Jesse Sarembock
SRMJES001
CSC2002S Assignment 2 Report

Introduction

This assignment required the design of a program to simulate the flowing of water across a terrain, using multiple threads in a thread-safe, yet efficient way. In this report, I will briefly discuss how I went about doing this, with regards to the design implementation and the use of java concurrency features.

Design and Implementation

My program consists of six classes in total: the three that we were provided with in the skeleton, and a further three I created. Starting with the classes I created myself from scratch, there is the Water.java class. This is a simple class to represent water on the terrain. It consists of a single field that is a 2D float array. It has a standard constructor, and the idea is that an object of type Water is created of the same size as the terrain, and each element in the array represents the current height of the water at that point. It also has various methods for manipulating or reading from the array of water values. All of these methods are static so that other classes can easily access them.

The next class I created is the WaterClickListener.java class, which extends MouseAdapter. This is a simple class with not much code, that tells the program what to do when the mouse is clicked on the terrain. In the event that the mouse is clicked on the terrain, the array of water heights is adjusted to represent that a water height of 0.03m is now present at the location where the mouse was clicked, as well as surrounding grid points, so that there is essentially a 3x3 block of water where the mouse was clicked, with each grid point having a 0.03m water height. The corresponding pixels are also changed to blue in this event.

The final class that I created is the Control.java class, which implements the Runnable interface. This class was created last, as it only became necessary to have when I implemented multiple threads instead of just one. The idea is that there is a constructor that takes an ArrayList<Integer> as parameter, and when each of the four threads is called in the Flow.java class, it is passed an instance of this class with an ArrayList a quarter the size of the original permute ArrayList as parameter. This is how I split the problem among four equal threads. The class has a run() method as it has to, which

continuously calls the runWaterFlow() method, which basically is what makes the water ‘flow’, while a Boolean flag is equal to true. This method also increments the timestamp.

In terms of modifications to existing classes, the class that saw the most alteration was the Terrain.java class. I added various static fields, such as a 2D array called waterSurface, which stores the current surface height at each grid point. ie, the terrain height plus the water height at a given gridpoint. Other static fields include a buffered image to display water on the terrain, instances of colours to paint pixels, and four Arraylists, each containing a quarter of the original permuted Arraylist.
Some of the fields and the methods that were provided from the start were changed to static to support my implementation.

Most of the methods that I wrote were done in the Terrain class. These include a getWaterImage() method which returns the buffered image which represents water on the terrain, as well as a deriveWaterImage() method, which works similarly to the deriveImage() method but is for the buffered image displaying water.
I also wrote a splitPermute() method which simply splits the large permuted Arraylist into 4. The splitPermute() method and deriveWaterImage() are both called in the readData method.

The most important method I wrote was also done in the Terrain class. This is the static void waterFlow(ArrayList<Integer> permute) method, which is essentially responsible for the flow of water over the terrain. It takes as parameter an Arraylist, so that the different threads can all use it with different permuted lists as parameters. What it does is, for every gridpoint, if the gridpoint is on the perimeter, it clears water there, by both adjusting the water array accordingly, and painting the corresponding pixel as transparent. For every gridpoint that is not on the perimeter, it checks its neighbours one by one to see if it is lower than itself and lower than all its other neighbours. If it is, water is added to that neighbour and subtracted from the current gridpoint. Finally, after this, if the current gridpoint has no water, its corresponding pixel is painted transparent.

The FlowPanel.java class remained mostly the same. The only changes are that I use the drawImage method in the paintComponent method to paint the water buffered image. This is done underneath the drawImage call to paint the greyscale buffered image. I then call repaint() at the end of the paintComponent() method. Finally, I altered the class so as to no longer implement Runnable, and removed the run method as it no longer served a purpose after I created the Control.java class.

Finally, the Flow.java class also remained much the same. I only added buttons and actionListeners for those buttons, as well as a MouseListener. I will briefly explain what I coded in each actionPerformed method.

For the Reset button, in the ActionPerformed method, I have a double for-loop that iterates through the entire terrain and sets the values in the water array to 0 and paints every pixel as transparent. I also set the timestamp to 0.

For the Pause button, I simply set the value of the Boolean flag to false.

For the Play button, I set the value of the Boolean flag to true and instantiated the 4 threads, each taking in an instance of the Control class as parameter, and each thread's instance of Control taking a different sub-permuted list as parameter.

For the End button, I set the value of the Boolean flag to false and dispose of the frame. One last alteration was to add the JLabel which displays the timestamp.

Model-View-Controller Explanation

My design conforms to the Model-View-Controller pattern in the following way: I have a class Water.java and the Terrain.java class which both constitute the Model part of the architecture. Those are the classes that are responsible for the data and for updating and manipulating the data as necessary. The View part of the architecture consists of the Flow.java class, which is responsible for the GUI, and the FlowPanel.java class, which is responsible for continuously repainting the view. The Model, ie the Terrain and Water classes, essentially update the view which is repainted when a change is made to the Model.
Finally, the Controller part of the architecture consists of the Threads which are initialized in the Flow.java class, and the Control.java class. The Control.java class manipulates the Model by constantly calling the Model's methods in a while loop. The User uses the Controller classes, only actually running the Flow.java class, and does not interact directly with the View or Model classes.

Concurrency Discussion and Motivation

There were not many places I identified in my code where concurrency features would be of any use, seeing as I did not create many methods which alter the data that the threads access. There were however a few places where I used locks, one place where I used an atomic variable and one place where I used a volatile Boolean flag.

In my Water class, I did not make the getWater() method synchronized, as this is only a read method. The placeWater method is a write method but I did not make this synchronized as this is only ever used by the main thread, when the mouse is clicked, thus there is no chance of multiple threads using this method simultaneously.
All the other methods in the Water class were made synchronized, as they write to memory and are accessed by multiple threads at once. This is how I achieved “fluid conservation” by not allowing any extra water to be added, subtracted, or cleared that shouldn’t be. The addWater and subtractWater methods are used a lot in the critical section of my code (the waterFlow method in the Terrain class), so synchronizing them does affect performance, but it is hardly noticeably slower and is a fair trade off for fluid conservation.

In the terrain class, I made the getPermute method synchronized, as well as the line of code in the waterFlow method which alters the waterSurface 2D array. This is to prevent the race condition that the incorrect index position might be calculated, or the incorrect index position in waterSurface being calculated, respectively.

In my Control class, I used an AtomicInteger type for the timestep counter instead of a regular int, to avoid the race condition where the timestep is incremented too many times by bad interleavings of threads.
I also used a volatile Boolean flag to avoid the situation where one of them is given a stale state of the flag.

In terms of synchronizing the threads on each timestep, I did this in my Flow.java class using the join() method. Each of the 4 threads I created are constantly running on sub-permuted lists and are thus always “doing something”, i.e. they are always active and “live”, apart from when the simulation is paused.

Conclusion

In conclusion, I first coded everything without the use of any concurrency features and could see in the simulation after adding concurrency features, that despite the simulation running slightly slower, the water seems to flow in a more consistent fashion, with, for example, less water getting stuck in one place.