Marie Cho
mhc2323
September 5, 2024
CS314H

Prog 1 Report

1 Overview

Goals

This lab was focused on developing code to create various filters and effects to change the digital appearance of images. Through implementation of such alteration techniques, I explored array manipulation, the fundamentals of RGB color values, and expressed my creativity through my resulting digital images. By the end of the lab, I successfully created multiple filters that applied specific alterations to images.

Insights and Surprises

A few insights I gained from this lab were about how the RGB values worked together to create all the different shades and tones of colors. As I completed filters such as *NoRed()* or *Threshold()*, I discovered that even just altering one value to 0 or 255 can drastically change an image's coloration. Furthermore, I was able to build algorithms that efficiently and effectively switched pixels from either side of the image when implementing the Reflect effects. Finally, this lab also pushed me to consider edge cases, such as handling 1 x 1 images or null arrays, and I developed thoughtful solutions to ensure the code could handle these scenarios appropriately.

2 Solution

Assumptions

(a) All pixels are created using a standard RGB format.
(b) Each color channel (R, G, or B) has a color depth of 8 bits, ranging from 0 - 255.
(c) All input images do not exceed screen dimensions (MacPro 14 inch: 3024 px x 1964 px).
(d) All input images are stationary (not a GIF or video).
(e) Each pixel in the image (element in the array) is non-null.

Design Considerations

When designing the solution for the image effects, I prioritized efficiency due to the fact that we needed to use at least one, if not two, for loops to access all pixels inside the 2D int array *pixels*. This is evident in my Reflect effects, where I used a two pointer approach. I also considered the simplicity and flexibility of the code by taking minimal steps and combining lines of code. This is evident in the *Shrink()* and *Threshold*() effects, where I used ternary operators to avoid excessive if-else statements. These choices improved both readability and ease of debugging.

Algorithms and Implementation

Each effect was designed with edge cases in mind. If the input array is null or has a zero height/width, the original array is returned. For dimension-altering effects, if the height is non-zero but the width is zero, the returned array is adjusted accordingly by doubling or halving dimensions. Specific implementations for each effect are described below:

(a) NoRed, NoGreen, NoBlue(): Using a nested for loop, for every element in the array, set the respective value (R, G, or B) to 0.

(b) RedOnly, GreenOnly, BlueOnly(): Using a nested for loop, for every element in the array, set the respective values (G and B, R and B, or R and G) to 0.

(c) VerticalReflect(): Using a for loop, iterate through each row in the array and set two pointers, one at the start and one at the end of the row. Using a while loop, swap the two elements at the pointers. Increment the start index and decrement the end index until the method breaks out of the loop.

(d) HorizontalReflect(): Using a for loop, iterate through each column in the array and set two pointers, one at the start and one at the end of the column. Using a while loop, swap the two elements at the pointers. Increment the start index and decrement the end index until the method breaks out of the loop.

(e) BlackAndWhite(): Using a nested for loop, for every element in the array, average the RGB values of the pixel. If this average was greater than 127, set it to 255 (white); otherwise set it to 0 (black). Set the RGB values to this average.

(f) Grow(): Create a new array that is twice the height and width of the *pixels* array. Using a nested for loop, for every element *pixel[i][j]* in the original array, copy the element into a 2 x 2 block for the new indices and neighboring positions (*pixel[i*2][j*2]*, *pixel[i*2 + 1][j*2]*, *pixel[i*2][j*2 + 1]*, and *pixel[i*2 + 1][j*2 + 1]*).

(g) Shrink(): Create a new array that is half the height and width of the *pixels* array. If such a dimension is not possible (< 1), use 1 as the dimension. Using a nested for loop, for every element in the new array, a new pixel is created using the average RGB values of the corresponding 2 x 2 block in *pixels* (*pixel[i*2][j*2]*, *pixel[i*2 + 1][j*2]*, *pixel[i*2][j*2 + 1]*, and *pixel[i*2 + 1][j*2 + 1])*, except those not within the height or width range.

(h) Threshold(): Using a nested for loop, for every element in the array, check to see if each of the RGB values > param value (ex: 127); if yes, set the respective value (R, G, or B) to 255; otherwise set the respective value (R, G, or B) to 0.

3 Further Discussion

Scope

This project has real-world applications in fields such as UI/UX design, graphic design, and photography. Image manipulation techniques developed in this lab such as filtering, reflecting, and resizing are suitable for basic image editing applications. Since this lab was also a “black

box model," applying such effects can also be leveraged to create a variety of modified images to enhance datasets used for training machine learning algorithms.

Yet, this program is also limited by the number of effects that were developed. While this project only implements 12 filters, much more would be necessary in the real world, such as saturation, vibrance, brightness, exposure, crop, etc. Furthermore, I was only able to manipulate the RGB values for color alteration, but using other techniques such as HSL (hue, saturation, lightness) would open options for more specific and variety of changes.

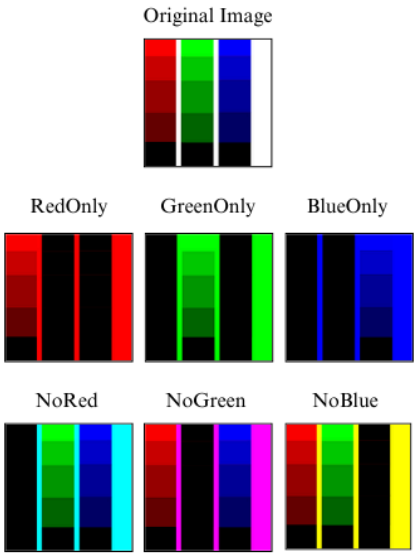
Solution Quality

The implemented methods succeeded in the following:

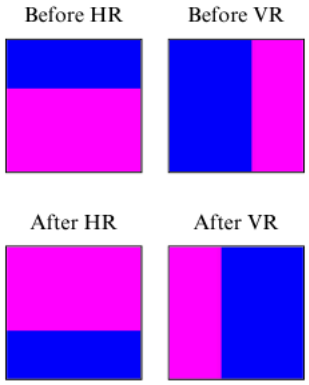
(a) Outputs had consistent results with no errors.
(b) Methods were able to handle edge/unique cases, including nulls and dimensions of 0 or 1.
(c) Methods performed within a reasonable time frame.
(d) Outputs maintained proportionality and quality, except for *Grow()* and *Shrink()*, which may compromise the quality due to pixel duplication or condensation.
(e) Code is clean, simple, and implements encapsulation for flexible changes in the future.

Interesting Results

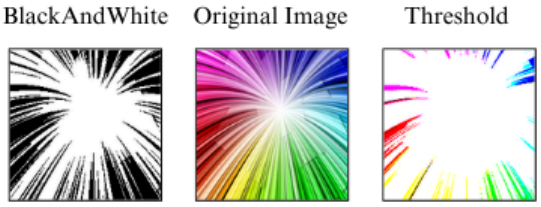
Shown below are the before and after images when each of the color manipulation effects is applied. As evident, in the top-left image, the *RedOnly()* filter leaves only the red tones, blacking out all other colors. Similarly, in the bottom-left image, the *NoRed()* filter removes all red tones resulting in a cooler, blue-green appearance.



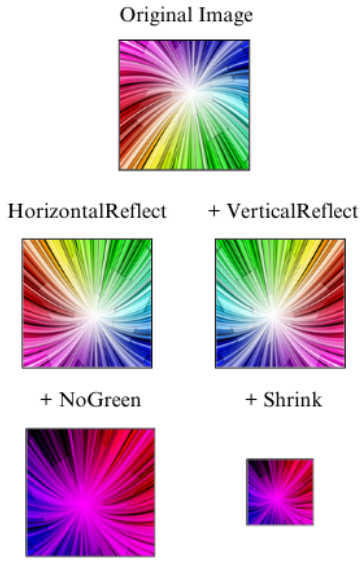
In the bottom-left image, the *HorizontalReflection()* and *VerticalReflection()* filters swap positions of the blue and magenta blocks, placing them on opposite sides (top/bottom, left/right).



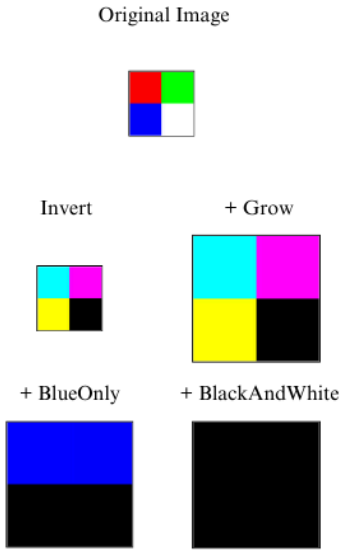
In the left image, the *BlackAndWhite()* filter leaves only black and white depending on the original image's RGB values. Similarly, in the right image, the *Threshold()* filter reduces the image to only eight distinct colors depending on the original image's RGB values.



Below is a sequence of transformations applied to an image to illustrate how a combination of multiple effects can produce visually compelling results. In this example, the following effects were used in this order: *HorizontalReflect()*, *VerticalReflect()*, *NoGreen()*, and *Shrink()*.



Below is another example of a series of transformations taking place on an image. In this instance, the following effects were used in this order: *Invert()*, *Grow()*, *BlueOnly()*, and *BlackAndWhite()*.



Problems Encountered

A few challenges I encountered were during the implementation of *Grow()* and *Shrink()*. In *Grow(),* I initially coded an algorithm that would stretch *pixels* in one dimension only, and copy each row into the following row. However, this approach proved inefficient, and I optimized it by simply copying each pixel into the respective locations of the 2 x 2 block in the new array. In *Shrink()*, there were more edge cases that I had to consider (ex: halving a dimension of 1). Additionally, hardcoding index calculations (ex: *i * 2 + 1*) led to *ArrayIndexOutOfBoundsException* errors for some test cases. These situations made me adopt more robust approaches that handled unexpected or uncommon scenarios.

4 Testing

Test Procedure

Using JUnit, I implemented methods to create new images and print out the inputted, outputted, and expected image results. For each of the effects in the *Transformations* class, I built a test method (ex: *testNoRed()*), where I created an input image with specified dimensions and color parameters, as well as an expected output image with its resulting characteristics. I passed in the input image to the respective transformation method and compared the actual output and the expected output images using *assertArrayEquals*(). If the statement was evaluated to be *true*, the *Transformations* class functioned as intended.

Test Cases and Rationale

For color manipulation methods, I used images that included the color being focused on, alongside additional colors, and compared the output to the expected result. For example, in *testNoRed()*, the input color was 0xFFFF00 (red and green) while the output should be 0x00FF00 (green only). Similarly, in *testRedOnly()*, the input color was 0xFFFF00 (red and green) while the output should be 0xFF0000 (red only). These tests verified that the methods either completely removed or retained the targeted color as expected. Methods like *testBlackAndWhite()* and *testThreshold()* had a similar procedure, but required multiple test runs using a variety of colors to ensure the transformations behaved correctly across different color ranges.

For reflection methods, I took a slightly different approach by filling sections of the image with two distinct colors: one for the top half and one for the bottom half (for *HorizontalReflection()*) or one for the left half and one for the right half (for *VerticalReflection()*). Because the two color placements should be switched after running the effects, the color parameters for the input image and expected output image are inverted as well. After applying the transformation, the images were checked to see if the colors were mirrored in their respective opposite positions.

For dimension manipulation methods, I inputted images of various widths and heights (ex: 2 x 2, 100 x 50, 100 x 1, 1 x 0) and compared the output and expected dimensions. For example, in *Grow()*, the dimensions should be double the input's for all cases except when the column length is 0, in which it will return an array with double the length of rows and 0 columns. In *Shrink()*, the dimensions should be half the input's for all cases except when one or both dimensions are not able to be halved without throwing an error. For instance, 100 x 1 should result in 50 x 1, 1 x 1, should result in 1 x 1, and 1 x 0 should result in 1 x 0. Checking all types of corner cases was used to ensure that the methods adjusted dimensions correctly and handled uncommon input images appropriately by returning either a null or an adjusted array.

5 References and Citations

(a) I participated in high-level discussions of concepts and strategy with Harini Majety.
(b) I received guidance in testing approaches and set-up from Andy Li and Ojas Phirke.