Software Requirement Specification (SRS)

GRID_HERO

Introduction

Team Details:

Serial No.	Name	Roll no	Email address
1	Pragya Rai	IMT2023529	Pragya.Rai@iiitb.ac.in
2	Kaushalraj Puwar	IMT2023063	Kaushalrajsinh.Puwar@iiitb.ac.in
3	Bhushitha Nagendra Kumar	IMT2023048	bhushithanagendra.kumar@iiitb.ac.in
4	MS Vijay Vignesh	IMT2023031	ms.vijayvignesh@iiitb.ac.in
5	Mannat Kaur Bagga	IMT2023071	MannatKaur.Bagga@iiitb.ac.in
5	Krish Kathiria	IMT2023045	krish.kathiria@iiitb.ac.in

Project Overview:

The **Grid Hero** is a strategic pathfinding game where the player must navigate through a grid, avoiding an enemy that dynamically finds the shortest path to capture them. The player's goal is to collect gold coins on the grid before being caught by the enemy. These coins are used to progress through three levels of increasing difficulty and use them in the final boss battle.Once the two levels are completed, the player engages in a final battle with a dragon, marking the end of the game.

Scope:

In-Scope (Part of the Final Product):

1. **Grid Gameplay Mechanics:**
    - A grid-based system where the player and enemy move.
    - Enemy uses the shortest pathfinding algorithm (Dijkstra) to chase the player.
2. **Gold Coin Collection and Usage:**
    - Players earn gold coins based on the number of gold coins collected before being caught by the enemy.
3. **Level Progression:**
    - Two levels with increasing difficulty and distinct coin collection requirements.
    - Each level introduces new challenges or grid modifications (e.g., obstacles,smaller grid size).
4. **Final Boss Battle:**
    - A final confrontation with a dragon after completing the two levels.
    - The player must use collected coins and weapons to defeat the dragon.

5. **Win/Loss Conditions:**
    - The player wins the game after finishing all the levels and defeating the dragon.

Out-of-Scope (Not Part of the Final Product):

1. **Multiplayer Mode**: The game will be single-player only, without any multiplayer or online components.
2. **Advanced AI or Learning Enemy**: The enemy will use basic pathfinding algorithms; it will not adapt or learn from previous movements or strategies.
3. **Open-World Exploration**: The game will be confined to a structured grid with defined levels and boundaries, not an open-world format.
4. **Character Customization**: There will be no options for customizing the player or enemy characters.
5. **Monetization Features**: There will be no in-game purchases, microtransactions, or real money currency.

GitHub Repository Link

https://github.com/KaushalrajPuwar/GridHero

Objectives

The primary objective of the **Grid Adventure Game** is to create an engaging, strategic pathfinding game with two levels and increasing difficulty. The project will focus on implementing core features that enhance player experience and progression through the game.

1. **Grid-based Movement System**:
    - Implement a grid structure where both the player and the enemy can navigate.
    - The player can move in multiple directions (up, down, left, right) within the grid boundaries.
2. **Pathfinding for Enemy**:
    - Develop an enemy using the shortest pathfinding algorithm (Dijkstra) to chase the player by finding the shortest path.
3. **Gold Coin Collection Mechanic**:
    - Implement a system where the player earns gold coins based on the number of coins collected before the enemy catches him.
4. **Level Progression**:
    - Design two distinct levels, each with increasing difficulty, such as more complex grids, additional obstacles.
    - Implement a level progression system where players must meet building-construction goals to unlock the next level.

5. **Final Boss Battle**:
    - Create a final challenge where the player faces a dragon after completing all three levels.
    - Develop a boss battle mechanic where the player must defeat the dragon using coins to buy weapons acquired throughout the game.
6. **Win/Loss Conditions**:
    - Ensure the game is winnable by defeating the dragon after completing all three levels.

These objectives will ensure the game is both challenging and enjoyable, with clear player goals and progressive difficulty as the game advances.

System Overview

Technical Specifications:

1. **Frontend**:
    - **Java**: The user interface will be developed using Java, leveraging its capabilities for creating desktop applications (use of Command Line).
2. **Backend**:
    - **C++**: Processes the user's choice and executes the relevant game logic.
    -
3. **Communication(Middleware)**:

    The JNI-based middleware acts as the communication bridge between the Java frontend and C++ backend, enabling method calls from the frontend to execute corresponding game logic in the backend. It translates Java objects and data into formats the C++ backend can process and vice versa.

Input/Output Requirements:

Player Input Requirements:

1. Movement commands from the player (e.g., up, down, left, right) to control the player's position on the grid.
2. During the fight with the dragon, the player must select one of the options displayed at each step.

Game Output Requirements:

1. Updates on the player’s position, enemy position, and gold coin count are displayed on the console.

2. Notifications for significant events, such as reaching a new level, or being caught by the enemy.
3. Output messages indicate whether the player has completed a level and whether they can advance to the next level.
4. Information about the player's success or failure in the final battle against the dragon, including health status and victory messages.

Functional Requirements

Player Movement

1. Description: The player can move in four directions (up, down, left, right) on the grid.
2. User Interaction: The player uses keyboard arrow keys or WASD keys to navigate.
3. Expected Functionality:
    a. The player’s position on the grid updates based on the input.
    b. Movement is constrained within the grid boundaries.
    c. Note: if the player has completed a level he gains xp which gives him extra power to cross accessible cells and loses the required xp.
4. Special Conditions: The movement is ignored if the player tries to move outside the grid or onto an inaccessible cell or goes back to the previous cell just before.

Enemy Pathfinding

1. Description: The enemy dynamically finds the shortest path to the player using pathfinding algorithms.
2. User Interaction: The enemy automatically moves two cells along the shortest path calculated after each player's movement.
3. Expected Functionality: The enemy recalculates its path after each player moves, adjusting its position accordingly.

Gold Coin Collection

1. Description: Player collects gold coins while moving on the grid before the enemy catches him.
2. Expected Functionality: The total number of collected coins is displayed in the console.

Level Progression

1. Description: Players progress through two levels, each with unique grids with progressing difficulty levels and gain xp.
2. User Interaction: Upon passing the required condition, players are prompted to proceed to the next level.

3. Expected Functionality: Each level introduces new obstacles players must meet the monument requirement to advance.
4. Special Conditions: If the player fails to meet the requirements, they cannot proceed to the next level.

Final Boss Battle

1. Description: After completing the two levels, players face a dragon in a final battle.
2. User Interaction: Players engage in combat by selecting attack, defence, and weapon options through a menu.
3. Note: Expected Functionality: Players use weapons as needed to fight the dragon and defeat them.
4. Special Conditions: The battle has its own set of rules and victory conditions, separate from regular gameplay.

Game Over and Win Conditions

1. Description: The game ends when the player is caught by the enemy or defeats the dragon.
2. User Interaction: Player is shown a message indicating game over or victory, with options to exit which restarts the game.
3. Expected Functionality: If caught, the player is shown a loss message if the dragon is defeated, a victory message is shown.

Use Cases:

1. **Use Case: Player Movement**
    - **Actor**: Player
    - **Scenario**: The player navigates the grid.
    - **Steps**:
        1. The player gives input W/S/A/D to move on the grid.
        2. The game checks if the move is within grid boundaries and is valid as per the grid conditions.
        3. The player's position updates to the new location.
        4. The game renders the new position on the grid.
2. **Use Case: Enemy Pursuit**
    - **Actor**: Enemy
    - **Scenario**: The enemy pursues the player.
    - **Steps**:
        1. The enemy detects the player's last known position.
        2. The enemy calculates the shortest path to the player.
        3. The enemy moves two cells toward the player’s location.

        4. If the enemy reaches the player's position, it triggers the game over condition.
3. **Use Case: Gold Coin Collection**
    - **Actor**: Player
    - **Scenario**: The player collects coins.
    - **Steps**:
        1. While moving on the grid to escape, the player collects gold coins.
        2. The coin count increases, and the updated total is displayed when the level is over.
4. **Use Case: Level Progression**
    - **Actor**: Player
    - **Scenario**: The player progresses to the next level.
    - **Steps**:
        1. The player successfully meets the number of gold coins needed to pass the level .
        2. The game initializes to the next level with a new modified grid with inaccessible cells and smaller size.
5. **Use Case: Final Boss Battle**
    - **Actor**: Player
    - **Scenario**: The player battles the dragon.
    - **Steps**:
        1. The player enters the final battle(final battle).
        2. The player selects attack(fight dragon), defense(dodge), or enter store options to buy health and weapons from the store or enter cave to fight smaller boss dragons.
        3. The game processes the player's choices and updates the battle status.
        4. The player wins or loses based on the battle outcome.
6. **Use Case: Game Over**
    - **Actor**: Player
    - **Scenario**: The game ends when the player defeats the dragon.
    - **Steps**:
        1. The player defeats the dragon.
        2. The game displays a game-over and victory message.
        3. The player is given the option to exit the game.

Development Setup:

C++: Ensure a C++ compiler is installed (e.g., GCC, Clang, or MSVC). Recommended version: `GCC 9.3+` or equivalent.

Java: Install Java Development Kit (JDK). Recommended version: `JDK 11+`.

JNI: Ensure JNI headers (`jni.h`) are available and include directory is in the PATH, typically included with the JDK installation.

Build Tool: Specific commands for building support files and binaries given in Environment Setup.

Environment Setup:

Step 1: Install Required Software

Install **JDK**:

Command: sudo apt install openjdk-11-jdk # Linux
Command: brew install openjdk@11 # macOS
Run the installer from Oracle official website #Windows

Install **GCC**:

Command: sudo apt install build-essential # Linux
Command: brew install gcc # macOS

Step 2: Clone the Repository (Using terminal or GitHub Desktop Application)

Command: git clone https://github.com/KaushalrajPuwar/GridHero.git
Command: cd GridHero

GridHero
|
|---cli
|__controller

Step 3: Compile and Build Backend (C++)

1. Command: cd controller
2. Command: g++ -std=c++17 -o game player/Player.cpp enemy/Enemy.cpp enemy/Dragon.cpp Game.cpp main.cpp inventory/Ammo.cpp inventory/Storage.cpp

Step 4: Set Up Frontend

1. Command: cd cli
2. Command: javac Frontend.java
3. Command: java -Djava.library.path=. Frontend **(To run the frontend)**

Setting up JNI:

Command: cd GridHero/controller
Command:

For linux:
g++ -std=c++17 -shared -fPIC -o libgamebackend.so main.cpp player/Player.cpp enemy/Enemy.cpp enemy/Dragon.cpp Game.cpp inventory/Ammo.cpp inventory/Storage.cpp

For macOS:
g++ -std=c++17 -shared -fPIC -o libgamebackend.dylib main.cpp player/Player.cpp enemy/Enemy.cpp enemy/Dragon.cpp Game.cpp inventory/Ammo.cpp inventory/Storage.cpp

For Windows:
g++ -std=c++17 -shared -fPIC -o libgamebackend.dll main.cpp player/Player.cpp enemy/Enemy.cpp enemy/Dragon.cpp Game.cpp inventory/Ammo.cpp inventory/Storage.cpp

MOVE THIS FILE TO FOLDER "cli".
The path of the file should be: ../GridHero/cli/libgamebackend.dylib/.so/.dll (same as Frontend.java)

Workflow:

Backend: Logic Implementation

The backend logic will handle the game mechanics and processing, including the following:

- **Game Initialization**:
  - Set up the game grid, player, enemy, coins, and inaccessible cells.
  - Adjust configurations based on the current level.
- **Player and Enemy Interaction**:
  - Manage player movements (e.g., avoid inaccessible blocks, collect coins).
  - Calculate the enemy's next move using pathfinding (shortest path via BFS).

- **Level Progression**:

    - Handle transitions between levels based on coins collected and player performance.
    - Reset or reconfigure the grid for new challenges (e.g., new obstacles in higher levels).
- **Dragon Slayer Battle**:
    - Integrate dragon battles in the final level, allowing the player to use weapons to attack dragons and monitor health/damage stats.
- **Game Over and Restart**:
    - Implement logic for game completion or reset if the player is caught by the enemy or fails to meet objectives.

Frontend: User Interface Interaction

The frontend will handle the visualization and interaction between the player and the game using the following:

- **Grid Display**:
    - Show the game grid with distinct markers for the player, enemy, coins, inaccessible areas, and empty cells.
- **Input Handling**:
    - Capture user input for movement (`W`, `A`, `S`, `D`) and display the updated grid.
    - Allow the player to select weapons and strategies for battling dragons in higher levels.
- **Status Updates**:
    - Provide real-time feedback about coins collected, player's position, enemy's position, and other stats.
    - Display messages for level transitions, game over, or success notifications.
- **Interactive Menus**:
    - Create menus for starting a new game, selecting levels, and equipping weapons.

Frontend-Backend Communication

JNI will manage the interaction between the frontend and backend through the following:

- **Command Parsing**:
    - Interpret user inputs from the frontend and invoke the corresponding backend functions (e.g.,move player,initialize grid).
- **Level and Configuration Management**:
    - Handle level progression and ensure backend logic aligns with the player's choices and progress.

Important Files and Folders:

Game.cpp:

The `Game` class encapsulates the core game logic, including grid initialization, player and enemy placement, coin distribution, and movement mechanics.

—----------

Player folder:

Player.cpp:

- **Player Stats & Methods:** Manages attributes like health, coins, XP, and weapons, with methods for earning coins, buying items, and displaying status.
- **Combat and Progression:** Handles taking damage, gaining XP, and tracking health; allows players to buy weapons and health using coins.

Enemy folder:

`Enemy.cpp`:

- **Enemy Movement:** Enables enemies to move towards the player and update their position on the grid.
- **Basic Positioning:** Provides methods to get and set the enemy's position on the map.

`Dragon.cpp`:

- **Dragon-Specific Attributes:** Adds health, damage, and name to represent the dragon's unique stats.
- **Combat Logic:** Handles taking damage, checking if the dragon is defeated, and displaying the dragon's status.

—----------

Inventory folder:

`Ammo.cpp`:

- **Ammo Attributes:** Defines ammo with name, durability, and damage, with methods to track its usability.
- **Usability Check:** Contains methods to use ammo (decrease durability) and check if it's still usable.

`Storage.cpp`:

- **Item Storage:** Manages a collection of ammo items with the ability to add new items to storage.
- **Item Checking:** Provides a method to check if a specific ammo item is present in storage.

—----------

Main file:

main.cpp:

- This file integrates with Java using JNI (Java Native Interface). It allows communication between Java and C++ by exposing C++ functions as Java methods moveplayer(), performAction() and others. This is done by using the `extern "C"` block to define JNIEXPORT methods, which are callable from Java. This is useful for integrating C++ game logic with a Java-based frontend.

—----------

Frontend.java

- **Game Flow Management:** The `Frontend` class handles the interaction with the user, allowing them to select levels, make moves, and perform actions like fighting dragons or entering stores, while communicating with the backend for game state updates.
- **Backend Integration:** It uses Java Native Interface (JNI) to invoke native methods (`gamebackend`), managing game logic like starting levels, moving the player, and performing various in-game actions based on user input and backend responses.

Conclusion:

- **Frontend and Backend Integration**: The project integrates a user-friendly frontend with a game backend to enable seamless gameplay.
- **Interactive Levels**: Players can engage in various levels of the game, each with unique challenges and objectives.
- **Player Actions**: Players can perform various in-game actions, such as movement and combat, enhancing gameplay.
- **Combat with Dragons and Monsters**: The game features combat scenarios with dragons and monsters, adding excitement and challenge.
- **Store for Health and Weapons**: A store allows players to buy health and weapons, providing a strategic element to the game.
- **Cave Exploration**: Players can explore caves, encountering additional challenges and monsters.

- **JNI for Backend Communication**: The project leverages Java Native Interface (JNI) to facilitate communication between the frontend and backend.

UML Diagram: