
- average user rating is 4/5 or higher
- aesthetically pleasing UI
- useful UX
- easy to understand and customize settings

0.5. Research

0.5.1. Similar Solutions

There are a few similar products on the market that test dexterity in various ways. Understanding these existing solutions helps position DoubleTapp in the competitive landscape and justify its development.

0.5.1.1. Tetris

Tetris is one of the most recognized dexterity-based puzzle games worldwide. While it effectively tests hand-eye coordination and spatial reasoning, it differs from DoubleTapp in several key ways:

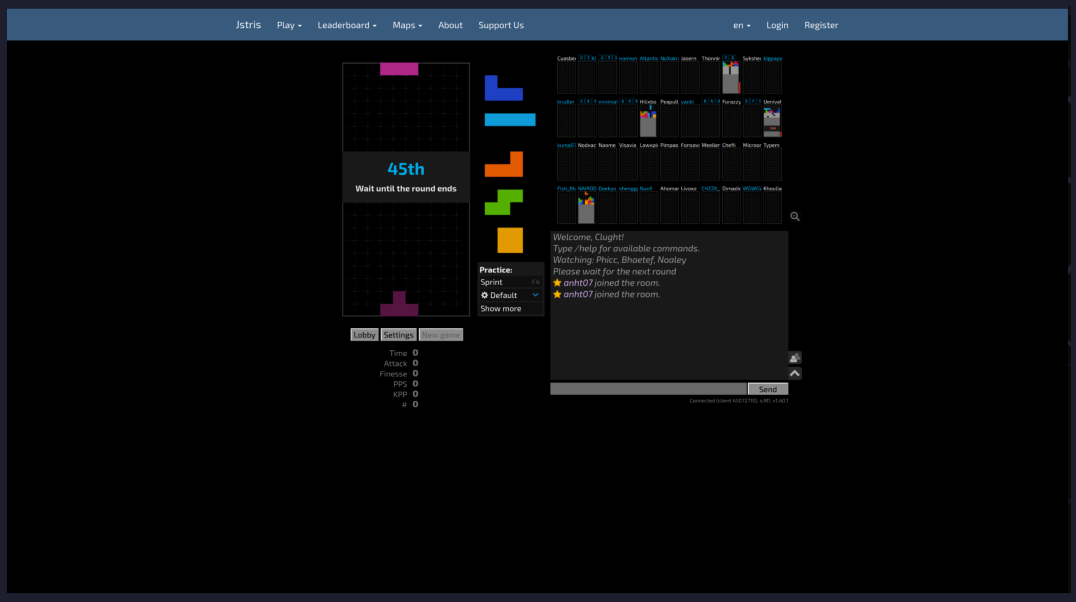

Figure 1: Tetris UI

- Tetris focuses primarily on single-hand dexterity, with players typically using one hand for directional controls and the other for occasional rotation/drop buttons
- It has a significant learning curve with complex strategies around piece placement and line clearing, i.e T-spins, Wall Kicks
- Players focus more on strategic planning of where to place pieces rather than pure dexterity training
- The modern competitive versions of Tetris (like Tetris 99) do incorporate multiplayer aspects, but interaction between players is indirect through "garbage lines"

Tetris has multiple useful features which I will be taking inspiration from, particularly Delayed Auto Shift (DAS)[1], which allows for precise control of pieces, this allows for people to have more accurate control over their piece placement and allows for timing optimization

0.5.1.2. Tapp

Tapp, developed by Alexander Tahiri at Studio Squared, is the direct predecessor to DoubleTapp and shares the most similarities:

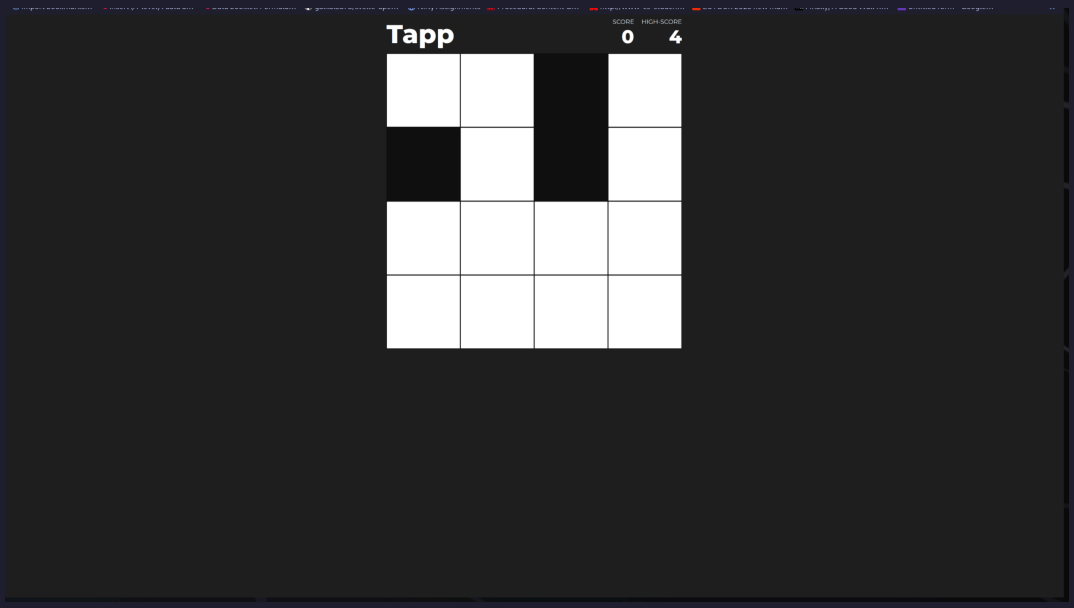

Figure 2: Tapp UI

- Uses a grid-based interface (4x4) with active and inactive tiles
- Tests dexterity through rapid target acquisition
- Focuses on score maximization without mistakes
- Simple, accessible gameplay with minimal learning curve

However, Tapp is limited to single-hand dexterity training, using only mouse input. It lacks the simultaneous dual-hand coordination that DoubleTapp aims to develop. Additionally, Tapp has no built-in multiplayer functionality or competitive leaderboard system.

0.5.1.3. Other Dexterity Training Applications

Various other applications exist for dexterity training, including:

- Typing games that test two-handed coordination but in a highly structured, predictable pattern (monkeytype, nitrotype)
- Rhythm games (like Dance Dance Revolution or osu!) that test reaction time and coordination but typically focus on timing rather than spatial navigation
- Aim trainers (for FPS games) that focus exclusively on mouse precision, although sometimes incorporate simultaneous dexterity, i.e counterstrafing, bhopping, edgebugging

0.5.2. Multiplayer

for implementing multiplayer, there are multiple solutions that work, i.e unidirectional HTTP requests, custom UDP handling, and websockets

Method	Pros	Cons
HTTP [2]	• Simple implementation • Reasonably performant • Easily Debuggable • widely supported	• Slow with many simultaneous users • Requires entire connection sequence for each request • relatively high latency • not designed for bidirectional communication
UDP [3]	• Very performant • allows for low level optimisations • minimal overhead	• susceptible to packet loss, and is not guaranteed to have data parity (important for doubletapp) • complex to implement, and difficult to interconnect with existing libraries without significant performance declines • often blocked by firewalls • no ordering guarantees
Websockets [4]	• allows for fast and safe data transmission • relatively complex to implement, as need to handle assignment of websockets to individual games • compatible with existing web server libraries • fully duplex, no need to reestablish connection sequence each request	• websockets don't recover when connections are terminated • some networks block the websocket protocol, limiting accessibility • high memory usage per connection compared to UDP/ HTTP

I have decided to use websockets, as they are a reasonable balance of complexity, performance, and ease of implementation, while still providing a high degree of reliability and safety.

0.5.3. PRNG's (Pseudorandom Number Generators)

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 with modulus 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,

PRNG Algorithm	Pros	Cons
ARC4 (Alleged RC4)	• Simple implementation • Fast for small applications • Variable key size	• Cryptographically broken • Biased output in early stream • Vulnerable to related-key attacks
Seedrandom.js	• Browser-friendly • Multiple algorithm options • Good for web applications	• JavaScript performance limitations • Depends on implementation quality • Not cryptographically secure by default
ChaCha20	• Cryptographically secure • Excellent statistical properties • Fast in software (no large tables) • Parallelizable	• Complex implementation • Overkill for non-security applications • Higher computational cost
Xorshift	• Extremely fast • Simple implementation • Good statistical quality	• Not cryptographically secure • Simpler variants have known weaknesses • Some states can lead to poor quality
Linear Congruential Generator (LCG)	• Simplest implementation • Very fast • Small state	• Poor statistical quality • Short period for 32-bit implementations • Predictable patterns
Mersenne Twister	• Very long period • Good statistical properties • Industry standard in many fields	• Large state (2.5KB) • Not cryptographically secure • Slow initialization
Xoshiro256+/++	• Excellent speed • Great statistical properties • Small state (256 bits) • Fast initialization	• Not cryptographically secure • Newer algorithm (less scrutiny)

		• Some variants have issues with specific bits
PCG (Permuted Congruential Generator)	• Excellent statistical properties • Small state • Good performance • Multiple variants available	• More complex than basic PRNGs • Not cryptographically secure • Relatively new

PRNG Algorithm	Estimated Time	Cycle Length	State Size	Performance
ARC4	Medium	$10^{\{100\}}$	256 bits	Moderate
seedrandom.js	Medium	(multiple selectable algorithms)	Varies by algorithm	Moderate (JS limited)
ChaCha20	High	2^{256}	384 bits	High for crypto
Xorshift	Very Low	$2^{128} - 1$	128-256 bits	Very High
Linear Congruential Generator (LCG)	Extremely Low	Up to 2^{32}	32-64 bits	Extremely High
Mersenne Twister	Medium	$2^{19937} - 1$	2.5 KB (19937 bits)	Moderate
Xoshiro256+/++	Very Low	$2^{256} - 1$	256 bits	Very High
PCG (Permuted Congruential Generator)	Low	2^{128} or more	64-128 bits	High

after testing, xoshiro256+ has provided the best results, in terms of speed and simplicity of implementation, while still providing a high degree of randomness, and a large cycle length, which is important for a game such as DoubleTapp, where we want to ensure that the game is fair and that the same seed will not be repeated for a long time.

0.5.4. Statistics(anti-cheat)

for the anticheat,I will be comparing the consistency of player movement timings, and the optimality of their paths, to approximately determine if they are using any forms of cheating, be it a bot, or a human using external software.

0.5.4.1. Player timings

for player timings, I will be using the standard deviation of the player's move timings, and comparing it to a sampled standard deviation based on my own move timings, a high standard deviation indicates that the player is more human, as different grid positions require different amounts of thought to move optimally

0.5.4.2. Path optimality

for calculating optimal paths, there are a few different algorithms that can be used, each having different time and space complexities, it is important that the algorithm calculates the optimal path, not a close approximation, as this will be used to detect potential cheaters. performance is inherently critical for this part, as it will be run on every "submission" of a move, and will need to be done concurrently.

Algorithm	Time Complexity	Space Complexity
A-Star	$O(b^d)$	$O(b^d)$
Djikstra's	$O(V+E)$	$O(V)$
Manhattan Distance	O(1)	O(1)

overall, manhattan distance is the best option for this project, as at max the grid would be 6x6, in which using A-star would be overkill, and manhattan distance is the fastest, while djikstra's is the slowest, and would be too slow for the game.

0.6. 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. the prototype has no game verification, but contains the core gameplay mechanics, and the UI design.

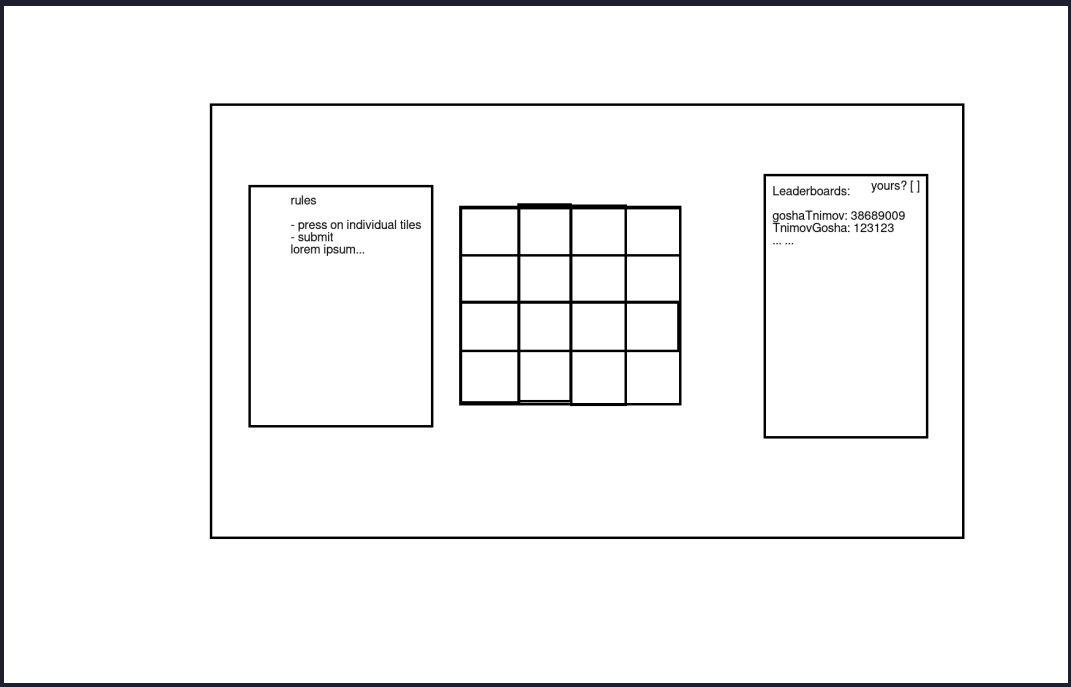

Figure 3: Initial Doubletapp WireFrame UI

this was the initial UI design sketch,which shows the general layout of the game

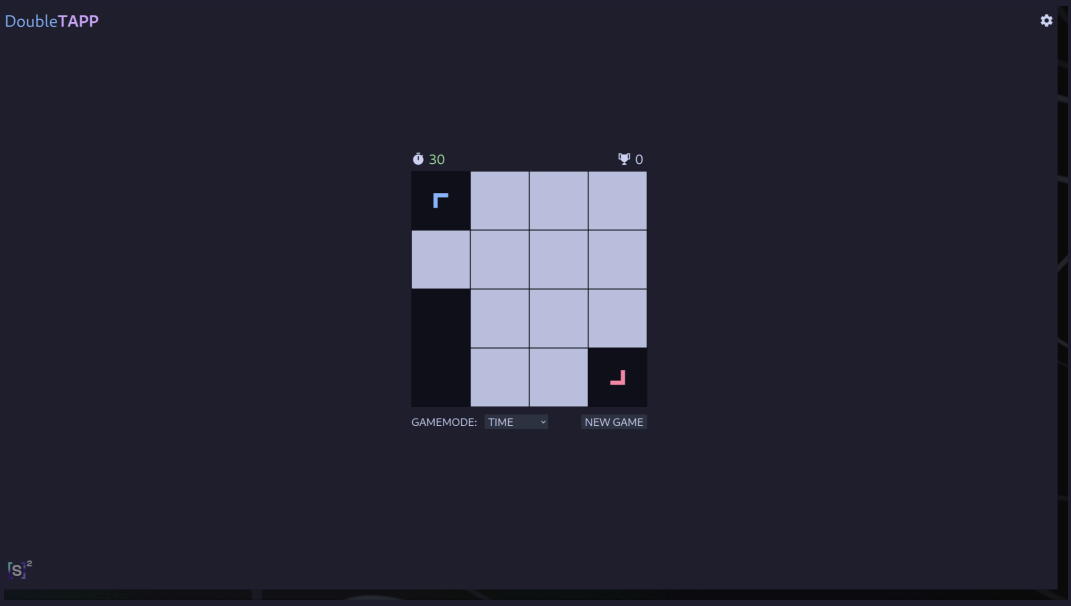

Figure 4: Initial Doubletapp WireFrame UI

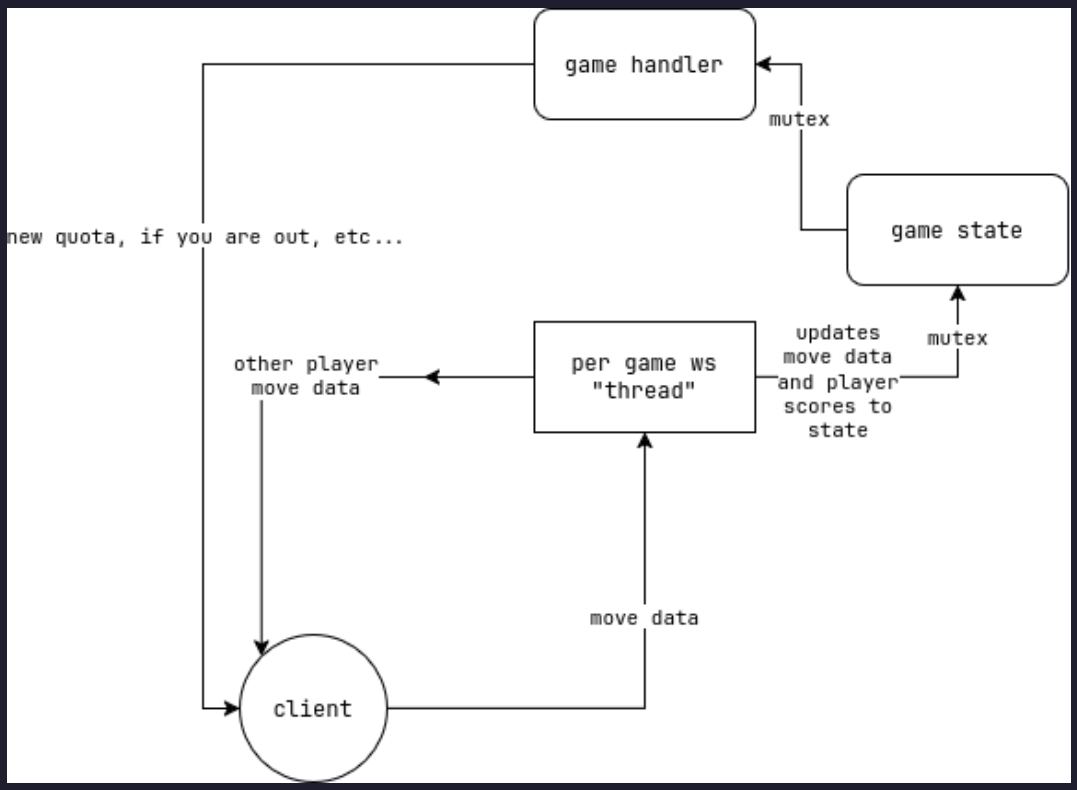

Figure 5: Game Handler Prototype Flowchart - Early design of the game processing pipeline

0.7. Critical Path

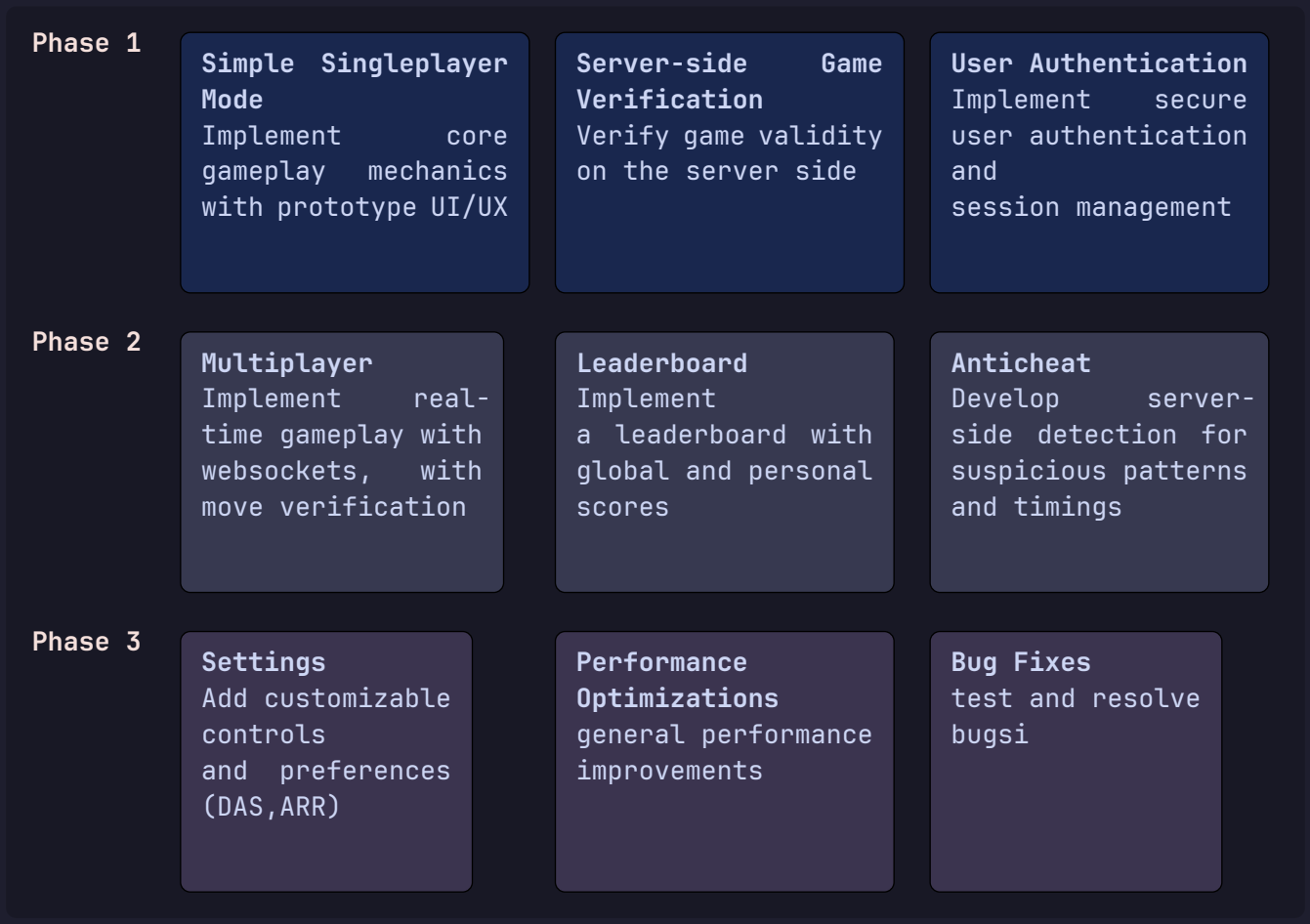

Figure 6: Intended Critical Path

0.8. Objectives

0.8.1. 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

0.8.2. 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

0.9. Documented Design

0.9.1. Libraries Used

0.9.1.1. Frontend Libraries

Name	Version	Reason	Link
Svelte	4.2.7	Reactive UI framework with minimal boilerplate, used for the frontend to provide a performant, easily maintainable UI/UX	svelte.dev
SvelteKit	2.0.0+	Full-stack framework built on Svelte, allowing for simplification of operations between the frontend and the backend	kit.svelte.dev
Tailwind CSS	3.4.4	css library, which allows you to define your css classes embedded in the html, allowing for a more readable and quickly iterable codebase	tailwindcss.com
Tailwind Catppuccin	0.1.6	Client-requested color scheme	GitHub
Svelte Material Icons	3.0.5	Icon library for Svelte, MIT licensed	npm
UUID	11.0.4	frontend library for generating UUID's, used for game management	npm
Xoshiro WASM	Local	Custom WASM implementation of Xoshiro256+	in code
TypeScript	5.0.0+	Typed JavaScript for better development	typescriptlang.org
Vite	5.0.3	Modern frontend build tool, used in frontend to allow for fast development and optimized production builds	vitejs.dev

Vite Plugin WASM	3.4.1	Vite plugin for WebAssembly integration	npm

0.9.1.2. Backend Libraries

Name	Version	Reason	Link
Axum	0.7.5	Modern Rust web framework with WebSocket support, one of the fastest web frameworks currently available, asynchronous and type-safe	GitHub
Axum-Extra	0.9.4	Extension crate for Axum with additional features like cookie handling and typed headers	GitHub
Tokio	1.39.2	Asynchronous runtime for Rust, required by axum and used for thread handling in websockets	tokio.rs
SQLx	0.8.0	Async SQL toolkit with compile-time checked queries, used for database operations, inherently supports pooling and multithreading.	GitHub
Serde	1.0.205	Serialization framework for structured data, allows for parsing JSON and other data formats into Rust objects, speeding up development time and reducing the amount of code needed to be written	serde.rs
Serde_json	1.0.128	JSON implementation for Serde, used for parsing and generating JSON data in WebSocket communication	GitHub
Bcrypt	0.17.0	Password hashing library, used before storing passwords in the database, salted and performant, although slightly outdated	crates.io
Tower-HTTP	0.5.2	HTTP middleware stack, baseline from axum, used for low level websocket handling	GitHub
UUID	1.7.0	Library for generating UUIDs, used for game management	crates.io
ULID	1.1.3	Sortable identifier generation, used for game management	crates.io

Validator	0.20.0	Data validation library, used for validating user input	crates.io
Chrono	0.4.37	Date and time library with timezone support, used for handling timestamps and durations to verify games	crates.io
SCC	2.1.11	Concurrent collections for server applications, performant asynchronous hashmaps	crates.io
Silly-RNG	0.1.0	Custom RNG implementation, used for the game, based on xoshiro-wasm	Local package
Cookie	0.18.1	HTTP cookie parsing and cookie jar management, used for session handling	crates.io
Dotenvy	0.15.7	Loads environment variables from .env files, used for configuration management	crates.io
Futures	0.3.31	Async programming primitives, used for handling asynchronous websocket operations	crates.io
Rand	0.8.5	Random number generation utilities, used for game seeding	crates.io
Thiserror	2.0.11	Error handling library that simplifies custom error types, used for robust error management	crates.io
Tracing-subscriber	0.3.18	Utilities for implementing and composing `tracing` subscribers, used for logging and diagnostics	crates.io

0.9.2. Algorithms

0.9.2.1. Xoshiro256+

xoshiro256+ is my chosen RNG, as it is performant and has a relatively low state size, allowing for many concurrent games to be played on a single machine, it is also very simple to implement, and has a relatively high cycle length, allowing for a more consistent game experience, it is also very fast, and has a low memory footprint, making it a perfect fit for the game. xoshiro256+ has a time complexity of O(1), and a space complexity of O(1), as it only requires a single pass through the seed array, and a single pass through the result array, which is constant time, and constant space, as the size of the seed and result arrays are constant.

```
        // 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)
```

0.9.2.2. 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())
}
```

0.9.2.3. Manhattan Distance

the manhattan distance is a distance metric, which is the sum of the absolute differences of their Cartesian coordinates, in my case i am using it to calculate the distance between the cursors, which allows for a more accurate calculation of the distance between the cursors, which allows for a more accurate game experience. I considered other algorithms, i.e djikstras, A-Star, but they are not needed for calculating the distance between the cursors, as the manhattan distance is a more efficient algorithm for this purpose.

the time complexity of the manhattan distance is O(1), as it only requires a single pass through the coordinates, and a single pass through the result, which is constant time, and constant space, as the size of the coordinates and result are constant.

```
fn manhattan_distance(x1: f64, y1: f64, x2: f64, y2: f64) -> f64 {
    (x1 - x2).abs() + (y1 - y2).abs()
}
```

0.9.2.4. 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

```
fn merge_sort<T: Ord + Clone>(arr: &[T]) -> Vec<T> {
    if arr.len() <= 1 {
        return arr.to_vec();
    }

    let mid = arr.len() / 2;
    let left = merge_sort(&arr[..mid]);
    let right = merge_sort(&arr[mid..]);

    merge(&left, &right)
}

fn merge<T: Ord + Clone>(left: &[T], right: &[T]) -> Vec<T> {
    let mut result = Vec::with_capacity(left.len() + right.len());
    let mut left_idx = 0;
    let mut right_idx = 0;

    while left_idx < left.len() && right_idx < right.len() {
        if left[left_idx] <= right[right_idx] {
            result.push(left[left_idx].clone());
            left_idx += 1;
        } else {
            result.push(right[right_idx].clone());
            right_idx += 1;
        }
    }

    result.extend_from_slice(&left[left_idx..]);
    result.extend_from_slice(&right[right_idx..]);

    result
}
```

0.9.2.5. Standard deviation

the algorithm for standard deviation is as follows:

$$\sigma = \sqrt{\frac{(\sum(x) - \mu)^2}{N}}$$

where N is the number of elements in the array, x_i is the ith element in the array, and mu is the mean of the array.

which can be implemented quite neatly in rust, using iterators, and their respective methods.

```
fn std_dev(arr: &[T]) -> T {
    let sum = arr.iter().sum::<T>();
    let mean = sum / arr.len() as T;
    let variance = arr.iter().map(|x| (x - mean).powi(2)).sum::<T>() / arr.len() as T;
    return variance.sqrt()
}
```

0.9.2.6. Delayed Auto Shift

Delayed auto shift (DAS for short) is a technique implemented in tetris, where you wait for a period of time before starting to move the pieces, while the key is being held down, bypassing the operating systems repeat rate. This is useful for optimizing movements in games similar to DoubleTapp, or tetris, people can customize their DAS and their ARR(auto repeat rate) to be optimal for their own reaction time, so if they need to move a piece they can move it to the corners very quickly, but only after X time has passed, instead of the OS default of 1 second for delay and 100ms per repeat, in my algorithm I used the provided javascript api's of setTimeout and setInterval, wrapped inside an asynchronous function to allow for multiple consecutive inputs, I separately handle keyDown and keyUp events, where on key down the interval is added to an array of intervals (thanks to javascripts type safety), in which the interval is cleared when an OS keyUP is detected, this comes with caveats as there are operating systems which send these events at different times, which can introduce some uncertainty. But due to the timings being customizeable, this isn't much of a problem.

```
// Example for one direction, repeated for others
case $state.keycodes.wU:
    if (dasIntervals[0] == false) {
        dasIntervals[0] = setTimeout(() => {
            dasIntervals[0] = setInterval(() => {
                wcursorY = Math.max(wcursorY - 1, 0);
                if ($state.gameMode === 'multiplayer') {
                    ws.send(JSON.stringify({
                        type: 'Move',
                        data: { player_id: `${temp_id}`, action: 'CursorBlueUp' }
                    }));
                }
                moves.push(['CursorBlueUp', Date.now() - lastActionTime]);
                lastActionTime = Date.now();
            }, $state.das);
        }, $state.dasDelay);
    }
```

0.9.3. Database Design and Queries

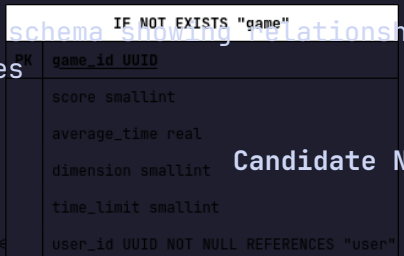

Figure 7: Entity Relationship Model - Database schema showing relationships between game entities

ERM

0.9.3.1. User Authentication Queries

```
SELECT id, password FROM "user" WHERE username = $1
```

this query is quite simple, it just selects the id and password from the user table, where the username is the same as the one provided, as the password is hashed before being stored, this method is secure. additionally it is run on the server side, preventing any XSS attacks, or SQL injections.

0.9.3.2. User Registration Query

```
INSERT INTO "user" (id, username, password) VALUES ($1, $2, $3)
```

0.9.3.3. Session Management

```
INSERT INTO session (ssid, user_id, expiry_date)
VALUES ($1, $2, NOW() + INTERVAL '7 DAYS')
SELECT u.id, u.username, u.admin, u.cheater
FROM "user" u
INNER JOIN session s ON u.id = s.user_id
WHERE s.ssid = $1 AND s.expiry_date > NOW()
```

0.9.3.4. Leaderboard Queries

```
-- Get global leaderboard
SELECT "game".score, "user".username
FROM "game"
JOIN "user" ON "game".user_id = "user".id
WHERE dimension = $1
AND time_limit = $2
ORDER BY score
OFFSET ($3 - 1) 100
FETCH NEXT 100 ROWS ONLY
-- Get user's personal scores
SELECT "game".score, "user".username
FROM "game"
JOIN "user" ON "game".user_id = "user".id
WHERE dimension = $1
AND time_limit = $2
AND "user".id = $4
ORDER BY score
OFFSET ($3 - 1) 100
FETCH NEXT 100 ROWS ONLY
```

0.9.3.5. Game Submission

```
INSERT INTO "game" (game_id, score, average_time, dimension, time_limit, user_id)
VALUES ($1, $2, $3, $4, $5, $6)
```

0.9.3.6. Statistics Trigger

```
CREATE OR REPLACE FUNCTION update_statistics_on_game_insert()
RETURNS TRIGGER AS $$
BEGIN
UPDATE user_statistics
SET
games_played = games_played + 1,
highest_score = GREATEST(highest_score, NEW.score)
WHERE user_id = NEW.user_id;
UPDATE statistics
SET
total_timings = total_timings + NEW.average_time,
total_score = total_score + NEW.score,
games_played = games_played + 1;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER game_insert_trigger
AFTER INSERT ON game
FOR EACH ROW EXECUTE FUNCTION update_statistics_on_game_insert();
```

0.9.4. Data Structures

0.9.4.1. 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

0.9.4.2. 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.

0.9.4.3. 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

0.9.5. Diagrams

0.9.6. Frontend

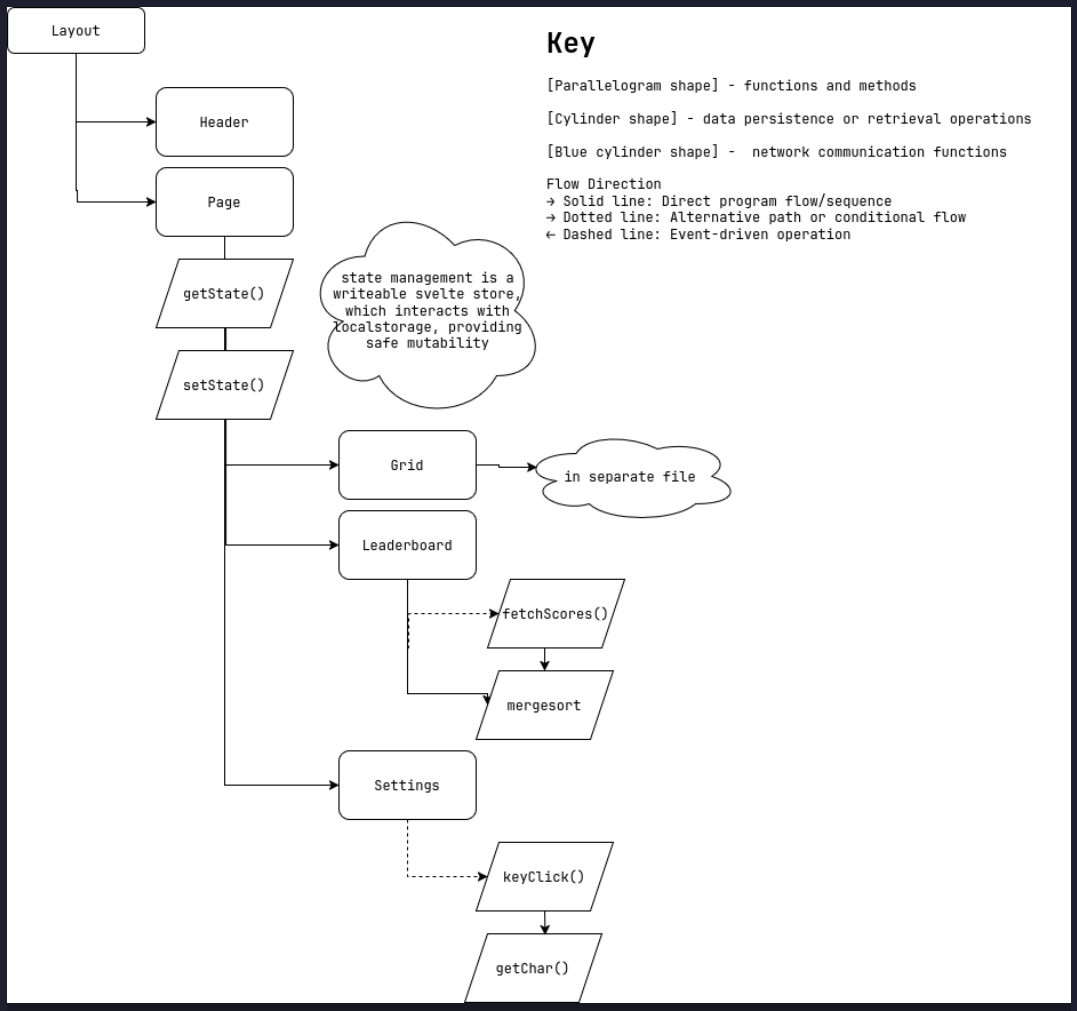

Figure 8: Client Component and Flow diagram

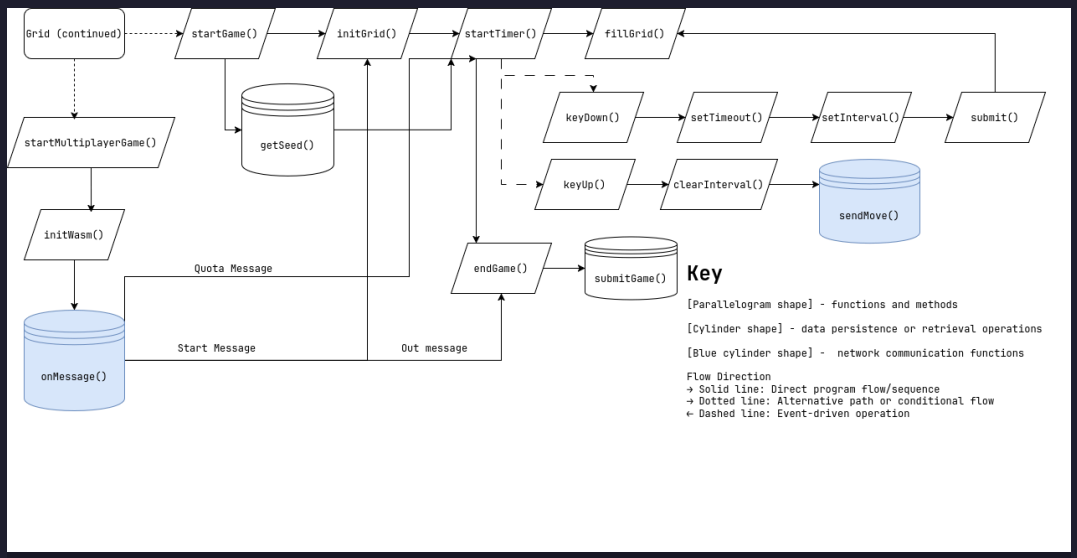

Figure 9: Grid Component

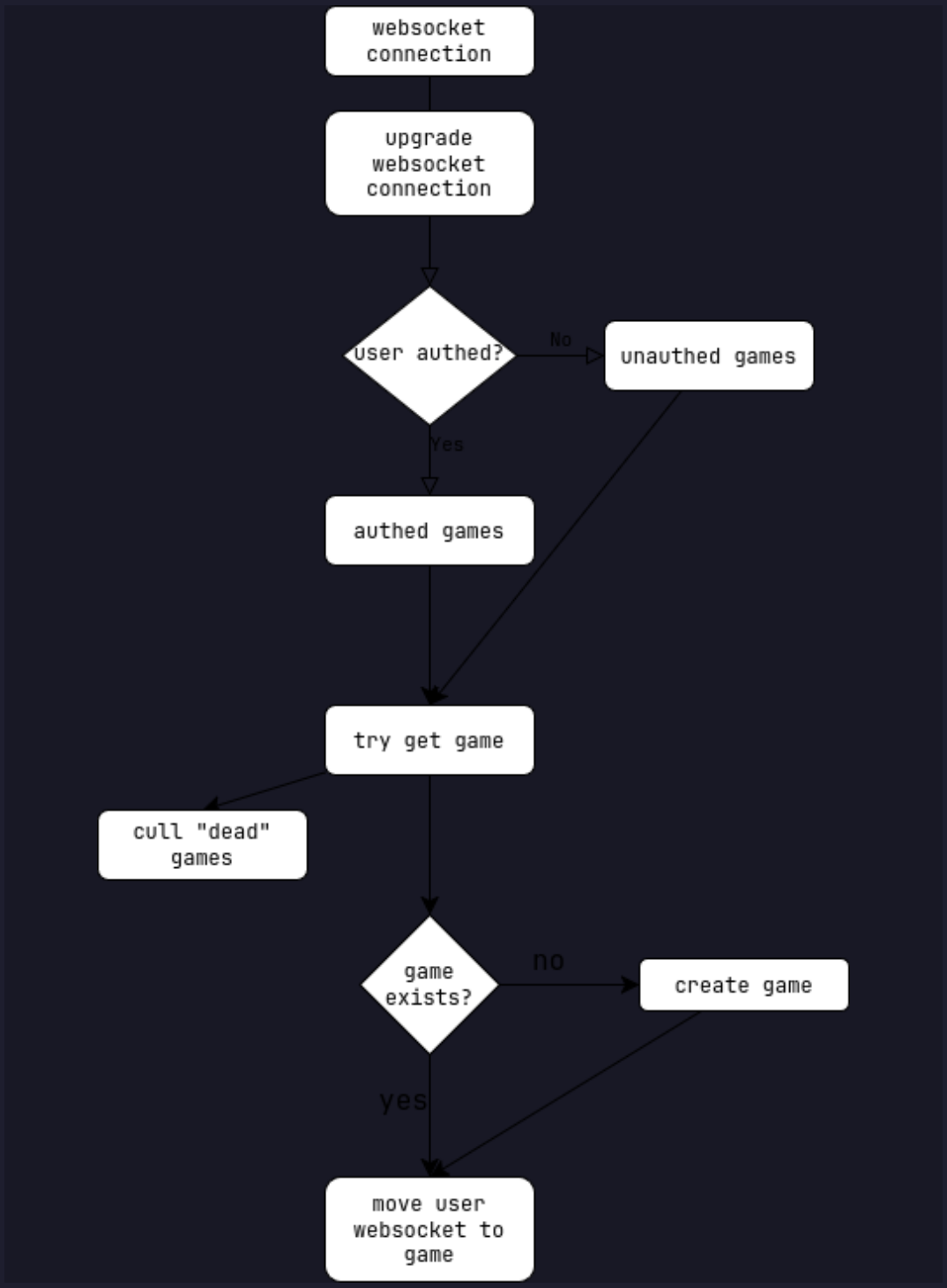

Figure 10: Game Handler Flowchart

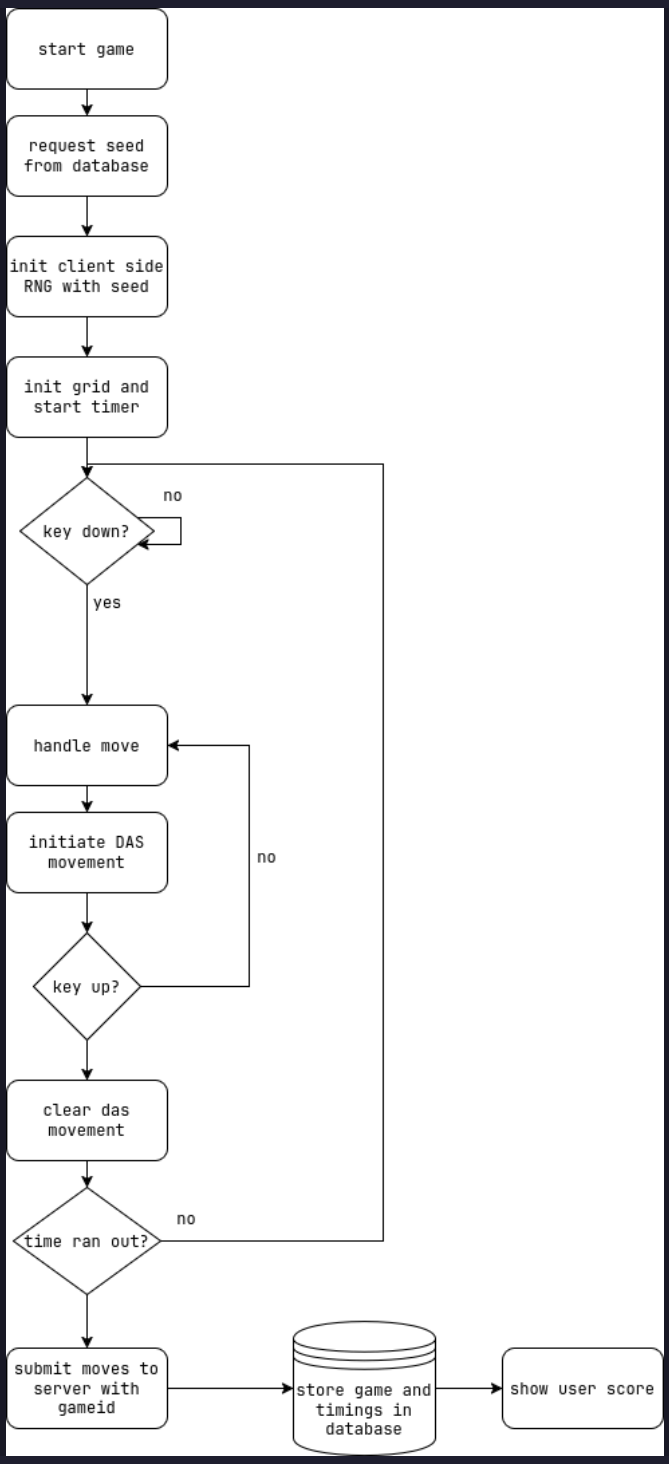

Figure 11: Singleplayer Game Flowchart

0.9.7. Backend

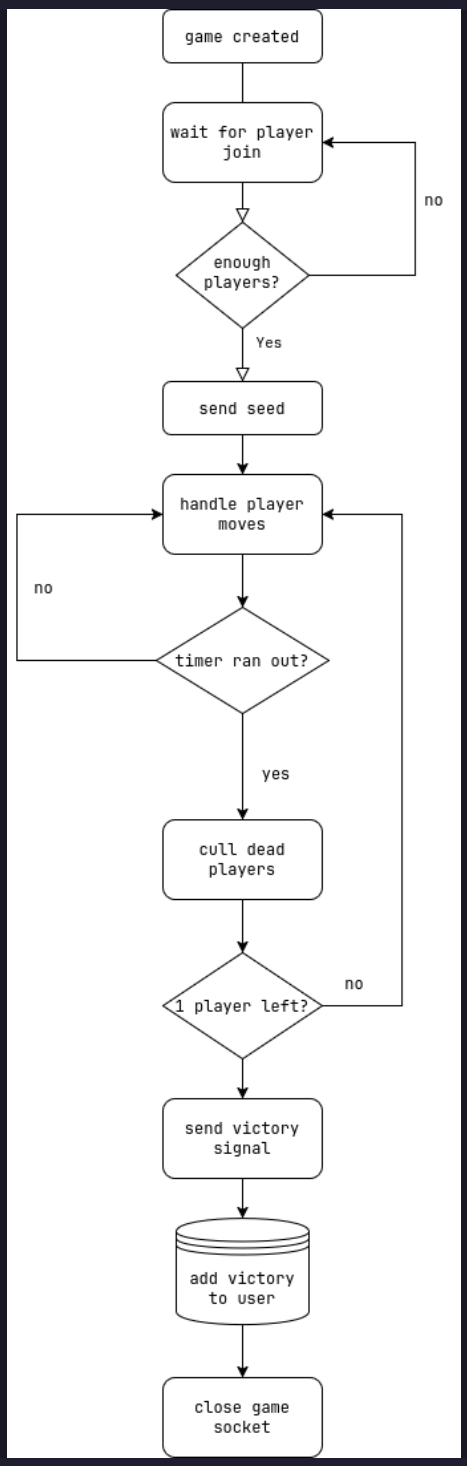

Figure 12: Multiplayer Game Flowchart

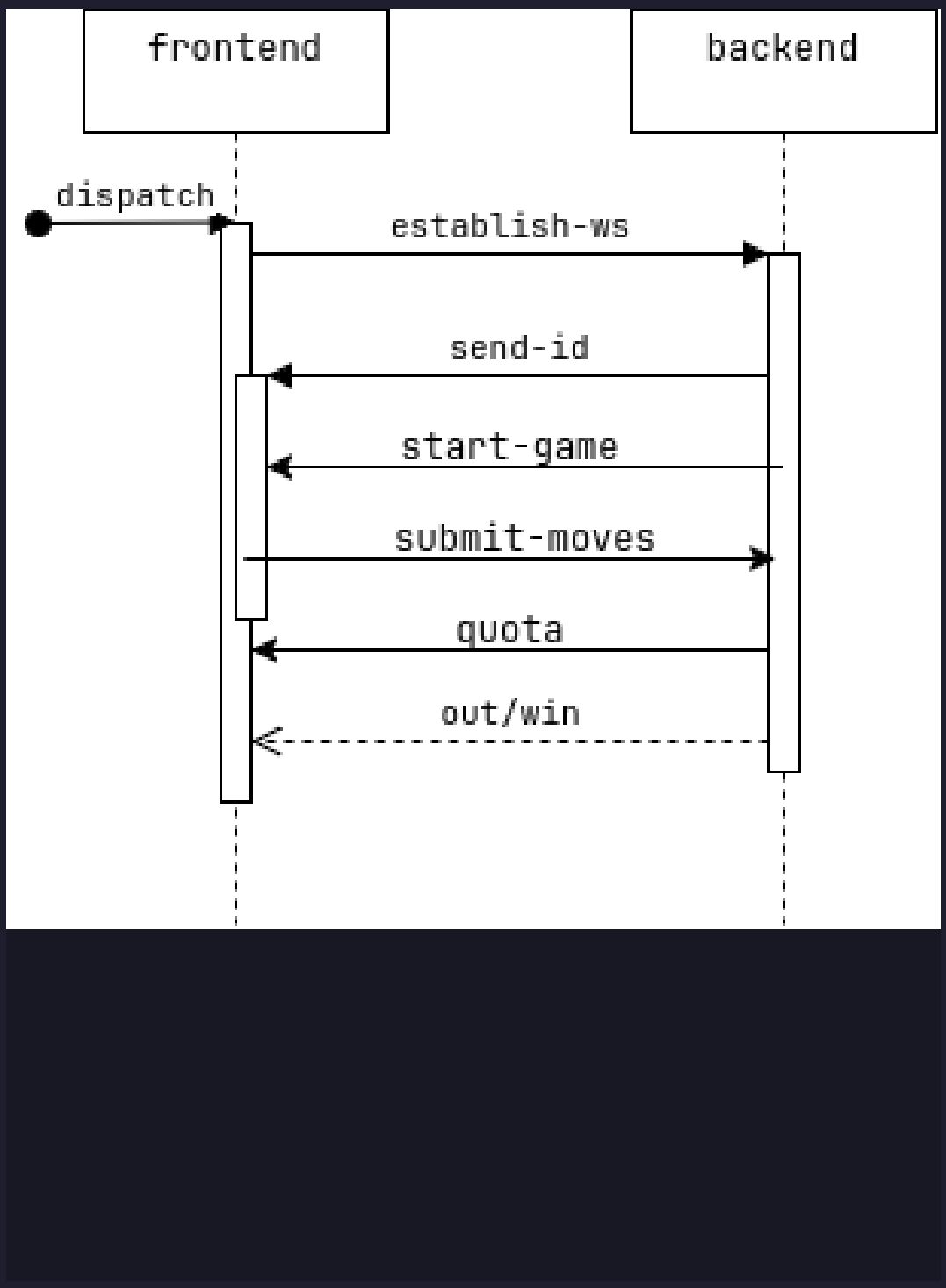

Figure 13: WebSocket message diagram

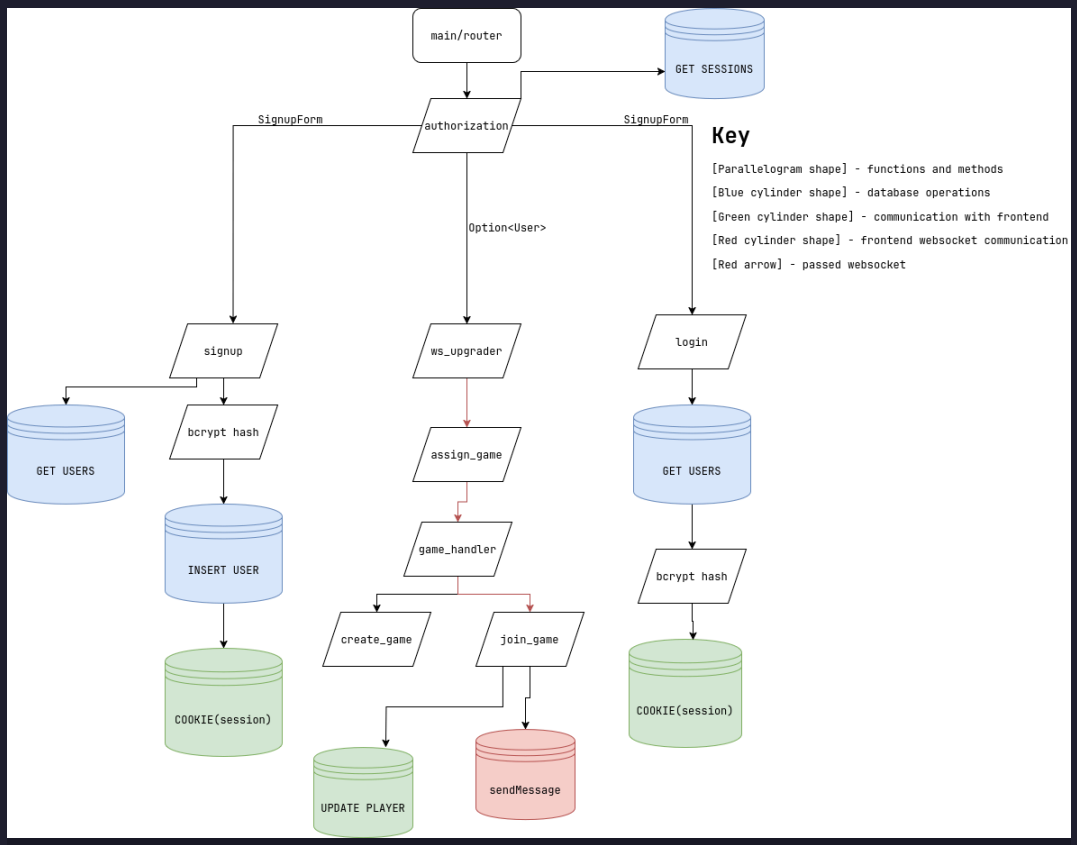

Figure 14: Backend Multiplayer Flowchart

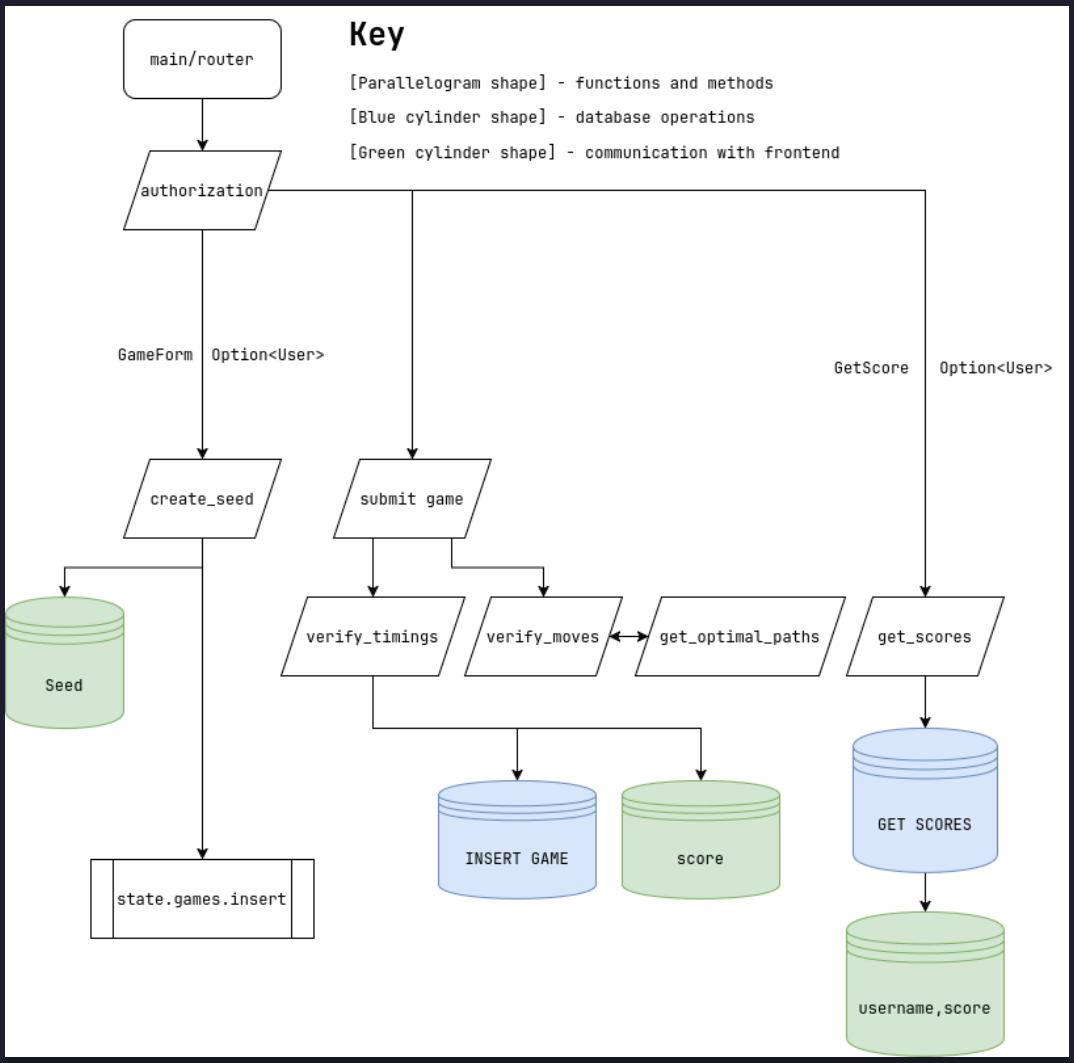

Figure 15: Backend Flowchart

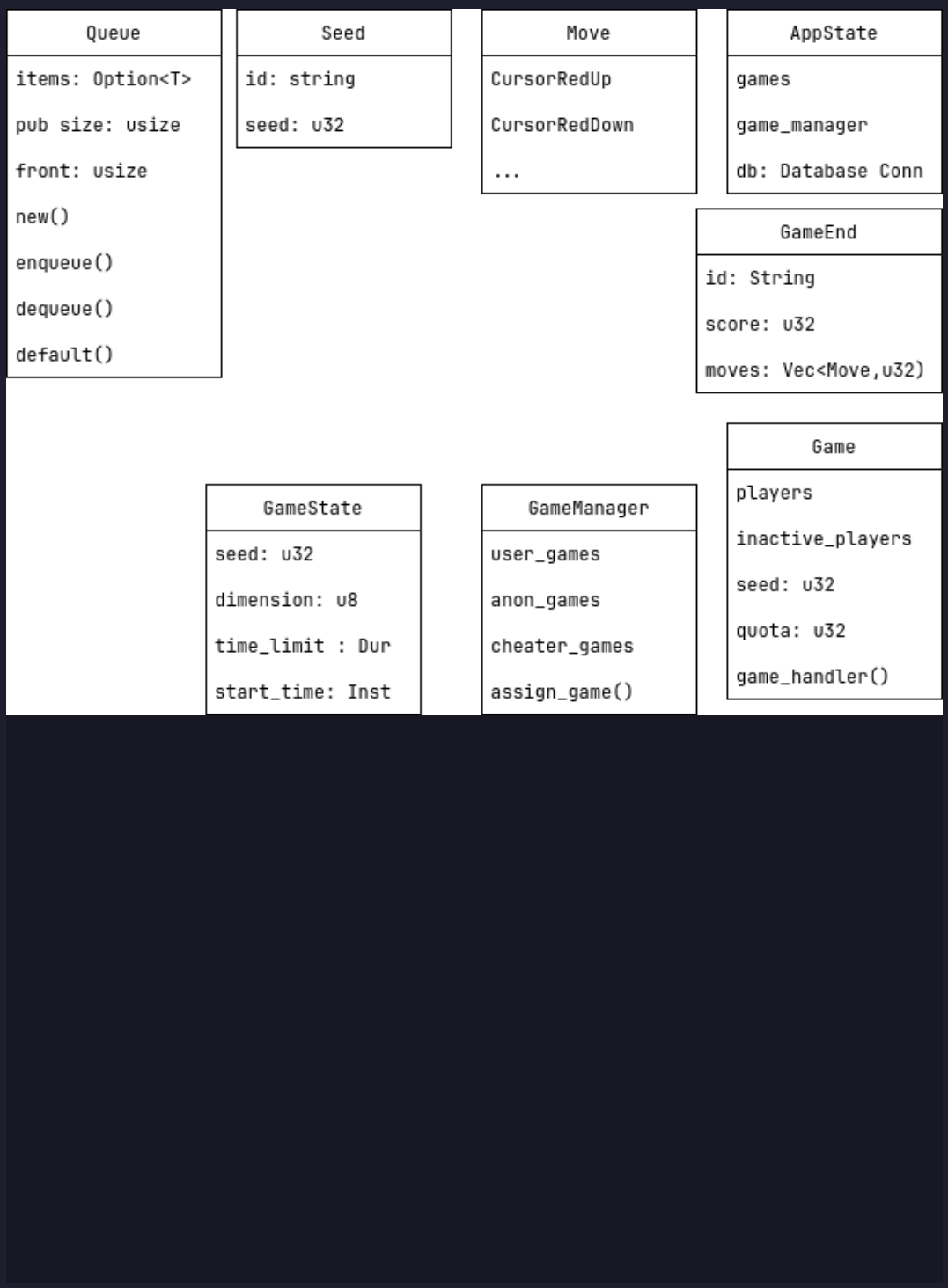

Figure 16: Class Diagram

0.10. Technical Solution

0.10.1. Code Contents

Component	Description	Path/Location
Grid	Core game grid display and interaction	/src/lib/Grid.svelte

	component, handles cursor movement, tile activation, and game state	
Game Handler	Manages game state, scoring logic, and game loop timing	/backend/src/game.rs
Settings	User preferences management for DAS, ARR, keybindings, and game parameters	/frontend/src/routes/settings.svelte
Authentication	User registration, login, and session management	src/routes/signup/+page.svelte
Leaderboards	Score display system with filtering, pagination, and sorting using MergeSort	/src/routes/leaderboard/+page.svelte
WebSocket Client	Client-side WebSocket connection handling for multiplayer mode	/src/lib/grid.svelte
Multiplayer	Multiplayer game orchestration, player matching, and state synchronization	src/lib/grid.svelte
DAS Implementation	Delayed Auto Shift functionality for improved input responsiveness	src/lib/grid.svelte

0.10.2. Skill table

Group	Skill	Description	Link
A	Complex client-server model	Full-featured multiplayer game system with real-time WebSocket communication	Multiplayer

0.10.3. Completeness of Solution

0.10.4. Complex Algorithm Implementation

0.10.5. Code Quality

my coding style follows rust's programming principles, i.e error handling through result and option types, and a focus on readability and maintainability, i.e i use descriptive variable names, and i try to comment my code to explain why behind the code, i also try to use meaningful variable names, and i try to keep functions small and focused, i.e single responsibility.

0.10.5.1. Grid Component

0.10.5.2. Game Handler

0.10.5.3. Settings

0.10.5.4. Authentication

0.10.5.5. Leaderboards

0.10.5.6. WebSocket Client

0.10.5.7. Multiplayer

0.10.5.8. DAS Implementation

0.10.5.9. WebSocket Server

0.10.5.10. Game State

0.10.5.11. Authentication API

0.10.5.12. Database Operations

0.10.5.13. Anti-Cheat

0.10.5.14. Session Management

0.10.5.15. Xoshiro256+

0.10.5.16. Circular Queue

0.10.5.17. Path Finding

0.10.5.18. Statistics

0.11. Testing

Test Description	Status	Proof
Test user registration with valid credentials	Pass	
Test user registration with existing username	Pass	
Test user login with valid credentials	Pass	
Test user login with invalid credentials	Pass	

Test session persistence across page reloads	Pass	
Test session expiry after timeout	Pass	
Test grid initialization with correct size (4x4)	Pass	
Test grid initialization with correct size (5x5)		
Test grid initialization with correct size (6x6)		
Test initial cursor positions (blue at 0,0 and red at size-1,size-1)		
Test initial grid has exactly 'size' active tiles		
Test blue cursor movement in all directions with keyboard		
Test red cursor movement in all directions with keyboard		
Test cursor movement boundary limits (cannot move outside grid)		
Test DAS (Delayed Auto Shift) functionality for cursor movement		
Test valid submission when both cursors are on active tiles		
Test invalid submission when cursors are on the same tile		
Test invalid submission when one cursor is not on an active tile		
Test score increment on valid submission		
Test score reset on invalid submission		
Test visual feedback (green) for correct submissions		
Test visual feedback (red) for incorrect submissions		
Test new active tiles appear after valid submission		
Test deactivation of submitted tiles after valid submission		
Test timer countdown functionality		
Test game end when timer reaches zero		
Test game statistics display after game end		
Test leaderboard display with correct pagination		

Test leaderboard filtering by grid size		
Test leaderboard filtering by time limit		
Test leaderboard filtering for personal bests		
Test multiplayer game joining functionality		
Test multiplayer game quota system		
Test multiplayer game player elimination		
Test multiplayer game final rankings		
Test WebSocket connection establishment		
Test WebSocket message handling for different action types		
Test WebSocket reconnection on connection loss		
Test server-side move verification with valid moves		
Test server-side move verification with invalid moves		
Test server-side timing verification for normal play		
Test server-side timing verification for suspicious patterns		
Test server-side path optimization detection		
Test PRNG (Xoshiro256+) deterministic output with same seed		
Test game state persistence in database		
Test user statistics update after game completion		
Test keybind customization persistence		
Test settings reset to defaults		
Test game performance with rapid inputs		
Test game performance with simultaneous inputs		

0.12. Evaluation

0.12.1. Overall Effectiveness

0.12.2. Evaluation Against Objectives

0.12.3. User Feedback

0.12.4. Response to Feedback

0.12.5. Future Improvements

1. Bibliography

[1] Hard Drop - Tetris Wiki, “Delayed Auto Shift - Understanding Modern Tetris Controls.” Accessed: Dec. 17, 2023. [Online]. Available: https://harddrop.com/wiki/DAS

[2] MDN Web Docs, “HTTP - Overview of the Hypertext Transfer Protocol.” Accessed: Dec. 20, 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

[3] IETF - Network Working Group, “User Datagram Protocol (UDP) - Internet Protocol Suite.” Accessed: Dec. 20, 2023. [Online]. Available: https://www.ietf.org/rfc/rfc768.txt

[4] Mozilla Developer Network, “An Introduction to WebSockets.” Accessed: Dec. 10, 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[5] David Blackman and Sebastiano Vigna, “xoshiro / xoroshiro generators and the PRNG shootout.” Accessed: Dec. 15, 2023. [Online]. Available: https://prng.di.unimi.it/

[6] Svelte Team, “SvelteKit Documentation.” Accessed: Dec. 19, 2023. [Online]. Available: https://kit.svelte.dev/docs/introduction

[7] Rust Language Team, “Asynchronous Programming in Rust.” Accessed: Dec. 02, 2023. [Online]. Available: https://rust-lang.github.io/async-book/

[8] Tailwind Labs, “Tailwind CSS Documentation.” Accessed: Dec. 05, 2023. [Online]. Available: https://tailwindcss.com/docs

[9] Catppuccin Team, “Catppuccin Color Scheme.” Accessed: Dec. 08, 2023. [Online]. Available: https://github.com/catppuccin/catppuccin

[10] Towards Data Science, “Understanding the Sigmoid Function in Mathematics.” Accessed: Dec. 14, 2023. [Online]. Available: https://towardsdatascience.com/understanding-the-sigmoid-function-385f5c01ae3e

[11] Joshua Glazer and Sanjay Madhav, *Multiplayer Game Programming*. Accessed: Dec. 12, 2023. [Online]. Available: https://www.informit.com/store/multiplayer-game-programming-architecting-networked-9780134034300