Report

Assignment 1

Karthik Karavatt

Student ID
20619965

Curtin University
10/09/2023

Contents

Multithreading Design

App

This is mostly unchanged from the demo code other than some the way it is launched because it is done in koltin

JFX Arena

This class is also mostly the same other than the game object. The game object is initialized here and the objects like the robots and wall are accessed from the game object and rendered to the UI.

Game

This class handles all the all the threads other than the JFX threads.

It uses two Synchronized containers

- walls
- robots
- the containers themselves are thread safe without using locks
- The contents inside are not thread safe though

In there class 3 threads are created and 1 thread pool

- Robot spawn thread
- Robot AI thread pool
- Wall thread
- Score thread

Robot Spawn Thread

- This thread is responsible for spawning the robots on the map
- It communicates by adding a robot to a MutableMap
- This resource is shared with the JFX thread, and Robot AI thread
- It is using a Synchronized MutableMap so adding the robots is thread Safe
- The thread is interrupted when the game is ends or the cross button is pressed on the window

robot AI thread

- for every robot a thread is created that moves their position on the map
- The robots position changes, this is updated on the JFX thread when requestLayout is called
- Because robots need to know if other robots are near them multiple robot threads can access the position of a robot
- The JFX thread can also access the position of the robot
- We do not want the robot's position to be accessed and updated at the same time so a lock is placed on the robot class when the position is accessed
- This is using a executioner service (thread pool)
- This thread will end when the robot is destroyed
- This is so that number of robots is not that much more than the threads available on the CPU otherwise, the robots will freeze
- When the game is over the executioner service will also be destroyed

Wall Thread

- Is responsible for handling placing the walls on the map
- This uses a blocking queue to communicate with the JFX thread
- Every time a grid is clicked on the map a wall is placed in the queue and a variable wall amount it incremented
- Once the queue is not empty, the wall Thread stops blocking and adds the wall to the grid
- Then adds a delay if needed
- The gird can be clicked multiple times on the JFX thread, and the walls will be added to the queue (only up to 5)
- The WallThread will continue to add them to the screen at a steady rate and block until the queue is empty
- The blocking queue avoids the deadlock
- This way the placement of the walls can be delayed while the ui is still responsive
- Once grid is clicked the wall amount variable is incremented
- The represents the number of walls in the queue
- The max number of walls is 10
- To increment and decrement this variable, a lock is placed (wallStatusLock)
- This lock is mainly so the UI elements can be updated properly
- As it is using a Simple String Property which is not thread safe
- When clicking the grid, to ensure the number of walls placed do not exceed the max capacity another lock is used
- The wallPlace lock is used in the squareClicked Listener so too many walls cannot be placed
- This thread ends when the game ends or the exit button is clicked on the window

Robot

- Represents the robot
- It will lock itself when it's position is being udpated or accessed
- As the JFX thread can access the position
- And at the same time the RobotAi thread can update the position

Wall

- Represents a wall
- Multiple threads can access the damaged variable
- It is atomic to avoid avoid it being set by multiple threads at once

Point

- A data class that represents a point

Architectural issues

We want the ui to be responsive

- Each robot has its own thread
- Currently the executioner service uses a fixed thread pool, using the maximum number of available processors
- If we are out of process and the robot AI threads are still moving, the robots may start to freeze
- This could be solved using
  - A system with more cores
  - cachedThreadPools
  - reuse the tread for a different robot when the robot is delayed
- This increases the complexity of the code
- As multiple threads can move the robot it increases the coupling as well
- making the code more error prone and harder to test
- To check if a robot can is at a grid, the game has to iterate over the entire grid
- This is very inefficient for very large grid sized
- To solve this we could use
  - better data structures such as quad trees or octrees
  - use better algorithms with lower time complexity
- This also increases complexity
- At a certain point, the gird size can still be large enough to slow the game down

Users should be able to see other users actions with minimal delay

- Assumption: The game is now multiplayer as it has lots of users
- If the game become multiplayer, we must insure the game is Synchronized with different users
- We now have to introduce a server, to process user requests
- This server must insure all users are Synchronized
- So when a user places a robot another user can see it on the grid
- We could use distributed network
  - When dealing with networks security is an issue
  - We need anti cheat detection, so a user cannot cheat and place more walls than needed