# CS 203 HW #8 — Digital Images

Due Friday, April 3

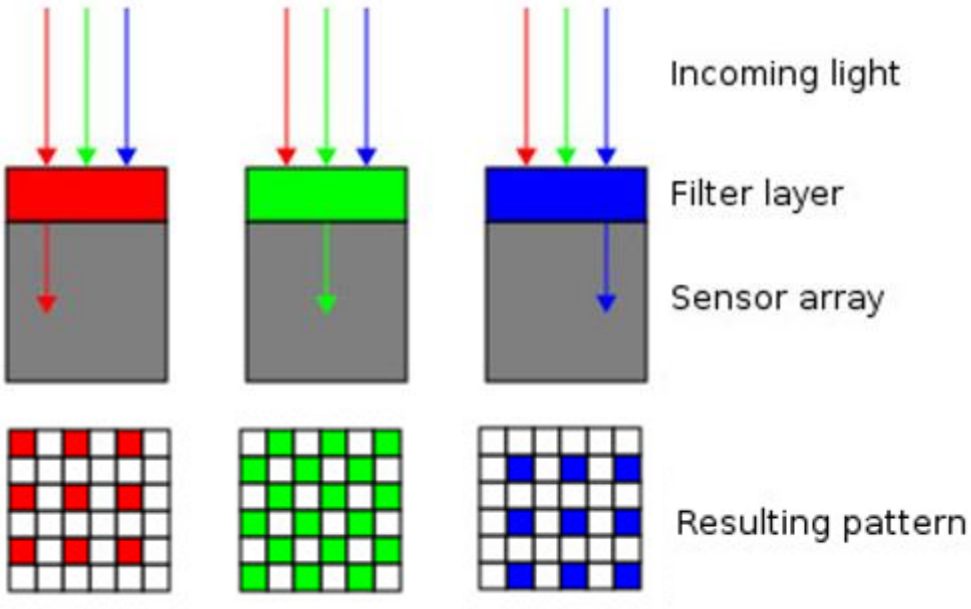
You will write a set of image filters for this homework assignment. The DigiCam company has hired you to design internal algorithms for their digital cameras and algorithms to process images. The main algorithm that you will implement is the internal algorithm for creating a full-color picture based on light intensity levels recorded by the sensors in the camera. Below you will find more information about how such digital cameras work and about the classes that comprise the starter code. Throughout this project, feel free to use your own images.

## Background

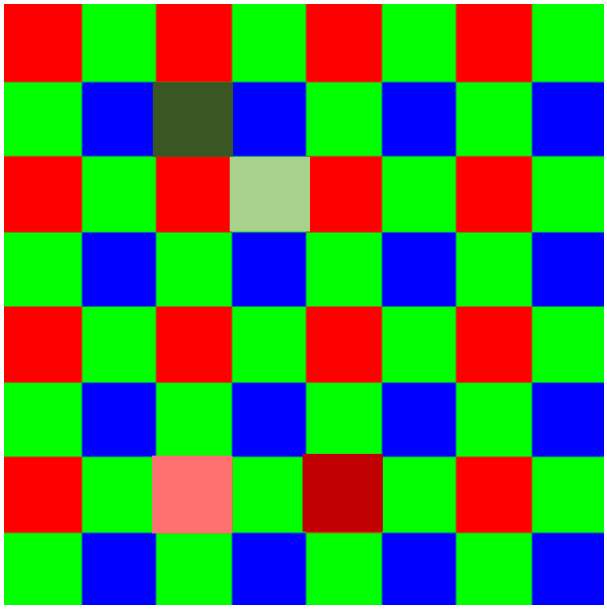
**How Color Images are Represented Digitally:** An image is a two-dimensional array of pixels, which are individual dots that make up the image. Each image has a specific resolution - the width and height in pixels of the grid that makes up the image. Each pixel has three components describing the intensity 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 (as 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 particular sensor may record only the intensity of green color at that pixel location. The camera then calculates the blue and red intensities for that pixel based on the values recorded in the pixels nearby.

One type of filter used in digital cameras is the Bayer filter. Because the human eye is most sensitive to green, Bayer filters use roughly 50% green sensors, 25% blue sensors, and 25% red sensors. Bayer filters are arranged in a pattern like this *(see next page)*:




The resulting patterns are combined into a single 2D array like this:



As a result, the camera has one correct intensity (red, green or blue) and two unknown intensities for each pixel in the image. The program embedded in the camera must estimate the value of the two unknown color intensities for each pixel by examining known intensities nearby. These algorithms are called de-mosaicing algorithms because they convert the mosaic of separate colors into an image of true colors. Your main task is to implement such an algorithm *(see specification below)*.

## Starter Code

The starter code is comprised of seven classes, but, for the most part, you won't need to modify them. Instead, you will create a new Java class for each of the 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.

**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 (a 2D array of Pixel objects) and modify the data. Important methods in this class are `getWidth()`, `getHeight()`, `getData()`, and `setData(Pixel[][] data)`. ***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.* Each instance of this class represents a single pixel of the image. *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 component; `green` is the value of the green component; `blue` is the value of the blue component, and `digCamColor` tells you which component you know is correct. Do not create new `Pixel` objects when modifying the image data; instead, modify the instance variables of the existing `Pixel` objects. If you have a `Pixel` variable named `currentPixel`, then you can access the red component by calling `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 converts a full-color image file to the corresponding data that would be gathered by the sensors in a camera with a Bayer filter.

**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 a new filter, add it to the list of filters in the configure method. Look at the configure method in this class to see how to add new filter buttons.

**Image Files:** Two 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.

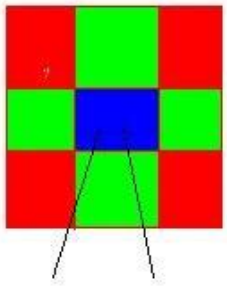
## Specification

**Note:** Each of the filter classes are independent, so you can implement them in any order. If you get stuck on one filter, you may find more success on another filter. Then you may be able to take your learning from completing a filter to go back to the filter you got stuck on.

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.
- Implement the `filter()` method so that it demosaics the image. Your method should start by calling the `getData()` method on the given `PixelImage` in order to get a two-dimensional array of `Pixel` objects representing the image. 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. Study the `Pixel` class to understand how to use it. Each `Pixel` object has a single accurate color component (one that you should not modify). The accurate component can be determined by calling `getDigCamColor()`. The last line of code in your filter method should be a call to `setData()`, which writes the array of Pixel objects back to the image.

**Example:** Let's say we have a pixel that was produced by a blue light sensor. This means that the green and red channels are unknown (set to a default value of zero) before your demosaic filter is executed. The filter's job is to estimate the correct values for the red and green components of this pixel. To perform the calculation, the filter should look at the surrounding neighbors of pixels, find the ones that sensed the color you are trying to calculate, and average these values.



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

- Add a corresponding Demosaic button for your filter in the application by adding a line to the configure method in the `SnapShopConfiguration` class.
    - Test your demosaic 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). The colors should be very similar, but the image may be a bit blurry or grainy. It is ok if your image no longer resembles the original if you press the Demosaic button repeatedly.
- Write a class called `FlipHorizontalFilter` that implements the Filter interface. This class should flip 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 as an example of how to flip the picture vertically across the horizontal midline. Test your filter to be sure that it works as expected. You should be able to press this button repeatedly to make the image flip horizontally each time.
- Write a class called `DarkenFilter`. This should darken the colors of the entire image by a small but noticeable amount. 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 `ShiftRightFilter` that implements the Filter interface to shift the picture one pixel to the right. The right-most column of pixels should become the new left most edge of the image, so it looks like the image 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 right one column of pixels. Multiple clicks on the button should keep moving the picture right.
- Write a class called `EdgeFilter` that implements the Filter interface to detect the edges in the image. 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 **eight** 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. You can experiment with the difference in brightness that yields a good result, but 20 seems to be a reasonable number.
- Write an original image manipulation filter of your choice. 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. Be sure to explain in the class header comments what this filter is supposed to do. Also, let the reader know whether the output should change if the button is pressed repeatedly.

## Logistics and Hints

- 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 horizontal 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.
- 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.**
- 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 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.

## Code Quality

A good computer program not only performs correctly, it is also 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.
- The comment header should also include a list of known bugs or deficiencies, or a statement that the program has no known deficiencies.
- Variables have names that indicate the meaning of the values they hold and use conventional case.
- 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 a statement or on a line by itself directly after it.
- Within the code, major tasks are separated by a blank line and prefaced by one or more single-line comments identifying the task.
- 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.
- Your program does not contain extraneous or commented out code. It does not contain out-of-date or irrelevant comments.
- When a constant value is used repeatedly in your program, declare a `final` variable with a descriptive name to contain that value. Use it instead of a literal.

## Turning in this Assignment

- Be sure your name is in the comment header at the top of all of the .java files you created.
- 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, include a copy of those images, and make sure the resulting .zip file is smaller than 10MB in size.
- Submit your .zip file on Moodle.