

Lab 11 – posted 11/16, due 12/5 – you DON'T have to use Eclipse for this lab. Please submit the java files.

Propagation Simulations

Scientists use computers to map and predict the changes and movement of many phenomena: weather, disease, pollution, economic prosperity, the damage caused by explosions, even social behaviors like the spreading of rumors. The state and movement of these phenomenons are often portrayed on a map that changes over time. The map is divided into squares (called Nodes), and each square on the grid represents the Value of a complicated equation, translated into a color.

Some are familiar to us. One aspect of global warming is portrayed here, where the color of each square on a grid is a measure of the Earth's absorption of heat:

<http://profhorn.aos.wisc.edu/wxwise/AckermanKnox/chap2/Albedo.html>

Some are quite striking. This is a map of x and y points, with colors given to a point's likelihood of resulting in a large number when placed in a certain mathematical equation:

<http://aleph0.clarku.edu/~djoyce/julia/MandelbrotApplet.html>

Note the behavior of any Node. Its color depends on its current Value at one moment in time, and its current Value depends on the results of some equation that may take into account many variables: the Node's place on the X-Y grid, the lapsed time since an incident, or the Values of the Nodes around it, probably at a previous instance in time. In that way, we might see the concentration of pollution moving east from its source in Cleveland to the inner harbor in Buffalo over 40 hours time. At each hour, the Value of a Node depends on easterly winds, and the previous concentration of pollution in the Node's neighbors.

From what we know about Object Oriented programming, each Node can be an object of properties and behaviors (called a Model), the map can be a JFrame of graphics where the color of each pixel depends on the Value contained in a correlated Node (called a View), and we might control parameters that affect behavior using a frame of JComponents (called a Controller).

The Model-View-Controller Pattern - (this is from a book on Essential Actionscript 2.0 by Colin Moock, http://www.adobe.com/devnet/flash/articles/mv_controller.html)

The Model-View-Controller (MVC) pattern says that we can and should separate an object's behavior from our view of that behavior. The basic principle of MVC is the separation of responsibilities. In an MVC application, the model class concerns itself only with the application's state and logic. It has no interest in how that state is represented to the user or how user input is received. By contrast, the view class concerns itself only with creating the user interface in response to generic updates it receives from the model. It doesn't care about application logic nor about the processing of input; it just makes sure that the interface reflects the current state of the model. Finally, the controller class is occupied solely with translating user input (provided by the view) into updates that it passes to the model. It doesn't care how the input is received or what the model does with those updates.

Separating the code that governs a user interface into the model, view, and controller classes yields the following benefits:

- Allows multiple representations (views) of the same information (model)

- Allows user interfaces (views) to be easily added, removed, or changed, at both compile time and runtime
- Allows response to user input (controller) to be easily changed, at both compile time and runtime
- Promotes reuse (e.g., one view might be used with different models)
- Allows multiple developers to simultaneously update the interface, logic, or input of an application without affecting other source code
- Helps developers focus on a single aspect of the application at a time

Instructions

In this Lab, you will create a Model (the Node class) and a simple Viewer (a Viewer class containing a grid and a paint method that paints the grid). The constructor of the Viewer class will be the ultra-simple Controller.

The Node (Model) Class

Create a Node Class that will represent some behavior. An object of this class will be created for every square on our grid.

For now, your Node class should have the following *properties*:

- its position on an X and Y grid. Every object instantiated from this class will know its place on a large grid. It is recommended the constructor if this class receive the X and Y location during creation.
- a floating point number that represents a Node's single, important **VALUE**.

The Node class should have the following *methods*:

- the Constructor, which simply accepts the Node's X and Y position as input parameters. In this way, each node receives a unique X and Y position.
- a means to calculate the Node's Value. When called, this method will calculate the Node's Value based on any parameters we later choose. This might be an equation that is a function of the Node's X and Y position. $Value = (X^2 + Y^2)$ is a good Value, since it increases as a function of distance from the start of the grid, and might reflect something like inverse gravitational pull, or heat from a central source, or the effect of an explosion.

The Viewer Class

The purpose of the Viewer class is to create a grid of Nodes, assess the Value of each Node, translate the Value into an appropriate color (say, red for small Values and yellow for large Values), and draw some pixels on a JFrame whose position correlates to a Node's position on the grid, and whose color was derived from the Node's Value. Follow these steps:

1. Write a Viewer class that extends JFrame.
2. In the Viewer class constructor, create a JFrame. 400 x 400 pixels ought to do it.

3. In the Viewer class constructor, create a grid of 100 x 100 Node objects. You can either use one of the Collection classes that we will discuss in class, or a simple 2-dimensional array of Node objects.

If you use a multidimensional array, remember that the array definition:

```
private NodeClass[] [] grid = new NodeClass[100][100];
```

merely creates enough room for an array grid that is comprised of 100 x 100 Node objects. Each element of the array must be populated by an additional Node object instantiation:

```
grid[i][j] = new NodeClass(i, j);
```

which creates the Node object, which is an element of the grid array at location i and j and will initialize the Node's awareness of its own place on the grid.

4. In the Viewer class constructor, create each Node object (there will be 10,000) and initialize each one by instantiating it. A helpful control structure is the “*nested for loop*”, which conveniently traverses all Values of x and all Values of y, which are represented here by i and j:

```
for (int i = 0; i<=99; i++){
    for (int j = 0; j<=99; j++) {
        grid[i][j] = new NodeClass(i, j);
    }
}
```

When you do this, you might want to keep track of the maximum Value, and the minimum Value of the collection of Nodes on the grid. This will give you a range of possible Values when translating a Value into a Color.

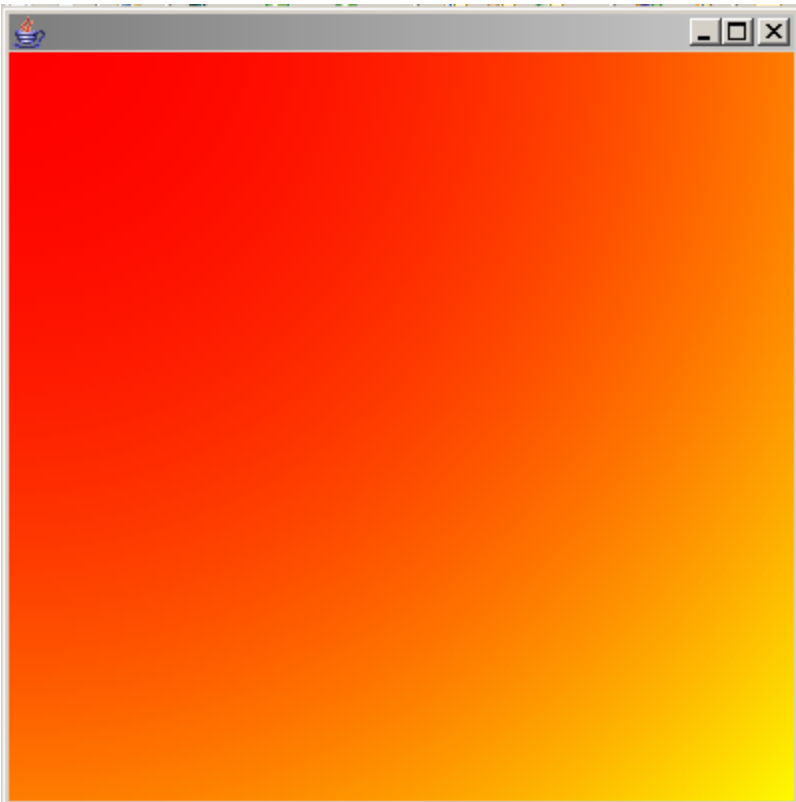
5. Write a paint method that gets the Value of each Node within the grid, translates the Value to an appropriate color, and draws a colored oval or rectangle at a point on the JFrame corresponding to a Node's position on the grid. In this way, each Node on the grid corresponds to a group of pixels on the JFrame, and the pixel group's color depicts a Nodes Value. A Node that has the maximum Value might be colored white, and a Node that has the minimum Value might be colored black, and Nodes with Values in the range get all colors in between.

6. Pick any appropriate equation $Value = f(x,y)$ and run the simulation to produce an interesting picture.

7. (Optional) – in the Node, you can make your value calculation a function of time (a timer value gets passed into your calculation method). Then, in the Viewr class, set up a timer. Each time the timer ticks, *repaint()*; your paint method (each time calling all the node calculation methods with the new timer value), and watch your simulation change over time.

An Example Solution

Here's my JFrame, depicting my grid of Node Values:



I used a 400 x 400 JFrame to represent my 100 x 100 Node objects. That means each Node got a 4 x 4 square of pixels. I drew a filled rectangle representing a Node's Value at (Node position $X * 4$) and (Node position $Y * 4$) where X and Y range from 0 to 99, representing the 100 X 100 grid of Nodes.

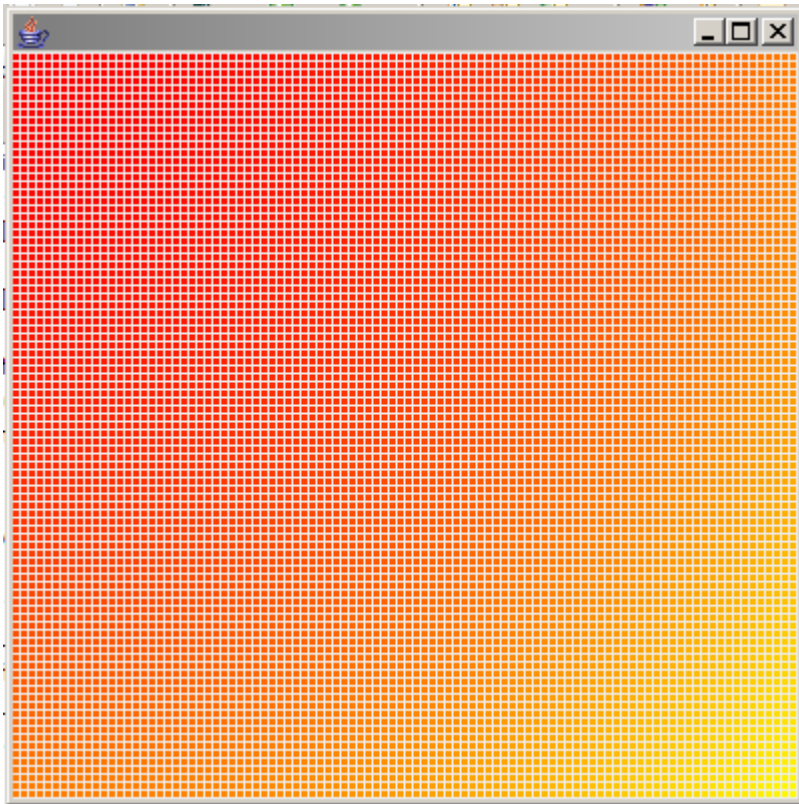
I got each Node's initialization Value to derive a Color. Reading the JavaDocs for the Color class, I see that I can create a Color using three Values: a red, green, and blue component (each a number from 0 to 255).

Since my initialization equation was $Value = (X^2 + Y^2)$, I got a range of numbers from 0.0 to 19602.0. I "normalized" the result on a scale of 0 to 255, where 0 translated to 0, 19602 translated to 255, and every Value in between was proportional to 255. That is, I used a normalization equation like this:

$$colorValue = 255 * (Value / 19602);$$

Then I could do my setColor for each rectangle using `new Color(0, colorValue, 255)` which gave my pixels a color of red for low Values, and yellow for high Values. I see from my map, that the farther I am away from the frame's origin, the more yellow my Node.

Changing the each rectangles size to 3 x 3 pixels, I could see each Node's color representation individually:



Or, changing my setColor calculation to `setColor(new Color(75, colorValue, colorValue));` really shows the changing of heat to cold as a function of distance from the source.

