System Architecture Overview – Minesweeper

Program Summary

This project is a Python implementation of Minesweeper using the Pygame library. The system runs on a main event loop that listens for user input, updates the game state, and redraws the screen. Version two of this program extended the original architecture with modern features such as new AI game modes, profile creation, user authentication, custom profile pictures, background themes, sound and music effects, a game timer, and high score tracking.

The program operates in four states:

- Menu – Displays the title screen, a menu interface, profile options, and mine count selection.
    - Settings
        - Users can choose between three game modes: Manual, AI Interactive, or AI Automatic.
        - When playing in an AI mode, users can select between Easy, Medium, or Hard difficulties.
        - A Dark Theme and Light Theme are supported for visual customization.
    - Profile
        - Users can create personal profiles to track high scores across game sessions.
        - Users can personalize their profile by uploading custom profile pictures.

- Playing – Core gameplay: reveal tiles, place flags, check win/lose.
    - AI Integration
        - In AI Interactive mode, the AI takes turns with the player, deciding which space to reveal based on difficulty.
        - In AI Automatic mode, the AI autonomously plays the game until completion.
        - Difficulty will impact the AI play:
            - Easy: Random selection
            - Medium: Plays like a human (uses logic & will flag)
            - Hard: Knows all mine positions & will never lose
    - Score
        - Each game session produces a score based on the selected mine count and the speed of completion.
        - Scores are automatically recorded and displayed within the player's profile and high score list.
- Win – Displays victory message, reveals board, returns to menu on click.

- Lose – Displays loss message, reveals board, returns to menu on click.

The architecture separates input handling, game logic, grid data, and rendering, while Pygame provides system-level services for drawing and event management. Additional dedicated managers are used for sounds, timing, authentication, and theming.

Diagram 1 – High-Level Flow

Diagram 1 shows the overall data flow:

- [New] AI turn processing
- User Input (mouse clicks, quit events).
- Event Handling interprets actions.
- Game State Manager determines whether the system is in Menu, Playing, Win, or Lose.
- Game Logic & Board System enforces rules (mine placement, recursive reveals, win/lose checks).
- Rendering Engine updates visuals (grid, buttons, counters, messages).
- User Feedback displays the new state on screen.

This diagram emphasizes the linear loop of input to processing to output.

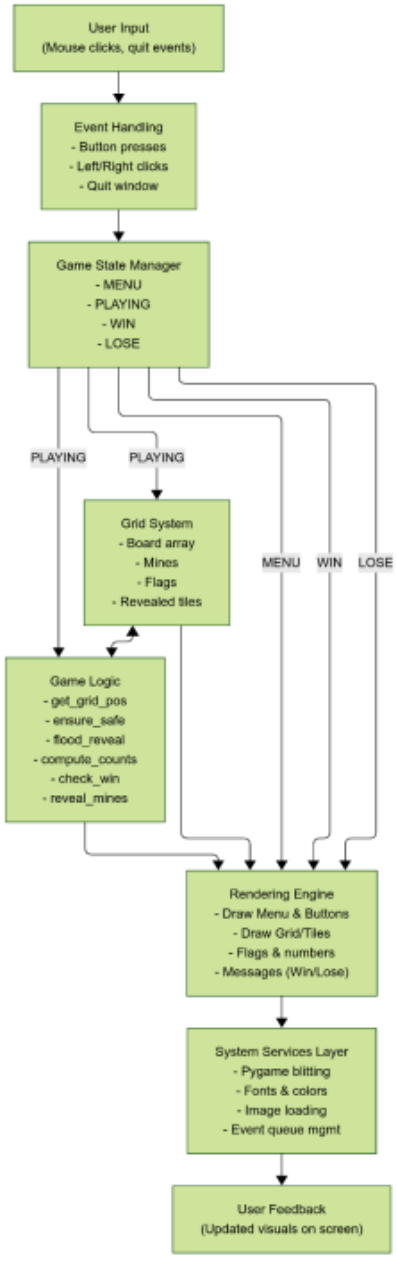

Diagram 2 – Component Interaction

Diagram 2 shows the internal interactions:

- The Main Loop drives all states (Menu, Playing, Win, Lose).
- In Playing, the Grid System (board arrays for mines, flags, revealed tiles) and Game Logic (rules and reveal logic) work together.
- All states feed into the Rendering Engine, which draws menus, grids, and messages.
- The System Services Layer (Pygame fonts, images, colors, event queue) supports rendering and input.
- The loop ends with User Feedback, showing updated visuals to the player.

This diagram emphasizes the interconnected nature of states, logic, and rendering.

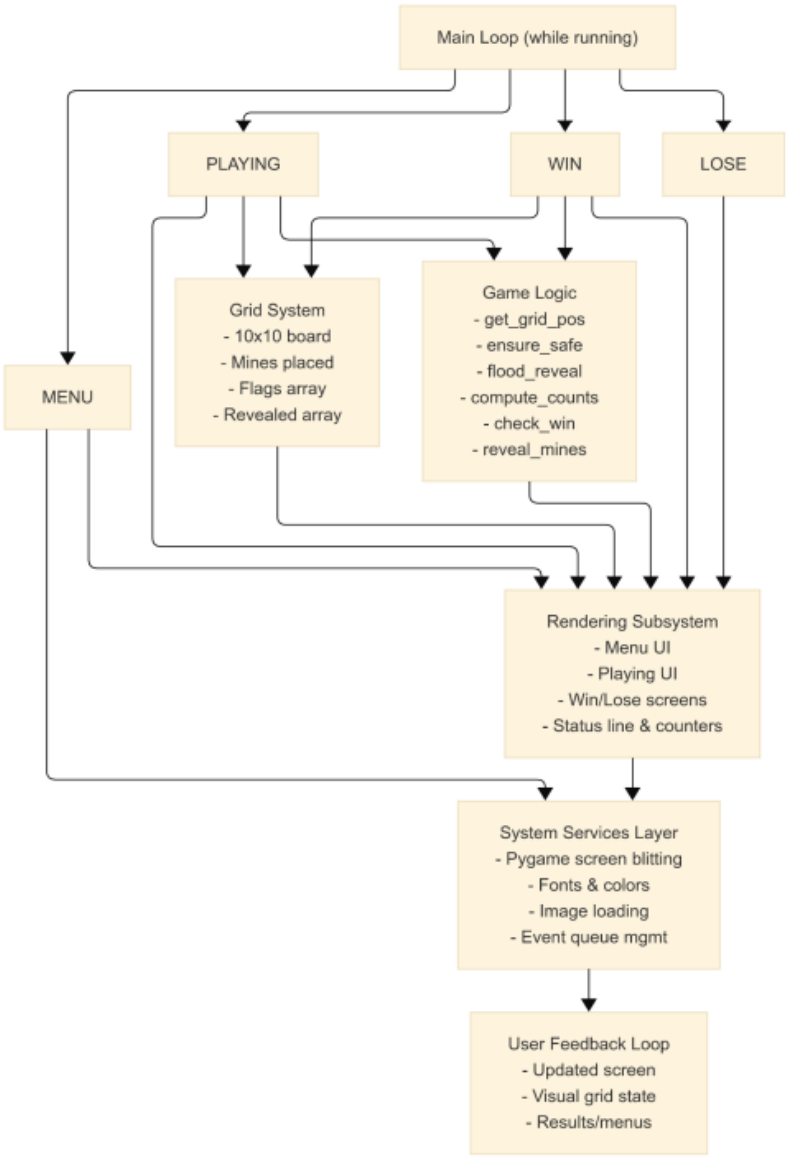

Diagram 3 – Class & New Feature Relationships (UML)

Diagram 3 shows how the new features were integrated into the existing game.

- The MinesweeperUI class serves as the central controller, coordinating gameplay logic, input handling, rendering, and communication between subsystems.
- The AuthContext, GameTimer, SFX, ThemeManager, and HighscoreManager classes each represent independent modules that the UI manages directly.
- Each module operates independently but communicates through clearly defined interfaces with the MinesweeperUI, ensuring modularity and ease of maintenance.
- Helper functions such as save_profile_image() and load_circular_profile() exist outside the main class hierarchy, supporting profile image management without introducing direct class dependencies.