Ahmed Khan – KHNAHM006

CSC2002S Assignment 2 report

In this report you will find an overview of all methods and variables I created for each class in my program, followed by an explanation of how I validated my program, and then a description of my Model-Controller-View pattern as well as any extra features I have added to the base program.

An explanation and justification of each of my concurrency features within each class, has been included within the description of each respective class.

I have also included a section at the end speaking about various bugs my program is likely to encounter.

Classes

Water

This is the only additional class that I created. It is used to define and manipulate the water object as specified by the assignment brief.

Variables

private AtomicInteger depth;

Constructors

public **Water**()

// Initialises depth to 0

Methods

public float **getDepth**()

// Returns the depth in a float format

synchronized public void **giveWater**(Water other, Terrain land, int x, int y, int sx, int sy)

// Takes one unit of water from current water object and gives it to the second object

public void **setDepth**(int d)

// Adds d units of water to the current depth

public void **incDepth**(Terrain land, int x, int y)

// Increments depth and updates the gui

public void **depWater**(Terrain land, int x, int y)

// Decrements depth and updates the gui

public void **clearWate**r()

// Sets depth to 0

Concurrency

1. **This class stores its depth variable within an atomic integer.**
    - This is necessary in order to ensure mutual exclusion when these depth variables are being incremented or decremented.
    - This is important especially when water is added to the grid in a position where water may already be flowing, in order to ensure that the new water is safely added, without the outflowing water to interfere with it.
2. **The giveWater() method has been synchronized**.
    - This is important as giveWater() calls both the incDepth() method as well as the depWater() method in order to take one unit of water from its own depth and give it to another Water object. Without the synchronization, there is an opportunity for another thread to butt in between these two calls, and possibly interfere with the way in which water is spread.
    - Therefore, this synchronization is necessary in order to prevent bad interleaving.

Terrain

There are a range of variables which I added to the terrain object in order to allow it to operate closely with the Water object.

Variables

```
int Count;

AtomicInteger ThreadCount;

Water[][] wtrGrid;

BufferedImage waterImg;

volatile boolean running;
```

Concurrency

1. **The Count variable counts the current timestep of the simulation.**
    - There was the possibility that this variable could be made Atomic in order to ensure that the timestep is accurately counted. However, this was not necessary, as my code was written so that a thread would only increment this count if it knew that all other threads were asleep. Therefore, data races in this variable would be impossible as only one thread is active to increment it. The thread would then wake up all other threads in order to start the next timestep
2. **ThreadCount is a counter which is incremented by each thread once they have completed their work for the current timestep**.
    - This variable was made atomic atomic in order to prevent data races.
    - This is vitally important as a data race in this variable could cause the threads to call the wait() function unnecessarily, and this would greatly increase risk of deadlock
3. **The running variable is declared volatile**

- This was completely necessary in order to stop the main While loop from going to sleep at the end of a timestep.

Methods:

public long **getCount**()

// Returns value of Count variable

public void **incCount**()

// Increments value of Count variable

public int **getThreadCount**()

// Returns value of threadCount variable

public void **incThreadCount**()

// Increments value of threadCount variable

public boolean **getRunning**()

// Returns the value of Running variable

public void **setRunning**(boolean run)

// Sets value of Running variable to run

public void **deriveWaterImage**()

// Initialises a transparent image to be used for water image overlay

public BufferedImage **getWater**()

// Returns water image for overlay

public void **spreadWater**(int fx, fy, sx, sy)

// Spreads water from a first set of xy co-ordinates to a second

public void **clearX**()

// Removes water form upper and lower grid boundaries

public void **clearY**()

// Removes water from left and right grid boundaries

public void **resetWtr**

// Clears grid of water and sets all water depths to 0

public void **placeWater**(int x, int y)

// Places 3 units of water in every position within a 3x3 block surrounding an xy co-ordinate

public void **updateWater**(int x, int y)

// Updates gui to show the movement of water units

public void **removeWater**(int x, int y)

// Updates gui to show the removal of water from a set of xy co-ordinates

FlowPanel

There are a few methods that I have added to the FlowPanel class in order to receive input from the GUI as well as handle multithreading across the terrain grid

Variables

int[] loc;

int lo;

int hi;

Object locker;

Constructor

FlowPanel(Terrain terrain , int low, int high, Object key)

// Constructs an instance of the FlowPanel in order to do multithreading

Methods

public void **reset**()

// Is called by GUI when reset button is clicked

public void **play**()

// Called by GUI when play button is clicked

public void **pause**()

// Called by GUI to pause the program when pause button is clicked

public void **wtrClick**()

// Called by GUI on mouse-click to place water

public void **checkThreads**()

// This method is used to synchronize the four running threads at the end of each timestep

public void **run**()

// Executed the water flow logic

Concurrency

1. **An Object Locker has been declared as a lock to be used in co-ordinating the 4 threads.**
2. **The locker object as described above was used for synchronization within the checkThreads() method**.

- This synchronization was vitally important in order to ensure that the program did not run into a deadlock. The checkThreads() method calls the wait() method in order to allow each thread to pause at the end of each timestep, this means that synchronization was compulsory.

3. **I used the locker object again within the run() method when allowing water to flow.**
    - This synchronization was absolutely necessary as the program was heavily prone to race conditions without it. By tracking the count of water units in the grid, I was able to observe that without this synchronization step, water was being unnecessarily deleted from the grid due to bad interleaving, and if allowed to run indefinitely, the number of water units would eventually become a negative number.

Flow

Within the Flow class, the setupGUI method was altered in order to add the various JButtons and JLabels as specified by the assignment instructions. Additionally, I placed a Timer inside the GUI in order to refresh the frame every 33 milliseconds (30 fps).

How I validated my system

There were four ways that I went about validating my program:

1. Deadlock check:
    - This was simple, the only portion of my code that threatened to give a deadlock is when a thread is called to wait() at the end of each timestep. I knew the program was valid once the deadlock stopped occurring
2. Check for race conditions:
    - I added a small piece of code into the Water class which would notify me if a Water object with a depth of 0, was trying to give water to another class despite not having any water to give. This allowed me to outline sections of code where bad interleaving caused a Water object to lose its water, before it had finished its operation of giving away that water.
3. Fluid conservation:
    - I added a loop in the FlowPanel class that would count the total number of water units on the grid at the end of each timestep. Through this I was able to verify that water was never being created or destroyed on the Terrain, except through mouse-click and boundary runoff. The results from a few of these tests have been included in a data file included with my program. These files simply include a list of numbers: the total number of water units on the grid at each timestep. Looking at the files, it is clear from the fact that the water count never increases, that water is never being duplicated. It can also be assumed from the fact that the water count stops decreasing when all water has either passed the boundary or accumulated in basins, that water is not being unnecessarily deleted. Thus, fluid conservation is intact.
4. Visual comparison
    - This is by far the weakest form of validation I used, but in order to make sure that my program was running consistently, I took pictures and videos and compared the behaviour I was seeing onscreen to that of previous running's of my simulation.
    - I also compared my own program behaviour to the one or two clips of the simulation that were shared by other students in order to ensure that my program was functioning similarly to that of other students.

Model-View-Controller pattern

My program takes the Model-View-Controller pattern as follows:

- View: The GUI generated by a single instance of the FlowPanel class (This specific FlowPanel is used to set up the GUI but its run() method is never called)
- Controller: The controller is made up of the 4 new FlowPanel threads which are run simultaneously
- Model: The model is made up jointly of the classes Water, and Terrain, which are then visually linked to together by the FlowPanel.

The **view** provides the user with access to the **controller** through buttons on the GUI. The **controller** runs the calculations needed to manipulate water flow, and then updates the **model** through the classes Water and Terrain. The Water class will then call a method in order to visually change the display on **view** which can then be seen by the user. In this way, my program conforms to the Model-View-Controller pattern.

Extra Features

1. Water Colour
    - The only extra feature I added was the usage of extra shades of blue to display water depth.
    - A water depth of 1 unit will display as light blue
    - A depth of 2 units will display in a shade of medium blue
    - And a depth of 3 or more will display in dark blue.

Bugs

1. Visual glitches
    - My program encounters one or two visual glitches within certain specific pixels on the terrain. I've noticed it in 2 or 3 different places. What happens is that a single pixel of water kind of flickers from side to side, as if a single unit of water is being passed back and forth between two objects. I was not able to determine the cause of this glitch and do not know how to fix it.
2. Slowness
    - My water simulation runs incredibly slow, at least in the virtual machine. It is likely that I was over paranoid with my usage of synchronization and as a result sacrificed far too much processing speed.

Conclusion

To summarise this report, I have created the water flow simulator as described by the assignment outline, and it conforms to the Model-Controller-View pattern. My simulator functions as intended and is able to uphold fluid conservation. My program utilises concurrency techniques in order to ensure thread safety however this came at the cost of speed. There were a few minor glitches which I was not able to sort out.