Stephen den Boer

19761257

Software engineering Concepts Assignment1 report

11/09/2023

# Explain design and multithreading

* Which classes are responsible for starting threads and what are these threads used for?
  + App.java adds an event listener to the GUI that creates a thread. The event handler creates a new thread because the wall creating task has the possibility to block, which is something that we want to avoid the GUI thread doing.
  + The MovementManager class creates threads using a thread pool. These threads are associated with the movement queue and are responsible for processing the movements that are added to that queue.
  + The Robot class creates a thread for each robot object to trigger a movement action for the robot after a delay.
  + The RobotCreator class creates a single thread for creating a new robot every 1500ms.
  + The ScoreManager class also creates a single thread. This is used to update the score every second.
  + The WallManager class creates a single thread for processing the build wall requests every 2000ms as they get added to the wall queue.
* How do the threads communicate?
  + Threads either use a blocking queue, private class fields or a monitor to communicate. There is also calls to platform.runLater to communicate with the gui thread
  + A blocking queue is used for the movement manager, threads that create movement runnable add them to the queue and the threads in the thread pool run those tasks.
  + The when communicating with the gui thread, methods from the JFXArena class are called using platform.runlater
  + Threads communicate through the 2D grid that holds the grid states. Whenever making changes to the grid, it is locked through a mutex, so that decisions can be made based on the state of the grid at that time without it changing. It also communicates to the movement blocking queue by submitting movement runnables.
  + The Robot class allows 2 different threads to communicate to each other using a private field locked by a monitor. When the field has been updated, it notifies the other thread.
  + ScoreManager and WallManager, both use a StringProperty for updating a label. When changing this value, they communicate the update to the GUI using platform.runlater.
  + WallManager uses a blocking queue to communicate wall building requests. It also uses a mutex to lock access to the number of walls so it doesn’t get changed at the same time by multiple threads.
* How do the threads share resources without incurring race conditions or deadlocks?
  + The movement blocking queue ensures that movement runnables can be added and taken from the queue easily. The blocking queue ensures that any waiting threads will be notified when there is new items in an empty queue or space in the queue for more items. This is also true for the WallManager which makes use of a fixed size blocking queue.
  + Other race conditions are avoided by having mutexes or a monitor to lock resources that can get changed by other threads whenever they are being read or changed.
  + Since the gui cannot afford to block waiting for shared resources, it has a duplicate copy of the robots and the walls (the robots is just their positions). This allows the gui to keep making its updates without having to wait for other threads which are modifying the robot or wall objects. This comes with some issues too. Whenever updates are made to the wall or robot objects, (their position anyway), these changes also need to be made in the gui. Particularly important is that when creating a robot, it needed to be added to the gui before the robot object creation. Otherwise a move request for that robot could be processed before the gui has a copy of the robot, and won’t be able to find it in time to process the move request. To solve this, the Robot was created on the gui thread using platform.runlater, after the gui had attained it’s copy.
* How do the threads end?
  + All the threads call blocking methods at some point, catching the InterruptedException and doing nothing. This allows the thread to end. Often they are running an infinite loop and this will exit the loop, ending the thread’s task. To cause these InterruptedExceptions, an event handler is added in App.java to handle to OnCloseRequest event. This occurs when the user hits the exit button, and is also manually triggered when the game is over. This event handler calls all the relevant methods in the relevant classes to send call interrupt() on all their threads. It will then display an alert with the final score.

# Consider Architectural issues relating to scalability

## What issues would it raise

* What are the kinds of non-functional requirements we might care about?
  + Response time between clicking a square, and wall in queue number updating.
  + Time taken for robots to move.
  + Delay between robot movements (compared to their expected delay)
  + Number of objects it can support before it crashes or becomes unusable.
* What problems would a very large number of game objects create?
  + There would be a lot more game updates. More walls, and robots are trying to get CPU time to perform their actions. This could overwhelm the CPU and it might not be able to keep up with processing the actions.
  + Since there is a thread for each robot, there would also be a lot more threads. This also means more threads are trying to access the same resources and the race conditions are made more severe. This could lead to the problem where due to CPU scheduling some threads get favoured over others. 1 thread could be left waiting for a long time while others get to perform their actions multiple times.
  + As the number of threads increases, it will become much larger than the number of CPU threads available. Since not all of these threads can physically run at the same time, the CPU will have to switch between these threads. At a large enough scale, these threads will greatly outnumber the CPU threads and will cause a lot of context switching between threads which is expensive (relatively speaking). This could lead to CPU thrashing, where more time is spend switching threads than actually completing the task. Even if something like 10% of the time is spent switching threads, it could have a noticeable impact on performance.
  + There would be a lot of gui updates as there are far more robots and walls changing and potentially changing more frequently (if there is shorter time intervals for movement). This could cause the gui to freeze or slow down and become out of sync with any user input (clicking to add a wall).

## How might we address these issues?

* How could you re-engineer the application’s architecture to address these problems?
  + Instead of creating a thread for each new robot object, a central class could be used for managing the robot threads. This class could make use of a thread pool with a limited size and robots would have to queue their tasks for the thread pool and would block when the queue is full, giving the opportunity for a different thread to be active on the CPU. This would help address the issues with having too many threads.
  + To avoid the gui freezing, it could potentially be made multithreaded, with different parts of the screen (like maybe ¼ of the grid) being rendered in a different thread. Likely this will cause more issues than it solves. A better approach is to be absolutely ruthless in the tasks that the gui gets assigned and do everything possible to make them lightweight. One good approach might be to only render the layout occasionally rather than after every single update to a robot or wall. (this could look like after every Xth update or after a certain amount of time (probably would require another thread to schedule the update)
  + A more general approach would be to profile the code and identify the parts of the code that take the most time. This will highlight parts of the code that get run often, especially by multiple threads. These parts of the code could then be targeted for optimisation to help them run more efficiently and reduce the strain on the CPU.
* What trade-offs might you be forced to make?
  + Convenience vs performance. While it might be easier to program things to work one way, (such as a single thread for each robot) it might not be feasible from a performance standpoint and a more complex approach might have to be taken.
  + Another trade off might be accuracy vs usability. The primary example here is with the update rate of the gui. To ensure the best accuracy, it should update after every change. However, this might cause the gui to be overloaded and freeze or slow the gui so that it becomes unsightly (slow moving animations) and unresponsive (user doesn’t see their clicks register walls being added to the queue). The trade off will be between how much accuracy, and how much usability are valued.
  + There is also a trade off when considering the number of threads. More threads might lead to better usability since more objects are able to have their own thread and don’t have to wait for one to be free. However too many threads will overwhelm the CPU with context switching. There is a trade off on either side of that sweet point in the middle which maximises performance. (this sweet spot will depend on the CPU and how many game objects/user input there is).
  + There is also a trade off of memory vs performance. With that many game objects the system may not be able to store them all in RAM. So while having things like duplicate copies of all the robots and walls, or having a 2D array for each square in the grid are convenient and increase performance, there is a trade off to be made by removing these things in favour of reducing the memory usage.