CSC2002S
Assignment 2
MLLDYL002

Introduction

The focus of this assignment is on concurrent programming. As per the assignment leaflet, we were tasked with designing a multithreaded/thread-safe Java program (a water flow simulator) implementing concurrency.

More specifically, given an incomplete Java implementation of a water flow simulator, a GUI interface and water simulation engine (utilizing 4 main threads) was required to be developed, ensuring synchronization across threads per timestep.

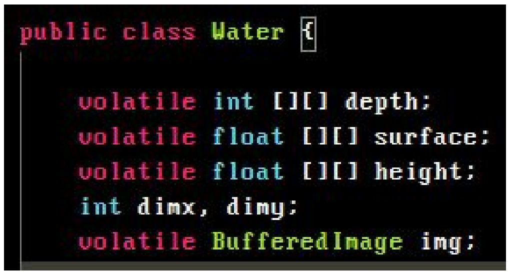
Additionally, the water depth computation, should utilize the list of permuted grid positions across threads and computations should be mutually exclusive across threads and not interdependent on each other in any other way.

Given the incomplete Java implementation a series of classes were developed as well as modifications to existing classes. These will now be detailed.

Water Simulator Class: Water.java

A Java class (Water.java) was created to handle the core simulator logic for the water flow simulation of the assignment.

Initial class variables to hold the water depth, water surface, water height, terrain dimensions as well as a BufferedImage class (for the water image overlay) are shown in **Fig.1**



```
public class Water {

    volatile int [][] depth;
    volatile float [][] surface;
    volatile float [][] height;
    int dimx, dimy;
    volatile BufferedImage img;
```

Fig.1 - Initial member variables (Water.java)

Following this, a series of member functions were written to handle:

- Class initialization

- Boundary Check
- Clearing of water boundary
- Adding water patches given (x,y)
- Clearing water
- Drawing water
- Simulation steps (water flow algorithm)

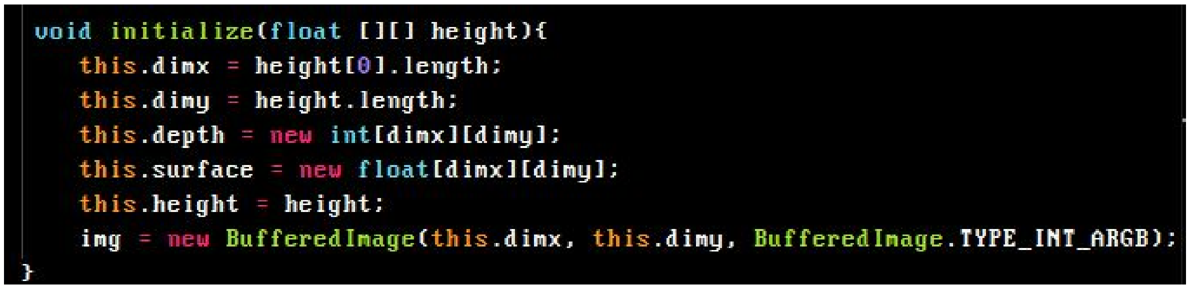
These member functions will now be detailed individually.

Class Initialization : initialize()

Fig.2 shows the code listing for the initialize function prototype. The initialize function takes as input, a single two dimensional float array corresponding to the terrain height values which is then copied to a similar internal array (called height) which is used for reference during computation of the simulation step.

Additionally, the x and y dimensions of the array are computed and stored in the dimx and dimy width variables.

Finally, the depth and surface multidimensional arrays are initialized and well as the BufferedImage object that is used to display the blue water pixels per grid unit

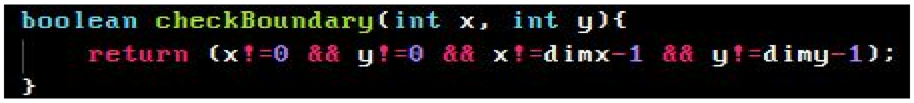


```
void initialize(float [][] height){
    this.dimx = height[0].length;
    this.dimy = height.length;
    this.depth = new int[dimx][dimy];
    this.surface = new float[dimx][dimy];
    this.height = height;
    img = new BufferedImage(this.dimx, this.dimy, BufferedImage.TYPE_INT_ARGB);
}
```

Fig.2 - Class initialization (Water.java)

Boundary Check : checkBoundary()

Fig.2.1 shows the code listing for the boundary check class method, this method simply takes as input a pair of grid coordinates and checks if they reside within the grid boundary.



```
boolean checkBoundary(int x, int y){
    return (x!=0 && y!=0 && x!=dimx-1 && y!=dimy-1);
}
```

Fig.2.1 - Boundary check (Water.java)

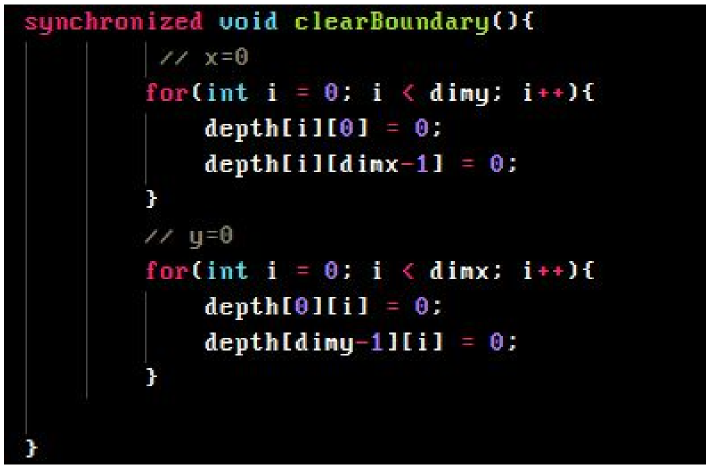
Here it is not necessary to define the class method as synchronized since the class method is only called directly through step() - the core simulation function, which is already synchronized.

Clearing of water boundary : clearBoundary()

The clearBoundary() member function is used to clear the water around the terrain edges , i.e for (x=0 & y=0) and is called during the initial stages of the water simulation step.

The method is declared with the synchronized keyword which is used to ensure that only a single thread at a time may call the function.

The function consists of two for loops that simply zero out elements within the terrain grid border, i.e for x = 0 and y = 0



```java
synchronized void clearBoundary(){
        // x=0
        for(int i = 0; i < dimy; i++){
            depth[i][0] = 0;
            depth[i][dimx-1] = 0;
        }
        // y=0
        for(int i = 0; i < dimx; i++){
            depth[0][i] = 0;
            depth[dimy-1][i] = 0;
        }

}
```

Fig.3 - Clear water boundary (Water.java)

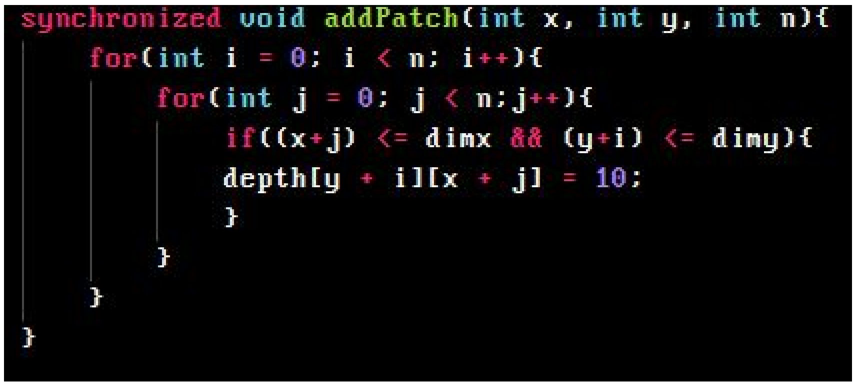
Adding Water Patches : addPatch()

The addPatch() member function is shown in **Fig.4**.

It takes as input three integers, the first describing the x position of the patch of water in grid units, the second describing the corresponding y position and a third parameter describing the rectangular dimension of the patch of water to add (in grid units).

It consists of a double for loop, where for each iteration, the dimensions are compared and the corresponding water patch position is checked if it is within the terrain grid bounds.

If the value is found to be within bounds, a value of 10 is assigned to the water depth for the particular water unit. Again this member function is declared as synchronized to ensure that only a single thread at a time may add a patch of water to the terrain grid.



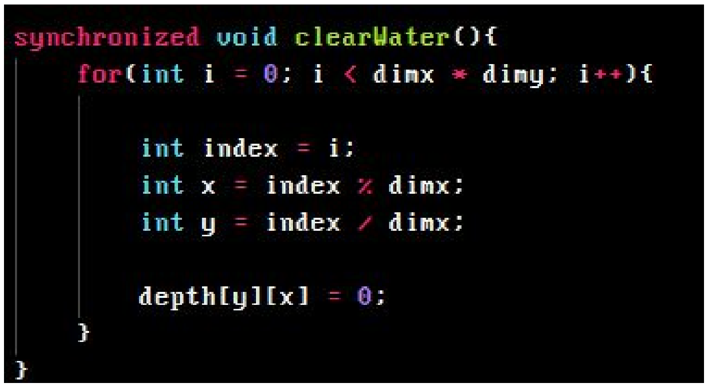
```
synchronized void addPatch(int x, int y, int n){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n;j++){
            if((x+j) <= dimx && (y+i) <= dimy){
            depth[y + i][x + j] = 10;
            }
        }
    }
}
```

Fig.4 - Class method to add water patch (Water.java)

Clearing Water : clearWater()

The clearWater() member function is used to clear all water on the global water grid, it simply consists of a for loop where the water depth of each point is set to zero.

Fig.5 shows the code listing for the clearWater() member function.



```
synchronized void clearWater(){
    for(int i = 0; i < dimx * dimy; i++){

        int index = i;
        int x = index % dimx;
        int y = index / dimx;

        depth[y][x] = 0;
    }
}
```

Fig.5 - Class method to remove/reset water (Water.java)

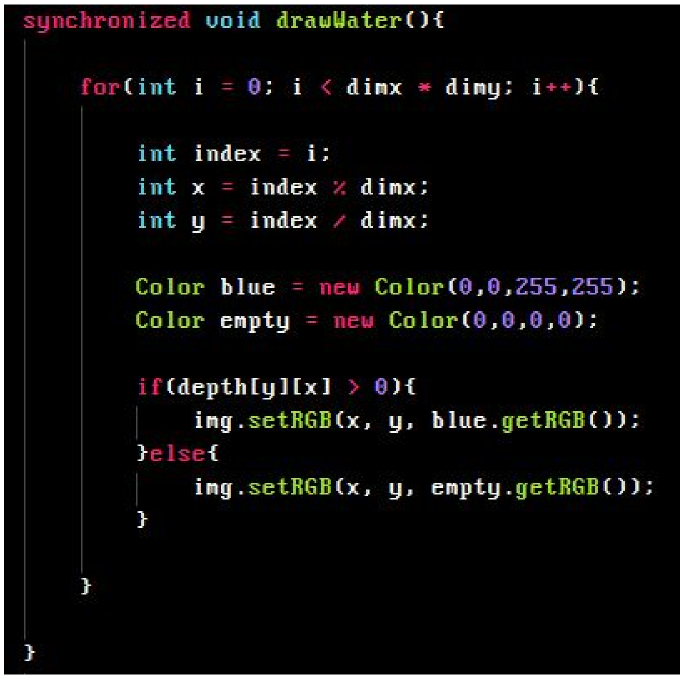
Drawing Water : drawWater()

Fig.6 shows the code listing for the drawWater member function, which is used to draw the pixels representing grid water depth values > 0.

The Color class is utilized to create the blue and transparent color classes which are parsed to setRGB along with the relevant x and y water grid coordinates.

It consists of a for loop (similar to **Fig.5**) where for each depth value in the water grid, if the water depth is > 0, setRGB is used on the BufferedImage (img) to set the corresponding water grid position (pixel color) to a blue, or transparent for a depth value = 0

The class method utilizes the synchronized keyword to ensure that only one thread at a time may call drawWater(), this helps prevent against race conditions that may occur on shared variable data



```java
synchronized void drawWater(){

    for(int i = 0; i < dimx * dimy; i++){

        int index = i;
        int x = index % dimx;
        int y = index / dimx;

        Color blue = new Color(0,0,255,255);
        Color empty = new Color(0,0,0,0);

        if(depth[y][x] > 0){
            img.setRGB(x, y, blue.getRGB());
        }else{
            img.setRGB(x, y, empty.getRGB());
        }

    }

}
```

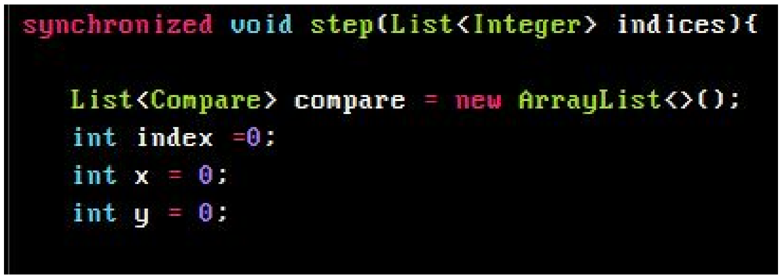
Fig.6 - Class method to draw water pixels (Water.java)

Simulation Steps : step()

Fig.7 shows the initial definitions for the class method step, which implements the core functionality of the water flow simulation timestep function (step).

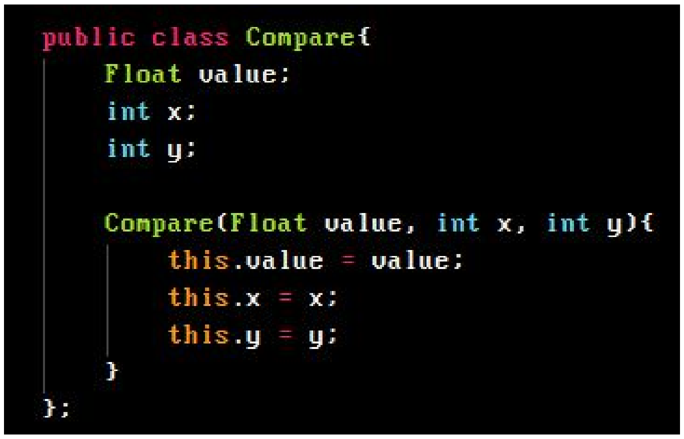
It takes as a single parameter a list of integers representing the permuted water grid positions to compute after completing a single simulation step, this is to be divided among the 4 threads such that for each thread, the size of indices for a single thread is divided equally (approximately).

It consists of a List of Compare objects (defined in the code listing shown in **Fig.8**) which is used to store the neighbouring water surface values along with their corresponding x and y coordinates.



```java
synchronized void step(List<Integer> indices){

    List<Compare> compare = new ArrayList<>();
    int index =0;
    int x = 0;
    int y = 0;
```

Fig.7 - Initial function definitions (Water.java)

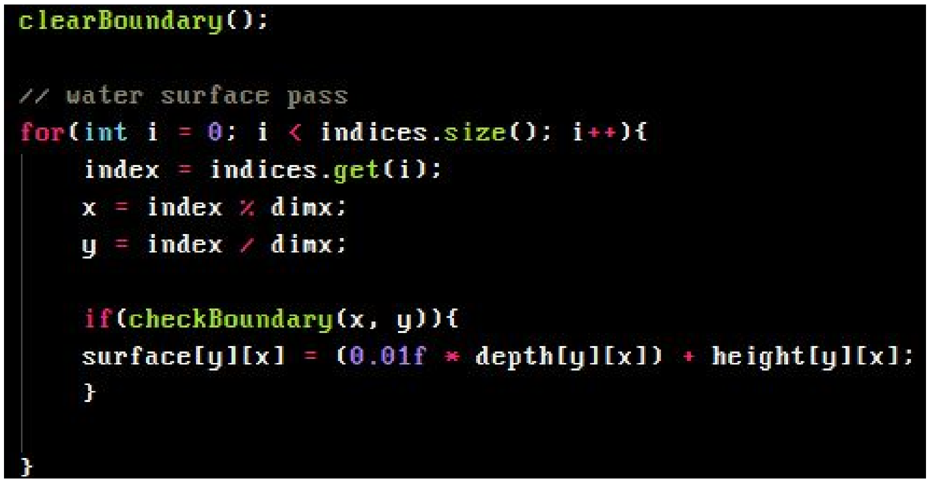


```java
public class Compare{
    Float value;
    int x;
    int y;

    Compare(Float value, int x, int y){
        this.value = value;
        this.x = x;
        this.y = y;
    }
};
```

Fig.8 - Compare class (Water.java)

A simulation step consists of two main 'passes', that is, two main processing stages implementing the water flow algorithm as defined in the leaflet.

The first stage is shown in **Fig.9** and is known as the water surface pass stage. Here the water surface boundary is initially cleared, then for each entry in indices the corresponding x and y grid position is boundary checked and it's corresponding surface value calculated using the formula defined in the leaflet.



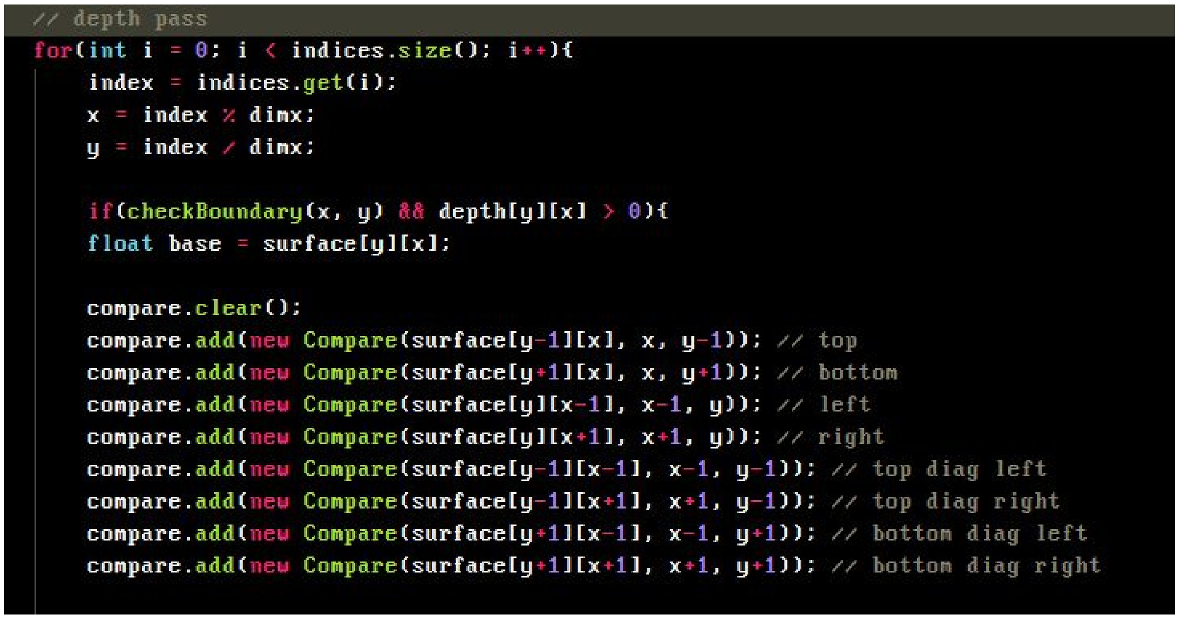
```java
clearBoundary();

// water surface pass
for(int i = 0; i < indices.size(); i++){
    index = indices.get(i);
    x = index % dimx;
    y = index / dimx;

    if(checkBoundary(x, y)){
    surface[y][x] = (0.01f * depth[y][x]) + height[y][x];
    }

}
```

Fig.9 - First processing stage of step() (Water.java)

Fig.10 shows the partial code listing for the second processing stage of step(). The main goal of the second processing stage is to perform water surface comparisons with a particular grid position defined by an (x,y) coordinate pair and then transfer the corresponding water depth unit to the lowest neighbour.

As can be seen from **Fig.10**, initially the x and y coordinates are calculated for the specific indices position after which a boundary check as well as water depth check is performed.

If passed, the water surface value for the corresponding (x,y) grid position is calculated. Then for each grid neighbour it's corresponding surface value as well as x and y positions are added to a list of type Compare (defined in **Fig.8**)

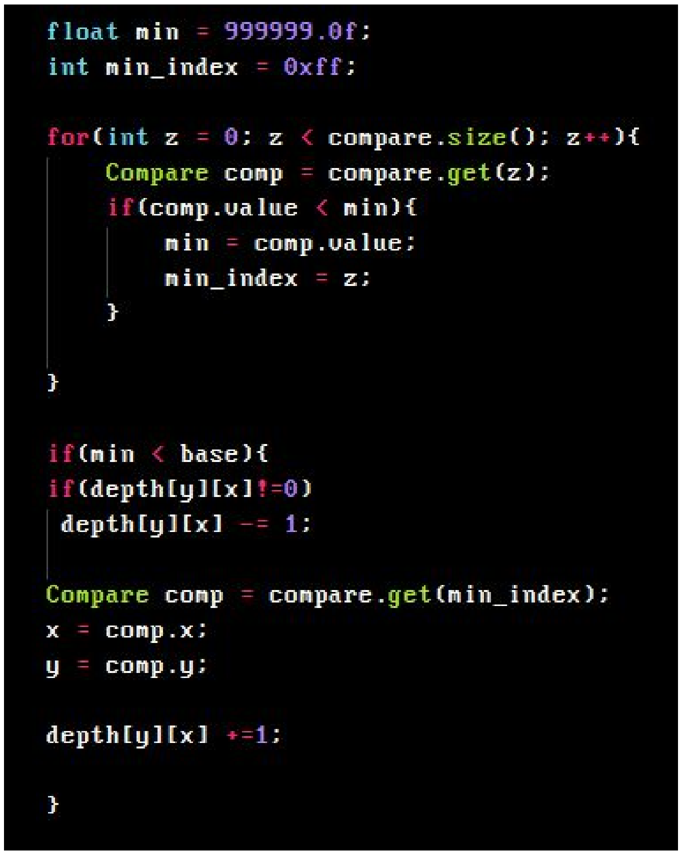


```
// depth pass
for(int i = 0; i < indices.size(); i++){
    index = indices.get(i);
    x = index % dimx;
    y = index / dimx;

    if(checkBoundary(x, y) && depth[y][x] > 0){
    float base = surface[y][x];

    compare.clear();
    compare.add(new Compare(surface[y-1][x], x, y-1)); // top
    compare.add(new Compare(surface[y+1][x], x, y+1)); // bottom
    compare.add(new Compare(surface[y][x-1], x-1, y)); // left
    compare.add(new Compare(surface[y][x+1], x+1, y)); // right
    compare.add(new Compare(surface[y-1][x-1], x-1, y-1)); // top diag left
    compare.add(new Compare(surface[y-1][x+1], x+1, y-1)); // top diag right
    compare.add(new Compare(surface[y+1][x-1], x-1, y+1)); // bottom diag left
    compare.add(new Compare(surface[y+1][x+1], x+1, y+1)); // bottom diag right
```

Fig.10 - Second processing stage (partial listing) (Water.java)

The second partial listing of the second processing stage is shown in **Fig.11.** Here, for each entry in compare the minimum water surface as well as corresponding index is calculated and compared to the (x,y) grid position of interest.

If it is strictly lower a unit of water depth is subtracted from the point of interest and added to the lowest water surface neighbour.



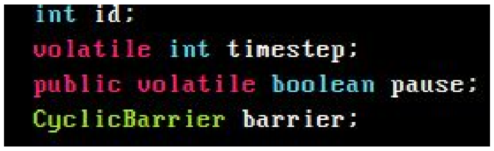
```
float min = 999999.0f;
int min_index = 0xff;

for(int z = 0; z < compare.size(); z++){
    Compare comp = compare.get(z);
    if(comp.value < min){
        min = comp.value;
        min_index = z;
    }

}

if(min < base){
if(depth[y][x]!=0)
 depth[y][x] -= 1;

Compare comp = compare.get(min_index);
x = comp.x;
y = comp.y;

depth[y][x] +=1;

}
```

Fig.11 - Second processing stage (partial listing) (Water.java)

Class Modifications: FlowPanel.java

Additional variables shown in **Fig.12** were added to the FlowPanel class, these include:

- integer id to store the thread id
- volatile integer to store the current timestep
- public volatile boolean to enable/disable simulation
- CyclicBarrier to ensure thread/simulation step synchronization



```
int id;
volatile int timestep;
public volatile boolean pause;
CyclicBarrier barrier;
```

Fig.12 - Additional variables added to FlowPanel

In addition the incomplete run method provided in the Java sample was completed and is shown in **Fig.13**. Here the thread name (parsed as a string id) is converted to an integer and stored into the id class variable.

In addition the length of the permute list is calculated and a variable (poolSize) assigned a value equal to the permute length / 4.

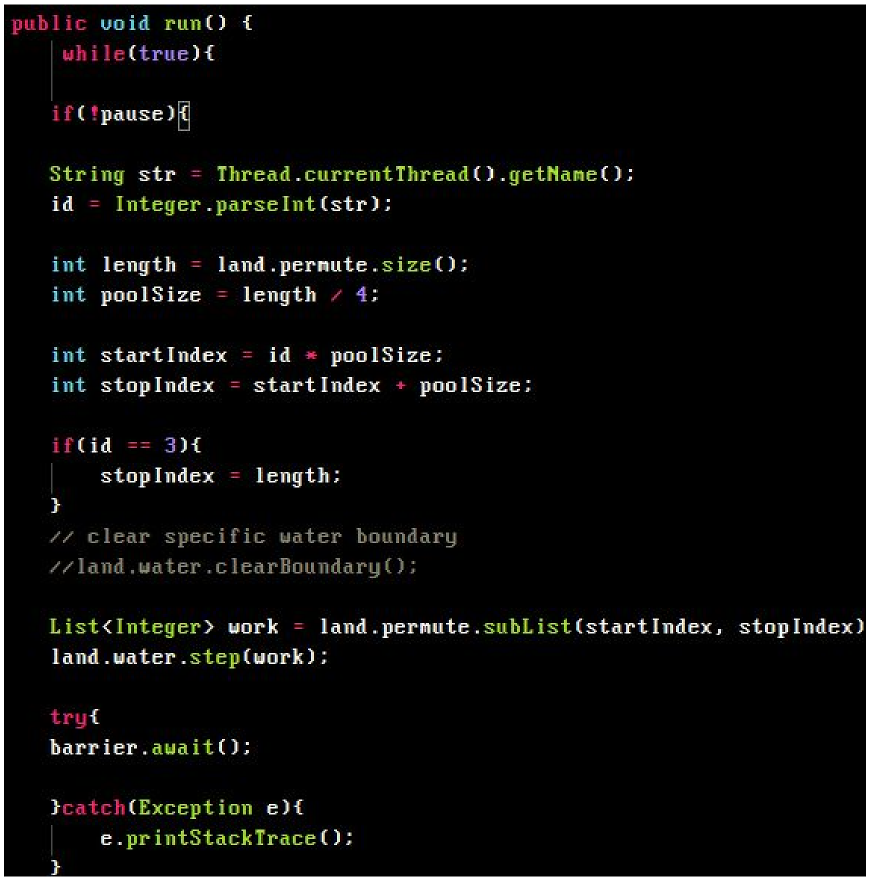
Next the work list is calculated by calling the subList method on the permute list, where startIndex and stopIndex define the work division (in terms of number of elements) per thread, this ensures that each thread approximately shares an equal workload in terms of number of terrain grid elements that need processing.

Following this, the step() core simulation class method is called on the work list and the water simulation processed for all grid positions defined by the work integer list.

Next a call to barrier.await() is made to ensure that all threads complete execution before continuing with the next iteration of the simulation step().

It is worth noting that the pause class variable defined internally is used to prevent the next iteration of the while loop and is used to pause/restart the simulation loop.

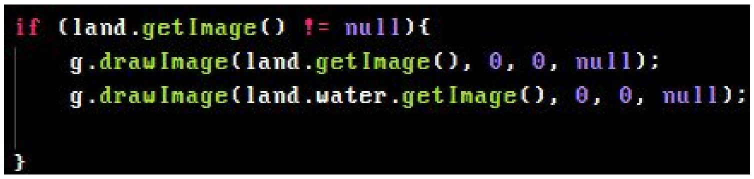
Outside of the pause (if) statement a call to drawWater() and repaint() is made, this ensures that the rendering of the terrain runs faster than the simulation steps.



```
public void run() {
    while(true){

    if(!pause){

    String str = Thread.currentThread().getName();
    id = Integer.parseInt(str);

    int length = land.permute.size();
    int poolSize = length / 4;

    int startIndex = id * poolSize;
    int stopIndex = startIndex + poolSize;

    if(id == 3){
        stopIndex = length;
    }
    // clear specific water boundary
    //land.water.clearBoundary();

    List<Integer> work = land.permute.subList(startIndex, stopIndex)
    land.water.step(work);

    try{
    barrier.await();

    }catch(Exception e){
        e.printStackTrace();
    }
```

Fig.13 - Completed run() within FlowPanel

Finally additional code was added to overlay the water BufferedImage over the GUI frame and is shown in **Fig.13.1**

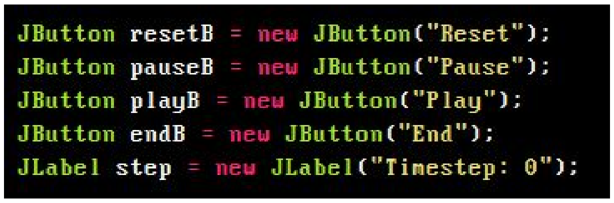


```
if (land.getImage() != null){
    g.drawImage(land.getImage(), 0, 0, null);
    g.drawImage(land.water.getImage(), 0, 0, null);

}
```

Fig.13.1 - drawImage for water image

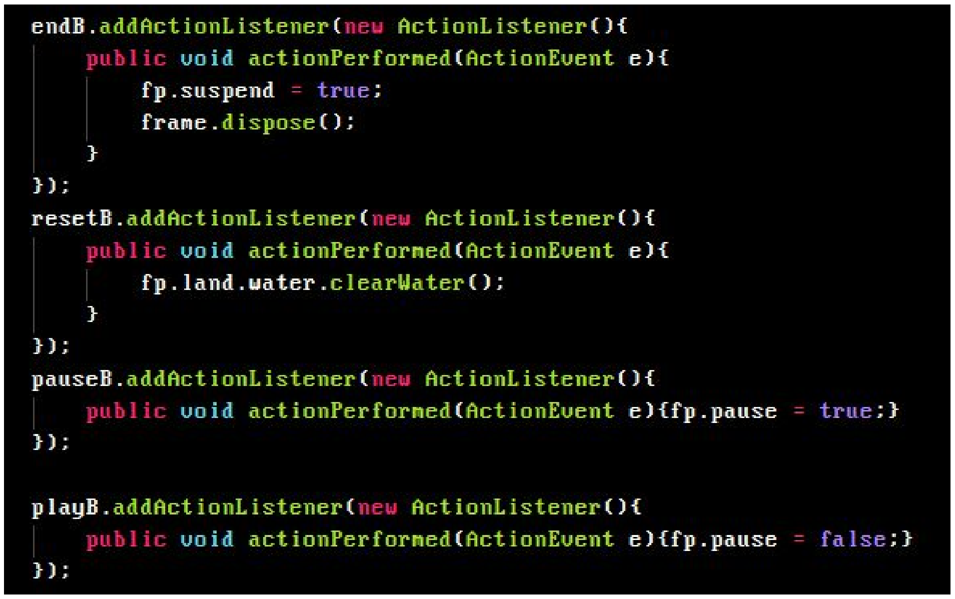
Class Modifications: Flow.java

Modifications to the Flow.java class were made to add the various GUI elements required by the simulator. Specifically, the 4 JButton objects as well as JLabel objects were declared in the setupGUI method and are shown in **Fig.14**.



```
JButton resetB = new JButton("Reset");
JButton pauseB = new JButton("Pause");
JButton playB = new JButton("Play");
JButton endB = new JButton("End");
JLabel step = new JLabel("Timestep: 0");
```

Fig.14 - Required GUI elements created



```
endB.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        fp.suspend = true;
        frame.dispose();
    }
});
resetB.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        fp.land.water.clearWater();
    }
});
pauseB.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){fp.pause = true;}
});

playB.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){fp.pause = false;}
});
```

Fig.15 - ActionLister Event Handlers (GUI)

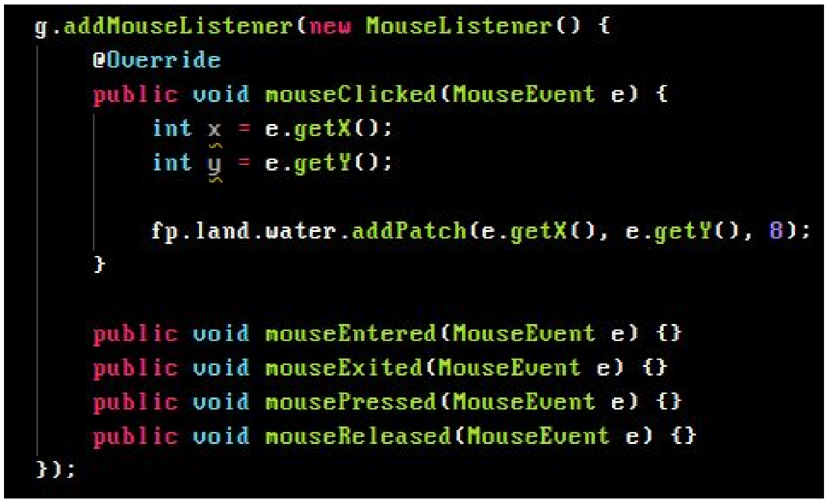
Fig.15 shows the various ActionLister event handlers used to handle play, pause, reset and end functionality in the simulator.

For the endB actionListener, the value of true is assigned to the suspend thread variable which ensures that threads have a chance to terminate before frame.dispose() is called.

For the reset actionListener a call to clearWater() is made to clear the water from the global water grid space.

For the pause/play actionLister's a simple true or false boolean assignment is made on the fp.pause variable.

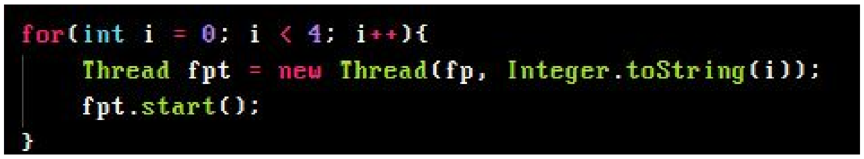
Fig.16 shows the MouseListener code listing used to handle mouse events on the GUI overlay. On an input mouse event the relevant x and y coordinates are obtained and a call to addPatch made with a value of n=8, which places a patch of water 8x8 units wide at the location (x,y).



```
g.addMouseListener(new MouseListener() {
    @Override
    public void mouseClicked(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();

        fp.land.water.addPatch(e.getX(), e.getY(), 8);
    }

    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
});
```

Fig.16 - ActionLister Event Handlers (GUI)

Fig.17 shows the thread creation mechanism within Flow.java, here 4 new threads are spawned each with a unique id and the run() method initialized with a call to start()

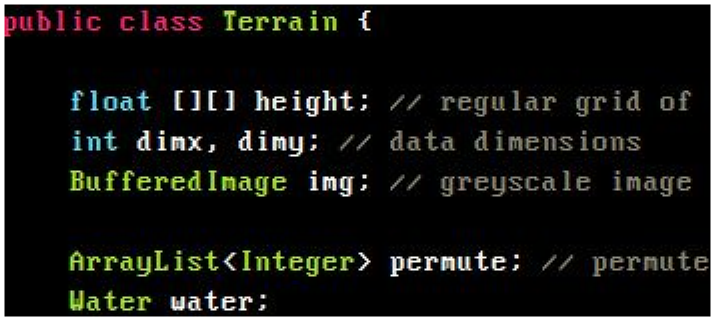


```
for(int i = 0; i < 4; i++){
    Thread fpt = new Thread(fp, Integer.toString(i));
    fpt.start();
}
```

Fig.17 - Thread spawning

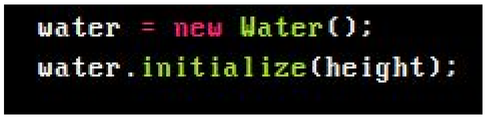
Class Modifications: Terrain.java

For the Terrain.java class, a few minor modifications were made which include the addition of the water class variable (implementing the water simulation core), shown in **Fig.18** as well as the corresponding initialization code shown in **Fig.19**.



```
public class Terrain {

    float [][] height; // regular grid of
    int dimx, dimy; // data dimensions
    BufferedImage img; // greyscale image

    ArrayList<Integer> permute; // permute
    Water water;
```

Fig.18 - Addition of water member variable



```
    water = new Water();
    water.initialize(height);
```

Fig.19 - Water class initialization

Thread Safety via Volatile

For various shared variables it was necessary to declare variables as volatile to avoid read/write synchronization issues across threads and ensure variable memory visibility.

For example in **Fig.1**, the variables depth, surface, height and img are declared as volatile. Declaring a variable as volatile ensures that access to that variable is automatically synchronized and that subsequent variable reads and writes across threads do not compete/ result in deadlock.

Therefore these variables are required to be declared as volatile since access to the depth, surface, height and img class members occurs across 4 threads and requires both read and write operations.

In contrast the dimx and dimy class variables are only written to once during a call to initialize() and therefore any access to the variable across different threads is read only and therefore does not require an explicit volatile qualifier.

Thread Safety via Synchronize

To prevent data races occurring within specific Water class methods, it was necessary to qualify specific class methods with the Java synchronized keyword.

The java synchronized primitive ensures that a -- class -- method declared as synchronized cannot be called on more than once instance objects at any one point in time.

That is, if multiple threads concurrently call a synchronized method, the thread currently executing the class method holds a 'lock' on the object being called on and other threads have to wait for this lock to be released before being able to perform a subsequent invocation of the class method.

For example, the clearWater() routine shown in **Fig.6** is used to clear the water depth variable for all grid positions (depth class member defined in **Fig.1)**. This method is called from an external thread responsible for handling GUI events.

The clearWater() requires a write operation to a member variable that is used across various threads. If the clearWater() method was not declared as synchronized and was called during a simulation time step, an attempt to write to the same variable at the same time could occur or some other unknown side-effect leaving to a data races

Therefore without this mechanism, the probability of data races across threads increases dramatically.

Not all class methods require this qualifier however, the checkBoundary() class method shown in **Fig.2.1** simply involves a data read operation and then performs a logical comparison.

Thread Synchronization via CyclicBarrier

One of the requirements for the simulation component of the assignment was that threads remain synchronized during a single simulation timestep.

That is, for each of the 4 threads handling a portion of the terrain grid each thread should finish depth processing (as per the step() algorithm) before the next time step begins. To ensure this, the CyclicBarrier class was utilized.

Fig.13 (bottom) indicates a call to barrier.await() is made to ensure that all threads complete execution before continuing with the next iteration of the simulation step().

Conclusion

In conclusion Java provides a useful concurrency API together with many built in synchronization qualifiers.

These make it possible to write thread safe multithreaded applications.