Assignment Overview

The assignment given was to implement a number of basic (and not so basic) filters and image effects – such as removing the red component, reflecting the image across an axis, and implementing a threshold filter. For this specific assignment, the nitty gritty details were largely dealt with by provided code (including the loading, saving, and representing of images), and the primary goal was to familiarize oneself with the Java programming language as well as some basic data manipulation concepts.

As I was already largely proficient in understanding Java code and syntax, my personal goal during this assignment was to explore some interesting ways to implement the requested filters, as well as explore a number of more interesting and curious filters (such as smooth, denoise, and edge detection).

Solution Design

As the actual code for loading, saving, and handing the application of image effects was already provided (and could not be changed), the primary focus of my design was on novel and functional ways to implement the various filters which the assignment requested. Each and every effect had its own class, derived from a small utility class called SimpleImageEffect which simply inherited ImageEffect and provided an overloading to specify a custom name for the effect (as well as automatically instantiating the params array).

The interesting part about my chosen implementation was that I chose to make wide use of Java 8 Lambdas, which allow for a (somewhat restricted) shorthand syntax for defining and passing around anonymous functions. With the ability to do this, I could adopt a limited functional style in my implementations, whereby I described the transformation in the image in terms of functions which were mapped over each pixel in the image. In general, there were three types of “filters” or “mapping functions” for which I could describe all of the image transformations:

- **Pixel Mapping (Pixel -> Pixel)**: Pixel mappings take a function which maps an input pixel to an output pixel, and then simply apply that function to every single pixel in the image.
• **Positional Mappings (Position -> Position):** Positional mappings take a function which maps an input position to an output position; the input position represents a position in the output image, and the output position represents the related position in the original image. In order to determine the value of a pixel in the output image, its position is passed to the mapping function (to produce the *result position*) and the pixel is set to the value of the pixel at the result position in the original image. In short, for every position in the output image:

\[
Output[\text{position}] = Input[\text{position} \_\text{mapping}(\text{position})]
\]

The reason for the seemingly curious choice to map output positions to input positions (instead of the more intuitive opposite, of mapping input positions to output positions), is that the mapping function is by nature one-to-one. Thus, for scenarios where the size of the output image varies from that of the input image, or a pixel in the input image may occur multiple times in the output image, the input to output mapping would not work as it may not be one-to-one. The only option, then, is to iterate through the positions of the output image as those are guaranteed to be unique.

• **Neighborhood Mappings (Position -> View, View -> Pixel):** Neighborhood mappings are the most powerful and complex of the three mapping types, and are constructed in two steps. For each pixel:

1. A *view* (represented by the ImageView class) is constructed using a Position -> View function which takes the pixel’s position as input and produces a view of the image which the next function will use. A view is simply a constrained set of pixels which the filter will operate on, such as all of the pixels in a 3x3 square around the current pixel.
2. The pixels contained in the view are aggregated or otherwise operated upon using a View -> Pixel function which takes the input view, performs some aggregating operation on it (such as average, median, min, max, etc.), and produces the final resultant pixel in the output image.

After reading the basic descriptions of each of these three mapping types, one may wonder: *what on earth could the advantage of these complicated sounding filters be?* The short and simple answer is that these filters are highly generic - they eliminate and encapsulate the majority of the code for actually iterating or manipulating the image itself (the *How*), and instead allow the users to describe only how the image needs to be manipulated (the
What). To take an example, see the different implementations of the apply function for the Invert class:

**Original**

```java
public int[][] apply(int[][] pixels) {
    int width = pixels[0].length;
    int height = pixels.length;
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            pixels[y][x] = pixels[y][x];
        }
    }
    return pixels;
}
```

**With Pixel Mapping Filter:**

```java
public int[][] apply(int[][] pixels) {
    return FilterUtils.applyPixel(pixels, pixel -> ~pixel);
}
```

As can be seen from this example, the actual details of the iteration and array manipulation are removed from the filter implementation - the entirety of the filter is described with the lambda “pixel -> ~pixel”! The utility of these filters is increased by the fact that they deal with uneven image sizes and other out of bounds issues automatically.

In order to support decent lambda operations, a number of custom functional interfaces which describe the actual function had to be created - resulting in a number of new files and classes to simply describe a single function (one of Java 8’s shortcomings). Several utility classes, such as PixelUtils and FilterUtils, implement the actual code which executes these lambdas on the images, as well as code for manipulating Pixels and views.

The description of the remaining filters will use one (or more) of the previous three mapping functions in order to implement its functionality.
Filters Implemented:

- **Invert (Pixel Mapping):** The invert filter simply maps each pixel to its binary inverse in order to invert the colors of the image, via the tilde (~) operator.
- **NoRed (Pixel Mapping):** The no red filter maps each pixel to a pixel where the red component has been replaced with 0.
- **NoGreen (Pixel Mapping):** The no green filter maps each pixel to a pixel where the green component has been replaced by 0.
- **NoBlue (Pixel Mapping):** The no blue filter maps each pixel to a pixel where the blue component has been replaced by 0.
- **RedOnly (Pixel Mapping):** The red only filter maps each pixel to a pixel where the green and blue components have been replaced by 0.
- **GreenOnly (Pixel Mapping):** The green only filter maps each pixel to a pixel where the red and blue components have been replaced by 0.
- **BlueOnly (Pixel Mapping):** The blue only filter maps each pixel to a pixel where the red and green components have been replaced by 0.
- **Black And White - Light (Pixel Mapping):** This black and white filter maps each pixel to a grayscale pixel by taking the average of the highest valued component and lowest valued component, and setting all components to that average.
- **Black and White - Average (Pixel Mapping):** This black and white filter maps each pixel to a grayscale pixel by taking the average of all three color components, setting all components to that average.
- **Black and White - Luminosity (Pixel Mapping):** This black and white filter maps each pixel to a grayscale pixel by taking a weighted average of the color components, setting all components to that average. Setting the weights to values which reflect human sight (Red = 0.21, Green = 0.72, Blue = 0.07) usually produces the best black and white filter.
- **BlackAndWhite (Pixel Mapping):** This filter is simply a duplicate of the Black and White - Luminosity filter where the inputs have been fixed to the recommended values of 0.21, 0.72, and 0.07.
- **Vertical Reflect (Positional Mapping):** This positional mapping reflects the image across the horizontal axis, by mapping the input rows $r$ to $(image.height - r)$.
- **Horizontal Reflect (Positional Mapping):** This positional mapping reflects the image across the vertical axis, by mapping the input column $c$ to $(image.width - c)$.
- **Variable Grow (Positional Mapping):** This positional mapping filter takes as input the integral ratio by which to scale up the image,
constructs an output image which is width * scale wide and height * scale high, and then maps every output pixel at \((r, c)\) to \((r / scale, c / scale)\).

- **Grow (Positional Mapping):** This filter simply is a duplicate of the Variable Grow function, with the scaling ratio fixed at 2.

- **Variable Shrink (Neighborhood Filter):** This neighborhood filter takes as input the integral ratio by which to scale down the image, constructs an output image which is width / scale wide and height / scale high. For every position \((r, c)\) in the output image, a view of the rectangle \((r * scale, c * scale)\) to \((r * scale + scale, c * scale + scale)\) is constructed and the component-wise average is taken to produce the final pixel.

- **Shrink (Neighborhood Filter):** This neighborhood filter is a duplicate of the Variable Shrink filter, with the shrink factor fixed at 2.

- **Threshold (Pixel Mapping):** This pixel mapping takes as input a threshold value, which describes the minimal value a component must have to continue to exist in the image. Every component of every pixel is compared to the threshold – if it is greater, it is set to 255; otherwise, it is set to 0.

- **Smooth (Neighborhood Filter):** This neighborhood filter “smoothes” an image by, for each pixel, constructing a square of a provided radius around each input pixel and then taking the average of each color component of all the pixels in that square.

- **Sharpen (Neighborhood Filter):** This neighborhood filter “sharpens” an image by using a convolution matrix on each pixel (implemented via a neighborhood filter). The center of the 3x3 convolution matrix is 9, with all other elements being 0.

- **Edge Detection (Neighborhood Filter):** This neighborhood filter employs the Sobel method for edge detection by applying two different convolution matrixes, \(G_x\) and \(G_y\), to the image separately to produce two grayscale images which show the horizontal and vertical edges respectively. The final image is produced by combing each related pixel in each of the two interim images using the function

\[
R = \sqrt{(G_x)^2 + (G_y)^2},
\]

where \(R\) is the intensity of the resultant pixel. See [http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/edge.html](http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/edge.html) for information on how the filter works in detail.

- **Denoise (Neighborhood Filter):** This neighborhood filter “denoises” an image by, for each pixel, constructing a square of a provided radius around the pixel and taking the median of each color component of all of the pixels in the square.
• **Erode (Neighborhood Filter):** This neighborhood filter “erodes” the color in an image by, for each pixel, constructing a square of a provided radius around the pixel and taking the minimum of each color component of all of the pixels in the square.

• **Dilate (Neighborhood Filter):** This neighborhood filter “dilates” the color in an image by, for each pixel, constructing the square of a provided radius around the pixel and taking the maximum of each color component of all the pixels in the square.

• **Darken (Pixel Mapping):** This pixel mapping darkens the image by multiplying every component of every pixel by a provided darken percentage (which is less than 1), to obtain a pixel which has proportionally reduced component values (and is thus darker).

• **Brighten (Pixel Mapping):** This pixel mapping brightens the image by multiplying every component of every pixel by a provided brighten percentage (which is greater than 1), to obtain a pixel which has proportionally increased component values (and is thus brighter).

• **Horizontal Mirror (Positional Mapping):** This positional mapping mirrors the left half of the image across the vertical meridian line, by doing nothing for all \( r < \frac{width}{2} \) and mapping \( r \rightarrow \frac{width}{2} - (r - \frac{width}{2}) \) otherwise.

• **Vertical Mirror (Positional Mapping):** This positional mapping mirrors the top half of the image across the horizontal meridian line, by doing nothing for all \( c < \frac{height}{2} \) and mapping \( c \rightarrow \frac{height}{2} - (c - \frac{height}{2}) \) otherwise.

• **Transpose (Positional Mapping):** This positional mapping transposes an image by simply swapping the row and column, eg mapping \((r, c) \rightarrow (c, r)\).

• **Shift Components (Pixel Mapping):** This pixel mapping shifts the components in each pixel one to the left, mapping (red, green, blue) to (green, blue, red).

• **Shift Rows (Positional Mapping):** This positional mapping shifts each row in the image by one more than the previous row (with the first row being shifted none), creating an interesting diagonal effect. This can be done by mapping \((r, c) \rightarrow (r, c + r)\).

• **Shift Columns (Positional Mapping):** This positional mapping shifts each column in the image by one more than the previous column (with the first column being shifted none), creating an interesting diagonal effect. This can be done by mapping \((r, c) \rightarrow (r + c, c)\).

• **Oil Painting (Neighborhood Filter):** This complex neighborhood filter will take as input two parameters: the radius of the square to construct for each pixel, as well as the number of intensity levels.
each pixel, a square of the provided radius is constructed and the intensity of each pixel is calculated, by computing it’s intensity in grayscale, dividing it by the maximal value (255), and then multiplying it by the number of intensity levels and rounding in order to produce a relative intensity level of the pixel from 0 to the max intensity level. The frequency of each intensity level in the square is then computed, and the average of the pixels having the most frequent intensity level is computed as the final pixel. See http://www.codeproject.com/Articles/471994/OilPaintEffect for the source of inspiration for this filter.

- **Cartoon Filter (Neighborhood Filter):** This neighborhood filter employs a similar method as the oil painting to produce “intensity levels” for each pixel. In order to apply the “cartoon effect” via edge detection, a 3x3 square is constructed around each pixel and the relative differences between opposing pixels around the center pixel are summed up. If they are greater than a provided threshold, the pixel is set to black. Otherwise, it is set to white. See http://softwarebydefault.com/2013/06/29/oil-painting-cartoon-filter/ for the source of inspiration for this filter.

### Post-completion Considerations

The assignment was highly educational, overall, primarily as in exercise in seeing how relevant functional programming ideas could be in a Java setting. The results were (mostly) positive, something which was hopefully reflected through the relative conciseness of many of the implementations of image effects and filters.

The program, though likely not bug free, is not particularly picky in the input it processes or functions it employs. There are no requirements on image size, shape, or characteristics for (to my knowledge) any of the filters, save a few filters where having a picture which is not 0x0 in size (though why you would want to process that of course is beyond rationale). The overall quality of the program is also of course debatable, but the usage of lambdas and other functional concepts greatly increased readability and encapsulation, which went a long way towards long term maintenance (was this project to be of any importance, that is) when combined with decent commenting and a moderately fleshed out class tree.

Some of the most amusing and interesting parts of the program were seeing how some of the more complex filters (such as the edge detection
filter) were implemented, particularly in how concisely relatively-complex transformations could be implemented. Many of the extra filters added (which, unfortunately for the length of this report, was quite a few) were either implementations of these interesting filters, or random filters from silly “what if” questions.

The major issue which I came across while looking at how I could implement the filters was through the existing framework I had to write the transformations in. It worked, of course, but the usage of reflection to load all of the effects from a single file (which had questionable support for spaces in the path) forced me to write a class for every single effect where a single class with differing instance variables would suffice. The restriction that the given classes could not be modified also hampered some creative license, as more fancy effects which did things such as combine 2 images (or other utilities, such as an “Undo” button), could not be implemented.

**Testing Specifications**

The primary method of testing for this assignment was standard black box testing, in which images (particularly a certain picture 11) were subjected to every filter and the results were checked using external image editing tools such as Paint.NET and GIMP to ensure that they actually worked properly. For simple color modifying filters, the simple color gradient and solid color images were used to check functionality first; for more complicated filters, a range of pictures (such as picture 11) were tested and then the effect compared to implementations in external image editing tools.

In order to check for technical correctness, a small suite of Junit tests was also composed for each of the required filters. Each test simply generated a large number of random matrixes/images, applied the filter to each image, and then checked to see that the output image satisfied the invariants which the filter was supposed to implement. For example, the test for the NoRed filter checked to see that all output pixels had a red component of 0. The random generation of test matrixes served to emulate a broad range of input cases which the program may encounter in normal operation, and also to generate random cases which may be otherwise unaccounted for.
The testing process was of course not exhaustive, and greater rigor could have been implemented with explicit edge case testing, but it offered a decent indication as to the functionality of the program.

Sample Images - Inputs

![Sample Images - Inputs](image1.jpg)

Sample Images - Outputs

![Sample Images - Outputs](image2.jpg)

(Repeated Sharpening)  (Oil Painting + Cartoonify)
(Oil Painting)