CSC3150 Assignment 2 Report: Implementing The Greatest Adventurer Game

Name: Jiaju Wang Student ID: 121090544

Your report should follow the template with the following section structure.

No page limitation

1 Introduction [2']

In CSC3150 Assignment 2, I implemented "The Greatest Adventurer" game as a multithreaded program. This game challenges a player to navigate through a dynamically changing dungeon to collect gold shards without colliding with moving walls. My implementation utilizes multiple threads to manage the game objects and their movements, ensuring real-time interaction and game progression.

2 Design [5']

The design of "The Greatest Adventurer" focuses on multithreading and user interaction within a terminal-based environment.

Here's a breakdown of the main components and their designs:

Dungeon:

The game space is a 17x49 grid, with borders represented by '+', '—', and '-'. The interior is dynamically updated with the adventurer ('0'), walls ('='), and gold shards ("$").

Multithreading:

Separate threads handle the movement of walls, gold shards, and the adventurer's navigation to achieve concurrent gameplay. This design choice enhances the game's complexity and realism.

Game Objects and Movement:

Adventurer:

Placed at the dungeon's center, controlled by user input (W/A/S/D for movement and Q for quit). Walls and Gold Shards: Their initial positions and movements are randomized. Walls move horizontally across rows, while gold shards move in random directions. Threads are assigned to each moving object, including the adventurer, for independent control.

Game Logic:

The game checks for collisions, collects gold shards, and ensures the adventurer does not cross the dungeon's borders. The status is continuously monitored to determine the game's outcome (win/lose).

Here's the coding designs:

Libraries

'iostream', 'stdio.h': Standard libraries for input and output operations.

'string.h', 'unistd.h': Libraries for string manipulation and POSIX-compliant system call wrappers, respectively.

'time.h', 'random': Used for time-related functions and generating random numbers.

‘termios.h‘, ‘fcntl.h‘: Handle terminal I/O characteristics and file control options.

‘ncurses.h‘: A library for creating text-based interfaces in a terminal.

‘vector‘, ‘thread‘, ‘mutex‘, ‘condition_variable‘:

Support dynamic arrays, multi-threading, mutual exclusions (mutexes), and condition variables for thread synchronization.

Macro Definitions and Global Variables

‘ROW‘, ‘COLUMN‘, ‘FREQ‘: Define the dimensions of the game map and the frequency of events occurring.

‘ISOVER‘, ‘ISQUIT‘: Flags to indicate whether the game is over and whether the player has quit the game.

The ‘Dungeon‘ Class

This class encapsulates the core game logic and state management.

Private Members:

‘game_mutex‘: A mutex used for synchronizing access to shared resources among threads.

’cv_mutex‘: Another mutex, possibly intended for use with the condition variable but not directly used in the code.

‘adventurer_threshold_cv‘: A condition variable used to synchronize the movement of walls, goals, and the adventurer.

‘thread_ids‘: An array to identify threads for the walls’ movement. Not explicitly used for identification in the provided code.

‘goals_move_dr‘: A vector storing the initial movement direction of each goal.

‘map‘: A 2D vector representing the game map.

‘times_count‘, ‘thread_count‘: Counters used for managing the movement frequency and synchronization between threads.

‘colectionsGoalsCount‘: Tracks the number of goals collected by the adventurer.

‘Node adventurer‘: A structure to store the adventurer’s position.

Public Methods

‘getInstance()‘: Implements the Singleton pattern, ensuring only one instance of the ‘Dungeon‘ class exists.

‘initialize_map()‘: Initializes the game map, placing walls, goals, and setting the adventurer’s starting position.

‘draw()‘: Clears the screen and redraws the map, showing the current state of the game.

‘check_status(int x, int y)‘: Checks the game state based on the adventurer’s new position, such as encountering walls, goals, or moving out of bounds.

‘adventurer_move(int t)‘: Handles the movement of the adventurer based on keyboard input and updates the game state.

‘walls_move(int t)‘: Manages the movement of walls in the dungeon.

‘goals_move(int t)‘: Manages the movement of goals in the dungeon.

‘kbhit‘ Function

Checks for keyboard input to control the adventurer’s movements. This function uses terminal I/O settings to non-blockingly read a character from the standard input, allowing the game to respond to player inputs without pausing.

Thread Functions:

‘goals_move_thread(int aa)‘: Wrapper function to start the ‘goals_move‘ method in a thread.

‘walls_movee_thread(int aa)‘: Wrapper function to start the ‘walls_move‘ method in a thread.

‘adventurer_move_thread(int aa)‘: Wrapper function to start the ‘adventurer_move‘ method in a thread.

‘main‘ Function

Initializes the ncurses library for handling input/output in a terminal.

Calls ‘initialize_map()‘ to prepare the game.

Creates threads for handling the movement of walls, goals, and the adventurer.

Waits for all threads to finish.

Displays the game outcome based on the ‘ISOVER‘ variable.

(Initializes the ncurses library and the game map. Launches threads, including six for controlling the walls, one for controlling the coins, and one for controlling the adventurer. Waits for all threads to complete, then ends the game session and prints the game’s outcome based on the global flags ‘ISOVER‘ and ‘ISQUIT‘.)

The Extra Credit

Simplified Thread Management

The game uses only two threads: one for moving walls and goals, and another for the adventurer’s movement. This simplification from the previous version reduces complexity in thread synchronization and resource sharing, focusing on the core gameplay mechanics.

Game State Management

The global variables ‘ISOVER‘ and ‘ISQUIT‘ manage the game’s state, indicating whether the game has ended due to a win/loss condition or if the player has quit. This global state control allows threads to coordinate the termination of the game loop.

Singleton Pattern

‘Dungeon‘ class implements the Singleton pattern through its ‘getInstance()‘ method. This ensures that only one instance of the dungeon exists, centralizing the game state and mechanics. The Singleton pattern is appropriate for scenarios like this game, where a single game state must be accessible from various parts of the program.

Threading and Synchronization

The use of threads for controlling game elements (walls, goals, and adventurer) ensures that the game world is dynamic and interactive. The ‘std::mutex‘ and ‘std::condition_variable‘ are used to synchronize the adventurer’s movement with the automated movements of walls and goals, preventing race conditions and ensuring that game updates are coherent.

3 Environment and Execution [2’]

Attention If you get stuck after running the program, please end the process and run it again and it will be ok, because this is a system refresh problem and the code of this job can run normally.

Tip: If the corresponding library is not installed, here is a code to help you.

code: sudo apt-get install libncurses5-dev libncursesw5-dev

Environment Setup:

The game runs in a Linux environment, specifically Ubuntu 20.04.6 LTS with a GCC version of 9.4.0, ensuring compatibility with the assignment’s requirements.

Execution Steps:

1. Compile the program with ‘g++ hw2.cpp -lpthread -lncurses‘.

2. Run the game using ‘./a.out‘.

3. Control the adventurer with W/A/S/D and quit the game with Q.

4. The game ends when all gold shards are collected or the adventurer collides with a wall.

The Extra credit

Attention: Before running the second file, delete the a.out file (the file generated in the first question) to avoid incorrect execution.

Execution Steps:

1. Compile the program with ‘g++ hw2_extra_credit.cpp -lpthread -lncurses‘.

2. Run the game using ‘./a.out‘.

3. Control the adventurer with W/A/S/D and quit the game with Q.

4. The game ends when all gold shards are collected or the adventurer collides with a wall.

4 Conclusion [2’]

This assignment provided me with hands-on experience in multithreaded programming and game design within a constrained environment (terminal-based interface). I learned about thread synchronization, condition variables, and the challenges of managing multiple entities in a shared space. The project underscored the importance of careful design and testing in developing interactive applications. Moreover, it highlighted the potential of terminal-based games as a valuable tool for understanding core programming concepts such as multithreading and real-time input handling.

The Extra Credit

Using a singleton pattern for the ‘Dungeon‘ class ensured a single instance of the game state, simplifying state management across different threads. This project deepened my understanding of practical multithreading applications, game design principles, and advanced C++ features like mutexes, condition variables, and the ncurses library for terminal manipulation.