Final Project Report

Title:

"Minesweeper game using OOP principles in C++ with the SFML framework"

Submitted to:

Ma'am Sumayyea Salahuddin

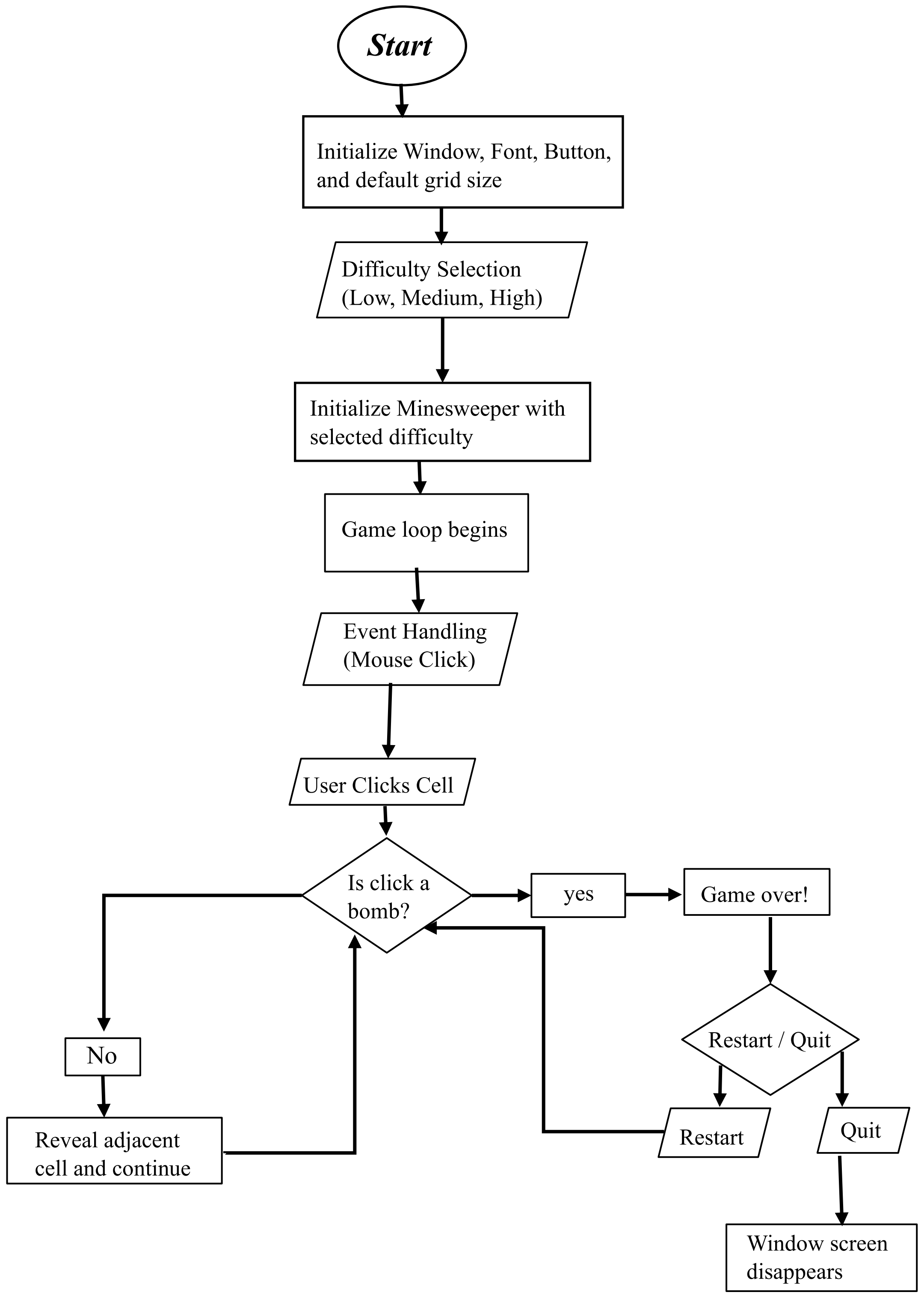
Section: A

3rd Semester

Group members:

Warisha shiraz (23PWCSE2267)
Laiba khan (23PWCSE2306)
Warisha Ahmad (23PWCSE2317)

“Flowchart”



Start
Initialize Window, Font, Button, and default grid size
Difficulty Selection (Low, Medium, High)
Initialize Minesweeper with selected difficulty
Game loop begins
Event Handling (Mouse Click)
User Clicks Cell
Is click a bomb?
yes
Game over!
No
Reveal adjacent cell and continue
Restart / Quit
Restart
Quit
Window screen disappears

Introduction:

The Minesweeper game presented in this code uses several important Object-Oriented Programming (OOP) concepts, including **Classes**, **Encapsulation**, **Abstraction**, **Inheritance**, and **Polymorphism** (though not extensively). This lab report focuses on analyzing and explaining how these concepts are implemented in the given code, providing a deeper understanding of how OOP is utilized to design and structure the Minesweeper game.

OOP Concepts Used in the Minesweeper Game:

1. Class:

In OOP, **a class** is a blueprint for creating objects, providing initial values for state (member variables) and implementations of behavior (methods). In the given code, the primary class used to represent the game is the Minesweeper class.

Minesweeper Class:

The Minesweeper class encapsulates all the data and logic required to play the Minesweeper game. This class contains:

- **Member Variables (Attributes):**
  - **gridSize**: Specifies the size of the grid (i.e., the number of rows and columns in the Minesweeper grid).
  - **cellSize**: The size of each cell in the grid.
  - **bombCount**: The number of bombs on the grid.
  - **grid**: A 2D vector (std::vector<std::vector<int>>) that stores the grid state, where -1 represents a bomb, and other values represent the count of neighboring bombs.
  - **revealed**: A 2D vector (std::vector<std::vector<bool>>) that keeps track of which cells have been revealed.
  - **gameOver**: A boolean that tracks whether the game has ended.
  - **font**, **bg_texture**, **bg**: Used for rendering the visual aspects of the game using SFML.
- **Member Methods (Behaviors):**
  - generateBombs(): Generates bombs on the grid and updates the surrounding cells to indicate the number of bombs in adjacent cells.
  - handleClick(int x, int y): Handles the logic for a user click, revealing a cell or ending the game if the user clicks on a bomb.
  - revealCell(int x, int y): Reveals a cell and recursively reveals neighboring cells if the cell is empty.
  - draw(sf::RenderWindow& window): Draws the current state of the grid to the window, including revealed cells, bombs, and numbers.

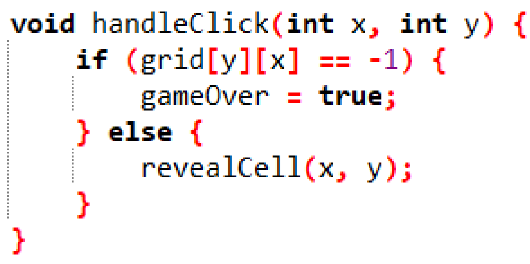
- isGameOver(): Checks if the game has ended.
- reset(int size, int bombs): Resets the game with a new grid size and bomb count.

2. Encapsulation:

Encapsulation is the concept of bundling the data (attributes) and methods (behaviors) that operate on the data into a single unit or class. It also restricts direct access to some of an object's components, which can help prevent accidental or unauthorized changes to the data.

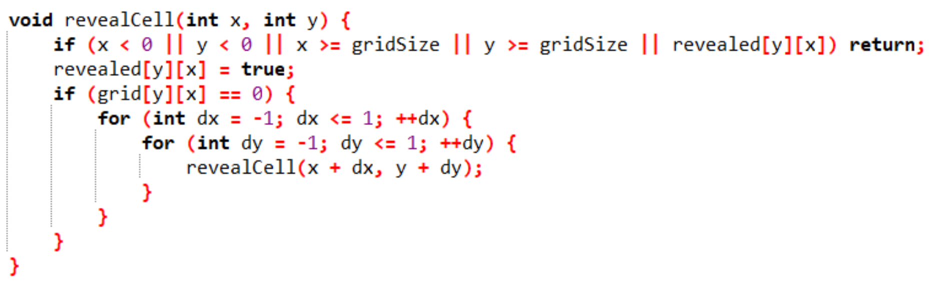
In the provided code:

- The **private member variables** (grid, revealed, gameOver, etc.) are encapsulated within the Minesweeper class, meaning external code cannot directly access or modify these values. This is important to maintain the integrity of the game's state.
- Access to these variables is only possible through **public methods** like handleClick() and revealCell(). This ensures that the data is only manipulated in valid ways, such as revealing cells or ending the game if a bomb is clicked.



```
void handleClick(int x, int y) {
    if (grid[y][x] == -1) {
        gameOver = true;
    } else {
        revealCell(x, y);
    }
}
```

- For example, to reveal a cell, the program uses the revealCell() method, which checks if the cell is already revealed or if it's out of bounds. This protects the game's internal state from being corrupted by external code.



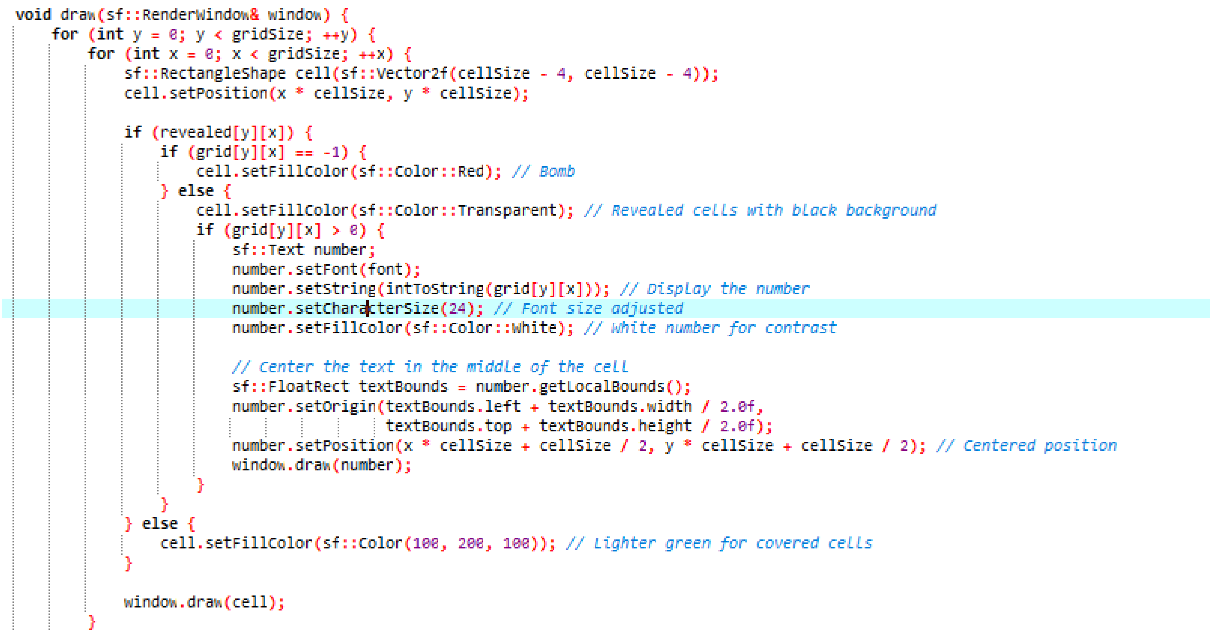
```
void revealCell(int x, int y) {
    if (x < 0 || y < 0 || x >= gridSize || y >= gridSize || revealed[y][x]) return;
    revealed[y][x] = true;
    if (grid[y][x] == 0) {
        for (int dx = -1; dx <= 1; ++dx) {
            for (int dy = -1; dy <= 1; ++dy) {
                revealCell(x + dx, y + dy);
            }
        }
    }
}
```

3. Abstraction:

Abstraction is the process of hiding the implementation details and showing only the essential features of the object. In this code, abstraction is used effectively to simplify interactions with the Minesweeper game logic.

- The Minesweeper class abstracts the details of how the grid is represented, how bombs are placed, and how cells are revealed. The user of the class doesn't need to understand the internal workings of these features to play the game. They only interact with the game through simple methods such as handleClick() and draw().

For example, the draw() method abstracts the process of rendering the grid, the bombs, and the numbers to the screen. The player doesn’t need to worry about the exact position or size of each cell; they just see the final game state displayed in the window.

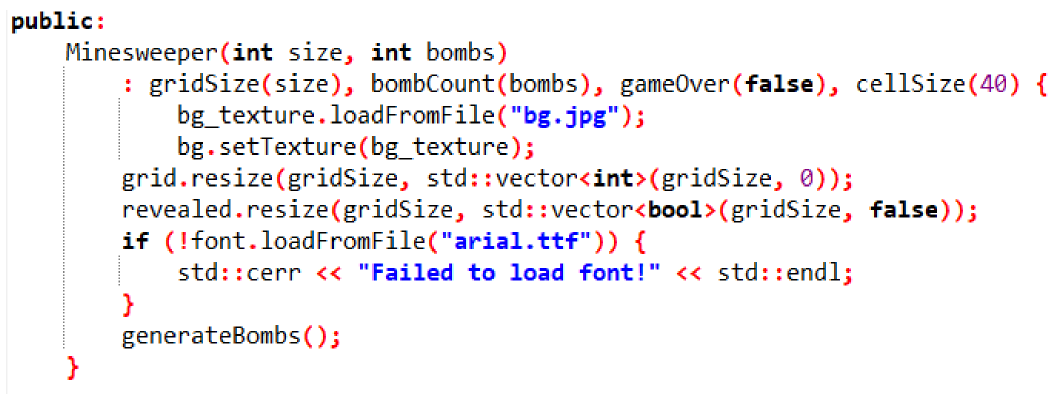


```cpp
void draw(sf::RenderWindow& window) {
    for (int y = 0; y < gridSize; ++y) {
        for (int x = 0; x < gridSize; ++x) {
            sf::RectangleShape cell(sf::Vector2f(cellSize - 4, cellSize - 4));
            cell.setPosition(x * cellSize, y * cellSize);

            if (revealed[y][x]) {
                if (grid[y][x] == -1) {
                    cell.setFillColor(sf::Color::Red); // Bomb
                } else {
                    cell.setFillColor(sf::Color::Transparent); // Revealed cells with black background
                    if (grid[y][x] > 0) {
                        sf::Text number;
                        number.setFont(font);
                        number.setString(intToString(grid[y][x])); // Display the number
                        number.setCharacterSize(24); // Font size adjusted
                        number.setFillColor(sf::Color::White); // White number for contrast

                        // Center the text in the middle of the cell
                        sf::FloatRect textBounds = number.getLocalBounds();
                        number.setOrigin(textBounds.left + textBounds.width / 2.0f,
                                         textBounds.top + textBounds.height / 2.0f);
                        number.setPosition(x * cellSize + cellSize / 2, y * cellSize + cellSize / 2); // Centered position
                        window.draw(number);
                    }
                }
            } else {
                cell.setFillColor(sf::Color(100, 200, 100)); // Lighter green for covered cells
            }

            window.draw(cell);
        }
```

The user interacts with the game via a **high-level abstraction**: clicking cells and watching the game state change. They don’t need to know about the internal grid manipulation or bomb generation.

4. Constructor and Destructor:

In OOP, constructors are used to initialize objects when they are created. The Minesweeper class has a **constructor** that initializes the game grid, bomb count, and other parameters, and it calls the generateBombs() method to set up the game state.



```cpp
public:
    Minesweeper(int size, int bombs)
        : gridSize(size), bombCount(bombs), gameOver(false), cellSize(40) {
            bg_texture.loadFromFile("bg.jpg");
            bg.setTexture(bg_texture);
        grid.resize(gridSize, std::vector<int>(gridSize, 0));
        revealed.resize(gridSize, std::vector<bool>(gridSize, false));
        if (!font.loadFromFile("arial.ttf")) {
            std::cerr << "Failed to load font!" << std::endl;
        }
        generateBombs();
    }
```

There is no explicit **destructor** in the code, but it is not strictly necessary here since the SFML objects (like sf::Texture and sf::Font) are automatically cleaned up when the window closes. However, if there were dynamically allocated memory (using new), a destructor would be necessary to deallocate that memory.

Output:

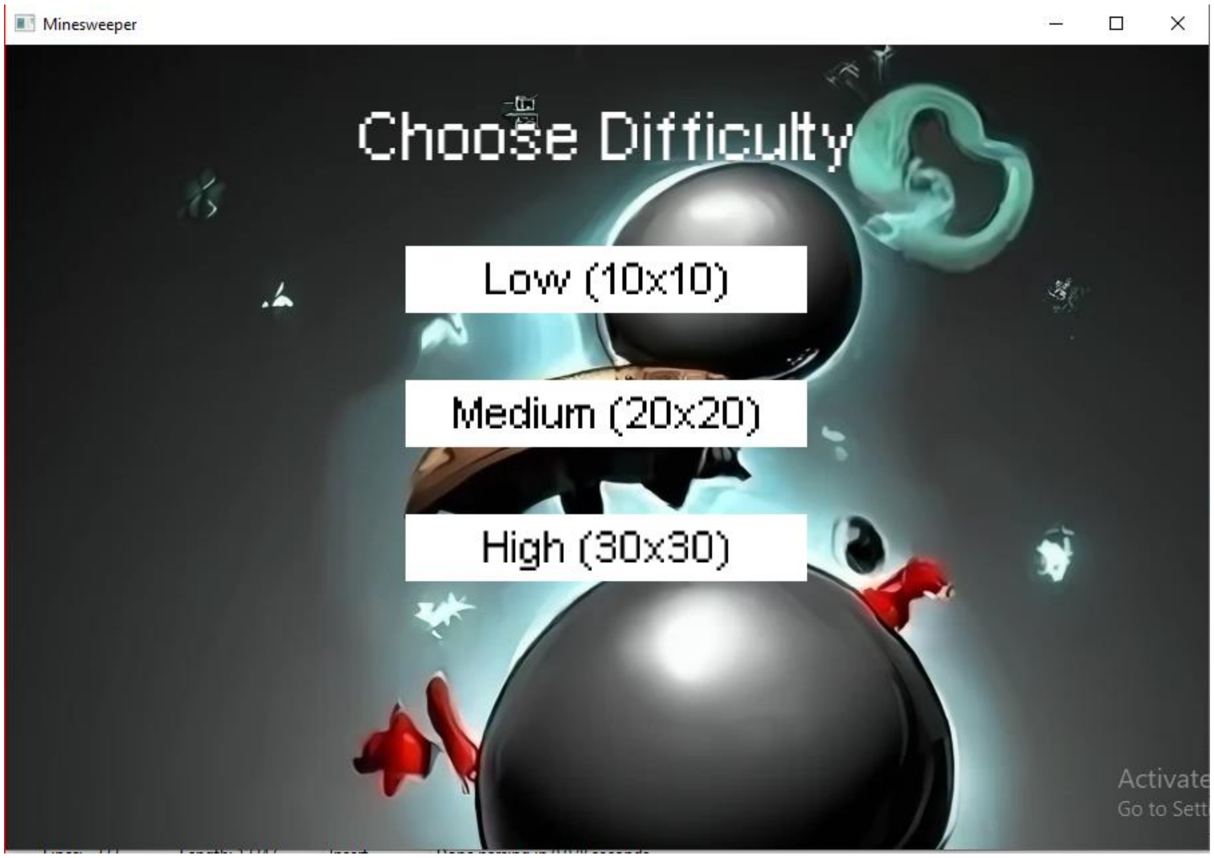


Figure i Menu of the game

Here, the game asks user about the level for game.

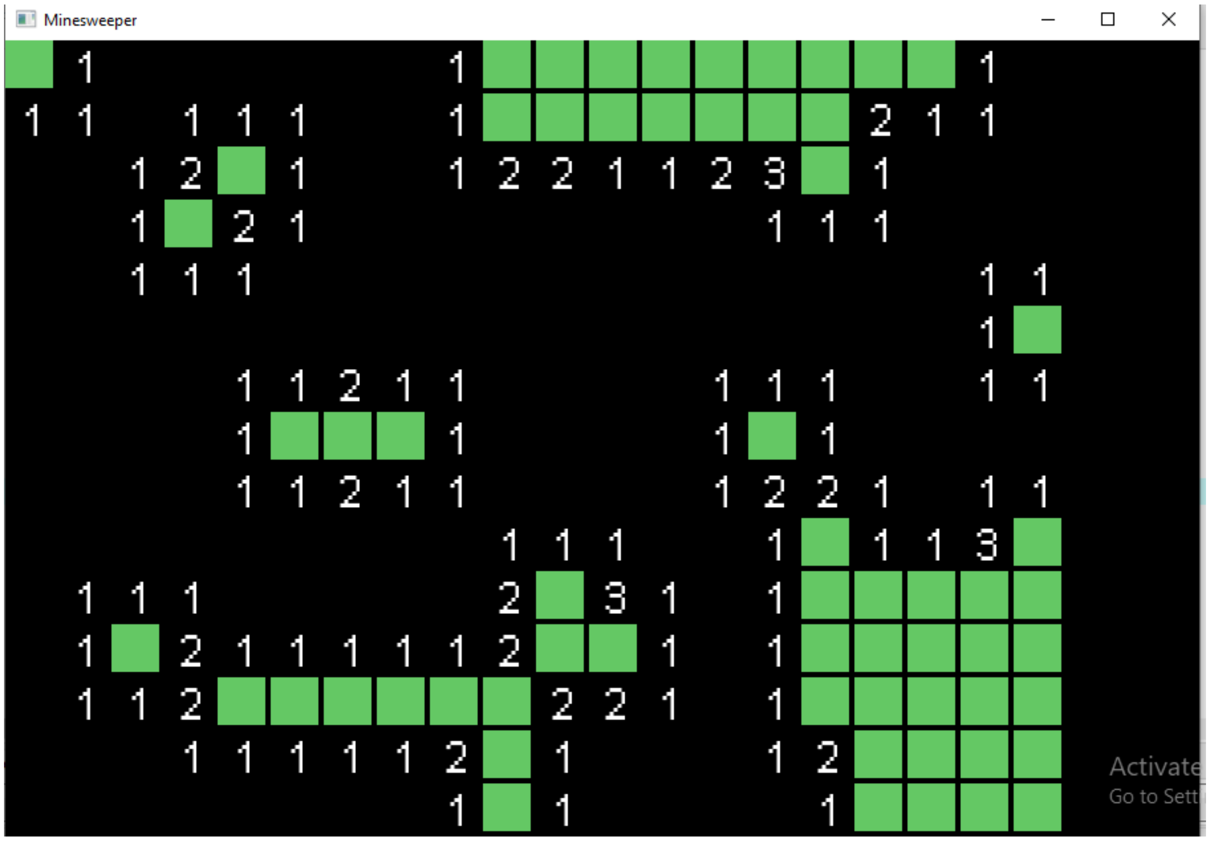
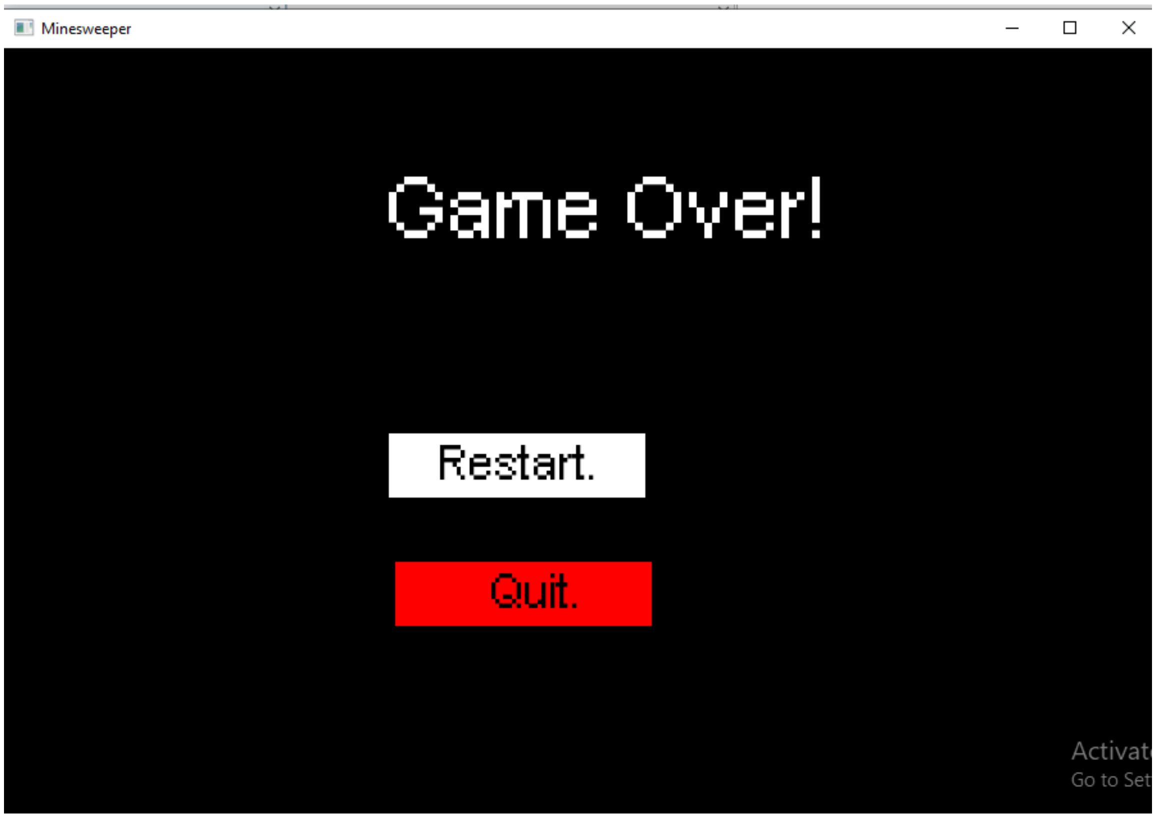


Figure ii Display

Here is how the game runs while revealing the cells we have to be careful of not to click the cell that has bomb which will lead to losing game.



This menu appears after the player clicks the bomb. If the player select restart the game will be restarted and if quit the game will be quit.