**CS203 HW #8
Spring 2012
Digital Cameras & Image Transformations**

**Due Dates:** Part A: Wednesday, March 28, 11 am (code for part A only)
                   Part B: Wednesday, April 4, 11 am (code for part A and part B and report)

You will write a set of image transformations for this homework assignment. The first algorithm that you will implement is what a digital camera does internally for creating a full-color picture based on light intensity levels the sensors in the camera record when a picture is taken. Below you will find more information about how digital cameras work and the Java classes that comprise the starter code. Throughout this project, feel free to use your own (small) images.

**How Color Images are Represented in a Camera or Computer:** A picture is a 2-dimensional array of pixels (individual squares that make up the image). Each image has a specific resolution - the width and height in pixels of the grid that makes up the image. Each pixel has three components describing the red, green, and blue values (the primary colors) that combine to produce the color of that particular pixel. The red, green, and blue values range from 0 (none) to 255 (full intensity). A black pixel, for example, is one that has values red=0, green=0, and blue=0; while a white pixel has values red=255, green=255, and blue=255. Values in between 0 and 255 for these components produce different colors. If the red, green, and blue values are equivalent, the resulting color is a shade of gray.
 **How Digital Cameras Work:** Digital cameras contain a 2D grid of light sensors to record the light value for every pixel of the picture. Cheaper cameras (i.e., the consumer ones, which you will be modeling in your project) use one sensor per pixel, so that each sensor captures a single color. Instead of recording a complete color image (with red, green, and blue values for every pixel), the camera stores the light value for just a single color at each pixel location. Software is used to generate the remaining two colors of each pixel. For example, a green sensor senses just the green value for an individual pixel. Suppose the value read is 150. The camera then uses software to determine the blue and red values for that pixel that has just a green sensor.

Digital cameras have Bayer filter patterns to determine the color sensors. Because the human eye is most sensitive to green, the pattern is comprised of roughly half green sensors, a quarter blue sensors, and a quarter red sensors. The Bayer Pattern may look like the following:


Once the camera captures the values for each pixel location, the algorithm embedded in the camera must interpolate the other two non-sensed color values to create the complete color image. These algorithms are called demosaicing algorithms since they convert the mosaic of separate colors into an image of true colors. Your main task in Part A is to implement this algorithm.

*References about Digital Cameras:*
<http://electronics.howstuffworks.com/digital-camera5.htm>,
<http://www.shortcourses.com/choosing/how/03.htm>

**Description of Starter Code:**

Seven files comprise the starter code, but you only need to modify **one** of the class files. You will create your own Java classes for different tasks outlined in this project. The classes that you should **not** modify are marked below.

**PictureShop:** Do not modify this class. This class creates the graphical user interface for the application and performs the loading operation for images. It displays the buttons that are part of the user interface. When a button is selected, the corresponding filter method in a class implementing the Filter interface is called. This class controls both the user view (the application window) and reacts to the buttons pressed by the user. You do not need to look at or understand this class file.

**PixelImage:** Do not modify this class. This class keeps track of the current test image (the image displayed on the left in the application). When creating a new filter class, you will need to get the data (2-Dimensional array of Pixels) and modify the data. Important methods in this class are getWidth(), getHeight(), getData(), and setData(Pixel[][] data). The data is stored in a 2-dimensional array of Pixel objects where the first dimension corresponds to the number of rows in an image and the second dimension corresponds to the number of columns in the image.

**Pixel:** Do not modify this class. This class represents Pixel objects. Each Pixel has four instance variables: red is the value of the red component; green is the value of the green component; blue is the value of the blue component; and filterColor is the color the digital camera actually senses for that pixel. If you have a Pixel object named currentPixel, then you can access the red value by currentPixel.getRed() in your code.

**Filter:** Do not modify this interface. Filter is an interface that all classes you write for this assignment must implement. Every class implementing the Filter interface must have a method called filter that does not return anything and takes as a parameter a PixelImage object.

**DigitalCameraFilter:** Do not modify this class. This class performs the conversion from a full-color image file to the corresponding data that would be gathered by the sensors in a camera. Because we're not working with real digital cameras, this class serves as the inverse operation of producing the full-color image from the sensor data. It is automatically executed before presenting the left-hand image in the application.

**FlipVerticalFilter:** Do not modify this class. This class serves as an example of a class that implements the Filter interface. The filter method in this class flips the image across the horizontal midline. The filter method is called when the associated button is pressed on the application window. Use this class as a template for all the filters you write for this assignment.

**PictureShopConfiguration:** You **should** modify this class. This class configures the buttons for the application and has the main method (which simply creates a new SnapShop object). When you create filters, add the filter to the SnapShop passed to the configure method. Look at the configure method in this class to see how to add new buttons to the application.

**Image Files:**  Three files are provided as image files in the *Images* directory of your starter package. [You are encouraged to use your own images for this project. Any .jpg image should work fine, but start with small images. Be sure to use images that are at most the size of the image in shortMixedGradients.jpg; otherwise, it may take a long time for your code to process these images.]

Every class you will write will look like this, with a variation of the loop(s) and code inside the loop(s):

/\*\*

 \* Description of filter goes here

 \* @author Your name here

 \* @version Date goes here

 \*/

public class FilterName implements Filter {

 /\*\*

 \* filter

 \* description of method goes here

 \* @param pi The PixelImage object to modify

 \*/

 public void filter(PixelImage pi) {

 Pixel[][] data = pi.getData(); // get image data

 for (int row = 0; row < pi.getHeight(); row++) {

 for (int col = 0; col < pi.getWidth(); col++) {

 // put code here for how you want to change the pixel

 // for example, if you want to set the blue and green

 // values to the red value, you could do

 // int red = data[row][col].getRed();

 // data[row][col].setBlue(red);

 // data[row][col].setGreen(red);

// remember: data[row][col] is a Pixel object

 }

 }

 // reset data into the PixelImage object pi

 // be sure to set the data back to the image, otherwise it will not display

 // the new colors

 pi.setData(data); //this is important to reset the pixels in the image

 }

}

**Specification:**

**PART A (7 points):**

* (6.5 points) Create a new Java class called DemosaicFilter that implements the Filter interface. Every class that implements the Filter interface must implement the filter method (see the FlipVerticalFilter class for an example or the code above).
* (.5 points) To add a corresponding button for this demosaic filter in the application, add a line to the configure method in the SnapShopConfiguration class.
* Complete the implementation of the filter method (see below for algorithm description). The data for the image is represented as a 2-dimensional array of Pixel objects where the first dimension represents the number of rows and the second dimension represents the number of columns. You should transform the image data in the DemosaicFilter class to modify two of the three color instance variables (red, green, and blue) for each Pixel object. Each Pixel object has just a single accurate color value (one that you should not modify) and has an instance variable named filterColor that is set to the accurate color. The other two colors have a value of 0.
	+ For example, let's say we have a pixel whose value was gathered by a red light sensor. Then, the green and blue values will be equal to 0 before your demosaicing filter is executed. Then, you would need to assign the green and blue values based on the neighboring pixels.
* Your job is to calculate the values for the two colors that did not get accurately sensed as the camera took the picture for each pixel in the image. To perform the calculation, look at the surrounding adjacent pixel neighbors, find the ones that captured the color you are trying to calculate, and average these values.
	+ Remember, color values go from 0 to 255. The values are integers.
	+ Also, remember not to go out of bounds (off the array) in the 2-dimensional array of Pixel objects.
	+ In the Pixel class, the getFilterColor() method returns the color that was captured at a pixel location. If p is a Pixel object, you may test if its accurate color is red by doing p.getFilterColor() == Pixel.RED.
	+ The following image shows how to calculate the green and red values for a pixel whose accurate color value (color sensor used when taking picture) is blue. Below is a description of the code you should implement.



Description of digital camera filter algorithm:

For each pixel p in image // for each row and column in the image

 Set green count, blue count, and red count to 0

 Set green total, blue total, and red total to 0

 For each of 8 possible direct neighbors

 If neighbor pixel has filter color of green

 Update green total by adding neighbor’s green value

 Add one to green count

 If neighbor pixel has filter color of blue

 Update blue total by adding neighbor’s blue value

 Add one to blue count

 If neighbor pixel has filter color of red

 Update red total by adding neighbor’s red value

 Add one to red count

 If pixel p’s filter color is red

 Set blue in p to (blue total / blue count)

 Set green in p to (green total / green count)

 Else if pixel p’s filter color is green

 Set blue in p to (blue total / blue count)

 Set red in p to (red total / red count)

 Else if pixel p’s filter color is blue

 Set red in p to (red total / red count)

 Set green in p to (green total / green count)

// after going through the entire image, state that all pixels now have all colors

For each pixel p in image

 Set filter color to ALL

**Hint**: Be sure to check the neighbor pixel exists before accessing the position in the array. Otherwise, you will go off the end of the array. For example, if the pixel p is in the top row, then there are no neighbors above it. So, you might have something like if(row > 0) {// get pixel at data[row-1][col]}.

**PART B (1 point per filter below)**

1. Write a class called Mirror that implements the Filter interface which mirrors the image across the vertical midline by reflecting the values on the left this midline to be the values on the right. The pixel values on the right-hand side of the image should be removed; instead, these values are found from the left-side of the image. See the FlipVerticallFilter class for an example of how to flip the picture vertically across the horizontal midline. Test your filter on an image after it has been demosaiced to be sure that it works as expected.
2. Write a class called Greyscale that implements the Filter interface. For each pixel in the image, add together the red, green, and blue values. Then, calculate the average value per pixel by dividing the total by 3. For example, you might find that the average value is 116.5. Then assign each of the red, green, and blue instance variables for that pixel to the calculated average. You may need to cast the average to (int). Test your filter on an image after it has been demosaiced to be sure that it works as expected.
3. Write a class called ShiftUp that implements the Filter interface to shift the image one row up. You should wrap-around the top-most row to be the bottom-most row. (Be sure to copy values to a temporary array if you need to assign those values later.) The effect should be that the picture scrolls upward when the button is pressed. Test your filter on an image after it has been demosaiced to be sure that it works as expected.
4. Write a class called FillBorder that implements the Filter interface to create a black border on the four edges of the image (top, bottom, left, right). The border should be 10 pixels wide (at the top, sides, and bottom) and overwrite whatever color was previously on the image. Black has values of 0 set to red, green, and blue. Test your filter on an image after it has been demosaiced to be sure that it works as expected.
5. Write a class called Brighten that implements the Filter interface to brighten the image (colors get closer to white, remember white has values of red = 255, green = 255, blue = 255). You may decide how to create this effect in your program (for example, add 5 to each color), but it should work for all possible colors. For example, if one of the colors is 255, then you cannot brighten it anymore. Test your filter on an image after it has been demosaiced to be sure that it works as expected.
6. Write a class called MixUp that implements the Filter interface to rotate the red, green, and blue values for every pixel in the image. For example, if a pixel has red=100, green=200, and blue=50, after rotating, the pixel should have red=50, green=100, and blue=200 (red gets blue’s value, green gets red’s value, and blue gets green’s value). (Hint: you’ll need a temporary variable to do the rotation…think of swap in selection sort.) Test your filter on an image after it has been demosaiced to be sure that it works as expected. After 3 MIxUps, you should get back the original picture.
7. Write an original image manipulation filter of your choice. To give you ideas: adjust the colors to the full range of 0 to 255, darken an image, make an image sepia tones (brownish), make an image only black and white, shift the image left or right, dither an image, create a watercolor painting, draw the outlines of items in the image (look for large contrasts in adjacent pixels and edge detection algorithms), etc. Test your filter on an image after it has been demosaiced to be sure that it works as expected.

**Additional Enrichment:**

* Add more image filters to your application.
* You may manipulate the color values for each pixel in the image. Try to make a photograph look like a watercolor painting by altering the colors. You may need to experiment with color combinations to get the desired colors. Implement a filter class for transforming the picture into an image more like a watercolor painting. It may be useful to simulate a paintbrush stroke by mixing nearby colors. Also, you could simulate different paintbrush sizes by randomly selecting the width of the paintbrush used for sweeping colors across the image. (See the Random class in the Java API).
* One important task in the computer science field of vision/graphics is recognizing objects in a picture. An important algorithm to help recognize objects is an edge detector. An edge detector finds edges of objects in a picture. You can find edges in colored pictures by looking for contrasting colors. Write a filter class that outlines the edges of objects in the picture.

**Logistics:**

1. Download the starter BlueJ project. Remember to use good style in your code and include comments.
2. **Follow these tasks when you start on the assignment:** Your first task is to familiarize yourself with the existing application. After downloading and compiling the Java files, run the starter code application by selecting PictureShopConfiguration and running main.

You should see a dialog box pop up. Click on the "load new file" button. This brings up a file browser window. Select one of the .jpg files that you downloaded for the project. You should see the original image on the right and a grainy image on the left. The grainy image is the data collected from the red, green, and blue color filters before the demosaicing algorithm transform the image to full color (which should look like the one you see on the right). There are two filters already in place. The Digital Camera Filter, when executed after button is pushed, creates the mosaic of separate colors. This filter also gets executed (on the left image) when you load a new image. FlipVertical flips the image across the horizontal center when the button is pushed.

**Note:** You can specify a default directory in the PictureShopConfiguration class in the configure method. Currently, the default directory is "Images/", which is where the provided images are stored.

A fully functioning example application is also posted to Moodle. Note, that you need to navigate to a .jpg file to load the image.

1. It might be helpful for you to create the Java documentation for the classes provided, so that you have an easy way to see what methods are available. In the BlueJ application, go to the Tools menu and click on "Project Documentation". This will create a web page with all the public information about the classes included in the project.
2. Look at the FlipVerticalFilter class for an example filter implementation.
3. The debugger is your friend if you are trying to see the values of the red, green, and blue components of a pixel.
4. **For Part A:** Turn in your complete BlueJ project (All Java files and those included in the starter project) as a single .zip file. Name it username\_HW8A.zip.
5. **For Part B:** Turn in your complete BlueJ project (all Java files that you wrote plus the files included in the starter project) and summary report. Zip up all files for the project and your summary report and turn in a single zip file. Name it username\_HW8B.zip.

**Grading Guidelines:**

Your program will be graded on a scale of 0 to 7 in two categories:

* (Part B only) Code Quality: Design, Implementation Style, Comments
* (Parts A and B) Code Operation: Functionality of code and adherence to project specification

Your summary report (Part B only) will be graded on a scale of 0 to 6 based on:

* Correct use of technical vocabulary
* Clarity
* Organization / Headings
* Evidence of Testing

**Report Guidelines and Format: (Use the template for HW1, but answer the questions below)

1. Introduction and System Use:** Provide a screenshot of the user interface of your application and describe briefly what each button does to the image.

**2. System Description:** Describe how the code executes for the image filter that you created for #7 on part B.

**3. Testing and Evaluation:**

**3.1 Specification:** If your program does not meet the specifications, please note these differences. If it meets the specification, write “Application meets all specifications.”

**3.2 User Test:** Have a friend use your application. Since your friend is probably unfamiliar with how a digital camera demosaics a picture, explain to him/her how that happens and press the button to activate this filter to create the full-color image. Then, have your friend try the other filters on the image. If your friend asks how the image is transformed, explain how the image data is changed for that effect.

Did your friend have any questions when using your application? If so, what did he/she ask?

What image transformation did he/she like the best?

Did your friend learn anything new about image transformations and/or digital cameras?

Did he/she have any suggestions for improvements or other image transformations?

**3.3 Results:** Include screen shots of resulting images from each of the seven part B filters. **For at least one of the images, use your own image other than the three images provided in the sample set.**

**4. Conclusion:** What was the most challenging part of this assignment?  What did you learn about images, processing images, and/or digital cameras?

1. How long did you spend on this assignment (both parts A and B)?

2. By typing your name here, you are acknowledging that the code and report you are submitting are your own.

**Appendix:** Copy and paste your code here – just the Filter classes that you wrote for part B (use **Courier** font so the characters line up correctly).