COMP3003: Software Engineering Concepts

Report

Authors:

Naseh Rizvi 20167671

Contents

1 Multi-Thread Design

1.1 Thread Creation

The classes that are responsible for creating threads are the RobotManager class, WallBuilder class and Score-UpdateRunnable class.

1.2 Main Application Thread

The primary application thread is responsible for starting the game loop, establishing the game environment, and initialising the graphical user interface. It serves as a doorway from which your application may be accessed.

1.2.1 RobotManager Thread

The RobotManager class is responsible for creating a thread to manage the construction and movement of robots. Within the run method of the class. a thread is created from a runnable task and submitted to the thread pool via the submit method of the thread pool. As soon as the robot is created it waits for a certain amout of milliseconds before moving to the next position. The moveRobot method in the RobotSpawner class is responsible for this, if the robot moves into a location with a wall or a citadel, the thread is then interrupted and the robot is removed from the area.

1.2.2 WallBuilder Thread

The WallBuilder class is responsible for creating a thread to manage the construction of the walls. In the run method in the class creates a runnable task that periodically adds the walls to the arena. A thread is created when this runnable is passed into the thread pool via the submit method. The purpose of this task is to simulate the continuous construction of walls in the game. It reguarly checks the queue, "buildQueue" for walls to be built and builds them to the game arena at regular intervals of 2000 ms. This thread ensures a continuous supply of walls for the game.

1.2.3 ScoreUpdateRunnable Thread

The ScoreUpdateRunnable class if responsible for updating the score continuously. It starts a thread to periodically update the score based on the game's progress. the score field is updated every second and whenever the collision between a robot and a wall.

1.3 Thread Communication

1.3.1 WallBuilder to Main Thread

The WallBuilder thread communicates with the main thread through the use of the Platform.runLater() method. When a wall is successfully constructed in the WallBuilder thread, it uses this method to update the GUI to reflect the addition of the new wall. Additionally, it communicates with the main thread to append log messages about the wall's construction.

1.3.2 ScoreUpdateRunnable to Main Thread:

Similarly, the ScoreUpdateRunnable thread communicates with the main thread via Platform.runLater() to update the score label in the GUI. This ensures that UI updates are done on the JavaFX Application thread, preventing GUI-related issues.

1.4 Shared Resource

1.4.1 Wall Construction

The BlockingQueue¡Wall¿ buildQueue is used to manage walls awaiting construction. Threads interact with this queue to add walls for construction (addWallToQueue) and to take walls for building (buildQueue.take()).

1.4.2 Synchronization

A synchronized block is used around the critical section of code that accesses the buildQueue. This ensures that only one thread can access the queue at a time, preventing race conditions where multiple threads might attempt to modify the queue simultaneously.

1.4.3 GUI Updates

When updating the GUI components like labels and the log, you wisely use Platform.runLater() to execute GUI-related code on the JavaFX Application thread. This avoids contention for GUI resources and potential race conditions.

1.5 Thread Termination

1.5.1 WallBuilder Thread Termination

The WallBuilder thread runs in an infinite loop, but it can be interrupted when necessary. When the application is shut down or no longer needs to build walls, the program calls threadPool.shutdownNow(), which interrupts the WallBuilder thread and gracefully shuts it down. The thread will then reach its termination point, releasing any resources it holds.

1.5.2 ScoreUpdateRunnable Thread Termination

Similar to the WallBuilder thread, the ScoreUpdateRunnable thread can also be stopped using the stopThread() method. When the application is closed or no longer requires score updates, stopThread() is called to set the running flag to false, allowing the thread to gracefully terminate.

In conclusion, the design demonstrates a thoughtful approach to multithreading in a JavaFX application. It effectively utilizes threads to handle concurrent tasks such as wall construction and score updates, ensures safe resource sharing, and manages thread termination to prevent resource leaks. Proper synchronization and communication mechanisms like BlockingQueue and Platform.runLater() contribute to the overall robustness and responsiveness of your application.

2 Scalability

2.1 Non-Functional Requirements & Problems

Performance is considered to be one of the most important non-functional requirements in this setting. The gaming experience that players have come to anticipate is one that is fluid and responsive, devoid of any discernible lag or delays. As the stakes of the game increase, it will be increasingly difficult to reach this level of performance. As a result, the architecture needs to be developed so that it can effectively handle a high load.

Scalability is another criteria that is absolutely necessary. The system ought to be able to elegantly accept an increasing number of game objects and participants without experiencing a decline in performance. Scalability assures that the game will continue to be fun even if the number of players continues to grow. Vertical and horizontal scalability are both involved in this process. Vertical scalability refers to the process of adding more resources to existing components, while horizontal scalability refers to the process of transferring the load across several instances or servers.

As the number of game objects increases, the need for careful management of concurrency becomes more pressing. The presence of numerous players simultaneously interacting with the game environment necessitates the implementation of appropriate synchronisation methods in order to avoid problems such as race situations and deadlocks. Controlling concurrency guarantees that several processes or threads can access shared resources in a secure manner without triggering any conflicts.

When playing a game on a larger scale, proper resource management quickly becomes one of the most important concerns. It is absolutely necessary to make effective use of all available resources, such as memory and the central processing unit (CPU), in order to avoid bottlenecks and crashes in the system. In order to keep optimal performance despite severe loads, it is necessary to put into practise tactics that optimise resource use.

2.2 Application Architecture

Modifications to the architecture are required so that these scalability issues can be addressed. A distributed processing architecture, in which different aspects of the game are executed on distinct servers or nodes, is one option that can be taken. This enables the load to be distributed among multiple servers, which makes it simpler to scale horizontally by adding additional servers as required. Mechanisms for load balancing ensure that the workload is evenly distributed among all of these servers, so avoiding any one server from becoming a bottleneck in the process.

When working with a game of a significant magnitude, database optimisation is absolutely necessary. It is recommended that the database schema be optimised for performance, and that methods such as sharding be used to distribute data over numerous database servers. The practise of caching data that is often accessed can further lessen the strain placed on the database and enhance response times.

Processing tasks in an asynchronous manner is a useful strategy for dealing with activities that do not call for immediate attention. Calculations that are not immediately necessary can be handled by background processes, freeing up resources for more important gaming actions. Scalability can be considerably improved with the use of this asynchronous technique.

2.3 Trade-offs

Nevertheless, these alterations to the architecture might need making some sacrifices. The addition of distributed components and microservices might result in a rise in complexity, which in turn can have an impact on the amount of time required for development. It can be difficult to strike a balance between low-latency gameplay and the ability to maintain a consistent game state across dispersed nodes. In addition, extending horizontally might result in a rise in operational costs, which means that trade-offs between cost and scalability need to be carefully considered.

To summarise, tackling scalability in a large-scale game application calls for a methodology that takes into account non-functional requirements, changes in architectural design, and trade-offs. The objective is to facilitate an increasing number of players while simultaneously maintaining a fluid and responsive gameplay experience. Scalability should have been incorporated into the architecture of the application from the very beginning in order to prevent the difficulties that are connected with retrofitting scalability into an already established system.