Software Engineering Concept (COMP3003)

Assignment 1 Report

Clement Diong Yong Bin 21228511

1. An explanation of your design in regards to multithreading.

In particular

- **Which classes are responsible for starting threads, and what are these threads used for?**

The JFXArena class is responsible for starting threads. There are two thread pools using ExecutorService. The first one is the "executorservice," which can run up to 100 threads concurrently. It manages tasks such as controlling the movement of robots, generating robots, and tracking the score. The second one is the "wallPlacementExecutor," which allows only one thread to run at a time. I implemented the "wallPlacementExecutor" because only one wall can be built at a time. Other walls can only be constructed if the previous wall is successfully built.

- **How do the threads communicate?**

Firstly, in the constructor of the JFXArena class, the generateRandomRobots() function is called to initiate the "executorservice" thread responsible for generating robots and tracking the game's score until it ends. Each thread task is assigned to individual robots by invoking the startRobotTasks(robot) function, allowing each robot to perform its movements with delays using Thread.sleep(). The startRobotTasks(robot) function also checks for collisions between robots and walls. When a robot is destroyed, it ends the specific robot's thread task.

If one of the robots enters the citadel, the executor service is shut down using the shutdown() method. This action terminates all threads under both the "executorservice" and "wallPlacementExecutor" thread pools immediately.

Additionally, there is an extra executor service called "wallPlacementExecutor," which allows only one wall to be built at a time. This thread is started only when the player clicks on a specific grid. Walls are built sequentially, one after the other, with the next wall thread executing only when the previous wall thread has completed its task.

- **How do the threads share resources (if they do at all) without incurring race conditions or deadlocks?**

  I utilize a blocking queue to store robot and wall objects, thereby establishing a mechanism that allows threads to enqueue and dequeue items without encountering conflicts during the addition and removal of elements. This design effectively mitigates race conditions.

  Furthermore, I employ an Executor Service to manage tasks such as robot movement and wall placement, ensuring the availability of a thread pool for concurrent task execution, which enhances overall system efficiency.

  To update JavaFX UI components, including labels and text areas, I adopt the Platform.runLater() method. This strategic choice schedules tasks to be executed on the JavaFX Application Thread, commonly known as the UI thread. This approach guarantees that graphical user interface (GUI) updates transpire in a manner that is inherently thread-safe, effectively preventing race conditions when making modifications to UI elements.

- **How do the threads end?**

  When one of the robots enters the citadel grid, the gameover Boolean is set to true. Upon the gameover Boolean becoming true, all threads are promptly shut down.

2. Consider architectural issues relating to scalability.

That is, what if your program has to deal with a very large version of the problem described? Consider a much larger grid, with thousands or millions of game objects, considerably shorter time intervals, greatly expanded player options.

1. **What are the kinds of non-functional requirements we might care about? And what problems would a very large number of game objects create?**

   I think the most important aspects are performance and scalability. We need to ensure that the game has low latency and high throughput to handle a large number of game objects and players without significant performance degradation. However, we should also consider resource management. It's crucial to optimize memory usage to prevent excessive resource consumption and ensure that the game code is efficient, not overly taxing the CPU. Additionally, we need to prioritize the user experience by providing a user-friendly and responsive user interface that can smoothly handle player actions and menu navigation. It's also important to consider accessibility features to make the game inclusive for a wide range of players.

   A very large number of game objects will create several problems. Firstly, it can lead to performance degradation, overwhelming the game engine and causing a significant drop in frame rates and gameplay performance. This can result in a sluggish or unplayable experience. Secondly, it will result in high CPU and memory usage. Managing a large number of game objects requires substantial CPU and memory resources. Inefficient coding or data structures can lead to excessive resource consumption, potentially causing crashes or slowdowns. Moreover, pathfinding complexity will increase. In tower defense games, pathfinding algorithms become more complex as the number of game objects increases. Calculating paths for a large number of units can strain the CPU and cause delays in unit movement.

2. **How could you re-engineer the application's architecture to address these problems? What trade-offs might you be forced to make?**

   I will employ Data-Oriented Design, which emphasizes organizing data for efficient processing. This approach involves using data structures and layouts that are cache-friendly, minimizing memory access times, and enhancing CPU performance. Additionally, I will implement object pooling to recycle game objects rather than creating and destroying them frequently. This strategy reduces memory allocation overhead and can significantly improve performance.

   Furthermore, I will select efficient algorithms for tasks such as pathfinding, collision detection, and AI. Profiling and optimizing these critical components can have a substantial impact on performance.

   As for trade-offs, I will be willing to sacrifice some level of realism in favor of better performance. This decision is based on the belief that performance significantly affects the user experience more than a higher level of realism.

   Moreover, I may simplify the player experience to manage system complexity by reducing the number of player options or game mechanics. This approach aims to make the game more accessible to a wider audience while maintaining performance and playability.