1 Problem Statement

Ever since humans have been able to capture images, editing was a crucial part of the process of enhancing the quality and artistry of images. Olden age photo editing was done in the dark room and physical manipulations while developing film ended up manifesting themselves in the image. In the digital age, modern social media such as Snapchat and Instagram emphasizes image manipulation through unique image filters intended to enhance user experience. In this assignment, my primary objective was to understand and implement a lightweight, modular image manipulation program that performs simple yet powerful and useful transformations. Furthermore, I wanted to foster good programming practices (specifically in Java) and learn how to make these practices an integral part of my solution design workflow.

2 Solution Design

When designing solutions in this program, I wanted to be very careful about how I wrote code and re-used it. This led to several modifications along the way that led to a satisfactory result.

2.1 Modularity

A key part of the Solution Design for this particular assignment involved modularity. Specifically, I wanted to make sure that it would be easy to add new manipulations and doing so would not interfere with the other manipulations. Furthermore, I wanted to provide a common set of functions or create code that would be applicable/usable in multiple scenarios. To do this, I classified the problems into two types:

2.1.1 Local Manipulation

Local Manipulation of images requires performing some sort of pixel manipulation referencing multiple pixels in the image to draw a conclusion about a singular pixel. These typically algorithms can be implemented easily with a copy of the original image because pixels values change during traversal. It might also be possible to find a neat way to do these types of manipulations without creating a copy of the image data. Neighborhood algorithms are classified under this type of problem.

2.1.2 Direct Manipulation

Direct Manipulation algorithms are where manipulations can be performed with a traversal (or multiple traversals) over the image. These problems are typically do not require a copy of the original image and do a "compute-n-replace" routine with the directly.

2.2 Evolution of Codebase

Though it may seem like my codebase currently repeats a lot of code, it used to be much worse in the beginning. By neatly packaging instructions into routines (like outsideBorder), I was able to save time in the debugging process.
2.3 Assumptions

Other than assuming that input images were at least When programming, I tried to account for as many cases as possible and made minimal assumptions about input images. I also was able to consider a few edge cases for the various manipulations. At times, fidelity of the output image is sacrificed for correctness of the desired image effect.

3 Manipulation Solutions

3.1 NoRed, NoGreen, NoBlue, RedOnly, GreenOnly, BlueOnly

These problems are direct manipulation problems and work directly with the colors and the makePixel() method. When a particular color needs to be removed, we traverse over every pixel and set that particular color value to 0 while retaining the other color values. When a color needs to be kept, we traverse over every pixel and retain that particular color value while setting the other two values to 0.

3.2 BlackAndWhite

This problem was a direct manipulation problem. All that I had to do was find the average of the red, green, and blue components of the images and then use that averaged value for all three of the components of the new grayscale pixel. This ends up working because if the values are all the same for each of the RGB components, then the colors lies on the white/black spectrum.

3.3 Vertical Reflect, HorizontalReflect

These two problems were local manipulation problems and required a copy to keep track of the original image before the reflection was applied. To perform the manipulation, the manipulated pixel was read in reverse from the copied data and assigned to the new location (top to bottom/left to right for vertical and horizontal, respectively).

3.4 Grow

I considered the grow problem as a direct manipulation problem because it relays on single pixel data for enlargement (and although the argument can be made for it being a local manipulation problem, it ends up not mattering for implementation). What we do here is we create a new array to serve as the container for the output and take a single pixel value and place it in 4 locations. Here, we had to be careful to increment the loop properly and keep track of the pixel positions to achieve the intended result. When keeping track of the non-enlarged pixel position using a single integer, we check if we have reached the bound every time so that we know when to reset and move on to the next column/row.

3.5 Shrink

I considered the shrink problem as a local manipulation problem because I needed to analyze a pixel region to draw a conclusion about a singular pixel in the resulting image. Here, I created an new array that would house the resulting shrunk image. I would then traverse the original image with a similar tracking system that I mentioned in the description for the Grow problem. This tracking system would allow me to both compute the average from the original image and place this averaged pixel in the smaller, resulting image.
3.5.1 Shrink Edge Cases

In cases where there is an odd number of pixels in either the height or widths of the image, integer division loses some of the information fidelity. It doesn’t end up affecting the experience that much because in most images that need to be edited, one row or column of pixels can be compromised. Here I was also careful here to consider cases where the image would be $1 \times n$ or $n \times 1$ in dimensions. In this case, I made sure to average and cut in half the image every time in the dimension that has more than 1 pixel.

3.6 Threshold

This was a direct manipulation problem because it didn’t require any access of other pixels during a given standard traversal. Here I setup three flags to keep track of which RGB components of a given pixel are greater than the thresholds. If a particular threshold is exceeded, that value will be set to 255 and if not it will be set to 0. This particular class also makes use of the params feature of ImageEffect. The threshold is taken as input and has a default value of 127 (half of 255).

4 [Extra Credit] Karma

4.1 Smooth, DeNoise, Erode, Dilate

These particular effects were all considered local manipulation problems. I setup a method called outsideBorder to make sure that there were no out-of-bounds array access calls. On top of this, I setup a routine to perform the neighborhood computations so that I would be able to reuse the code for each of the four scenarios. I then performed the required analysis specific to an effect and returned the output.

This part of the assignment actually reminded me a lot of convolutions and kernel functions. While in this case, our kernels were not that complicated, it was nice to see that we could easily modify our code into something much more powerful (with applications in areas like Computer Vision).

![Image from TikZ 2D Convolution Github Repository](image)

5 Problems Encountered

Naturally, I ran into many errors while writing the project. I ended up being able to fix many of the issues that I had using the debug methodology described in one of the subsequent sections. Here are some of the issues that I had:
5.1 Boundaries

5.1.1 General

I had to make sure that I was traversing the right bounds when manipulating images. Many times, I would be out-of-bounds because of. This occurred when I was both when I was working on the neighborhood filters for Karma. I had to make sure that I wasn’t trying to access a pixel that didn’t exist.

5.1.2 Pixel Shifts

This error was more subtle than the general boundary error. What had occurred was an off-by-one bug that ended up not resulting in a runtime error but instead shifted the whole image to the right or left. After identification of the problem, this bug was relatively easy to fix.

5.2 Edge Cases

Some edge cases, such as the one mentioned in the description for Shrink, gave me a hard time because I tried to come up with a generalized solution that would fix the problem and account for all cases. This ended up not being efficient and I ended up just writing in cases for when those particular scenarios occur.

5.3 Debug Methodology

5.3.1 Print Statements

My first reaction to a runtime error or an incorrect output is to print out the various relevant values. Because we were dealing with large images in this project, this quickly became very cumbersome to do and I ended up only using this debugging technique for smaller issues relating to bounds and such.

5.3.2 J-Grasp Debugger

In the J-Grasp IDE, the Debugger is very easy to use and still powerful enough to visualize the modifications being made to the various arrays in real time (or while stepping through at breakpoints). This was a major upgrade from the print statements because it allowed me to address more systematic problems effectively.

6 Testing Methodology

To test my program, I used the sample images and applied filters manually (in the black box testing approach). I ended up using a bit of logic (e.g. if I apply NoRed, NoGreen, and NoBlue I should get a black image) to make sure that various filters were working. Furthermore, it became evident that some types of pictures were more useful than others when it came to testing the image. Here are the images I mainly relied upon and considered "useful" in the test set:

- **NoRed, NoGreen, NoBlue**: 20.jpg, 18.jpg, 17.jpg (red, green, and blue), 13.jpg, 14.jpg,
- **RedOnly, GreenOnly, BlueOnly**: Logic Check, 13.jpg, 14.jpg
- **BlackAndWhite**: 11.jpg (Dr. Lin’s Face)
If I were to test someone else’s code, I am fairly confident that this routine of images would work. I would also include more images that are of non-square dimensions. Sometimes this is a bug that may not be caught (because of the nature of the test set). Other testing approaches that would increase my confidence would include comparing the results that I produce to results that are produced by libraries included in Java or even stepping through each particular effect in the JGrasp debugger to make sure that every line of code is being put to good use.

7 Interesting Results Gallery

Before:
After:
Tower Image: 3 Denoise, 3 Erode, 3 Smooth
Holi Image: 1 Threshold, 1 Smooth, 3 DeNoise, 1 Vertical Reflect, 1 No Green
Mysore Palace Image: 2 Dilate, 1 Threshold, 1 Invert, 3 DeNoise, 1 Blue Only
8 Further Improvements

I thought that this project was an amazing starting point for some really cool things that can be done with image manipulation. Recent research focusing on computer vision uses some of the same fundamental concepts and it’s great to see that the framework provided is so easy to use and rapidly prototype filters and effects. Overall, this was a wonderful project and it helped me learn a lot about style and precision in writing code.