EECS 581: Minesweeper Project

Project System Architecture

& Person-Hour Estimates

Version 1.2

Table of Contents

System Architecture Overview

Components

There are 4 major components to the architecture of the Minesweeper game: Board Manager, User Interface, Input Handler, and Game Logic.

The Board Manager is responsible for creating/managing a 10x10 game board grid made through a 2D array. This component helps manage the game as a whole, as it aids with tracking cell status by indicating whether the cells are covered, uncovered, contain a bomb/mine, whether it is flagged or unflagged, if there is a number, or if it is an empty cell. The board manager also aids with housing a User Interface component of the game.

The User Interface component is responsible for interacting with the user's actions and performing the correct tasks associated with each action. It is also responsible for showcasing the correct game status to the user, such showcasing win and loss statuses. It aids with rendering the grid and making sure status indicators are visible to the user throughout the game (such as flag count, the number revealed on the grid, and clicking functionality). As a result, the User Interface component houses the Input Handles and Game Logic components.

The Input Handler component is responsible for processing the user input by identifying the specific actions that the user performs, such as left mouse-click indicates the user wants to select a cell to be uncovered, and right mouse-click indicates the user wants to place a flag on a cell. This component is responsible for communicating with the Game Logic component to carry out the user's desired actions.

Once the actions from the Input Handler component are communicated, the Game Logic component carries out various actions and handles gameplay rules depending on the different scenarios:

- The Game Logic will randomly generate bombs only after the user's first click, and also make sure no bomb is generated in the first clicked cell. This is to ensure that the user's first click is a guaranteed safe click.

- Based on where the bombs are generated, the Game Logic will compute the correct number that will be placed in each bomb-free cell to indicate to the user where a potential bomb resides on the game board
- Game Logic will also attempt to recursively reveal adjacent cells (containing numbers) that are bomb-free during the rest of the gameplay.
- Lastly, if all the bomb-free cells are uncovered, the Game Logic detects a win, but if the user clicks on a bomb, then all of the bombs are revealed to the user and a loss status is detected.

Data Flow

For the game, data is passed between the user, the input handler, the game logic, and the game board. There are two general flows: initial game flow and subsequent game flow.

In the initial game flow, the user interacts with the User Interface and inputs the desired bomb count for the game, and starts the game. The Input Handler takes in the number of bombs and creates the game board with the specified number of bombs by initializing the **Grid** object. The Game Board then creates the empty 10x10 board and creates each **Cell** object. The Game Logic then initializes each cell with the default attributes. Finally, the User Interface displays the newly generated board.

In the subsequent game flow, the user continues to interact with the User Interface by clicking or right-clicking/left-clicking on cells. The input handler determines which cell is clicked on by identifying the mouse position with respect to the grid. Before passing the position of the clicked cell to the Game Board, it checks if a bomb is clicked on or if all cells except the bomb are clicked, indicating the player won. The Game Board handles the input differently depending on the type of click. If it is a normal click, the clicked cell is revealed, as well as adjacent cells using a recursive algorithm. The Game Logic and User Interface updates each cell. Otherwise, the Game Board determines which cell to flag or unflag. If it is a left-click, the Game Logic unflags the cell. Otherwise, if it is a right-click, the Game Logic flags the cell. The User Interface displays the board continuously after each user input and displays a winning or losing screen accordingly.

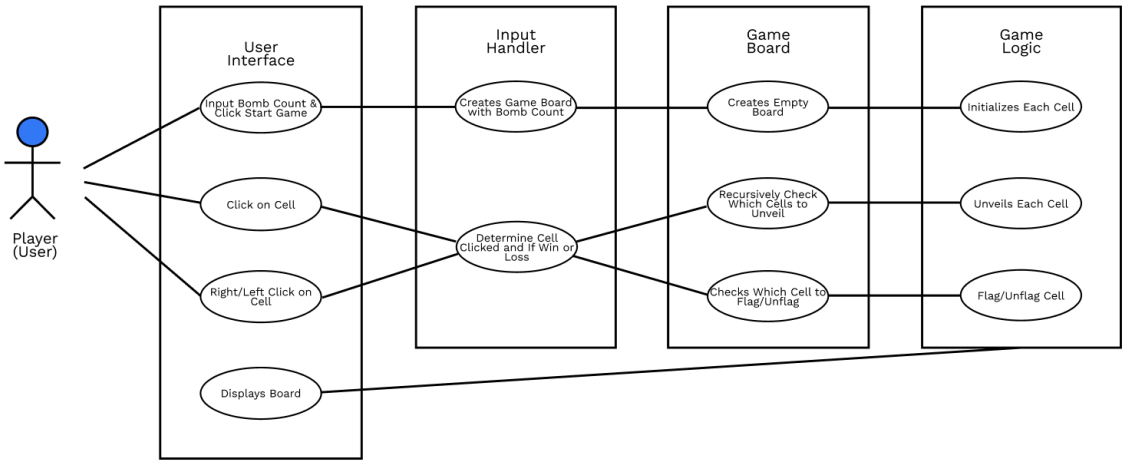

Diagram 1: Components and Data Flow

Key Data Structures

To keep track of the cell states in the grid, we created a **Grid** class, which has an object attribute called **self.grid_list**, which is a nested list, or a 2D array. Other object attributes of the Grid class include how many bombs there are for this game round, the flags placed on the grid, and the actual Pygame grid surface. Each entry in the 2D array constitutes one of the rows in the grid, and each element in one of these rows represents one of the cells. Therefore, this array is a 10 by 10 array, holding 100 elements within it, where each element is indexed by row (i) and column (j). Each element is of the **Cell** class type. The Cell class has object attributes like the image of the cell, the type of the cell, if the cell has been revealed or flagged, and the position of the cell.

When the grid is first initialized, the bomb cells will be placed randomly into different indices of the 2D array. Then, for each adjacent cell to a bomb, a number cell is computed for how many bombs are adjacent to it. Finally, the remaining cells are classified as empty when they have no adjacent bombs.

Therefore, we created a legend for how we label the type of each cell, which is stored in the Cell class's attribute, **self.type**:

- B = Bomb
- E = Empty
- N = Number

To retain the state of each Cell, we must also have attributes for if the cell is covered, flagged, or uncovered. Therefore, we have a **self.revealed** attribute and a **self.flagged** attribute in the Cell class. This will get updated each time a user clicks on one of the cells. If a user clicks on a non-bomb cell, then it will go to the dig algorithm function, where if the clicked on cell is a N type, then self.revealed = True. If the user clicks on a bomb, then it will reveal all the bombs since the game is over, by making self.revealed=True. For a flagged cell, if the user wants to flag a cell, then self.flagged = True. Cells will not be revealed if self.flagged = True.

For the game state object, we track the bombs by marking each bomb cell with the type "B". Since the user can decide what amount of bombs they would like on the grid, we have a **self.bomb_amount**, with a default value of 10 in the Grid class. This value will get updated based on what the user selects in the **Game** class, which handles the user input. For the flags remaining, we have a flags_remaining function that will update how many flags have been used by the player. This will be updated on the game board in the Game class, so that the user has a live update at all times. Finally, the win/loss status will appear as a UI component in the Game class. If a user clicks on a bomb, it will reveal the bombs and show a "Game Over" page. If the user clicks on all non-bomb cells, then they will go to the "You Win" page. The status of winning or losing is directly tied with how many cells are uncovered and the type of cells uncovered, which are stored in the attributes of the classes described above.

User Assumptions

In the Minesweeper game, the number of bombs generated matches the user's chosen value at the beginning of the game, ranging between 10 and 20 (inclusive).

Figure 1: Demonstrating the user input of bombs

The first cell selected by the user is guaranteed to be safe and will not contain a bomb. To ensure that there is variety, bomb locations are randomized for each new game/round and will not be the same/identical across rounds.

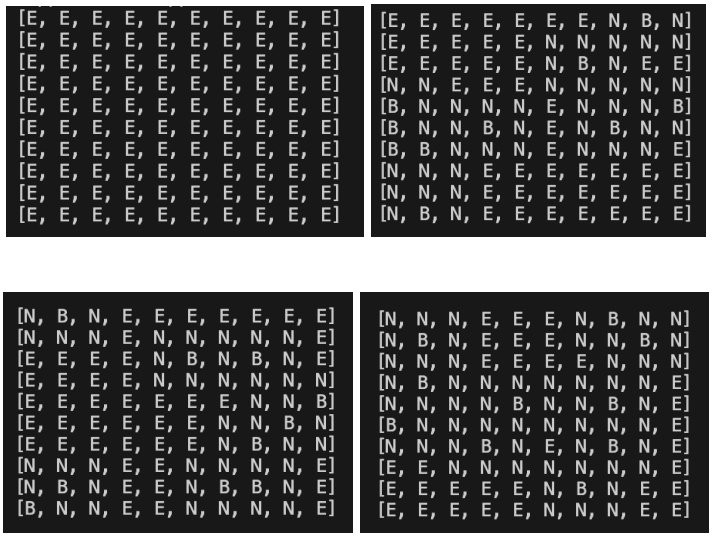

Figure 2: Demonstrating the cell types and randomization

When a user selects a cell with zero adjacent bombs, the game will automatically and recursively uncover surrounding cells until the numbered cells are reached. Numbers displayed on uncovered cells accurately indicate the count of bombs in the surrounding eight cells, guiding the player's decisions. The grid size is fixed at 10 rows by 10 columns for all the games. Users can freely place and remove the flags on covered cells to mark suspected bomb locations/cells. The remaining flag (shown on the top left "Flags Left: ") counter correctly updates to show the number of bombs left to be unflagged(shown on the top right "Mines left: "). Uncovering a bomb immediately ends the

game with a loss. A win is achieved only when all the non-bomb cells are uncovered without detonating any bombs.

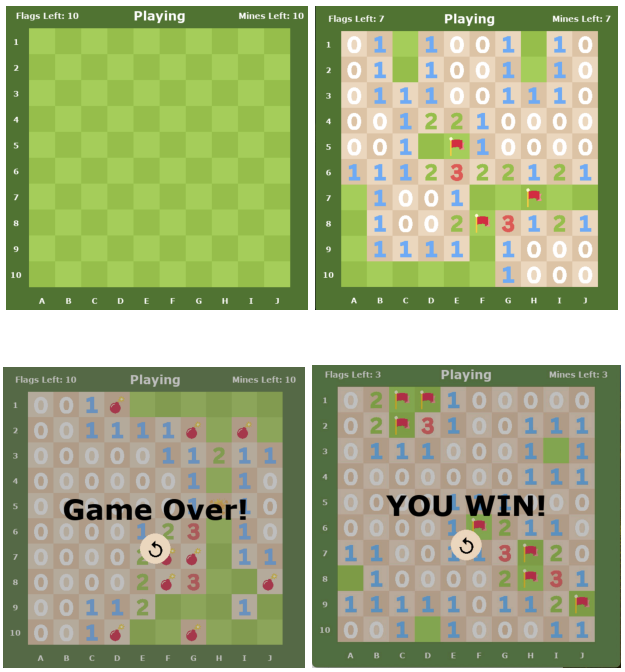

Figure 3: Demonstrating the different game board states

Person-Hour Estimates

Methodology

First, we created a spike to research the potential components in the Minesweeper game. This helped give us a holistic overview of how Minesweeper works, the different features that we would need for the game, and the different classes that could be integrated into our code. We then ranked each feature based on the complexity of each component and the dependencies on other parts of the code.

From there, we were able to create a backlog of ticket issues that contained all the high-level features that were needed to achieve the completion of the game. Then, using the methodology

learned in class, called “Planning Poker”, we did a preliminary round of estimation of person-hours. We accounted for buffer time, meetings for planning sprints, and testing the code.

As we discussed each feature more, we created sub-tickets under the features that held more granular tasks for each person to complete. We estimated each one by having each person vote on the number of hours they thought it would take to complete the ticket. If there were disputes on the ticket estimation from different people, each person would explain why they thought it would take that many person-hours, and we would do another round of estimation. Once we had reached a consensus about the ticket, that would be our final estimate.

Feature: Creating Empty Grid Board + Random Bomb Generation	
Tickets	Estimated Person-Hours
Create Game Class to Set Up Grid UI #18	2 hours
Create the Settings File To Load Assets #19	1 hour
Create the Cell Class and its Functions #20	1 hour
Create a Grid Class That Contains Cells #21	1 hour
Create the Unique Bomb Generation on an Empty Grid #22	4 hours

Feature: Implement Number Generation for Adjacent Bombs + User Input	
Tickets	Estimated Person-Hours
User Input Number of Bombs #24	2 hours
Generate Numbers for Cells Next to Bombs #25	1 hour

Create Game Over Feature #27	1 hour
Create First-Click Feature #28	3 hours

Feature: Implement Map Recursion	
Tickets	Estimated Person-Hours
Implement Bombs Revealed and Flags Checked #38	2 hours
Implement Map Recursion	2 hours

Feature: Implement Flags	
Tickets	Estimated Person-Hours
Create Flags, Flag Count #29	2 hours

Feature: Game UI + End States	
Tickets	Estimated Person-Hours
Create the Assets for Game #46	2 hours
Make new UI Changes to Front Page #34	2 hours
Create Game Won Logic + Page #48	3 hours
Create Game Over Logic + Page #27	1 hour

Actual Person-Hours

Feature: Creating Empty Grid Board + Random Bomb Generation

Tickets	Actual Person-Hours	Date	Assignee
Create Game Class to Set Up Grid UI	5 hours	08/29/2025	Kusuma
Create the Settings File To Load Assets	1 hour	08/29/2025	Nikka
Create the Cell Class and its Functions	1 hour	08/29/2025	Anna
Create a Grid Class That Contains Cells	1 hour	08/29/2025	Nimra
Create the Unique Bomb Generation on an Empty Grid	4 hours	08/29/2025	Sophia

Feature: Implement Number Generation for Adjacent Bombs + User Input			
Tickets	Actual Person-Hours	Date	Assignee
User Input Number of Bombs	3 hours	09/01/2025	Sophia
Generate Numbers for Cells Next to Bombs	1 hour	09/01/2025	Anna
Create First-Click Feature	5 hours	09/01/2025 - 09/02/2025	Kusuma

Feature: Implement Map Recursion			
Tickets	Actual Person-Hours	Date	Assignee
Implement Bombs Revealed and Flags Checked	3 hours	09/01/2025	Nimra
Implement Map Recursion	2 hours	09/01/2025	Nikka

Feature: Implement Flags			
Tickets	Actual Person-Hours	Date	Assignee
Create Flags, Flag Count	2 hours	09/01/2025	Nimra

Feature: Game UI + End States			
Tickets	Actual Person-Hours	Date	Assignee
Create the Assets for Game	5 hours	08/29/2025	Anna
Make new UI Changes to Front Page	4 hours	08/29/2025	Sophia
Create Game Won Logic + Page	4 hours	09/02/2025	Sophia
Create Game Over Logic + Page	1 hour	09/01/2025	Nikka

Additional Actual Person-Hours from Group Meetings:

08/29/2025: 3 hours

09/01/2025: 2 hours

09/06/2025: 2.5 hours

09/15/2025: 2 hours

All members of the Group 30 team will regularly manage, update, and commit to the GitHub Repository. The repository will be publicly available for viewing and accessing here: Minesweeper Group 30 GitHub.

**** Note: All our Project Meeting Logs will be housed in the GitHub Repository on the Wiki Page. Please reference it as needed. We also have our Sprint Meetings and tickets created on the GitHub page.**