Assignment 2 - WaterFlow Simulator

Abstract

In this assignment, you will design a multithreaded Java program, ensuring both thread safety and sufficient concurrency for it to function well. This builds on the problem presented in the parallel computing solution from the first assignment.

Introduction

For this assignment I implement a multithreaded water flow simulator that shows how water on a terrain flows downhill, accumulating in basins and flowing off the edge of the terrain.

In order to complete this assignment I first added the GUI components JButtons, to the skeleton code provided to us. From there I then created a new Layer ontop of the greyscale terrain image which allowed me to paint water ‘drops’ onto the terrain image. As noted in the prac outline, I hard coded the depth of the water unit to correspond to a depth of 0.01m This then followed by creating a water class to perform the surface height calculations and lead to me being able to move water to it’s neightbouring points if they were lower. By using the genPermute() method in the terrain class, the neighbouring points were assessed in a random order to prevent the ‘transportation’ of water. From here I now had a sequential version of the simulation onto which I applied four threads, and used various methods discussed later to ensure proper concurrency, but this produced a much faster version of iterating through the grid and allowed water to flow faster. A timeStep counter was also added to view how many times the grid had been traversed.

*This is extremely simplified as this assignment took hours to understand and days to code out, but I am happy with the end result.

Correctness

In order to check that the simulation was running correctly, during the stage where my program was running sequentially, I checked that the amount of water added by mouse clicks before the play button was pressed corresponded to the amount of water that had left the grid + the amount of water that was still on the grid. This was easy in sequential form, and resulted in the same answer. When I had finally added four threads to my program, I now had to check this was the same in order to ensure there was no water loss due to threads no synchronising correctly. By applying the same calculation on each thread, and ensuring correct synchronisation and wait() + join() methods were called, I can safely say that no water is lost during the simulation and thus am confident that the simulation is correct.

*Note that the calculations have been removed in the simulation now to ensure faster performance.

Classes and Modifications

Terrain.java:

There were no modifications made to the Terrian.java file given. I did copy the code from here to be the basis of my water class, as most of the methods were either the same or very similar.

Flow.java:

This class was used to create the individual threads, and then join them for synchronisation and run the FlowPanel class. I created all the necessary GUI Buttons and Labels to show the time step counter, and used a boolean to control when the simulation plays or is paused on the press of the buttons. A mouse listener is used to implement the adding of water on clicks.

FlowPanel.java:

This class creates the terrain image in greyscale and then creates the buffered image on top for the water to be displayed. The run method repaints the canvas and the paintComponent is synchronised.

Water.java:

In order to correctly implement mutual exclusion of water data this class was created. The water points are stored in a 2D array. It returns the buffered Image of the water when needed and has a method for coping the water Array when calculations are being performed (this is the safe way to do it during concurrency). Methods for adding and removing water to the array are coded here and called from Flow when needed. This simply accesses the array. The moveUnit() method moves water from one point to the next when told to do so from the FlowThread class. The final method in this class ensure that water that has left the canvas changes colour.

FlowThread.java:

This class handles the calculations performed by the threads. The method for finding the lowest neighbouring point for water is here, where it takes in a point and the water point and checks if the surface height of the water is above the point to determine if the water will flow. It returns the x,y coordinates of where the water should move to. In the run method which is synchronised, the water array is copied and then getPermute is called to randomise the order of searching and once complete, another synchronised water call is made to move the water unit. The canvas is then updated by calling deriveImage method.

Model-View-Controller pattern.

Good programming practice is to use the MVC pattern when programming, especially when GUI's are involved. As a result in this assignment I have impelmented this by creating 3 more model classes since the Controller and the View classes where provided.

In my assignment the following is true:

Flow.java is the controller class
FlowPannel.java is the view class
Terrain, Water and FlowThread are all models of data in the package.

We know the the model must only contain pure application data, which is why my classes only have the methods for calculations and moving of water data and thread implementation in them.

The view class presents the models data to the user, and allows access to the model's data, all of which is handled by FlowPannel. Here, the Images are created and passed to Flow.

The controller exists between view and model. The flow class listens to events triggered by the user in the View class and then executers the methods in the Models class. This then automatically updates the FlowPannel class which is again the view class.

Design and Implementation

UI Controls

- JLabel to Show Text = "Time Step counter" and the actual count number
- JButtons for Reset, Pause, Play and End

  Each button has a listener associated with it to allow for Swing to determine what code runs when the button is pressed. This is mainly handled by setiting the play boolean to ether true or false.
- JPanel to group the Buttons in one panel and another for the Label.
- A mouseAdapter (Specialised Listener) was added to get the coordinates of each mouse press to add water to the terrain.

Mutual exclusion on water.

This was achieved by creating a seperate water object with propteries shown above. This class handled all functions related to water and by doing this, creates a safer way to handle data during concurrent programming. The exclusion of this was fathered by creating a separate FlowThread class to ensure that all thread operations were also excluded from the main functions of the simulation and was again safer programming. Finally by forcing each thread to wait for the other threads to finish but including the join() try/catch block in the Flow.java class, we ensured that there was no conflicting operations and each lock had been released before the next iteration.

Simulation threads

By creating 4 individual threads in the Flow.java which were an instant and handled by FlowThread.java I was able to implement threading in this assignment. Each thread was past 1/4 of the terrain grid, which is would traverse independently, allowing for the flow of water to be faster. By excluding the calculations into FlowThread, we are able to handle the Threads easier. Each thread is created for every traverse and waits for the other threads to finish before the next iteration can take place.

Concurrency and Motivation

Deadlock

In order to avoid deadlock I never called a function outside of the module when I didn't know whether they will call back into the module without reestablishing invariants and dropping all module locks. What this means is that all calls from the threads were made to a synchronised method, were the calculations were performed and methods executed, the threads were then joined and only then was the next iteration undertaken.

By making a copy of the WaterArray for each thread, they were never handling the same dataset which prevents locks being held on data needed, and as well as each only handling a different part of the array, deadlock was prevented.

Race conditions

To prevent any race conditions from occurring the following methods were synchronised: **setupGUI (Flow.java)**, **painComponent(FlowPanel.java) run(FlowPanel.java), findLowest(FlowThread.java)**. On top of this in FlowThread.java run method the copying of the water array was Synchronised as was the moveUnit method by encapsulating them in a synchronised method call. By synchronising all methods that handled the water data and repainted the canvas, I was able to ensure that proper synchronisation was implement and no race conditions could occur.

Synchronisation

As stated above in the race conditions section, multiple methods were synchronised to prevent race conditions. By using the Java reserved keyword “synchronised”. This means all all methods with this call made can only have one thread executing in them at a time, and why none of the traversing methods use this as it would defeat the point of having multiple threads, but where not needed such as repainting the canvas or handling the water array, is safer and prevented both deadlock and race conditions.

Atomic variables - I used none for this assignment as I feel that they are quite unpredictable and there are better ways to ensure safety in concurrency.

It is important to mention here that the getPermute() method, by randomising the order of which water neighbours was checked, prevent water from teleporting (making big jumps across the terrain)

Conclusion

In conclusion there were several ways in which to safely implement the threading of this simulation, but I believe I have achieved the desired outcome, with some fairly simple implementation. The simulation executes as requested and follows all the assignment guidelines and looks impressive when run. There is no water loss during the simulation and water flows smoothly. I have used various implementation of thread safety as discussed above, and have used metal exclusion on water data. The outcome is a fully functioning simulation that I am proud of. Included in this submission along with this report is all the necessary code and files.