Introduction to Computer Science, IDC Herzliya, 2016

HW 5: Image Processing

Images are all around us, and so is image processing. Each time you move, rotate, shrink or enlarge some image, you are using image processing software. You may also be familiar with image editing programs like *Paint*, *Photoshop*, and *Gimp*. In this assignment you will learn about, and implement, some of the most commonly used functions used in image processing.

Although there is no need to use any available image processing software in this assignment, we wish to note two freely-available programs. Gimp is a popular open source alternative to PhotoShop that requires some installation effort. IfranView is a light-weight image editor that can be easily downloaded and installed. Once again, you don't need any of these programs for this assignment, so this is just a general reference.

In this assignment you will continue to develop two key programming skills: handling multi-dimensional arrays, and working with functions.

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 fundamental concepts of digital imaging.

Digital Imaging

Color

Color is a human perception of a sinus-shaped light wave that has a certain wavelength. The human brain is programmed to 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. 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*. All the fantastic colors that we are fortunate to see around us come from the way our brain mixes and combines different *intensities* of those three basic colors (for example, "very light red" combined with "medium green" and "dark blue").

The natural mechanism described above gives rise to a certain model (but not the only one) for representing colors using digital media. We can view each color as a vector of three integer values, each ranging between 0 and 255. Those three numbers will be used to represent the *intensities* of the basic colors red, green, and blue. Here is an example of how this system, known as "RGB", can be used to represent a few colors in a programming language like Java:

```
int[] red =        {255,   0,   0};   // the color red
int[] green =      {0  , 255,   0};   // the color green
int[] blue =       {0  ,   0, 255};   // the color blue
int[] black =      {0  ,   0,   0};   // the color black
int[] white =      {255, 255, 255};   // the color white
int[] yellow =     {255, 255,   0};   // the color yellow
int[] marineBlue = {0  , 102, 204};   // a color sometimes called "marine blue"
// Etc.            most colors have no names; the next paragraph explains why.
```

Since each basic color intensity runs from 0 to 255, the RGB system can represent $256^3 = 16,777,216$ different colors, which is about 6 million more colors than the human brain can discern. Not bad.

Pixel

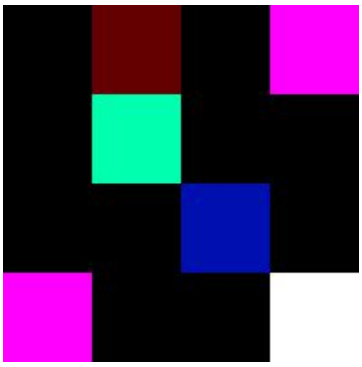
A *pixel* (shorthand for *Picture Element*) is the smallest controllable element of a digital image -- "the atom" from which digital images are made. A pixel is characterized by a color (three RGB values) and an (*i*,*j*) location in some two-dimensional space. Taken together, these values determine the color that will appear in this location.

Image

A digital image can be represented by a rectangular matrix of pixels: the (*i*,*j*) entry of the matrix represents a single pixel. Each such pixel is represented further as a 3-element vector: element 0 represents the red intensity, element 1 represents the green intensity and element 2 represents the blue intensity. We see that the entire image can be represented using a 3-dimensional array.

The image *resolution* is determined by how many pixels it contains. The more pixels, the sharper and more detailed the image. For example, consider an image with a resolution of 500 x 700 RGB pixels. Such an image is 500 pixels high and 700 pixels wide, containing a total of 500 x 700 = 350000 pixels. Since each pixel is represented by three integer values, such an image will occupy 500 x 700 x 3 = 1050000 integer values in memory. One way to cut down storage space is to use the smallest possible integer data type that the host language offers for representing 256 different values. In this project we will not worry about space efficiency, and will use `int` values throughout.

To illustrate the approach, consider the following image:



Using the conventions discussed above, this image can be represented using the following array:

```
{{{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}}}
```

Note that each square in the above picture represents one pixel. Since in reality these pixels are very small, the above image is actually very tiny (it was blown up 5000% before we plugged it into the text). There is a famous oil painting, drawn by Salvador Dali, which uses low-resolution for creating an enigmatic portrait of a famous person. Can you tell who this person is?

Image file

The matrix representation described above works well once the image has been "read" somehow into the program. In order to store images persistently, and transfer them from one computer to another, we must decide how to represent them using text files. There are many different file formats for storing digital images, JPEG, GIF, PNG and BMP being well-known examples that serve different purposes. In order to cut down storage and communications costs, most of these formats store images in a compressed way.

Compression is of course very important, but it complicates the tasks of reading and writing image files. Some formats, like PPM (*Portable Pixel Map*), are uncompressed, and easier to work with. PPM is recognized by most image editors including Gimp and IfranView, and that's the format that we will also use in this assignment. 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 *the file header*. `P3` is an agreed-upon code for describing which type of image is stored. Let us not worry about it, remembering to always use `P3` in all our image files. Next comes the number of columns and the number of rows in the image (in this example, 4 x 4). Finally, we have the maximum color code value, which is 255 in all the images used in this assignment.

Following the header comes the *body*. The image body contains the actual picture information. Every three consecutive numbers represent a single pixel. For example, the first three numbers (0 0 0) code that the color of the top-left pixel in the image is black, and the last three numbers (255 255 255) code that the color of the bottom-right pixel is white (as indeed is the case in our `tinyPic` example). White space is commonly used to make the data more readable to humans, and is ignored by computers. Following convention, we will write each row of pixels in a separate line.

The Assignment

In this 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 already read the entire document, take a look at the supplied `ImageEditing.java` class. All the editing functions that you have to write must be added to this class, in no particular order.

You will notice that the supplied class already contains one function, named `show`. This function has one parameter: a 3-dimensional integer array, designed to represent an image according to the model described above. The `show` function renders on the screen the image that the given array represents, using the services of `StdDraw`. The `show` function will be very handy in debugging and testing your work. For now, there is no need to understand the code of `show`; just use it as you see fit.

Testing: The assignment revolves around writing several functions. It is absolutely crucial to test each function as soon as you finish writing it. We recommend putting all your tests in a `main` method. For example, here is a typical testing code segment:

```
// *** Testing reading an image file
int[][][] pic = read("tinypic.ppm"); // Reads image data from a file, into an array
display(pic);                        // Displays the array's data on standard output
show(pic);                           // Renders the image on the screen
// *** End of testing reading an image file
```

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

Functions

Reading an image file

The function `public static int[][][] read(String filename)` receives the name of a PPM file and returns an array containing the image's data. The file must be located in the assignment's folder.

Implementation tips: use `StdIn` to read the image data from the given file. To get started, your code must make a call to the function `setInput(String filename)`, which is part of the `StdIn` library. This function informs all the `StdIn` functions that from now on standard input will come from the file `filename`. You can assume that this file contains valid PPM code.

Write and test the `read` function. This must be done before continuing to do anything else in this assignment.

Printing the image data

The function `private static void print(int[][][] source)` writes to standard output the contents of the given 3-dimensional array. For example, if we call this function with an array that represents the `tinypic` image, we should get the following output:

```
  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
```

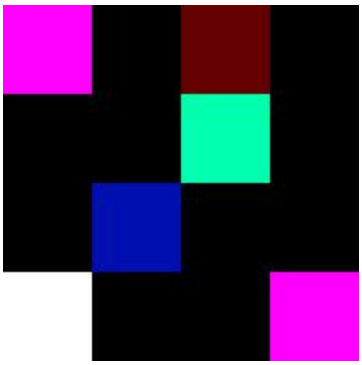
The `print` function is a private "helper" method which is used for debugging purposes. It provides a direct way for verifying that your program reads and represents images properly. Use it as you see fit, for your own debugging. Note that your output must be formatted nicely, to make it readable (use `printf`).

Horizontal Flipping

The function `public static int[][][] flipHorizontally(int[][][] source)` is designed to flip the given image, horizontally. The function receives an image matrix as input and returns a new image matrix as output. In each row of the new matrix, the order of the input pixels is reversed (remember that within each pixel though, nothing changes). For example, when applied to the `tinypic` image, the function returns the following matrix:

```
{{{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}}}
```

When rendered on the screen, this matrix displays the following image:



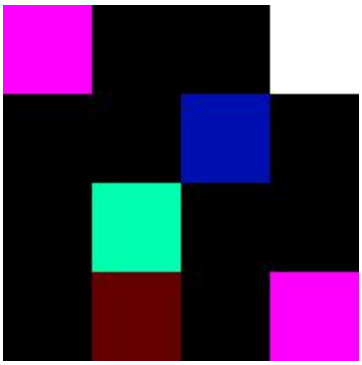
Write and test the `flipHorizontally` function.

Vertical Flipping

The function `public static int[][][] flipVertically(int[][][] source)` flips a picture, vertically. This function is very similar to the previous one, except that it reverses the order of the pixels of the source image along the columns instead of along the rows. For example, when applied to the `tinypic` image, the function returns the following matrix:

```
{{{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}}}
```

When rendered on the screen, this matrix displays the following image:



Write and test the `flipVertically` function.

Grey Scaling

The RGB system has the convenient property that when the three color intensities are all 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, while doing it in a way which is pleasant and sensible to the human eye. Here is an example of a colored image of a cupcake and its greyscaled version:

In order to implement such a transformation, we have to come up with a sensible way for mapping each one of the 16,777,216 possible RGB colors using one of the 256 possible shades of grey. Suppose that the RGB values (each being a number from 0 to 255) are represented by the variables *r*, *g* and *b*. We define *luminance* to be the following linear combination:

```
lum(r,g,b) = (int) (0.299 * r + 0.587 * g + 0.114 * 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}.

We see that luminance is a weighted average using the three weights 0.299, 0.587, and 0.114. These weights, which are based on the human eye's sensitivity to red, green, and blue, were determined after running many experiments with human subjects.

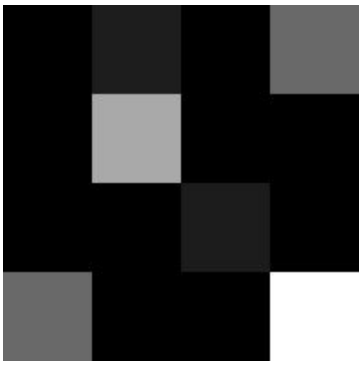
We'll divide the greyscaling implementation into two steps. First, write a function `public static int[] luminance(int[] pixel)` which takes an RGB pixel as input and returns the corresponding greyscale pixel, using the formula presented above. As usual, test the function before going on. For example, the function should transform the pixel {255, 0, 0} (which happens to be red) into the greyscale pixel {76, 76, 76}.

Now you are ready to write the greyscaling function `public static int[][][] greyScale(int[][][] source)`. This function takes an RGB image as input and returns a greyscaled version of this image. Needless to say, this function makes use of the `luminance` function described above.

For example, here is the greyscale representation of `tinypic` (produced using the debugging `print` function):

```
{{{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}}}
```

And here is the image itself:



Editor

We now describe a simple client program that uses the three image processing services described above. The program `Editor1.java` takes two command-line parameters: the name of a PPM file, followed by one of the operation codes `fh`, `fv`,

or `gr`. The program reads the image from the file, opens a graphics window, and displays in it a new image which is either the horizontally flipped, vertically flipped, or greyscaled version of the original image.

Implementation tips: the program should read the given file using the `read` function, call one of the three functions according to the given operation, and display the output image using the `show` function. All these functions are called from the `ImageEditing.java` class.

Write and test the `Editor1.java` program.

Scaling

Quite often, we want to change the proportions of 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 top 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 below it.

The scaling algorithm is as follows. Suppose that the width and the height of the source image are w_0 and h_0, and the width and height of the target scaled image should be w and h. With this notation in mind, the color of pixel (i,j) of the target scaled image should be set to the color of the pixel $(i * \frac{h_0}{h}, j * \frac{w_0}{w})$ 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 the color of pixel (4,6) of the source image.

The function `public static int[][][] scale (int[][][] source, int width, int height)` takes as input a digital image and two dimensions and returns a scaled version of the digital image according to the specified dimensions.

For example, `tinypic` is a 4 x 4 image. Suppose we want to scale it into a 3 x 5 image. If the array `pic` represents `tinypic` in our program, then the function call `scale(pic,3,5)` should produce the following matrix:

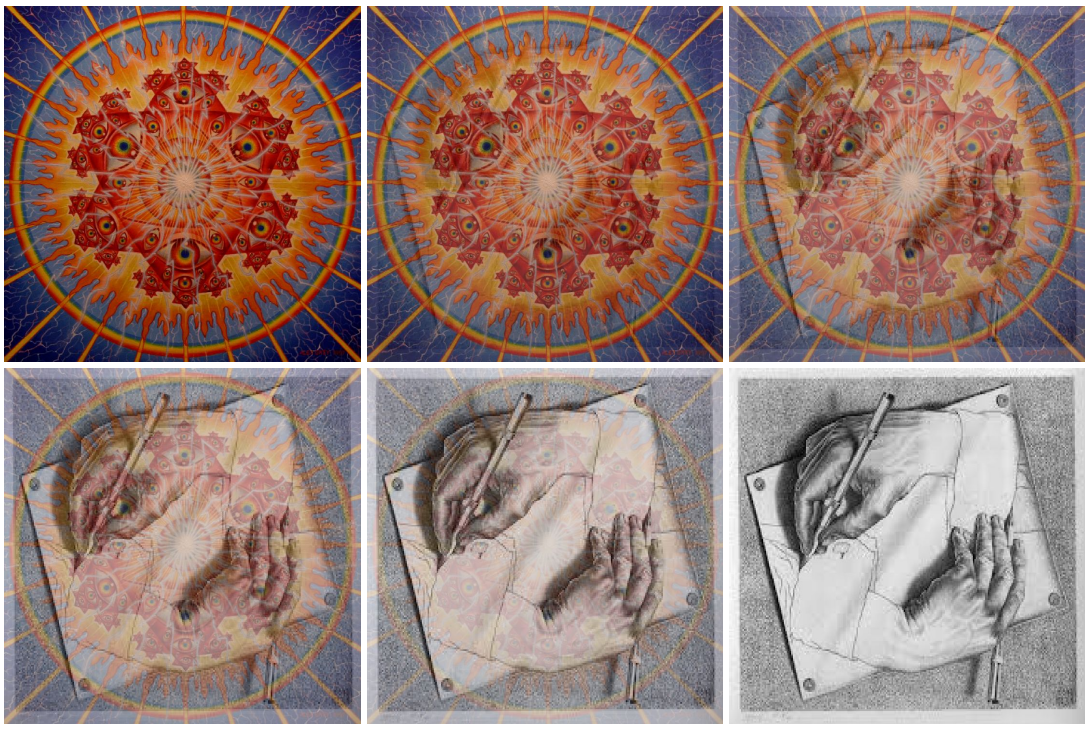
```
{{{0  , 0  , 0  }, {100, 0  , 0  }, {0  , 0  , 0  }},
 {{0  , 0  , 0  }, {100, 0  , 0  }, {0  , 0  , 0  }},
 {{0  , 0  , 0  }, {0  , 255, 75 }, {0  , 0  , 0  }},
 {{0  , 0  , 0  }, {0  , 0  , 0  }, {0  , 15 , 75 }},
 {{255, 0  , 255}, {0  , 0  , 0  }, {0  , 0  , 0  }}}
```

Note that scaling is a "lossy operation": some of the information contained in the original image can be lost during the scaling process.

Write the `scale` function. To test your implementation, write an `Editor2.java` program that takes three command-line arguments: a PPM file name, representing some image, followed by the desired width and height of the target scaled image. The program reads the source PPM file using the `read` function, scales it to the specified dimensions, and displays the result using the `show` function. All these functions are called from the `ImageEditing.java` class.

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 step-wise fashion. For example, here is an example of morphing some exotic Indian picture (the source image) into a famous Escher drawing (the target image), using 6 steps:



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

Blending: a blend of two pixels is a new pixel whose 3 RGB values are weighted averages of the RGB values of the two input pixels. The blending operation is parameterized by a real number $0 \leq \alpha \leq 1$ that determines how to blend the two inputs: the weight of the first input pixel is α, and the weight of the second input pixel is $1 - \alpha$. For example, suppose that the two input pixels are {100 ,40 ,100} and {200, 20, 40}. Blending them with $\alpha = 0.25$ produces the pixel {175, 25, 55}, as follows:

$$0.25 * 100 + 0.75 * 200 = 175$$
$$0.25 * 40 + 0.75 * 20 = 25$$
$$0.25 * 100 + 0.75 * 40 = 55$$

The function `public static int[] blend (int[] pixel1, int[] pixel2, double alpha)` returns a pixel blended according to the process described above. Note that the resulting pixel must consist of integer values. Write and test the `blend` function.

Combining: Two images of the same resolution can be combined by blending all the corresponding input pixels using a given α. The function `public static int[][][] combine (int[][][] source1, int[][][] source2, double alpha)` returns the `alpha`-blending of the two given source 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 `combine` function.

Morphing: suppose we want to morph a source image gradually into a target image in n steps. To do so, we stage a sequence of $0, 1, 2, \ldots n$ 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 `public static void morph (int[][][] source, int[][][] 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 starts by rescaling them as necessary. At the end of each blending step, the function uses the `show` function to display the intermediate result. Write and test the `morph` function.

Fading to Grey: to demonstrate your work, write a program `FadeToGrey.java`. The program takes two command-line arguments: the name of an input PPM file representing an image, and a number representing the desired number of fading steps. The program displays an animated step-wise view of how the given colorful image fades into grey (illustrating what will happen to your hair in some point of the future, a process which may be somewhat accelerated by working on HW 5).

If everything works well, after experimenting with various n values you'll be able to produce a nice animation of the fade effect between the two images.

Submission

Before submitting this assignment, take some time to inspect your code, check that your functions are short and precise. If you find some repeated code, consider making it into a function. make sure your program is written according to our **Java Coding Style Guidelines**. Also, make sure that each program starts with the program header described in the **Homework Submission Guidelines**. Any deviations from these guidelines will result in a point penalty.

Submit these files only:

- `ImageEditing.java`
- `Editor1.java`
- `Editor2.java`
- `FadeToGrey.java`

Deadline: Submit your assignment no later than December 27, 23:55.