ASSIGNMENT 4
McGill Paint Part 2 (Draft)

COMP-202, Summer 2011

Due: Monday, July 4th, 2011 (23:30)

This version is a draft of assignment 4 being released for people to get ahead on the assignment a bit. I may make some changes to the assignment, such as adding or removing a question.

Much of this assignment will depend on parts of assignment 3. As such, shortly after the final assignment 3 is handed in, you will be provided with an assignment 3 solution.

You must do this assignment individually and, unless otherwise specified, you should follow all the general instructions and regulations for assignments. Graders have the discretion to deduct up to 10% of the value of this assignment for deviations from the general instructions and regulations.

Beginning in this assignment, the 75% non-compilation penalty will apply. This means that if your code does not compile, you will receive a maximum of 25% for the question. If you are having trouble getting your code to compile, please contact Dan or one of the TAs or consult each other or the discussion boards.

Part 1	50 points
Part 2	30 points
Part 3	5 points
Part 4	15 points
Bonus	30 points
	100 points total

Part 1 : Matrix Utilities Part 2

In this section, you will add some methods to the MatrixUtilities class you wrote in assignment 3. The following should be added to the file `MatrixUtilities.java`.

1. `public static double[][] power(int[][] m, double power)` This method should take as input a 2d matrix and return a new matrix with the same size as the original matrix and with each entry being the value of the corresponding entry in the original matrix raised to `power`.

   For example, if the input matrix is: { {1,2}, {3,4} } and `power` is 2, then the resulting matrix should be: { {1,4 }, { 9, 16 } }

2. `public static int[][] doubleSize(int[][] m)` This method should return a new matrix with each of its dimensions doubled in size. The way to do this is that every single value in the original matrix should be duplicated 3 additional times so that it is present 4 times in the new matrix.

3. `public static int[][] halfSize(int[][] m)` This method should return a new matrix with each of its dimensions halved in size. The way to do this is to "compress" the matrix by taking each 2 by 2 block of the original matrix, averaging the 4 values, and replacing it by the 1 average. You may assume that the matrix has even dimensions in both directions.

4. `public static int[][] approximateDerivativeX(int[][] m)` This method will approximate the "derivative" of the matrix with respect to x. What this means, is as the matrix indices moves from left to right, how do the values of the matrix change?

   We can approximate this by creating a new matrix based off of `m` where the value of the new matrix m' is computed using the following formula:

   $m'[i][j] = (m[i-1][j+1] - m[i-1][j-1]) + 2*(m[i][j+1] - m[i][j-1]) + (m[i+1][j+1] - m[i+1][j-1]$

   Note that in cases on the edge, such as when i = 0 for example, you should treat the value at the location as being the same as the one that was not off the matrix. For example, if the formula says to take m[-1][0], you should use the value for m[0][0] instead.

5. `public static int[][] approximateDerivativeY(int[][] m)` This method will approximate the "derivative" of the matrix with respect to y.

   This formula will be similar to the derivative with respect to x, except it will be flipped slightly.

   $m'[i][j] = (m[i+1][j-1] - m[i-1][j-1]) + 2*(m[i+1][j] - m[i-1][j]) + (m[i+1][j+1] - m[i-1][j+1]$

   It is also possible to use the rotate method along with the approximateDerivativeX method to avoid doing more work here.

6. `public static int calculateDeterminant(int[][] m)` The *determinant* of a matrix can be calculated by taking performing the following for every entry in the 0th row of a matrix.

   (a) If the matrix has only 1 row and 1 column, then the determinant is just whatever that 1 entry is.

   (b) Otherwise, for a number in the ith column (starting from zero) and 0th row, cross off (temporarily) the entire ith column and 0th row.

   (c) The resulting matrix has 1 fewer row and 1 fewer column after crossing them off. Take the determinant of this matrix and multiply the result by the value in the ith column and 0th row.

   (d) If i is odd, multiply the result by negative 1.

   (e) Sum up all such results.

   An example will be done in class of this procedure. `http://www.youtube.com/watch?v=21LWuY8i6Hw` is a video demonstrating this on a 3*3 matrix

   USING RECURSION write a method to calculate the determinant of a matrix.

   All your code for this section should go into a file `MatrixUtilities.java` in a class `MatrixUtilities`

Part 2 : Image Manipulation Part 2

In this section, you will use the Matrix tools you wrote in the first part, along with some other tools provided to you, to write an image manipulation program.

See the assignment 3 specification for some more background information on this.

Building on top of what you wrote in assignment 3, you should additionally write the following methods in the class `ImageConverter`

- A method `doubleSize` This method should double the size of the image. It can do this by calling the method you wrote in `MatrixUtilities.java` on `imageData[0]` , `imageData[1]` , `imageData[2]` , and `imageData[3]` . It should then produce a new 2d array from the results and store a reference to this array into `imageData`

- A method `halfSize` This method should halve the size of the image referenced by imageData using a similar transformation to that done on matrices in part 1.
- A method `convertToBlackAndWhite()` The way to convert an image to black and white is by assigning to each pixel the average value of the red,green, and blue components. For example, if the original image had 200 for red, 150 for blue, and 100 for green, then the new image should have 150 for all 3.

  This method should convert the image referenced by imageData to black and white.
- A method `getNegativeImage()` One can convert a black and white image to a negative image by "flipping" the values of all components. To do this, each component of each pixel should have the value $255 - x$ where x was the original value. Write a method `getNegativeImage()` that converts a black and white image to the negative of the image.

  Hint: If you are clever about things, you can use the methods in `MatrixUtilities` here once again to avoid writing much code.

Part 3:Point class

Define a class `Point` . The class `Point` should have 2 private fields, `x` and `y`. In addition, it should have the following methods:

1. `public Point(int xcoord, int ycoord)`
2. `public int getX()`
3. `public int getY()`

Note that your method should NOT have a setter for x or y. This means that a Point object, once defined is *immutable*. You will have to create a brand new `Point`

Part 4: Computer Vision Algorithms

Now, we will add functionality to our program to detect various shapes. We will perform 2 different computer vision algorithms, that are central to detecting objects. 1)Detecting dark pixels and 2)Detecting lines

You have been provided an interface `PixelHighligher`. There is only 1 method a class has to define in order to implement this interface `choosePixels` . The method `choosePixels` takes as input an `int[][]` and returns an `ArrayList<Point>` . This ArrayList is chosen in different ways depending on how the class chooses to implement `PixelHighligher`

Note that the input is only ONE `int[][]` . This is because we are assuming that the image is already in black and white, meaning that all values are the same. You can pass in any of the channels–red, green, or blue.

Part 4a) Dark pixel chooser

Write a class `DarkPixelChooser` . This class should implement the interface `PixelHighligher`, meaning it must have the method `choosePixels` defined with the same input and output as in the interface definition.

The class should have a constructor `DarkPixelChooser` which takes as input an `int threshold` and sets a corresponding private instance variable equal to that value.

The method `choosePixels` should return an `ArrayList<Point>` where the values in the ArrayList represent all pixels with values greater than the value `threshold` that is passed to the constructor.

Part 4b) Using the dark pixel chooser

Add a private field `PixelHighligher pixelHighligher` to your class `ImageConverter`. For now, inside the constructor of `ImageConverter`, set pixelHighligher equal to a new `DarkPixelChooser` with the parameter `threshold` equal to 155.

Now add a method to `ImageConverter` called `public applyPixelHighlight()` This method should call the method `choosePixels` defined on all Objects of type `PixelHighlighter`. It should then set set all the values in the red,green, and blue arrays according to the following criteria:

If the pixel is present in the ArrayList returned by `choosePixels()` then each of the RGB values should be 0. Otherwise, each of the values should be 255.

You can now call this method and you should see the darker values exagerated.

Bonus: Edge chooser (30 points)

Write a class `EdgeHighligher` that also implements `pixelHighligher`. This method will choose pixels that are the edge of something in an image. The way we will detect edges is surprisingly straightforward. To detect an edge, we will look for areas of the image where there is a large change in the intensity (i.e. the numbers) of the pixels.

First, write a constructor `EdgeHighligher(int threshold` which takes as input a threshold and sets an instance variable equal to this value.

Next, write a method `choosePixels(int[][] m)` which should choose pixels on the edge. To do this, we will use more of the Matrix methods we defined in the first part. First, compute a matrix by the following:

1. Given the matrix m, first compute 2 matrices, L_x and L_y which are the derivative of the matrix with respect to x and y respectively. (Call the methods you wrote in part 1)
2. Now, compute a new matrix, in which each entry is equal to the value of the matrix in $\sqrt{L_x^2 + L_y^2}$
3. Return an ArrayList of Point representing the coordinates where the value is greater than `threshold`

You can now change the constructor of `ImageConverter` to create a new `EdgeHighligher` instead of a new `DarkPixelChooser`. Once you do this, you should see a new image that has all the edges selected. If you don't see this, experiment with numbers other than 155. (Note from Dan: I will provide a couple images that make this distinction more obvious)

Spiritual Growth Question: Corner chooser (not for credit)

I will provide more information on how to detect corners. For a head start on this, you can read about Harris Corner detection at *http://www.cim.mcgill.ca/ dpomeran/vision/project.pdf*

What To Submit

`confession.txt` - You should write in this file any information that you think is useful for the TA to mark the assignment. This should include things you were not sure of as well as parts of your code that you don't think it will work. Of course, like a confession, this will draw the TA's attention to the part of your code that doesn't work, but he/she will probably be more lenient than if he/she has to spend a lot of time looking for your error. It demonstrates that even though you couldn't solve the problem, you understand roughly what is going on.

`MatrixUtilities.java`
`ImageConverter.java`
`DarkPixelChooser.java`
`Pixel.java` (Sorry: I forgot to list this before)
`EdgeHighligher.java` (optional)
`CornerChooser.java` (optional)