Introduction to Computer Science, Efi Arazi School of CS, 2021

HW 7: Image Processing

Images are all around us, and so is image processing. Each time you move, crop, rotate, or resize an image, you are using image processing algorithms. In this homework assignment you will learn about, and implement, some of the most commonly used image processing functions. These functions lie at the very core of programs like Instagram and Photoshop.

As you implement these capabilities, you will continue to develop two key skills: handling two-dimensional arrays, and understanding modularity. You will also get a hands-on introduction to creating and using simple *objects*. This will provide a gentle hands-on introduction to object-oriented programming, a subject that we will take up in the second part of the course.

For an entertaining behind-the-scenes view of digital images, take a look at this TED talk (thanks to Dotan Beck for recommending it).

Image processing is an exciting field of theory and practice, and we hope that you will enjoy cutting your teeth into it. We begin by introducing some basic concepts.

Digital Imaging

Color

Color is a human perception of a light wave that has a certain wavelength. The human brain can distinguish between about 10 million different wavelengths, and human languages have given a few of these wavelengths names like *red*, *yellow*, *green*, *magenta*, and so on (notice the conceptual similarity to naming selected sound waves like *do*, *re*, *mi*, etc.). When light waves hit the human eye, specialized cells in the retina react to them according to their wavelengths. The human retina features three types of such sensor cells, each specializing in detecting different spectrums of wavelengths. Those wavelengths correspond to what we are used to call *Red*, *Green*, and *Blue* (RGB). All the fantastic colors that we are fortunate to see around us emerge from the way our brain mixes and combines different *intensities* of those three basic colors.

The natural mechanism described above gives rise to a certain mathematical model (but not the only one) for representing colors. We can view each color as a vector of three integer values, each ranging between 0 and 255. Those three numbers are used to represent the *intensities* of the three basic colors Red, Green, and Blue. Thus, mathematically, every color is a triplet of numbers ‹*r*, *g*, *b*›, where each of these values ranges from 0 to 255. We see that the RGB system can represent 256^3=16,777,216 different colors. This is about 6 million more colors than the

human brain can discern. Not bad.

`Color` objects: In Java, colors are represented as instances of a class called `Color`. Another way of saying this is that "In Java, colors are represented as objects of type `Color`". Unlike primitive types like `int` and `double`, the type `Color` is not part of the basic Java language. Rather, it is implemented by the `Color` class, which is part of the standard class library that extends the basic Java language. How classes can be made to represent new data types is something that we will learn later in the course. For now, suffice it to say that the Java class library features many such classes, and `Color` is one of them (`String` is another one).

The color objects are quite similar to arrays. For example, the following code declares and constructs several color objects:

```
Color red     = new Color(255,  0,  0);
Color green   = new Color(  0,255,  0);
Color blue    = new Color(  0,  0,255);
Color black   = new Color(  0,  0,  0);
Color white   = new Color(255,255,255);
Color yellow  = new Color(255,255,  0);
```

For example, the first statement says: "declare a reference variable named `red`, of type `Color`, and make it refer to a memory block in which the three numbers `255`, `0`, and `0` are stored".

In object-oriented programming, the data of an object is accessed using methods that are called "getters". Here is an example of using these getters:

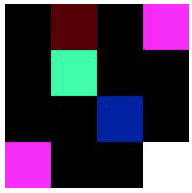
```
System.out.println(yellow.getRed());   // Prints 255
System.out.println(yellow.getGreen()); // prints 255
System.out.println(yellow.getBlue());  // prints 0
```

In object-oriented programming, a function that operates on an object is called *method*. The above example illustrates calling three methods on the same `Color` object, referred to by the variable `yellow`. As we see from the comments, each of these methods is designed to return an `int` value. This method calling illustrates a key difference between accessing array data and accessing object data. If we were to use an array named `yellow` for representing the color's data, we could get the RGB values by accessing `yellow[0]`, `yellow[1]`, and `yellow[2]`, respectively. If we use a `Color` object instead, as we do here, we cannot access these values directly; Instead, we access them using the three *get* methods. We will have much more to say about this style of object-oriented programming later in the course.

Here is the Color class API. The only `Color` methods that we will use in this project are the `get` methods, so there is no need to study other methods in this API.

Image

A digital image can be viewed as a grid of RGB values. For example:



```
(  0,  0,  0) (100,  0,  0) (  0,  0,  0) (255,  0,255)
(  0,  0,  0) (  0,255,175) (  0,  0,  0) (  0,  0,  0)
(  0,  0,  0) (  0,  0,  0) (  0, 15,175) (  0,  0,  0)
(255,  0,255) (  0,  0,  0) (  0,  0,  0) (255,255,255)
```

This little image can be implemented in Java as a 2D array of `Color` objects, as follows:

```
Color[][] tinypic = {
                       { new Color(  0,  0,  0), ... , new Color(255,  0,255) } ,
                       { new Color(  0,  0,  0), ... , new Color(  0,  0,  0) } ,
                       ...
                       { new Color(255,  0,255), ... , new Color(255,255,255) } ,
                     }
```

We see that every array element `tinypic[i][j]` refers to a `Color` object. In image processing applications, these array elements are called *pixels* (short for *Picture Elements*).

The image *resolution* is determined by how many pixels it contains. The more pixels, the sharper and more detailed the image. Typically, an image array contains hundreds of rows and hundreds of columns. Physically speaking, each pixel is drawn on the screen (or printed on paper) using a very tiny area. The image shown above, which is made of only 16 pixels, was blown up 5000% before we plugged it into this document.

We note in passing that if you are a great artist, you can get a lot of expression from just a few pixels. For example, Salvador Dali drew a low resolution yet pretty good portrait of Abraham Lincoln.

Image files

In order to store images persistently, and transfer them from one computer to another, we use files. Clearly, we must decide on some standard way to structure such files. In this homework we use a well known file format called PPM (*Portable Pixel Map*). For example, the PPM file of our `tinypic` image is as follows:

```
P3
4 4
255
  0   0   0   100   0   0     0   0   0   255   0 255
  0   0   0     0 255 175     0   0   0     0   0   0
  0   0   0     0   0   0     0  15 175     0   0   0
255   0 255     0   0   0     0   0   0   255 255 255
```

The first three lines are called *file header*. P3 is an agreed-upon code that says "I am a PPM file". Next come the numbers of columns and the number of rows in the image (in this example, 4 by 4). Next comes the maximum color code value in this image, which is 255.

Following the file header comes the *body*. The body contains all the pixel values. Every three consecutive numbers represent a single pixel. White space is commonly used to make the data more readable to the human eye, and is ignored by computers. Following convention, each row of pixels starts in a new line.

The PPM file format is recognized by most image editors and image viewing apps. When you double click a `.ppm` file on your computer, most likely it will be shown as an image. In order to see the RGB values that drive this image (the numbers), open the `.ppm` file using a text editor. At this stage we recommend that you take a look at some of the `.ppm` files supplied with this homework.

There are many different file formats for storing digital images. JPEG, GIF, and PNG are popular examples, each serving a different purpose. In order to cut down storage and communications costs, these file formats represent images in a compressed way. PPM files use no compression, making them easy to work with (but less practical).

Getting Started

Start by compiling, running, and understanding the supplied `ColorDemo` class. This class illustrates how to construct and print `Color` objects. It also illustrates how to construct arrays of `Color` objects. When you are done playing with `ColorDemo`, we recommend that you spend a few minutes practicing mixing colors.

In this homework assignment you will gradually develop a library of image editing functions. You will also write client code for testing and playing with these functions, and enjoying the fruits of your work. Before doing anything though, go ahead and read this entire document. There is no need to understand everything you read; this understanding will grow on you as you start working on the code.

So, assuming that you've read the entire document, take a look at the supplied `Instush.java` class. We now turn to describe the functions in this class.

Rendering an image on the screen: You will notice that the supplied `Instush.java` class features an implemented function named `show`. This function renders (displays) a given 2D array of `Color` values on the screen. The `show` function performs this important service by calling functions from the `StdDraw` class. You are welcome to look into how the `show` function works, but you can also use it abstractly, whenever you wish to render an image on the screen. If you want to understand how the `show` function works, start by reviewing the code of the supplied

`StdShowDemo` class, and playing with it. This activity is optional.

Printing the image data: In addition to rendering an image using the `show` function, developers need to inspect the "raw image" by listing its underlying data – the RGB values. This service is performed by a function named `print`, which is critically important for testing and debugging purposes. When we will start implementing various image editing functions, we will test them on small images, consisting of just a few rows and columns of pixels. The `print` function will enable you to inspect the numbers and check that the functions are doing what they are supposed to do. For example, here is a typical testing sequence:

```
// Tests the reading of an image from a file:
int[][][] image = read("tinypic.ppm");
print(image);   // prints the image data
```

You will have to write and execute similar testing code segments for each function that you implement. We now turn to specify all these functions.

Image Editing Functions

Reading an image from a file

The function `Color[][] read(String filename)` receives the name of a PPM file and returns an array containing the image data. The `.ppm` file must be located in the homework folder.

Implementation tips: Use `StdIn` functions for reading the image data from the given file. To get started, your code must call the `StdIn.setInput(String)` function, which sets standard input to the file whose name is the given string (the parameter of `read`). Assume that this file contains valid PPM code.

Printing the image data

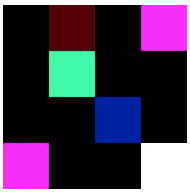
The void function `print(Color[][] image)` prints the `Color` data of all the pixels. For example, here is the result of printing the `tinipic` image (after reading it, of course, from the `tinypic.ppm` file):

```
(  0,  0,  0) (100,  0,  0) (  0,  0,  0) (255,  0,255)
(  0,  0,  0) (  0,255,175) (  0,  0,  0) (  0,  0,  0)
(  0,  0,  0) (  0,  0,  0) (  0, 15,175) (  0,  0,  0)
(255,  0,255) (  0,  0,  0) (  0,  0,  0) (255,255,255)
```

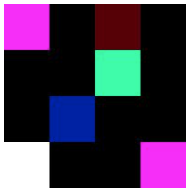
Implementation tip: To print individual `Color` objects, use the same code that we provided in the `ColorDemo` class (copy-paste the code that you need into `Instush.java`). To print a 2D array of `Color` objects, iterate through the elements of the array, and print each element individually. Your output should be formatted exactly the same as the example shown above.

Horizontal Flipping

The function `Color[][] flippedHorizontally(Color[][] image)` returns a new image which is a horizontal flip of the given image: In each row of the new image, the order of the pixels is reversed (within each pixel though, nothing changes). For example, here is the `tinypic` image (top) and its horizontally flipped version (bottom):



```
( 0,  0,  0) (100,  0,  0) (  0,  0,  0) (255,  0,255)
( 0,  0,  0) (  0,255,175) (  0,  0,  0) (  0,  0,  0)
( 0,  0,  0) (  0,  0,  0) (  0, 15,175) (  0,  0,  0)
(255,  0,255) (  0,  0,  0) (  0,  0,  0) (255,255,255)
```



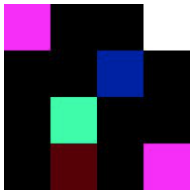
```
(255,  0,255) (  0,  0,  0) (100,  0,  0) (  0,  0,  0)
( 0,  0,  0) (  0,  0,  0) (  0,255,175) (  0,  0,  0)
( 0,  0,  0) (  0, 15,175) (  0,  0,  0) (  0,  0,  0)
(255,255,255) (  0,  0,  0) (  0,  0,  0) (255,  0,255)
```

Write and test the `flippedHorizontally` function.

Implementation tip: Start by creating a new image (a 2D array of `Color` objects) that has the same dimensions as the given image. Then fill it with the correct values. This tip applies to all the functions in this class that return a new image.

Vertical Flipping

The function `Color[][] flippedVertically(Color[][] image)` returns a new image which is a vertical flip of the given image: In each column of the new image, the order of the pixels is reversed. For example, vertically flipped version of tinypic is as follows:



```
(255,  0,255) (  0,  0,  0) (  0,  0,  0) (255,255,255)
( 0,  0,  0) (  0,  0,  0) (  0, 15,175) (  0,  0,  0)
( 0,  0,  0) (  0,255,175) (  0,  0,  0) (  0,  0,  0)
( 0,  0,  0) (100,  0,  0) (  0,  0,  0) (255,  0,255)
```

Write and test the `flippedVertically` function.

Grey Scaling

The RGB system has the following convenient property: When all the three color intensities are the same, the resulting color is a shade of grey, ranging from black (0,0,0) to white (255,255,255). The resulting 256 values are called "greyscale codes". With that in mind, "greyscaling" is a technique for transforming a colored image into a black and white image gracefully, in a way which is pleasant and sensible to the human eye. Below is an example of a colored image of a cupcake and its greyscaled version:

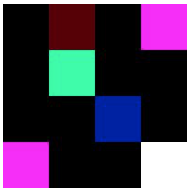
How to transform a 3-valued RGB color into a 1-valued greyscale code that “represents” that color? Suppose that the RGB values (each being a number from 0 to 255) are represented by the values *r*, *g* and *b*. We define *luminance* to be the following linear combination:

$$lum(r, g, b) \;=\; (int)\,(0.299 \cdot r \;+\; 0.587 \cdot g \;+\; 0.114 \cdot b)$$

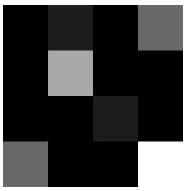
Since the luminance weights are positive and sum up to 1, and since the intensities are all integers between 0 and 255, the luminance ends up being an integer between 0 and 255. With that in mind, the resulting greyscale value is defined as (*lum*, *lum*, *lum*). The three weights 0.299, 0.587, and 0.114, which are based on the human eye's sensitivity to red, green, and blue, were determined after running experiments with human subjects.

We'll implement the greyscaling transformation in two steps. First, write the function `Color luminance(Color c)`, that returns the greyscale color of the given color, using the formula presented above. To check that your function is working properly, you may consult this resource.

Next, implement the function `Color[][] greyScale(Color[][] image)`. This function returns the greyscaled version of the given image. For example, here is the `tinypic` image and its greyscaled version:



```
(  0,  0,  0) (100,  0,  0) (  0,  0,  0) (255,  0,255)
(  0,  0,  0) (  0,255,175) (  0,  0,  0) (  0,  0,  0)
(  0,  0,  0) (  0,  0,  0) (  0, 15,175) (  0,  0,  0)
(255,  0,255) (  0,  0,  0) (  0,  0,  0) (255,255,255)
```



```
(  0,  0,  0) ( 29, 29, 29) (  0,  0,  0) (105,105,105)
(  0,  0,  0) (169,169,169) (  0,  0,  0) (  0,  0,  0)
(  0,  0,  0) (  0,  0,  0) ( 28, 28, 28) (  0,  0,  0)
(105,105,105) (  0,  0,  0) (  0,  0,  0) (255,255,255)
```

Editor 1

We now describe a simple client program that uses the three image processing services described above. The program `Editor1.java` takes two command-line arguments: the name of a PPM file that represents an image, followed by one of the strings `fh`, `fv`, or `gs`. The program reads the image from the file and the specified transformation, and displays a new image which is either the horizontally flipped, vertically flipped, or greyscaled version of the given image. For example:

```
% java Editor1 ironman.ppm fh
```

Implementation tips: the program creates an image by calling the `read` function, creates another transformed image by calling one of the functions `flippedHorizontally`, `flippedHorizontally`, or `greyscaled`, and finally renders the transformed image using the `show` function. All these functions are called from the `Instush.java` class.

Write and test the `Editor1.java` program.

Scaling

Quite often, we want to resize a given image. For example, reducing a given image into a small thumbnail image, zooming in on a satellite photograph, or making an image wider or taller. All these operations can be described as *scaling* either the width and/or the height of the image. For example, the left image below is 400 pixels wide by 600 pixels high. If we halve its height and double its width, we get the 800-by-300 image shown on the right.

The scaling algorithm is as follows. Let the width and the height of the source image be w_0 and h_0, and the width and height of the target image be w and h. With this notation in mind, pixel (i, j) of the target image should be set to pixel $(i \cdot \frac{h_o}{h}, j \cdot \frac{w_o}{w})$ in the source image. For example, if we are halving the size of an image, the scale factors are 2 in both dimensions. Therefore, and choosing an arbitrary pixel as an example, pixel (2, 3) of the scaled version should be set to pixel (4, 6) of the source image.

The scaling operation is implemented by the function `Color[][] scale(Color[][] image, int width, int height)`. The function returns a new image which is a version of the original image, scaled to be of the given width and height. For example, `tinypic` is a 4 x 4 image. We can scale it to a 3 x 5 image by calling `scale(tinypic,3,5)`. Below we see the original image (top) and its scaled version (bottom). You can work out the proportion factors and verify that the scaled image is built correctly.

```
(  0,  0,  0) (100,  0,  0) (  0,  0,  0) (255,  0,255)
(  0,  0,  0) (  0,255,175) (  0,  0,  0) (  0,  0,  0)
(  0,  0,  0) (  0,  0,  0) (  0, 15,175) (  0,  0,  0)
(255,  0,255) (  0,  0,  0) (  0,  0,  0) (255,255,255)
```

```
(  0,  0,  0) (100,  0,  0) (  0,  0,  0)
(  0,  0,  0) (100,  0,  0) (  0,  0,  0)
(  0,  0,  0) (  0,255 175) (  0,  0,  0)
(  0,  0,  0) (  0,  0,  0) (  0, 15,175)
(255,  0,255) (  0,  0,  0) (  0,  0,  0)
```

Scaling is a "lossy operation": some of the information contained in the original image may be lost during the scaling process. This means that once an image has been scaled, it may be impossible to scale it back to the original image.

Write the `scale` function, and test it on the `tinypic` image.

Editor 2

This client program is designed to test the scaling service described above. The program receives three command-line arguments: the name of the PPM file representing the image that should be scaled, and two integers that specify the width and height of the scaled image. For example:

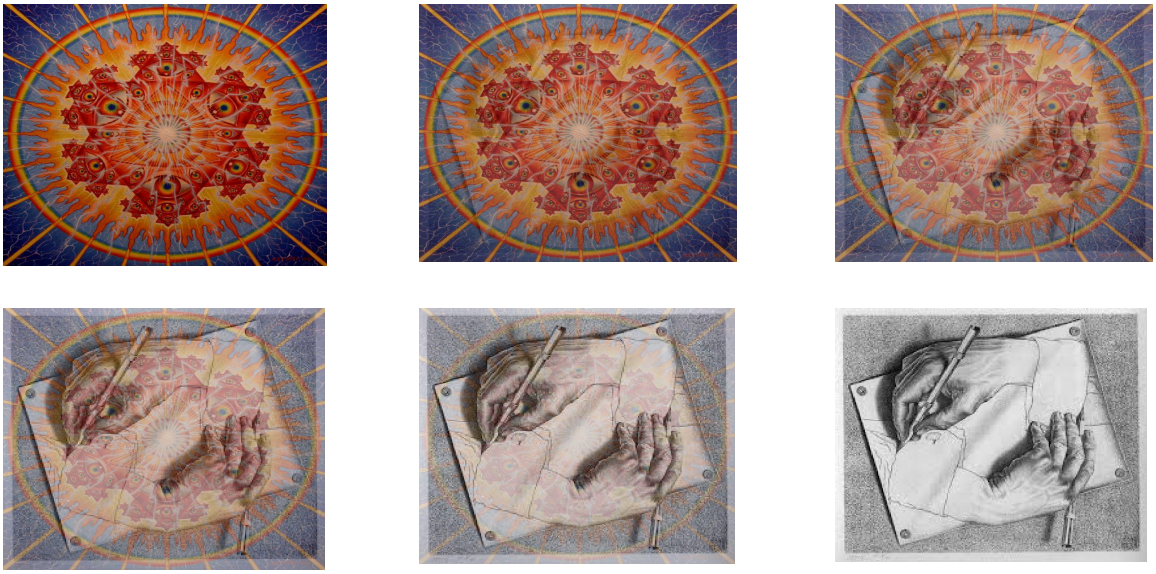
```
% java Editor2 ironman.ppm 100 800
```

Implementation tips: the program creates an image by calling the `read` function, creates another scaled image by calling the `scaled` function, and finally renders the scaled image using the `show` function. All these functions are called from the `Instush.java` class.

Write and test the `Editor2.java` program. It should be fun to scale `Ironman` and some other PPM images, as you please.

Morphing

We now turn to implement a striking visual effect called "morphing": Given a source image and a target image, we transform the former into the latter in a stepwise fashion that can be as smooth as we please. For example, here is an example of morphing a drawing by Alex Grey into a drawing by M.C. Escher, in 6 steps:



We will approach the morphing challenge by dividing it into several independent functions, which we now turn to describe.

Blending colors: a blend of two colors is a new color whose RGB values are weighted averages of the RGB values of the two input colors. The blending operation is parameterized by a real number $0 \leq \alpha \leq 1$ that determines how to blend the two colors: the weight of the first color is α, and the weight of the second input pixel is $1 - \alpha$. For example, suppose that the two input colors are (100, 40, 100) and (200, 20, 40). Blending them with $\alpha = 0.25$ produces the color (175, 25, 55), as follows:

$$0.25 \cdot 100 + 0.75 \cdot 200 = 175$$

$$0.25 \cdot 40 + 0.75 \cdot 20 = 25$$

$$0.25 \cdot 100 + 0.75 \cdot 40 = 55$$

The function `Color blend (Color c1, Color c2, double alpha)` returns a new blended color according to the process described above. Of course, the resulting color must consist of integer values. Write and test the `blend` function (the version that operates on two colors).

Blending images: Two images of the same dimensions can be blended by blending all the corresponding input pixels using a given α. The function `Color[][] blend (Color[][] image1, Color[][] image2, double alpha)` returns the alpha-blending of the two given images. The function computes each new pixel using the `blend` function. Assume that the two source images have the same resolution. Write and test the `blend` function (the version that operates on two images).

Morphing: suppose we want to morph a source image into a target image gradually, in *n* steps. To do so, we stage a sequence of 0, 1, 2, ... , n steps, as follows. In each step *i* we blend the source image and the target image using $\alpha = (n - i)/n$. For example, here is what happens when $n = 3$:

step 0: Blend the two images using $\alpha = 3/3$ (yielding the source image)
step 1: Blend the two images using $\alpha = 2/3$
step 2: Blend the two images using $\alpha = 1/3$
step 3: Blend the two images using $\alpha = 0/3$(yielding the target image)

The function `void morph (Color[][] source, Color[][] target, int n)` morphs the source image into the target image in *n* steps. If the images don't have the same dimensions, the function scales the target image to the dimensions of the source image. At the end of each blending step, the function uses the `show` function to display the intermediate result. Write and test the `morph` function.

Editor 3

This client program tests the morphing service described above. For example, consider the following program execution:

```
% java Editor3 cake.ppm ironman.ppm 300
```

This action transforms the cake into Ironman, in 300 steps. The greater the number of morphing steps, the smoother is the transition.

Implement the `Editor3.java` program.

Implementation tips: The program starts by getting the three command-line arguments. Next the program creates the source and the target images by calling the `read` function. Finally, the program calls the `morph` function. All these functions are called from the `Instush.java` class.

It should be fun to morph various images into each other.

Editor 4

This client program performs another striking visual effect: it morphs an image into its grescaled version. For example:

```
% java Editor4 thor.ppm 300
```

This action transforms the colored image of Thor into a black and white image of Thor.

Implement the `Editor4.java` program, and test it. Sit back, watch, and enjoy.

Implementation tips: The skeleton of this class is not given. You should write, document, and implement it yourself, from scratch.

Submission

Before submitting your work for grading, make sure that your code is written according to our Java Coding Style Guidelines. Zip the following five files into the file `HW7.zip`:

- `Instush.java`
- `Editor1.java`
- `Editor2.java`
- `Editor3.java`
- `Editor4.java`

In addition, create a single PDF document, named `HW7.pdf`, that lists the code of the 5 classes that you wrote in this homework in a way which is easy to read and review. Start each class on a new page. Use the font `Consolas`, or Arial, size 12. The PDF file can have many pages, that's fine.

In previous homework assignments we guided you to zip the Java files that you wrote together with the PDF file. We now change these guidelines: Submit to Moodle *two separate* files: `HW7.zip`, and `HW7.pdf`.

Submission deadline: Sunday, December 5, 2021, 23:55. You are welcome to submit earlier.