Uppsala University

Programkonstruktioner och Datastrukturer
VT 25

Minesweeper

Authors:
Konstantin Dimitriadis Lorenz
Erik Verastegui
Remu Mäenranta
Group 5

March, 2025

Contents

1 Introduction

Minesweeper is a classic puzzle video game renowned for its simplicity and lean learning curve. Our choice of making this game was motivated by its balanced complexity, making it a manageable and challenging project that would align well with our acquired knowledge during this course. The primary objective of this project was to use TypeScript to create a representation of the game as accurately as possible, strictly adhering to its game mechanics and rules without introducing new features. The expected outcome should be a fully playable game on the same level as the original minesweeper game.

The rest of this report is structured as follows: Section 2 provides an overview of the Minesweeper game mechanics. Section 3 dives into the implementation details, including grid structure and game logic. Section 4 gives a quick overview of the tests used and their relevance. Finally, Section 5 presents the results and conclusions drawn from the project. Optionally, we also provided an appendix that details our attempt to implement a GUI. In it we present our design considerations and challenges encountered regarding the GUI.

2 Usage

2.1 Background

[1] Minesweeper is based on a grid that contains a finite number of cells. At the start of the game, each cell is 'unrevealed' and clicking on a cell results in 'revealing' that particular cell. Once reavealed, cells can have three different states: an empty cell, a cell with a number, or a cell with a mine. The mines are randomly distributed across the board. If a 'revealed' cell contains a mine, the game ends instantly at loss. A cell with a number indicates how many mines are adjacent to that particular cell, diagonally, vertically, and horizontally. The game allows you to flag unclicked cells if you suspect a mine is beneath, which prevents you from accidentally clicking on these flagged cells. See Figure 1 as an example.

The game is won when all cells that do not contain a mine are revealed.

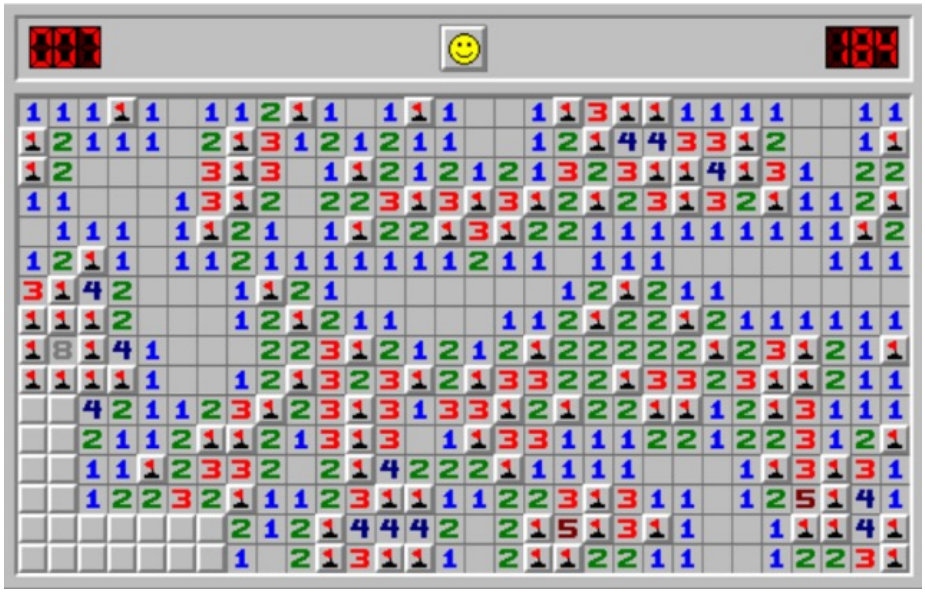

Figure 1: Example of a minesweeper game

2.2 Our game

The program is started by first compiling it with `tsc src/main.ts` and then running `node src/main.js`. When starting the game, the player is prompted to select the difficulty by specifying the grid size and the numbers of mines. Unlike the original minesweeper game where individual cells are clicked, our version requires the player to enter the coordinates of the desired cell by specifying the row and column. Once choosing a specific cell, the player can either reveal or "flag/unflag" an unrevealed cell by entering 0 to "flag/unflag" or 1 to reveal it. If the player wishes to restart an ongoing game, unfortunately, there is no flexible way to do so. The standard "Ctrl + C" to abort is not supported. The game ends strictly when the player either loses or wins. If you wish to start a new game during an ongoing session, you are required to open a new terminal window to begin a new game.

3 Program Documentation

3.1 Data Structures

Minesweeper revolves around two central data structures, a `Grid` which contains `Cells`. Implementing these data structures has been a central part in our development of minesweeper. These data structures are defined in `minesweeper/modules/classes.ts`.

3.1.1 Cell

Formally, a `Cell` is a `Class` with three public properties: `type` of type `CellType`, `state` of type `CellState` and a `neighboring mine count` of type `number`. These three properties hold all of the necessary pulic data of a `Cell`.

```
type CellType = "mine" | "empty";
type CellState = "revealed" | "unrevealed" | "flagged";

export class Cell {
    // Properties
    type: CellType;
    state: CellState;
    neighboring_mine_count: number;
}
```

The `type` describes if the cell is a mine or not, while the `state` describes if the cell is flagged/revealed/unrevealed, which will change depending on user action. The `constructor` for the `Cell` takes both `type` and `state` as parameters. The actual generation of cells is managed by the `Grid` which always initializes them with a `state` of `"unrevealed"`, and a `type` of `"empty"`.

As for the methods, `Cell` has getter/setter functions for both `type` and `state`. The main, important method is `get_neighboring_mine_count` which will be explained in subsection 3.2

A `Cell` only holds information about 1 cell in the `Grid`, this includes `type` and `state` which determine whether it is a mine/flagged/revealed/unrevealed, but also a `neighboring_mine_count` to track the number of neighboring mines. An important thing to note is that `Cell` exists independently of the `Grid`, since we have no mention of any `row` or `col` indices, this choice was deliberate, since `Cell` is meant to operate at a lower level of abstraction than `Grid`.

3.1.2 Grid

The `Grid` data structure accounts for most of the code in this project. Formally, it is a `Class` with 4 main properties `row_count`, `col_count`, `desired_mine_count` and `grid`. The `row_count`, `col_count` and `desired_mine_count` are passed to the constructor, and with these values the `grid` is generated, as a 2D `Array` of `Cell`s.

```
/**
 * Represents a minesweeper grid containing cell objects
 */
export class Grid {

    //Properties
```

```
    grid: CellGrid;
    row_count: number; // Number of rows
    col_count: number; // Number of columns

    mine_count: number; // Number of mine cells
    desired_mine_count: number // Desired number of mine cells

    safe_cells: number; // Number of unrevealed empty cells
    game_state: GameState; // State of the game
```

As described before, we initialize all of the `Cells` with a `type` value `"empty"` and `state` value `"unrevealed"`. This is done through 2 for loops over `row_count` and `col_count` that populate the 2D `Array` with cells:

```
this.grid = [];
for (let i = 0; i < row_count; i++) {
    const row: Cell[] = [];
    for (let j = 0; j < col_count; j++) {
        row.push(new Cell("empty", "unrevealed"));
    }
    this.grid.push(row);
}
```

In addition to getter/setter methods `Grid` has a number of critical methods, like `get_neighbors`, `reveal`, `reveal_neighbors`, `populate_with_mines`. These will be described in more detail in subsection 3.2.

At a high level, the main game loop operates on a global `Grid` which has its `Cells` modified, as per the users actions on them.

`Grid` represents the game board. It is represented as a 2D matrix of `Cells` with methods to generate them, and to modify them. With this the user can interact with and modify the board, by acting on `Cells`.

3.2 Methods

As mentioned before `Grid` and `Cell` have quite a few methods associated with them. Here we will take to documenting the most important ones.

3.2.1 populate_with_mines

The method responsible for placing the mines, the `populate_with_mines`, is a crucial part of the generation of the `Grid`. As parameters, it takes an `amount` (of mines) and `blacklist`. The `amount` is the number of mines one wants to populate the grid with, and the `blacklist` is used to exclude certain indices from mine placement.

The way the method works is via a helper function `add_mine`. This function takes an `amount` and a `whitelist` corresponding to all allowed `Cells`, that are not in the `blacklist`. The function chooses a random index within this `whitelist` and sets the `type` of the `Cell` at the given index to `mine`, this index is then removed from the `whitelist` and the process continues recursively until `whitelist` is either empty or `amount` is 0.

```
function add_mine(amount: number, whitelist: Array<[number, number]>) {

        const random_number = Math.floor(Math.random() * whitelist.length);

        const row = whitelist[random_number][0];
        const col = whitelist[random_number][1];

        grid.set_mine(row, col);

        whitelist.splice(random_number, 1);

        add_mine(amount - 1, whitelist);
}
```

The `whitelist` is calculated in the outer function by filtering all indices, which can be obtained through the `get_all_indices` getter method by whether they are in the `blacklist` or not.

3.2.2 set_mine

The `set_mine` method is extensively called by `populate_with_mines`. It is important because it handles both changing the type of a `cell` to a mine, and incrementing the `neighboring_mine_count` of any adjacent cells to that mine. This method works by taking a `row` and a `col` and getting all the neighbors of the `Cell` at the index (`row`, `col`), through the `get_neighbors` method. It then increments the `neighboring_mine_count` property for all of these `Cells` and sets the current `Cells` `type` to `"mine"`.

```
set_mine(row: number, col: number) {
    const neighbors: Array<[number, number]> = this.get_neighbors(row, col);

    // Increments mine_count for each neighbor of the mine
    neighbors.forEach((cell_index) => {
        const neighbor = this.cell_at(cell_index[0], cell_index[1])
        neighbor.increment_neighboring_mine_count(); });

    this.cell_at(row, col).set_type("mine");
}
```

3.2.3 reveal

The function that handles revealing cells, `reveal` , is a diverse but crucial method that is used for both changing the state of selected cells to revealed and for checking if this results in either victory or defeat. As input it takes a `row` and a `col` and sets the `state` of the `Cell` at index (`row`, `col`) to `"revealed"`. But in doing this there are some things to consider. When a `Cell` is revealed multiple things can occur: if it is of `type "empty"` then revealing it will cause the number of empty revealed cells to increase. This is because if this number reaches 0, the game is won. If the `Cell` has no neighboring mines, the neighbors should be revealed, and if it is of type `"mine"`, then revealing it will result in a loss.

```
// reveal method
cell.set_state("revealed");

if(cell.get_type() === "empty"){

    // Decrement number of remaining empty cells
    this.safe_cells--;

    if(this.safe_cells === 0) {
        this.game_state = "win";
    }

    if(cell.get_neighboring_mine_count() === 0) {
        this.reveal_neighbors(row, col);
    }
}

if(cell.get_type() === "mine") {
    this.game_state = "lose";
    this.reveal_all_mines();
}
```

This means that the `reveal` method is indirectly responsible for determining a win or a loss.

3.3 Game Loop

The main game loop is located in `minesweeper/src/main.ts`. It is responsible for interacting with the user and handling the entire game. This is done by first instantiating a `Grid` and a `while loop` that continuously prints out the grid, and then asks for user input. This input is fed into the API of the `Grid`, to modify it. The results of this are used to advance the game, until the user either wins or loses.

The game loop is part of a function `main` which handles the entire game. The game loop has three main components: user input, grid manipulation, and outputting to the terminal.

3.3.1 User Input

User input is collected on two occasions: at the start of a game, and at the start of each "round". In the `main` function the user is asked for `row_count`, `col_count` and `mine_count` values:

```
// Get settings for the game
const row_count = int_input("Input the desired number of rows: ");
const col_count = int_input("Input the desired number of columns: ");
const mine_count = int_input("Input the desired number of mines: ");
```

These values are then used to generate the `Grid` for the game:

```
const grid = new Grid(row_count, col_count, mine_count);
```

The function `int_input` takes as a parameter a `message`, `max` and an `error_message`. The function then prints out the message, asking for the specified user input using the "prompt-sync" npm module. The input is then parsed and returned. Any parsing errors, or if the inputted number exceeds the given `max` parameter, result in the `error_message` being printed out, and the user being prompted for input anew. `rows/cols/mines`.

```
function int_input(message: string, max: number = Infinity, error_message: string = "Invalic
    while (true) {
        let input: string | null = prompt(message);

        if (input === null) {
            console.error(error_message);
            continue;
        }
        const value: number = parseInt(input);

        if (isNaN(value) || value < 0 || value > max) {
            console.error(error_message);
            continue;
        }
        return Math.floor(value);
    }
}
```

Note that error handling is required to safely parse the input. In the game loop user input is collected through the `get_player_move` function, which collects the desired `Cell` to move to, and whether to flag or reveal.

```
function get_player_move(grid : Grid): Move {
    const row: number = int_input("Enter the row number: ", grid.row_count - 1);
    const col: number = int_input("Enter the col number: ", grid.col_count - 1);

    let move_type : Move["type"] = "reveal";
    let choice: number = int_input("0 to flag and 1 to reveal: ", 1);
    if (choice === 0){
        move_type = "flag"
    }
    return {col: col, row: row, type: move_type};
}
```

`int_input` is used to make sure that the input is valid, i.e. that it is a number within the boundaries of the `Grid` . Note that `get_player_move` returns a value of type `Move`. `Move` is the internal record type that represents a move. And it consists of two numbers `row` and `col` and `move_type` of "reveal" or "flag".

3.3.2 Grid Manipulation

`Grid` manipulation is handled entirely by the `player_move` function. Which is defined in `minesweeper/modules/game_module.ts`. It allows the frontend, i.e. the `main.ts` file to interact with the backend, i.e. `classes.ts`.

The `move` function takes a value of the previously discussed type `Move` as input, and a `grid`. `player_move` then manipulates the `Grid` based on the input.

Case 1: The `move` is the initial move

As alluded to in subsubsection 3.2.1 the first move of a game should be handled differently. We can check if the `move` is the initial move by checking if none of the cells are revealed, which is done via the `safe_cells` property of the `Grid`.

```
if(grid.safe_cells === grid.get_empty_count() && move.type === "reveal") {
    grid.game_start(move.row, move.col);
}
```

Case 2: General case of `"reveal"`

In the general case where the `type` field of the `move` is `"reveal"` we simply call the `reveal` function with `move.row` and `move.col`. Recall from subsubsection 3.2.3 that `reveal` both handles the revealing of a `Cell` and whether the game has been won or lost, this makes it so that we don't need to worry about determining such outcomes at this level.

Case 3: General case of `"flag"`

If `move.type` is `"flag"`, we call the `flag` method with `move.row` and `move.col`. `flag` simply toggles the `state` of the `Cell` between `"flagged"` and `"unrevealed"`.

`player_move` then returns `grid.get_game_state()` to be assigned as the `result` of a given move, which by default will be `"undecided"`, but could have been altered by `reveal` as previously discussed. This value is then checked in the `main.ts` gameloop to see if the game has ended.

```
if(result === "win") {
    console.log("You win!")
    break;
}

if(result === "lose") {
        console.log("You lose!")
        break;
}
```

3.3.3 Outputting to the Terminal

To output the grid to terminal the function `display_grid` is used. `display_grid` is called at the start of each round and displays a formatted version of the `Grid`. `display_grid` contains 2 for loops, that display indices and cell symbols. The first loop takes care of displaying column indices, it does this by printing a string containing the numbers 0 to `grid.col_count - 1`.

```
let col_number_string = "   ";
for(let col_number = 0; col_number < cols; col_number++) {
        col_number_string += ` ${col_number % 10}`;
}
console.log(col_number_string);
console.log();
```

The second for loop takes care of displaying row indices to the terminal, and displaying cell symbols. This is done in a similar manner as with the column indices. A string is created, the calculated row index is appended to it, and then the appropriate cell symbol is appended. The cell symbols were chosen arbitrarily and are represented by the following constants.

```
const MINE_SYMBOL = "*";
const FLAG_SYMBOL = "!";
const UNREVEALED_SYMBOL = "~";
const EMPTY_SYMBOL = " ";
```

The correct cell symbol to display is calculated using the `cell_symbol` function. `cell_symbol` contains a switch statement, that matches `cell.state` with its previously described values. In the case of a `state` of `"flagged"` the correct symbol will just be `FLAG_SYMBOL`, in the case of `"unrevealed"` it will be

UNREVEALED_SYMBOL. In the case of "revealed" there are two possibilities, so another switch statement is added in this case, that matches on cell.type. In the case of "mine" the MINE_SYMBOL is returned, but in the case of "empty" the cell.neighboring_mine_count is returned if it happens to be above 0. If it is 0 the EMPTY_SYMBOL is returned instead.

So the return value of cell_symbol(cell) for each cell of the current row is added to the representative string of that row, and is outputted to the terminal. This is then done for each row, and that is how the Grid is printed to the terminal.

3.4 Error Handling

This far we have avoided discussing or showcasing error handling. But it is crucial and has been implemented at every level of this project. We implement error at 2 levels: at the backend, through classes, and at the frontend.

3.4.1 Backend Error Handling

The module errors.ts is responsible for declaring errors that can occur in the classes.ts file. errors.ts contains 3 classes: InvalidMinesError, NegativeGridError and InvalidIndexError, all of which extend Error, so that you can use the standard Error API on them. These 3 classes cover all errors that can occur in classes.ts.

InvalidMinesError:

InvalidMinesError is triggered when one tries to create a Grid with a desired_mines_count that is either negative or greater than rows · cols, which exceeds the total number of cells. It is used at the start of the constructor function for Grid to check that the desired_mine_count is valid:

```
if (desired_mine_count < 0 || desired_mine_count > row_count * col_count - 1) {
    throw new InvalidMinesError(desired_mine_count, row_count, col_count);
}
```

NegativeGridError:

NegativeGridError is triggered when one tries to create a Grid with a row_count or col_count that is non-positive. It is used at the start of the constructor function for Grid to verify the input row_count and col_count values:

```
if(row_count < 0 || col_count < 0) {
        throw new NegativeGridError(row_count, col_count);
}
```

InvalidIndexError:

`InvalidIndexError` is triggered when one tries to access an invalid index in the `Grid` matrix. This error works well with the `Grid` method `is_valid_index` which checks if a given `row` and `col` constitute a valid index. Here is an example of how these are used in the `cell_at` method:

```
cell_at(row: number, col: number): Cell {
    if(this.is_valid_index(row, col)) {
        return this.grid[row][col];
    }
    else {
        throw new InvalidIndexError(row, col, this.row_count, this.col_count);
    }
}
```

3.4.2 Frontend Error Handling

The frontend errors are handled entirely by the `int_input` function which is documented under subsubsection 3.3.1. This is because `int_input` handles all the input from the user, which is where errors can occur at the frontend.

These two error handling methods overlap a bit. This is because we cannot utilize the error classes method for frontend errors, this is because errors at the fronted should be handled differently. When an *invalid input* error is encountered at the frontend, the response should not be to terminate the program, but to re-prompt the user for a new input. This is what the `int_input` function does. However, if the backend encounters a invalid value then there is no other choice but to terminate the program. Thus overlapping error handling does introduce more verbose code, and makes things slightly more complicated, than technically need be. However it is a choice that we have taken as part of a *defensive programming* initiative to make the code as safe as possible.

4 Tests and coverage

To ensure the correctness of our Minesweeper game, we've designed a comprehensive set of tests. These tests were made to verify the functionality of the game logic and to help ensure this functionality even with various edge cases. We verify that our grid correctly detects invalid indices, ensuring that "illegal" moves are handled appropriately. The progression and change of the the game state is tested by confirming that revaling all non-mine cells results in a win and revealing a cell with a mine results in a loss. Furthermore, we test manipulation of the grid, such as flagging, unflagging and preventing flagging of revealed cells. The tests also cover the acquisition of neighboring cells, by ensuring that `get_neighbor` returns correct adjacent cells and that the neighboring mine counts update correctly when a mine is placed. The distribution of mines is also

tested, ensuring that `populate_with_mines` correctly places the expected number of mines, and that `set_mine` correctly updates the `neighboring_mine_count` of cells adjacent to any placed mines. Finally, `game_start` is tested to confirm that mines are placed adequately while maintaining an initial safe zone for the players first move. In summary, the tests should help confirm that the Minesweeper game behaves as expected during various circumstances and ensures a reliable program.

5 Discussion

5.1 Design Strengths

5.1.1 Object Oriented Design

A possible way of representing a grid could have been a simple array matrix containing integers for each cell of the grid, together with functions that manipulate and read this grid. A cell's number could have represented the neighboring mine count for all empty cells, a number from 0 to 8, and for a mine a number outside of this range, say for example -1. We chose to instead represent the main parts of the game using objects, both the grid and its contained cells are objects with properties that track the ways in which the board is mutated over the course of the game. This object-oriented design allowed for a much higher level of abstraction, and a way to group relevant properties and actions together.

5.1.2 Modular Design

The different parts of the game, the game logic contained in `game_module.ts`, the grid and cell objects in `classes.ts` and the visual user interface in `main.ts` are all kept in different files. This was done mainly to ensure that the game should function with both a graphical and a terminal-based user interface, without either of these being dependent on eachother, but it also had the side-effect of greatly improving the organization of the project.

5.2 GUI Problems

We had initially planned to also develop a working GUI alongside the terminal version for the game, as we anticipated that playing minesweeper by inputting coordinates into a text-field would not make for the most exhilarating gameplay. Our plans would fall through however, and we noticed this as the deadline for the project drew closer, the report had yet to be written, and only one of us had any tangible previous experience with using a GUI-framework. There are a number of things that could have been done differently to avoid this outcome. First of all, prior knowledge of svelte would have prevented incompatibility in the first place, as we could have chosen a different framework instead. With that being said, we did have reasons to choose svelte, it is comparatibly lightweight, and minimal, which is ideal since we did not plan to do a lot with it. Secondly

we could have allocated more time and resources to the development of the GUI, then we might have been able to catch the shortcomings we faced earlier, and potentially have been able to switch frameworks, and continue the development.

Although the GUI was not successfully developed to full functionality, its failing didn't impede at all on the rest of the project, due to the modular design and planning of the project.

5.3 Conclusion

The primary goal of the project, that being a working implementation of the logical puzzle game minesweeper, was achieved. Work started by first defining the representation of a grid and its contained cells as objects with relevant properties and methods to represent various ways of mutating the grid. Afterwards the game module was built to handle any given action by applying appropriate methods on a grid. Finally, the visual terminal-based interface was built to make it possible for a player to be able to feed various actions into the game module, to manipulate the grid, and to receive the state of the game after each action.

A GUI documentation

The intention was to include a GUI written in *Svelte* that would wrap the backend and provide a user interface in the browser. However, due to reasons that will be discussed, this feature is not finished, and is thus not part of the final project. Nonetheless, the documentation for what has been done is provided.

A.1 Plan

Since we have a developed backend there is no need to rewrite the logic for the game again, and one can just rewrite the `main.ts` function in terms of `Svelte` components, that utilize the `Grid` and `Cell` APIs. This makes for the software architecture seen in Figure 2.

The **global grid** represents a global mutable variable that represents the grid. It is an instance of `Grid` and is read by the **svelte components** to display to the screen. The **user input values** represents the values `row_count`, `col_count` and `mine_count` which can be changed dynamically at any point, and which will modify the **global grid**.

A.2 Displaying the Grid

Two svelte components `Grid.svelte` and `Cell.svelte` are responsible for displaying the `Grid`. Cell uses the `cell_symbol` function to display a formatted

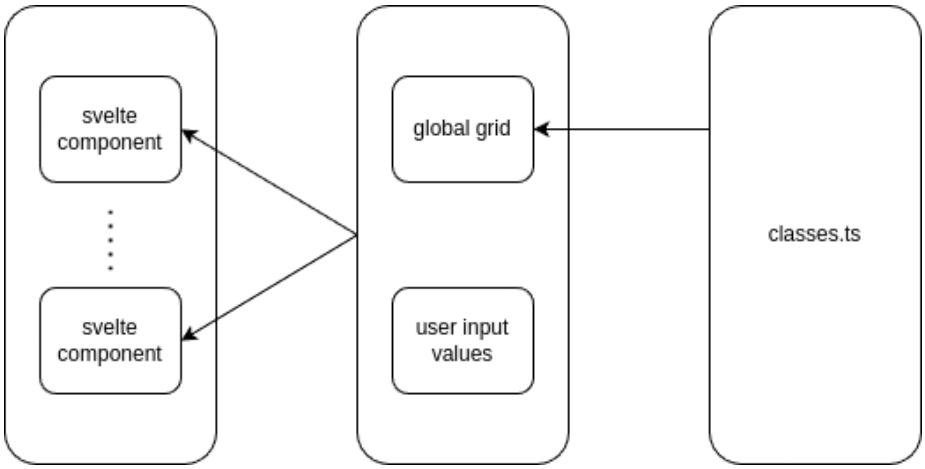

Figure 2: GUI software template

version of the same symbols, as in the terminal implementation. It also handles clicks, namely left click to move to a cell, and right click to flag a cell. Then based on these actions it modifies the **global grid**. The `grid.svelte` component displays the grid, it does so by using 2 for loops, over the **user input values**' `row_count` and `col_count` properties and displays a `Cell` for each one. Note that this process is rerun each time any of the **user input values** change. These two components are then displayed in the main component `+page.svelte`.

In Figure 3 you can see the **global grid** being displayed within the blue region. You can also see a *settings button* in the red region. When clicked on, it will display a modal, where you can change the **user input values**.

In Figure 4, the red, green and blue regions are of importance. The sliders within allow you to directly modify the **user input values**' `row_count`, `col_count` and `mine_count` fields. This component is called `modal.svelte` and becomes visible when the aforementioned button is pressed. In it the `value` properties of the `input` HTML tags are bound to the **user input values**' respective fields, using the `bind` functionality provided by svelte. When these values are nudged the grid automatically updates, without needing to refresh the page.

In Figure 5 you can see how interacting with the sliders affect the **global grid** by changing the amount of rows, columns or mines, which is then read by the svelte components and displayed.

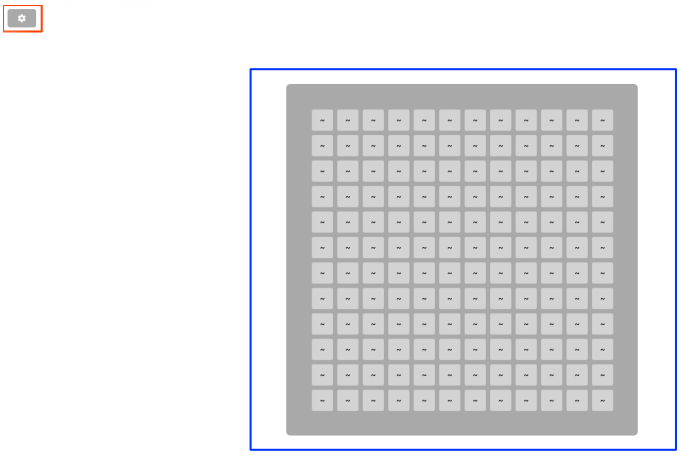

Figure 3: GUI grid showcase

A.3 What went Wrong

There are limitations to the methods showcased so far to make svelte interact with the backend. Particularly when it comes to the so called `$state` rune.[2] `$state` is a *svelte rune* that allows you to create a *reactive* variable, meaning you can update it, and the UI reacts on the change, and adjusts. `$state` is used to make the **global grid** and **user input values** modifiable across all components, and have the UI react on changes made to them. The **global grid** and **user input values** are records with fields, that are wrapped in a `$state` rune. This works great for simple fields, like `row_count` that only holds 1 value, the record can just be passed around between components, and be modified globally. However, this method does not work for `Grid`s. This is because `Grid` is a class with deeply nested properties, such as `grid.grid` which is an array, containing arrays of `Cell`s. Simply wrapping a instance of `Grid` in a `$state` rune, won't make the UI react on changes made to deeply nested components, such as the `Cell`s. This means that the UI does not react when you for example left click a `Cell` to reveal it, even though you handle it correctly:

```
function move_to_cell(event: MouseEvent) {
    event.preventDefault();
    const click: number = event.button;
    switch (click) {
        case 2:
            get_grid().cell_at(row, col).set_state("flagged");

        case 0:
            get_grid().cell_at(row, col).set_state("unrevealed");
```

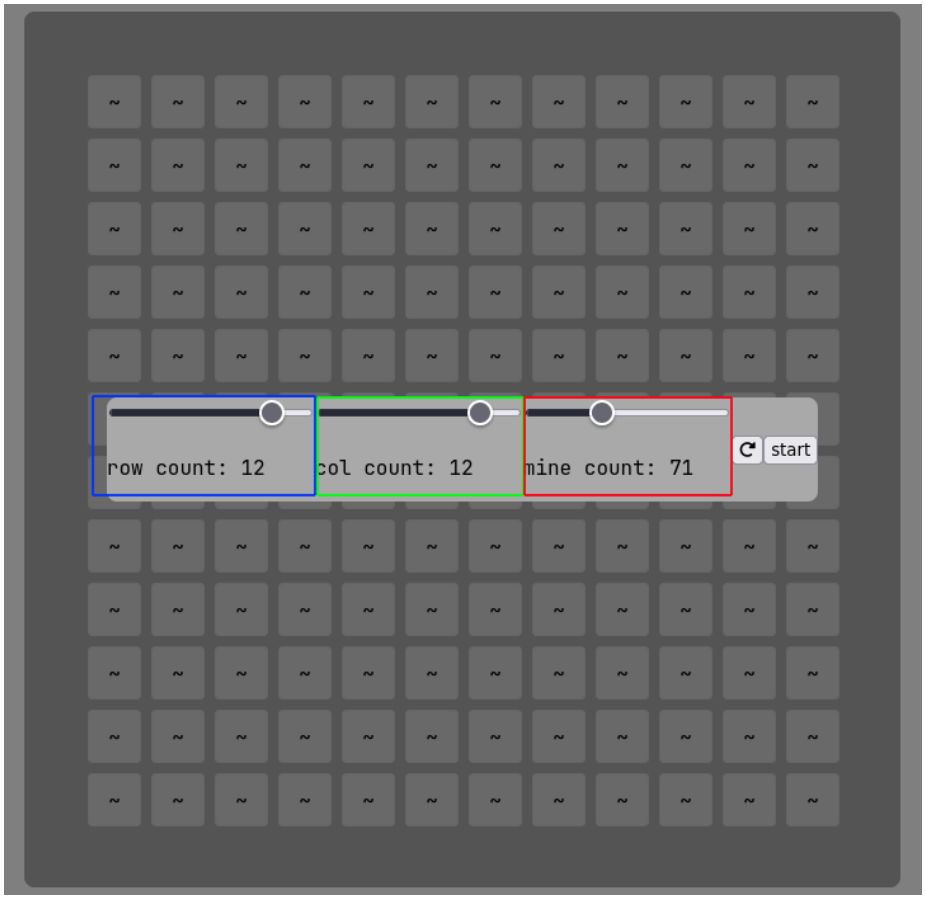

Figure 4: GUI settings showcase

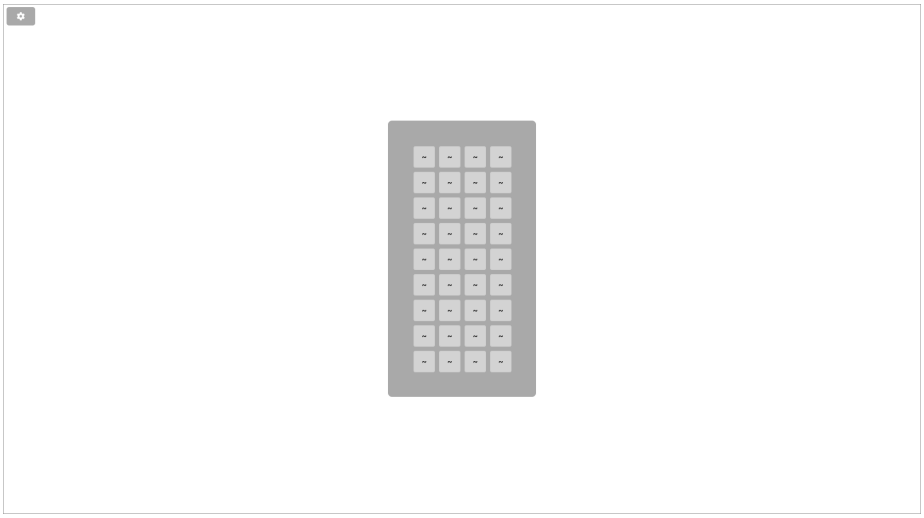

Figure 5: GUI rescaling showcase

```
    }
}
```

The `move_to_cell` function is supposed to handle a `Cell` being clicked, and set the `state` of the `Cell` to `"flagged"`/`"unrevealed"` depending on if it was a right click or a left click. However, these changes are not caught by the `$state` rune, and thus the UI remains unchanged.

A.4 A Possible Solution

A possible solution to this issue would be to wrap each property of the `Grid` class in a `$state` rune, this would make each property reactive, thus making the entire **global grid** reactive, no matter how deeply nested the modified value might be.

We have decided not to proceed with this fix though. That is because it would require changing the `classes.ts` file, and this is not how we wanted the GUI to be implemented. The GUI should not reinvent the backend, it should simply interact with its API, just as `main.ts` does. For that reason, we have left the GUI as it is, unfinished.

References

[1] Minesweeper Online, "Minesweeper.online," 2025. Accessed: 2025-03-08.

[2] Svelte, "$state," 2024. Accessed: 2025-03-09.