Concurrent Programming: River Flow Simulation

MGWGIF001

Table of Contents

1 Introduction

1.1 Background

In this assignment, whilst building on the parallel computing solution from the first assignment, it was tasked to design a multithreaded Java program which ensured both thread safety and sufficient concurrency for it to function well. For this to be achieved, it was required that we 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 as shown in Figure 1.

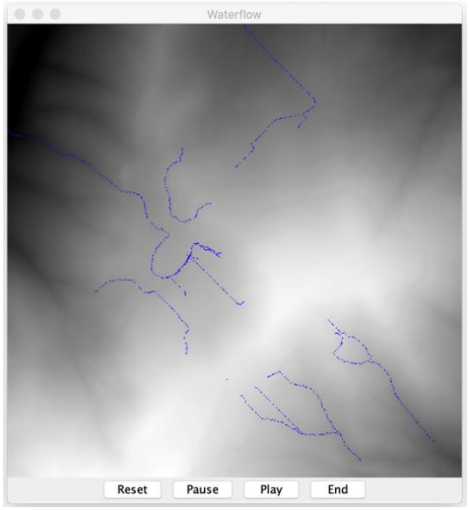

Figure 1: Sample screenshot of expected program display

1.2 Aim

For this assignment, skeleton code was provided for the assignment. When this skeleton code was executed, this skeleton provided an incomplete it displays the underlying terrain but did not include any simulation or display of the underlying water and the buttons shown in Figure 1 excluding the ‘end’ button. As such, the aim of the assignment was to modify the skeleton code to add the following features.

- A main display window that shows the landscape as a greyscale image, with black representing the lowest elevation and white the highest. Overlaid on this should be an image representing the locations of water in blue
- A counter that displays the number of timesteps since the start of the simulation
- A ‘reset’ button that zeroes both the water depth across the entire landscape and the timestep count
- A ‘pause’ button that temporarily stops the simulation.
- A ‘play’ button that either starts the simulation or allows it to continue running if it was previously paused
- An ‘end’ button that closes the window and exits the program
- Mouse input that allows the user to click on the display to add a square of water to the simulation at the corresponding position on the terrain, irrespective of whether the simulation is currently running or not
- A simulation responsible moving water over the terrain that is carried out by 4 threads, each responsible for a portion of a permuted list of grid positions

2 Methods

2.1 Code Architecture

For this assignment, the Model-View-Controller pattern was used which is represented in Figure 2.

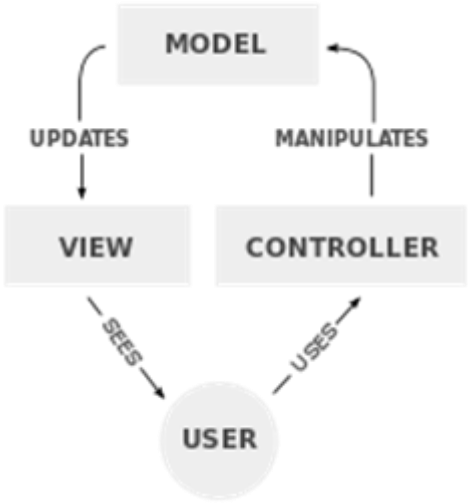

Figure 2: Model-View-Controller showing clear separation between the display of information and its internal representation

In keeping with this pattern, the existing classes in the skeleton code and extra classes were added and organized into their respective categories as shown in Table 1 which allocates each class to is relevant section.

Table 1: Classes created and their respective responsibilities and class category

Class Category	Class	Description
Model	Terrain.java	Class that stores information about underlying mountain terrain heights for each position in a 2-dimensional array of type float.
	Water.java	Class that stores information about the water depth at each position for the created water in a 2-dimensional array of type int.
View	Flow.java	Class responsible for providing the GUI interface which the user will be interacting with and adding all the relevant event listeners such a mouse and button clicks. Also contains the main method.

Controller	FlowPanel.java	This is the main controller class responsible mainly for refreshing the GUI view after a model manipulation and the creation of the simulation threads that will directly carry out the moving water simulation. It is also the point of entry for the View class.
	FlowController.java	This class extends the Thread class and is responsible for moving the water in a certain section of the permuted list of grid positions. For this assignment, four thread instances of this class would be created at runtime.
	WaterPainter.java	This class extends the RecursiveAction class of the ForkJoin Framework and is responsible for drawing the blue water overlays across the underlying terrain and was chosen to apply a Mapping algorithm to allow for more efficient drawing as the terrain dimension increases

In keeping with the above, the classes were implemented with multiple methods created. To get future insight on each method implemented for each class, please consult the relevant Java docs.

2.2 Simulation Approach

As mentioned above, there are 2 model classes storing the water depth and terrain heights respectively. Of note, it should be noted that although the water depth is stored as an integer, each water unit corresponds to a depth of 0.01m. As such, a value of 5 corresponds to a depth of 0.05m. With the following tasks detailed in () would be carried out by the respective for a single simulation timestep.

Table 2: Simulation Tasks and Class Responsible

Task	Reason	Class Responsible	Implementation in Class
Water is cleared from the boundary (x=0, y=0, x=dimx-1, and y=dimy-1) by setting values there to zero.	This represents water flowing off the edge of the terrain	FlowController.java	the method moveWater () checks if a point meets this condition and sets it to 0
All grid positions not on the boundary are traversed in a permuted order (see the getPermute () method in the Terrain class).	This helps to reduce unevenness in the speed with which water flows across the terrain.	FlowController.java	Upon starting the program, the genPermute () method was used to fill up the permute variable in the terrain instance. The list is then subdivided into 4 sections and each FlowController thread monitors a portion of it
For each grid position (x, y) the water surface ($s_{x,y}$) is calculated by adding water depth ($w_{x,y}$) to terrain elevation ($h_{x,y}$). Thus, $s_{x,y} = w_{x,y} + h_{x,y}$. The current water surface at (x, y) is compared to the water surface of the neighboring grid positions.	A single unit of water is transferred to the lowest neighbors, so long as the water surface of this neighbor is strictly lower than that of the current grid position. Otherwise no water is transferred out of the current grid position.	FlowController.java	The moveWater () method takes uses this reasoning to determine the lowest neighbor and then moves water from the current point to the neighbor if need be. Of note, to simplify the simulation, the program assumes 1 lowest neighbor at most

2.3 Concurrency Design

2.3.1 Thread Safety

On the notion of thread safety, the classes to which such care are below

Table 3: Classes to be aware of thread safety

Class	Cause of Concern
Terrain.java	This information is shared amongst multiple FlowController threads
Water.java	This information is shared amongst multiple FlowController threads
Swing Library	The library is not thread safe and hence care must be taken when using it

Of the above classes, the Terrain class contain information that remained unchanged for the entirety of the simulation run. As such, there was no reason to make it thread safe as there we never any dangers associated with it. As for the swing library, the FlowPanel and WaterPainter classes that interacted with it actively carried out mutually exclusive tasks that would not interfere with each other hence there was no need to add any protection.

Finally, for the Water class, data safety was required as the multiple threads reading and modifying it needed to ensure that they would not conflict with each other. As a result, all the data access and modifier methods were made to be synchronized. As an aside, initially, I made the Water class volatile for the FlowController to ensure that it was always up to date, however, the performance loss from this was too great and as such, I decided to move remove this attribute and allow caching to occur which could cause possible interleaving errors but not so drastic as to affect the simulation results as a whole.

2.3.2 Thread Synchronization

For the synchronization of the FlowController threads to be carried out, the code was written such that for one timestep, the 4 threads are created, started and waited for them to complete by means of using the join() method and only then would the step complete and the program move on to the next timestep.

2.3.3 Liveness and Deadlock

With regards to liveness, the FlowController's work was divided evenly and the water synchronized methods were made to be very small and not intensive. As such, the program has a high probability of achieving liveness. With regards to deadlock, the only methods themselves in the Water class could always run to completion and there was no use of dangerous methods like suspend () as such, deadlock is unlikely to be encountered.

3 Results and Discussion

3.1 Simulation of Medium Terrain Size (medsample_in.txt)

For the medium sized terrain, the simulation was with multiple permutations of play, pause and reset, aside from this, blocks of water were also at the boundaries and close to known basins to observe if the expected behavior was obtained. Due to the program, having mostly applied the concurrency safeties to the Water class, it was observed that water conservation was preserved, and no race conditions seemed apparent from the simulation tests. As an aside, the simulation was noted to be relatively quick.

3.2 Simulation of Large Terrain Size(largesample_in.txt)

For the large Terrain size, similar testing was carried out as in the medium sample terrain and similar results were obtained. The only difference between the two sets was the noticeably slower step time as compared to the medium sample terrain. This was attributed to how the simulation rendering was only carried out by 4 threads and as such had an O(n) performance as the terrain grew. An effective solution could have been an implementation of a reduce algorithm using a ForkJoin Framework however, since this broke the requirement of using 4 threads for simulation, I decided against it in the end and hence accepted the lower performance.

4 Conclusions

4.1 A Reduce Algorithm is preferred for water flow simulation

As mentioned above, it was noted that that use of 4 simulation threads though it allowed for easier thread synchronization, came at a cost of worse scalability as the terrain size increased in terms of simulation runtime. As such, a more appropriate approach is believed to be the use of multiple threads that handle smaller sections by means of using a Reduce Algorithm and the ForkJoin Framework.

4.2 Minor Errors are acceptable within simulation for improved performance

As noted above, though the use of the volatile attribute guaranteed no staleness in the Water class, the resulting performance loss led to it being less than ideal. Hence a class with minor staleness was seen to be more idea as the performance boost from the memory hierarchy meant the issue would quickly be resolved.