System Architecture

The system is divided into the game, which holds all data about the current state and logic of the game, including the state of the board, and the frontend, which handles visualizing the game state as a game of Minesweeper and handling input from the user. User input will modify the game state which in turn updates the frontend to match.

Game

The game state holds all data containing the state of the game:

`state: Enum{START, PLAYING, END::LOSS, END::WIN}`: status of the game

`mines: int[10.==20]`: number of mines in the game

`flags: int[0..=mines]`: number of flags set

`remaining int[0..=90]`: number of covered non-mines in the game

`board: Cell[10][10]` grid of cells and their state

Each `Cell` holds its own information:

`is_mine: bool`: whether the cell is a mine

`is_flagged: bool`: whether the cell is flagged

`is_covered: bool`: whether the cell is covered

`mine_count:` the number of directly bordering cells with a mine

Logic / User Flow

The game state begins at `START`. All state values are at their defaults. The `BOARD` is filled with uncovered, unflagged, non-mine, and zero mine count cells. The user is allowed to select a valid value for `mines` (implicating the value of `remaining`) and begin the game, transitioning to `PLAYING`. At the starting of the `PLAYING` state, `mines` cells are selected, set their `is_mine`, and increment their neighbors' `mine_count`.

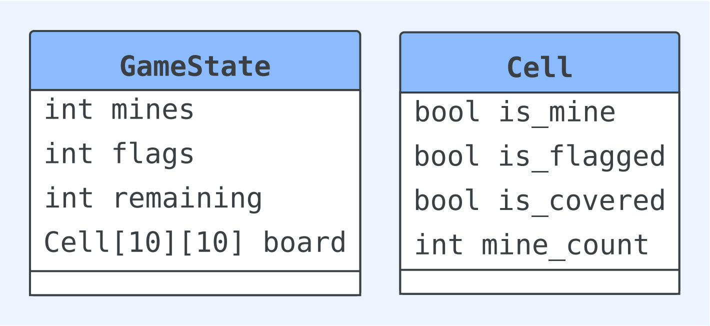
If the first cell that the user uncovers is a mine, the cell is `is_mine` is unset, neighbors' `mine_count` decrements, and a new, valid cell is selected. Otherwise, the standard logic for uncovering cells applies: if the cell is a mine, the game is sent to the `END::LOSS` state; if the cell has a zero mine count, a BFS is performed to uncover all connected cells which have zero minecount; and if the cell has a positive mine count, only that cell is uncovered. Each time a cell is successfully uncovered, its `is_uncovered` is set and `remaining` is decremented. If the user sets or removes a flag, the cell's `is_flag` is toggled and `flags` is updated. Actions that create an invalid `flags` count are ignored.

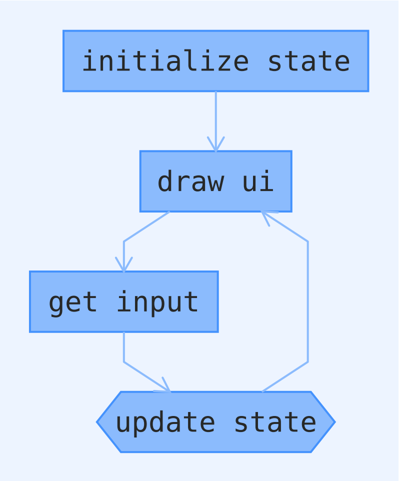
If the user successfully uncovers all cells, the game is sent to the `END::WIN` state. In either `END::LOSS` or `END::WIN` state, the user can retry to return the game state to `START` to begin a new game.

Frontend

The frontend uses the game state to determine how elements should be drawn, as well as which inputs can be performed and handled. For example, board input should only be allowed during in `PLAYING`, and mine count selection should only be allowed in `START`. User input is captured, classified, and sent to the game in order to process the logic as specified above. Mouse position are captured to determine what item is selected from either text input fields, buttons, or board cells. Mouse actions are captured and classified to determine which action should be performed to advance the game logic loop. Keyboard input is captured for the mine count selection field.

Diagrams (WIP)


GameState
int mines
int flags
int remaining
Cell[10][10] board
Cell
bool is_mine
bool is_flagged
bool is_covered
int mine_count


initialize state
draw ui
get input
update state