CS 203 HW #7 — Digital Cameras and Image Filters

Due Date: **Part A –** Wednesday, March 26th
Part B – Wednesday, April 2nd

You will write a set of image filters for this homework assignment. The DigiCam company has hired you to design internal algorithms for their digital cameras and algorithms to process images. The first algorithm that you will implement is the internal algorithm 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 images.

Grading

Your assignment grade for both parts of the assignment will be based on:

- Program functionality (90%)
- Code quality (10%)

Background

How Color Images are Represented in a Camera or Computer: A picture is a two-dimensional array of pixels - individual dots 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 intensity of red, blue, and green (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).

How Digital Cameras Work: Digital cameras contain a 2D grid of light sensors to record the intensity for every pixel of the picture. Cheaper cameras (i.e., the consumer ones, which you will be modeling in your project) use color filters over the sensors, so that each sensor captures a single color. Instead of recording a complete color image (with red, green, and blue values), the camera stores the light intensity for just a single color at each pixel location. Software is used to generate the remaining two colors of each pixel. For example, a single sensor may be recording the intensity of green and get a value of 150. The camera then uses software to determine the blue and red intensities for the pixel associated with the sensor.

Digital cameras have Bayer filter patterns in front of the 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 looks like the following:

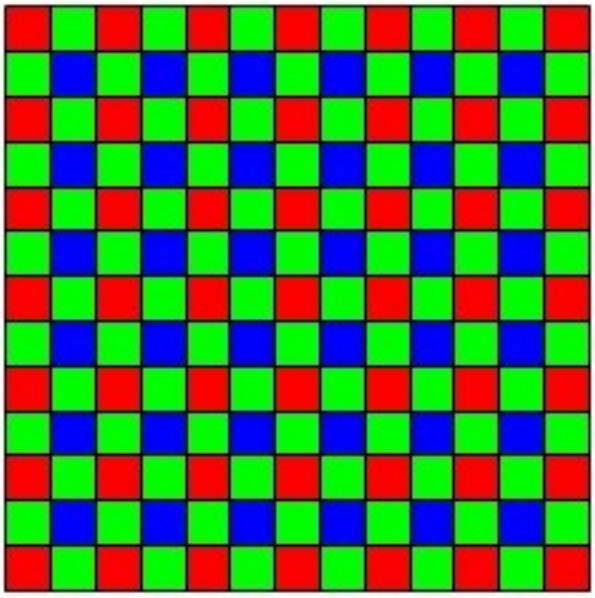


Illustration 1: Note: View this image in color

Once the camera captures the intensities for each pixel, the algorithm embedded in the camera must interpolate the other two color intensities 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 such an algorithm (see Specification below).

References about Digital Cameras:
http://electronics.howstuffworks.com/digital-camera5.htm
http://www.shortcourses.com/guide/guide1-3.html

Starter Code

The starter code is comprised of seven classes, but you only need to modify one of them. However, you will create new Java classes for different tasks outlined in this project. The classes that you should *not* modify are marked below.

SnapShop: *Do not modify this class.* This class creates the graphical user interface for the application and performs the loading operation for images. It controls 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 supports both the user view (the application window) and the engine to support the actions done 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 want 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 rows and the second dimension corresponds to the columns. ***Important:*** You should not call the `getData()` or `setData()` method more than once in any given filter. If you call this method repeatedly in a loop your code will run very slowly.

Pixel: *Do not modify this class.* This class represents Pixel objects. *You will be using this class the most so you should examine it carefully.* Each Pixel has four instance variables: `red` is the value of the red channel; `green` is the value of the green channel; `blue` is the value of the blue channel; and `digCamColor` is the color channel that the digital camera senses for that pixel. Do not create new Pixel objects when modifying the image data; instead, modify the instance variables of the Pixel objects themselves. If you have a Pixel named `currentPixel`, then you can access the red channel by `currentPixel.getRed()` in your code.

Filter: *Do not modify this interface.* Filter is simply an interface that all filter classes must implement. All classes implementing the Filter interface must have a method called filter that does not return anything and takes as a parameter a PixelImage.

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 on 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. **Hint:** *Study this class carefully as it provides a good model for the filters that you write for this assignment.*

SnapShopConfiguration: This class contains the main method that starts the SnapShop program. 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 filter buttons.

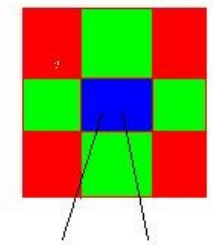
Image Files: Three files are provided as image files in the Images directory of your starter package. You are encouraged to use your own images for this project. Any .jpg image should work fine, but you should start with small images. Be sure to use images that are, at most, the size of the image in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

Part A: Specification

Complete the following steps:

- Create a new Java class called `DemosaicFilter` that implements the `Filter` interface. Every class that implements the `Filter` interface must implement the filter method. I recommend you examine the `FlipVerticalFilter` class for an example of how to do this.

- When you call the `getData()` method on the given `PixelImage`, you are given a two-dimensional array of `Pixel` objects where the first dimension represents the row and the second dimension represents the column. You should transform the image data in your `Filter` class to modify two of the three color instance variables (red, green, and blue) for each `Pixel` object. Each `Pixel` object has a single accurate color channel (one that you should not modify) and has an instance variable (whose value can be retrieved via the `getDigCamColor()` method) that is set to the accurate color channel.



Calculate green channel based on the surrounding green (up, down, left, right) pixels. Calculate red channel based on the surrounding red (corners) pixels.

Example: let's say we have a pixel that was produced by a red light sensor. Then, the green and blue channels will be equal to 0 before your demosaicing filter is executed. Your job is to calculate the values for the two colors that did not get sensed as the camera took the picture for each Pixel in the image. To perform the calculation, look at the surrounding neighbors of pixels, find the ones that sensed the color you are trying to calculate, and average these values. The diagram below is one example of how you can calculate the green and red channels for a pixel whose accurate color channel (color filter used when taking picture) is blue.

Hint #1: This is a difficult algorithm to implement. Take care. Think first. Then design on a piece of paper. Do not write any code until you are certain or this assignment will take a lot longer than it should.
Hint #2: Remember, color values range from 0 to 255.
Hint #3: Also, remember not to go out of bounds (off the array) in the two-dimensional array of `Pixel` objects. For example, `myImage[x-1][y]` will cause a crash if x is zero.

- Add a corresponding button for your filter in the application by adding a line to the configure method in the `SnapShopConfiguration` class.

- Test your filter and make sure it works. When your new Demosaic button is pressed the image on the left should appear virtually identical to the original (shown on the right).

Logistics and Hints for Part A

- Please download the starter BlueJ project (`DigCamStarter.zip`) from the course webpage.
- Run the program by executing the `main` method in the SnapShopConfiguration class.
- When you run the program, you should see a small 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 then 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 demosaicing algorithms transform the image to full color.
- 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 automatically gets executed (on the left image) when you load a new image. Executing it repeatedly has no additional effect.
    - The `FlipVerticalFilter` flips the image across the vertical center when the button is pushed.

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

- It might be helpful for you to view the Java documentation for the classes provided. 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.
- The demosaic algorithm is difficult. There are multiple valid solutions to this program. Some of them can be done in about a page of text. Others will require many pages of code and many hours of extensive debugging. Good design up front will help you avoid frustration later. Do not begin writing code until you have a clear algorithm in your mind. Draw a picture of a small image and execute your algorithm on it by hand. *Think things through.*
- A frequent issue with this algorithm is avoiding array out of bounds errors. If your code attempts to access a Pixel object like this: `data[x-1][y]` does your code guarantee that x is not zero?
- When you are doing modifying the array of pixels, don't forget to write it back to the image via the `setData` method!
- If your program doesn't work right away, *use the debugger* to determine what is going wrong. Watching your program execute will likely lead to some easy insights as to what is going wrong.

Part B: Specification

Complete the following steps:

- Write a class called `FlipHorizontalFilter` that implements the Filter interface (10%). This class should flips the image horizontally across the vertical midline by reflecting the values across this midline. An image with a person's head on the left should have an image of the head on the right after this filter is applied to the image. See the `FlipVerticalFilter` class for an example of how to flip the picture vertically across the vertical midline. Test your filter to be sure that it works as expected.
- Write a class called `DarkenFilter` (15%). This should darken the colors of the entire image by a small but noticeable amount. For most images, you should still see the original picture, just in darker colors. Repeatedly applying this filter will eventually create an image of all black pixels. When testing this filter, be sure to first demosaic the image to the full color version and then apply the filter.
- Write a class called `ShiftLeftFilter` that implements the Filter interface to shift the picture one pixel to the left (20%). The left-most edge should become the right edge of the image, so it looks like it is scrolling. (Be careful in updating values in the array! You may need a temporary array to store a column of Pixel objects.) Clicking on the button for this filter once should shift it left one pixel. Multiple clicks on the button should keep moving the picture left.
- Write a class called EdgeFilter that implements the Filter interface to detect the edges in the image (25%). An edge is a distinct change in the brightness of the

image. The brightness can be measured by taking the average of the color components (i.e., the red, green and blue values). If a given pixel is significantly brighter or darker than at least one of its neighbors then it is an edge. Your edge filter should convert all edge pixels to black and all the other pixels to white. The result should look somewhat like a coloring book page. **Hint:** Be careful not to make any of these color changes until *after* you've identified the edges.

- Write an original image manipulation filter of your choice (20%). Some ideas: adjust the colors to a narrower range, dither an image, create a watercolor painting, blur the image, reverse the colors of the image, make the image appear partially melted, etc.

Logistics and Hints for Part B

- Each time you implement a new filter, you'll need to add it to the SnapShop via the `SnapShopConfiguration` class.
- Use the `FlipVerticalFilter` as a template for creating new filters. This will likely be easier than starting from scratch.

Code Quality (10% each for 7a and 7b)

A good computer program not only performs correctly, it also is easy to read and understand:

- A comment at the top of the program includes the name of the program, a brief statement of its purpose, your name, and the date that you finished the program.
- Variables have names that indicate the meaning of the values they hold.
- Code is indented consistently to show the program's structure.
- The body of `if` and `else` clauses are enclosed in braces and indented consistently, even if they consist of only a single line of code.
- Opening braces are placed consistently, either at the end of the if-statement or directly under the 'i' of `if` or the 'e' of `else`. Closing braces are in the same column as the 'i' of `if` or the 'e' of `else`.
- Within the code, major tasks are separated by a blank line and prefaced by one or more single-line comments identifying the task, e.g., "draw a tree."
- Methods are separated by blank lines and prefaced by a multi-line comment describing what they do and what their parameters mean. (See starter code for examples.)
- Very long statements (such as long print statements or complex boolean expressions) are broken across lines and indented to show their structure.
- Now that you are familiar with loops and methods, your code should not contain redundant or repeated sections.

Turning in this Assignment

- You will turn in this assignment twice (once for Part A and once for Part B) using two different "Turn it in Here" links on the course web page.

- Be sure your name is in the comment header at the top of all your .java files.
- If you did any of the Additional Enrichment for Part B, document this functionality in the respective filter files. Also, add inline comments in `SnapShopConfiguration` that identify your extra filters.
- Compress your entire BlueJ project into a single .zip archive. Be sure your archive includes the directory itself so that, when unzipped, it creates that directory. If you used custom images, make sure they do not total to more than 10MB of data.
- Submit your .zip file via the associated "Turn it in Here" link on the course website.