ASSIGNMENT OVERVIEW

In programming assignment one, we were required to manipulate images to create a variety of different effects. The focus of the assignment was on modifying 2-d arrays of pixels rather than working with BufferedImage objects, so much of the lower level work such as image reading and writing was done for us. The goal of this assignment was to finish the required effects and ensure they work on both common and edge cases. In doing this and attempting more complex effects, I intended to learn about basic image manipulation techniques and algorithms. In a broader sense, the purpose of this lab was to demonstrate the power of even simple data structures such as arrays and give us a gradual introduction to testing ideologies.

SOLUTION

In general, all of my effect classes followed the same design. I used a nested for-loop to access each pixel in the 2-d array and heavily utilized the methods provided in ImageEffect to modify pixels. Specifically, I used the getRed(), getGreen(), getBlue() to garner information about the pixel and the makePixel() method to modify it.

Classes: NoRed, NoBlue, and NoGreen

These effects were fairly trivial and had a relatively simple implementation. For NoRed, I removed all red shades by simply looping through pixels[][] and setting the original pixel to a new pixel value that had a red value of zero but maintained the blue and green values of the original pixel. For NoGreen and NoBlue the procedure was extremely similar except instead of setting the red component of the pixel's RGB value to zero I set the green and blue components to zero, respectfully.
Classes: RedOnly, BlueOnly, and GreenOnly

These effects were, in essence, the opposite of the NoRed, NoBlue, and NoGreen effects. I implemented these by setting each pixel’s non-eponymous RGB values to zero while maintaining the component in the class name. For instance, in GreenOnly, I removed all blue and red shades by simply looping through pixels[][] and setting the original pixel to a new pixel value that had red and blue values of zero but maintained the green value of the original pixel.

Class: BlackAndWhite

Unlike the previous effects, BlackAndWhite required the use of an algorithm. At first, I decided to take a non-weighted average of each pixel’s RGB components and creating a new pixel in which the RGB components are all equal to this average. The key assumption in this case was that a pixel that has RGB values that are equivalent to each will be some shade of gray. This assumption is supported by color theory and doing so resulted in a fairly decent grayscale image. However, upon further research on how the human eye works, I realized there are better ways of converting an image to grayscale. I decided on the luminosity method which weights green higher than other colors since our eyes are more sensitive to green shades (Cook, 2009). The weighted average was calculated using this formula: 0.21 R + 0.72 G + 0.07 B.

Classes: VerticalReflect and HorizontalReflect

The goal with VerticalReflect was to flip all pixels across an imaginary vertical line that ran through the center of the image. One of the key assumptions was that the image would be at least two columns wide. A key difference between this effect and previous ones is that I only looped through half of the columns. This means that if the width was odd then the middle column didn’t move, whereas, if the width was even, pixels in all columns shifted to the opposite side. The reason for doing so was to avoid creating an image that looks like the original left half reflected across both sides. Within the nested loops, each pixel was swapped with another that was on the same row and equidistant from the vertical center line of the image.

For HorizontalReflect, the goal was to flip all pixels across an imaginary horizontal line that ran through the center of the image. One of the key assumptions was that the image would be at least two rows wide. A key difference between this effect and previous ones is that I only looped through half of the rows and all of the columns. Within the nested loops, each pixel was swapped with another that was on the same column and equidistant from the horizontal center line of the image.
**Classes: Grow and Shrink**

The Grow and Shrink effects were more complex to implement. For grow instead of modifying the original pixels array, I created a new temporary array called expandedPixels that was twice the width and height of the original. In my outer and inner for-loops, the bounds were set to height x 2 and width x 2, respectively. Within the loops, I set expandedPixels[row][column] equal to pixels[row/2][column/2]. Due to integer rounding error, this resulted in one pixel in the original image representing four pixels in the enlarged image.

Unlike the Grow class, Shrink required multiple checks to prevent IndexOutOfBoundsException errors. The central assumption I made with this class was that both the height and width of pixels would be greater than 1. This assumption was proved incorrect in later testing and forced me to modify the method to work for edge cases. Similar to Grow, I created a new temporary array called shrinkedPixels that was half the width and height of the original. The bounds of the nested loops were set to height and width, however, I incremented rows and columns by 2 to avoid overlap. Within the actual method, I averaged the RGB values of the current pixel, along the pixels directly to the right, directly below, and directly to the right and below. Before doing so, I ensured that each coordinate was within the bounds of pixels[][] to avoid index out of bound errors. I then set shrinkedPixels[row/2][column/2] equal to this average value.

**Class: Threshold**

Threshold was the only one of the required classes that needed a parameter. To implement this, I called the super class’s (ImageEffect) constructor within the constructor of Threshold which resulted in the declaration of the ImageEffectParam ArrayList params. I then added a new ImageEffectParam to the ArrayList called ThresholdValue which allowed the user to determine the threshold value. Additionally, it prevents the user from entering a value that is outside of the acceptable range for RGB values (0-255) and if an unusable input is given, the threshold is set to 127 by default.

This class reduces the colors in images from millions to 8 by setting RGB values in each pixel on or off depending if they are above or below some user-inputted threshold. To implement this, I created temporary variables called red, green, and blue and assigned them a value of zero. If the RGB values in a particular pixel were greater than the threshold values, the variable for that color would be set to 255. Finally I set pixels[rows][c] equal to makePixel(red, green, blue).
DISCUSSION

Over the course of this project, I gained some valuable insights about basic image manipulation techniques. Most of these insights were gained working on the “harder” required methods and the additional karma.

One of the most useful things I learned was the importance of testing for edge cases and not making assumptions about the size of the image. For instance, while testing Shrink I realized that the program would crash attempting to create an array with a height and/or width of zero if height and/or width equaled one. To counter this, I preemptively multiplied height and the width by two in such cases.

Working on the karma methods was one of the most informative parts of the project. In my project, I created a smooth, sharpen, denoise, erode, static, and orange-tint filter. The static filter was one I made up. It is essentially an extreme version of sharpen that compares a pixel’s RGB values to its neighbors and drastically reduces them if they are less than those of its neighbors and drastically increases them if they are greater. This the entire image to appear pixelated in areas of contrast. The effect was meant to mimic old color televisions. It essentially arose out a failed attempted at sharpen which took a painstaking amount of time to fine-tune. Finally for my sharpen method, I decided against using a multiplier and adopted an addition/subtraction based method. To my surprise, this returned a far crisper image. Another filter I included was OrangeTint. It essentially orange tints the entire image by maintain the red values, halving green values, and setting blue values to zero. This was fairly elementary but displayed some UT pride so I decided to include it.

Overall, I felt the most of my solutions were extremely limited in scope compared to what is professionally available. Since the required effects up to Grow were basic in nature, they likely don’t differ much from what is professionally available. However, I believe my Grow and Shrink methods were fairly limited in scope as they did nothing to prevent quality loss and blurring. This applies even more to the karma filters. For example, compared to the sharpen in Photoshop mine didn’t allow the user to choose the radius, degree of adjustment, or where to apply the sharpening algorithm. Furthermore, now most sharpening is done using a genetic algorithm rather than simply averaging and comparing pixels to their neighbors. Similarly, my Smooth effect was fairly rudimentary. In actual practice, more complex filter shapes and statistics are utilized to reduce distortion.
INTERESTING IMAGES

“Bleed Orange”: Created by applying the threshold filter at 140 and then applying the orange-tint effect.

“TV Dinner”: Created by applying the static filter.

TESTING METHODOLOGY

For this project, I utilized black box testing. I first determined what each effect was supposed to do to each image. I then tested each effect on every image and determined whether it matched my preconceived notions. I quickly realized this is not an effective way of testing many of the effects, so I switched over to a more logical approach. To determine the accuracy of the NoRed, NoBlue, NoGreen, RedOnly, BlueOnly, and GreenOnly effects I used the ColorSyncUltily app on my computer. To test
the Grow and Shrink functions I used Preview to determine if the image was being shrunk exactly.

Certain images were more useful than others in testing. For instance, images 3,4,5 made it possible to intuitively test NoRed, NoGreen, and NoBlue. Additionally, small and strangely shaped images such 7, 12, and 16 were helpful in determining whether Shrink worked in edge cases. For effects such as VerticalReflect, HorizontalReflect, and Threshold, I found the images that were actual pictures of something (such as 1,2, and 11) were useful in superficially examining whether the effect was working as intended.

**SOURCES**