GHRAHM004 – Assignment 2 – Concurrency

**Introduction (5)**

For this assignment, we looked to build on the previous assignment by simulating how water on a terrain flows downhill, accumulating in basins and flowing off the edge of the terrain.

The aim was to do this by:

- Using a multi-threaded Java program

- Ensuring both thread safety and sufficient concurrency

- Using a GUI to control the simulation

**Design and Implementation (15)**

Classes added and Modifications made

Flow:

* Provided in Skeleton Code:
  + Contains the main method and GUI which user would interact with.
* Modifications Made:
  + Distributed the simulation workload across 4 threads by creating 4 new FlowPanel objects in Flow that were each responsible for processing a different section of the grid. The original FlowPanel object was the only one that was visible to the GUI and would be updated by the FlowPanel objects sent to threads that were doing the backend processing. Thus, separating the display of the information (model) and its internal representation in accordance with **the Model-View-Controller pattern**.
  + Added a Timestep counter that displays the number of timesteps since the beginning of the simulation.
  + Added a Reset button that zeroes the timestep count and the water depth across the entire landscape.
  + Added a Pause button that pauses the simulation run.
  + Added a Play button that lets the simulation run.
  + Added an End button that closes the window and exits the program.
  + Added a Mouse Listener to add 1 meter of water to the terrain on click.

FlowPanel:

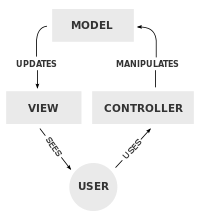
* Provided in Skeleton Code
  + Provides methods for manipulating the model and repainting the GUI.
* Modifications Made:
  + Edited the constructor to allow a Water object to be passed in by reference, as well as the beginning and end index of the section of the grid that the FlowPanel object would process.
  + Created the methods for the buttons to call when Resetting, Pausing, Playing and Ending the Simulation.
  + Edited the run() method to run the simulation for each threads respective section of the terrain, as well as synchronise the other threads so that the GUI and Timestep count are only updated once all threads have finished processing their section, for each iteration.

Terrain:

* Provided in Skeleton Code:
  + Stores information about and provides methods to help use and traverse the terrain.
* Modifications made:
  + Synchronized the getPermute() method to remove concurrency issues in the simulation.

Water:

* Added:
  + Stores information about and provides methods to help use, manipulate and traverse the water depths at each point in the terrain such as:
    - A method to zero water depth across the entire grid.
    - Methods to transfer 0.01 meters of water to the lowest neighbor.

Model-view-controller design (how code conforms)

The User interacts with the GUI created in the Flow class to add water, start, pause, reset and end the simulation. The grid displayed in the GUI is powered by only one FlowPanel Object.

Other FlowPanel objects that are invisible to the user (due to not being added to the GUI) are created, sent to threads and run in the background. During this process, the background FlowPanel objects continuously update and repaint the FlowPanel object that is responsible for the GUI but none of the processing.

The user observes these updates and use the controller to pause the simulation, add more water, and start it again. The model manipulation by the background threads can begin again.

The same Water and Terrain objects are shared by all FlowPanel objects so any manipulation by the processing FlowPanel objects is visible to the GUI generating FlowPanel object.

**Concurrency Discussion and Motivation (25)**

Concurrency features used, and why they were necessary

How wrote code to ensure:

* + Thread Safety and Synchronization (explain when you need to protect data and when not)
    - Thread Safety was needed for the accessor and modifier methods of the water class since multiple threads were reading and modifying it. Some of the methods were also quite long. There was, hence, a lot of opportunity for potential interleaving so many of the accessor and modifier methods in the Water class were synchronized.
    - Thread safety was **not needed** for most of the Terrain class since it stored information that never needed to be changed throughout the simulation. However, the getPermute() method in the Terrain class was not safe, since this method contained multiple steps and multiple threads were constantly calling it in the run loop at the same time. This resulted in a lot of interleaving that could be visibly seen during the simulation on most runs. Therefore this method was also synchronized.
    - Although the Swing Library in not thread safe, the Model-view-controller design pattern implemented which only allowed one FlowPanel object to control the GUI (with no model manipulation responsibilities) made it thread safe as it wasn’t directly involved in any Multithreading processors. Furthermore, due to some slick code, this GUI Flow Panel would only be called once all background threads had finished processing their section of the code. Therefore additional concurrency measures did not need to be implemented.
    - The static integer variables that would oscillate between 0 and 4 (the number of threads created) to help synchronize the background threads such that the GUI would only be updated once all the threads had finished processing their respective section of the code for each iteration, was not thread safe since interleaving during the accessing and modifying could result the program coming to a stand still. Therefore, they had to be made with the AtomicInteger class.
    - The static timestep counter had to be made Atomic as well since it was also incremented by multiple threads.
  + Liveness and No Deadlock:
    - There was no need to use methods like suspend() which runs the risk of causing a deadlock.
    - When writing the code to ensure that no painting would be done until all 4 background threads had finished processing their section, a deadlock problem was removed by using a second while-loop-lock. One while-loop-lock before repainting, and one while-loop-lock after repainting to ensure that all threads got a chance to progress before any other threads made it to the next iteration, which would have resulted in a deadlock.
    - Therefore, the liveness conditions that require a system to make progress and for some threads not to be starved were also met, since a thread could not be left suspended. The loops would hold until every thread has done its processing before resetting the loop-condition and there are no locks that wouldn’t be available to a thread once the other threads had been processed that would halt its processing.
    - In the simulation, each thread needed multiple locks, however they had to be obtained in the same order. Therefore, the philosophers deadlock or Livelock would not occur.
* Validation (incl. race conditions)

**Conclusions (5)**