Joseph Li

CS 314H

Assignment 1 Report

Section 1: Brief Overview and Goals

This assignment consisted of implementing various classes that would modify images to accomplish certain tasks including modifying the rgb values of the individual pixels and translating pixels to show reflection.

My goals in this assignment were firstly to understand what kind of assignments we will be doing in this class as well as understanding how grading works, where to find help, and how much time might need to be committed to future assignments. For this particular assignment, I was interested in understanding how the Java graphical user interface functions, which mostly consisted of interpreting the starter code. I also wanted to do something creative in this assignment, as I don't usually encounter assignments that allow for contributing one's own ideas.

Section 2: Description of Solution Design

In implementing one's own additions to starter or skeleton code, it is vital to understand how the starter code works. Obviously the ones who wrote the starter code know this, but I believe it is still important to write down my thought process for clarity. Classes in *Transformations* all inherit from the *ImageEffect* abstract class, with the key abstract function being *apply*. The GUI takes every *ActionEvent,* classifies it as an *ImageEffect* (or not), outputs the parameters associated with that *ImageEffect,* and modifies the image according to the *apply* function of the *ImageEffect*. It was important for me to know this to correctly implement the *Puzzle* and *Swap* class that will be discussed later.

In designing the functions for the solution, I assumed that the provided functions worked as intended and assumed that the integers input in the *pixels* array were valid values. Additionally, I assumed that adding classes to the *Transformations* file would automatically be shown in the *effects* tab. My code mostly revolved around using some standard logic and math rather than using any algorithms.

Section 3: Discussion of Completed Assignment

The scope of our part of the assignment included classes that would change images in certain ways which I implemented using basic loops and the provided functions.

In implementing the *NoColor, OnlyColor,* and *Reflections* effects, the implementation need not be discussed, though they were good introduction effects to how the whole project functioned.

For the *BlackAndWhite* effect I implemented a gray-scaling system which utilized the fact that any pixel with equal moments of red, green, and blue pigment turned into a shade of grey. Thus I simply averaged the sum of the pigments in each pixel.

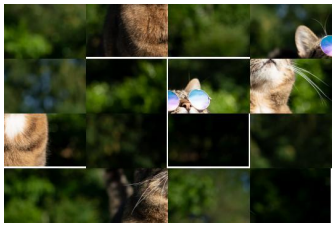
For the *Grow* and *Shrink* effects I scaled pixels with a 1 to 4 ratio, i.e. for *Grow* 1 pixel in the original image became 4 in the resulting image. This could be done conveniently with a little math where point(x,y) becomes the square from point(2*x, 2*y) to point(2*x + 1, 2*y+1) and vice versa.

The implementation for *Threshold* simply rounded the rgb values to 255 or 0 based on an input parameter. The *Dummy* class helped quite a bit in understanding how to take in input parameters.

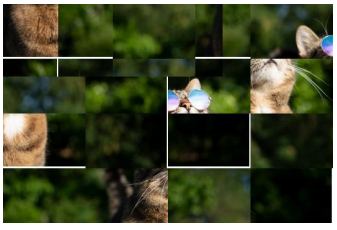
For karma I implemented the *Smooth* effect, which simply replaces a pixel's rgb value with the average of all its neighbors. I chose to implement this within a 3 x 3 space (though the space size doesn't matter too much, as essentially the larger the space size the more the image smoothes out) using two difference arrays for x and y representing the possible neighbors, and computed the average value for each of the colors red, green and blue.

Lastly, I thought it would be fun if the user could turn an image into a puzzle and then try to solve the puzzle, kind of like a mini-game. Thus I added two new effects: *Puzzle* and *Swap*. The *Puzzle* effect takes in a parameter *pieces* which represents the number of puzzle pieces per side (or the square root of the total number of pieces). It then adjusts the length and width if the number of pieces does not divide evenly into the pixels by taking the least multiple of *pieces* greater than the current length and width. As the picture itself will be smaller than the adjusted image, the added space defaults to white to look somewhat like a border. To actually convert the image into a puzzle, I keep track of each piece based on the coordinates of its top left corner, and use a random number generator to determine which of these pieces should be added to the new image in order. The *Swap* function utilizes three parameters (*pieces, first,* and *second*) to determine which pieces of the puzzle to swap (note that it is not necessary to create a puzzle before swapping). The function simply loops through each of the top left markers for each piece, and checks if that is the puzzle piece number the user intended to swap. I initially found it difficult to transfer the number of pieces in the puzzle to *Swap* (which is necessary because if I don't know the number of pieces then I cannot determine what the actual pieces are), but ended up relying on the user to accurately input the number of pieces as other methods may have required me to edit the starter code. If the user inputs an amount of pieces that is not equal to the *pieces* value used when creating the puzzle, the code will not "break" but it may result in a swap that is not desired (if the *pieces* inputted for swap is greater or if the values are co-prime).

Original 4 x 4 Puzzle

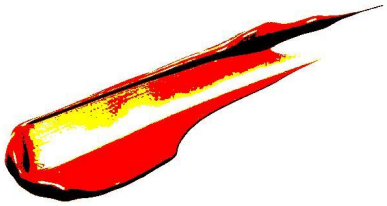


Swapped pieces 1 + 2 inputting *pieces* as 3 (you can see the top left corner is



messy)

In discussing interesting results, to be honest, I half expected most of my functions to not work as well as they did and was quite surprised at the results of each effect. Notably I thought greyscale would not look identical to the original image, as it simply averaged the rgb values of the pixel and I thought many different colors would result in similar grey colors. However, as the “amount” of each rgb value is associated with lighter and darker shades, the image still turns out to resemble the original image perfectly. Similarly, the threshold effect also does a surprisingly good job of retaining the same structure as the original image, even with only 8 colors. It was also interesting that the common color brown would be changed to a mix of red, yellow, and black using the threshold effect. This just goes to show that brown is actually a mixture of red and green pigments.



I also found it interesting that you could create a puzzle inside of another puzzle, and solve the nested puzzle, though it looks very wonky.

Section 4: Discussion of Software Testing

Testing code uses my solution as a black-box and verifies the solution through the results. To verify each of the *NoColor* and *OnlyColor* effects, I tested each of the functions on test image 13 (the given sample images). This image displays each of the rgb colors, which will show me the effect of each function on their respective colors.

For the reflections, I simply looked at whether the image (any image which looks different flipped) seemed to be reversed, and called it twice to check if the result would be the original image.

For the *BlackAndWhite* effect I wasn't exactly sure what the exact output should look like, but after testing this effect on sample images 1, 6, and 11, I concluded that since all images resulted in a grey coloring while largely retaining their original shape, the function should be valid as the actual code is quite simplistic and shouldn't have any edge cases.

For *Grow* and *Shrink* testing was similar to reflections, simply grow and shrink any image and check if the original image is retained (and if the proportions seem correct) should be enough to test if the functions are valid.

For *Threshold* I wanted to test if only the 8 base colors were used in the resulting image and if the resulting image remotely resembled the original. I tested image 13 because it allowed me to test different threshold levels which would result in different amounts of blocks rounding to black. I also tested image 21 because it had many shades of similar colors, and threshold should result in these similar colors equalizing, which it did.

For *Smooth* not too much testing was needed, as, like *BlackAndWhite* the code was relatively simplistic, and simply testing on image 11 (though almost any image isn't just a single color would suffice) was enough to see that image was indeed getting smoother at a slow rate.

Finally, in testing *Puzzle* and *Swap* I needed to ensure that 1) a randomized puzzle with the right number of pieces was being created 2) if the image does not divide evenly into *n* pieces, white border should be added 3) the correct pieces are being swapped, with the remaining image being the same 4) having out of bounds input does not break my code. Simply testing this on multiple images confirms 1 and 3. Additionally, I tested creating a puzzle where the pixels did not divide (e.g. 200 pixels into 6 pieces) and a white border was created. For 4, I simple tested inputting large values, and my program worked as intended.