CS 5004 Object-Oriented Design

Assignment 8: An Image-Processing Application

Due Date: Monday, April 1st, 11:59 pm.

This homework should be done individually.

1 Introduction

Many applications use color images. A good number of these provide a way to change their appearance in different ways. For example, Instagram has "filters" that convert a picture into something more interesting. They do this by editing the colors of individual dots in the image (called pixels).

In the next several assignments, you will incrementally build an application that works with such images, implement some simple but interesting operations on them, and build a program around them complete with "image scripting" and an interactive graphical user interface. This is the first of these assignments.

2 Images

Whatever the file format (jpg, png, bmp, etc.) an image is basically a sequence of pixels. Each pixel has a position in the image (row, column) and a color. For a color image, the pixel's color is stored using three numbers: red, green, blue. Any color is a combination of these three *base* colors. To see how various colors can be made using these base colors: open IntelliJ ... Type "Ctrl+Shift+A" on Windows or "Command+Shift+A" on Mac and type "show Color Picker". When selected, you can pick different colors and observe their red, green and blue values to get an intuition for how this works).

Each of these colors is called a "channel" in the image (i.e. a color image has 3 channels, ignoring transparent images). Each channel is usually stored using 8-bits, that is 256 distinct values. This is where the name "24-bit image" comes from (3 8-bit channels)!

Using all this information, an image can be thought of as simply as a 3D array of integers with $rows = height$, $columns = width$ and $depth = 3$. With 8-bit channels, each value is between 0 and 255. The provided `ImageUtils` class includes methods that allow you to read and write actual image files using 3D arrays. You can use them to work with any image files of standard formats.

3 Generating images

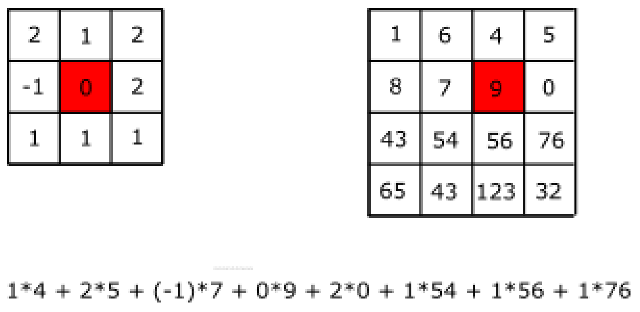
While many images are photographs, some images can be generated by a computer. These images often have a structure that makes this generation possible. Such images are useful in many contexts, and it is often more convenient to be able to generate them when needed, rather than read from a file. Generating an image is as simple as specifying the red, green and blue values for each pixel in the image.

Examples of images that can be generated by a computer include those with rainbow-colored stripes, checkerboard or Chinese checker patterns, etc. Flags of many nations (see `https://en.wikipedia.org/wiki/List_of_national_flags_by_design`) can also be generated computationally. Note that the flag sizes may be variable, but their proportions are prescribed.

4 Image Filtering

A basic operation in many image processing algorithms is filtering. A filter has a "kernel," which is a 2D array of numbers, having odd dimensions (3×3, 5×5, etc.). Given a pixel in the image and a channel, the result of the filter can be computed for that pixel and channel.

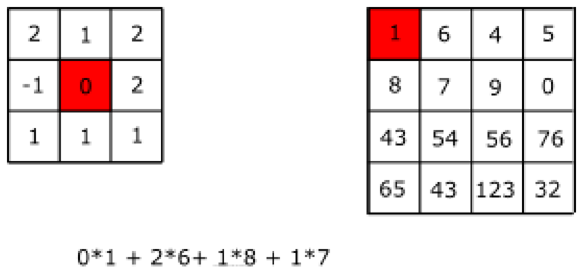
As an example, consider a 3×3 kernel being applied for the pixel at $(5, 4)$ (row 5, column 4) of the red channel of an image. We place the center of the kernel at the particular pixel (e.g. `kernel[1][1]` overlaps pixel $(5, 4)$ in channel 0. This aligns each number in the kernel with a corresponding number in that channel (`kernel[0][0]` aligns with pixel $(4, 3)$, `kernel[1][2]` aligns with pixel $(5, 5)$, etc.). The result of the filter is calculated by multiplying together corresponding numbers in the kernel and the pixels and adding them. If some portions of the kernel do not overlap any pixels, those pixels are not included in the computation. The picture below illustrates the filter operation.



2	1	2
-1	🟥 0	2
1	1	1

1	6	4	5
8	7	🟥 9	0
43	54	56	76
65	43	123	32

2*6 + 1*4 + 2*5 + (-1)*7 + 0*9 + 2*0 + 1*54 + 1*56 + 1*76



2	1	2
-1	🟥 0	2
1	1	1

🟥 1	6	4	5
8	7	9	0
43	54	56	76
65	43	123	32

0*1 + 2*6+ 1*8 + 1*7

4.1 Applications of filtering

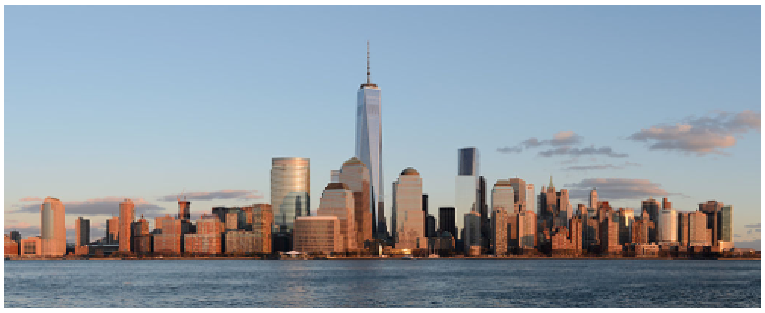
Many image processing operations can be framed in terms of filtering (i.e. they are a matter of designing an appropriate kernel and filtering the image with it).

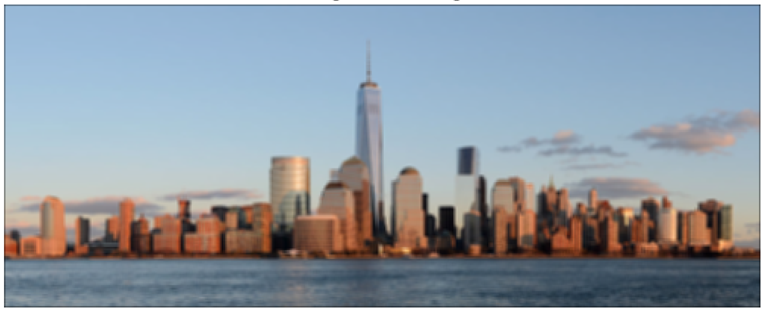
4.1.1 Image blur

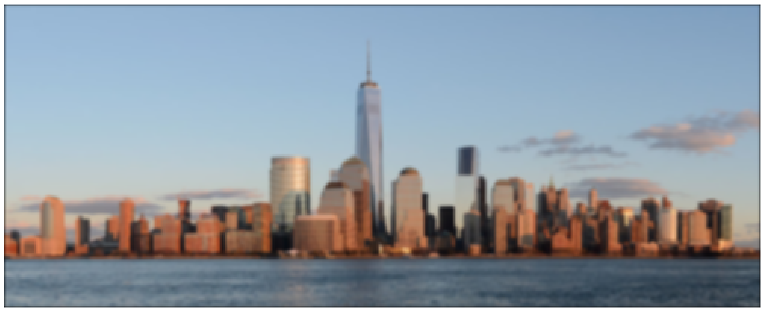
Blurring an image is probably the simplest example of filtering. We can use a 3×3 filter as follows:

$$\begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix}$$

Blurring can be done by applying this filter to every channel of every pixel to produce the output image. Here is an example of blurring an image this way:


Original Image


Blurred Image


More Blurred Image

The filter can be repeatedly applied to an image to blur it further. The last image above shows the result of applying the same filter to the second image.

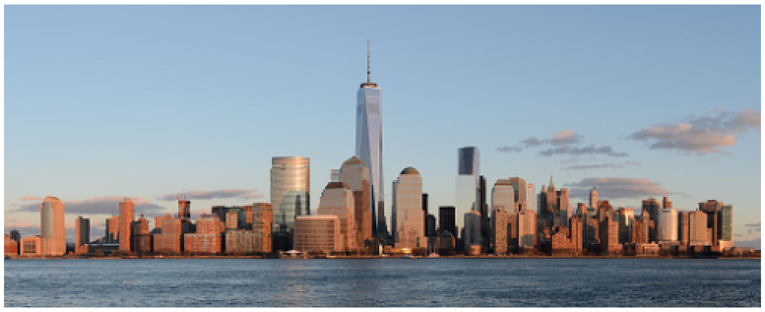
4.1.2 Image sharpening

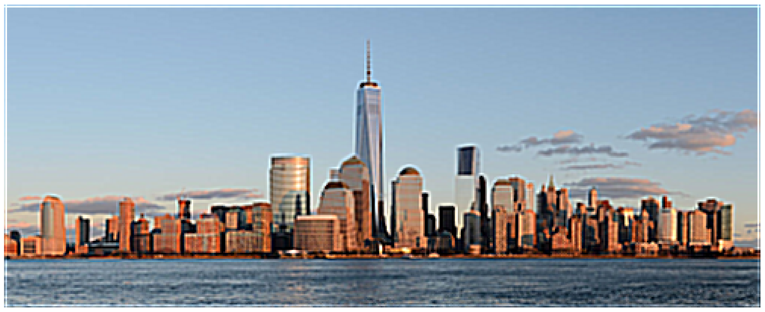
Image sharpening is in some ways, the opposite of blurring. Sharpening accentuates edges (the boundaries between regions of high contrast), thereby giving the image a “sharper” look.

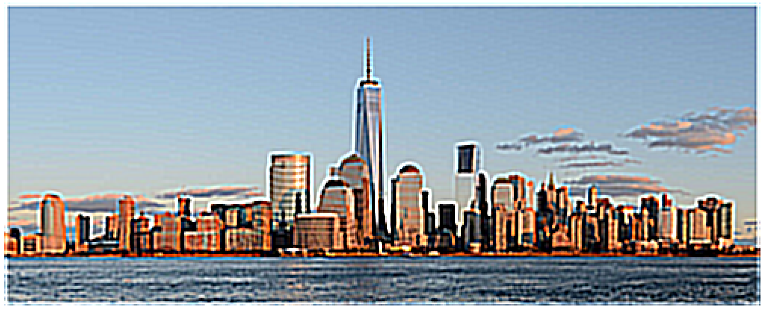
Sharpening is done by using the following filter:

$$\begin{bmatrix} -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \\ -\frac{1}{8} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{8} \\ -\frac{1}{8} & \frac{1}{4} & 1 & \frac{1}{4} & -\frac{1}{8} \\ -\frac{1}{8} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{8} \\ -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \end{bmatrix}$$

As with blurring, it is possible to repeatedly sharpen the image. Here is an example of sharpening an image:


Original Image


Sharpened Image


More Sharpened Image

4.2 Clamping values

Often filtering results in an image where some values are beyond their range. For example, using 8 bits per channel means each value in each channel is between 0 and 255. Filtering such an image may cause some values to be outside this range. We must "clamp" these values to avoid overflow and underflow while saving, and to display such images properly. Clamping requires two values: the permissible minimum and maximum. Then each value in an image that is lesser than the minimum is assigned to the minimum, and each value greater than the maximum is assigned to the maximum.

Clamping is implemented as the last step of any filtering operation, to ensure that the resulting image can be properly saved and displayed.

5 Color transformations

Filtering modifies the value of a pixel depending on the values of its neighbors. Filtering is applied separately to each channel. In contrast, a color transformation modifies the color of a pixel based on its own color. Consider a pixel with color (r,g,b). A color transformation results in the new color of this pixel to be (r',g',b') such that each of them are dependent only on the values (r,g,b).

A simple example of a color transformation is a linear color transformation. A linear color transformation is simply a color transformation in which the final red, green and blue values of a pixel are linear combinations of its initial red, green and blue values. For example, if the initial color is (r,g,b), then the final red value being $0.3r + 0.4g + 0.6b$ is a linear color transformation.

Linear color transformations can be succinctly represented in matrix form as follows:

$$\begin{bmatrix} r' \\ g' \\ b' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} * \begin{bmatrix} r \\ g \\ b \end{bmatrix}$$

where a_{ij} are numbers. Using the above notation, the final values will be:

$$\begin{aligned} r' &= a_{11}r + a_{12}g + a_{13}b \\ g' &= a_{21}r + a_{22}g + a_{23}b \\ b' &= a_{31}r + a_{32}g + a_{33}b \end{aligned}$$

Similar to filtering, clamping may be necessary after a color transformation to save and display an image properly.

5.1 Applications of color transformations

There are many image processing operations that can be expressed as color transformations on individual pixels (i.e. implementing them is a matter of using the correct numbers a_{ij} on all pixels of the image).

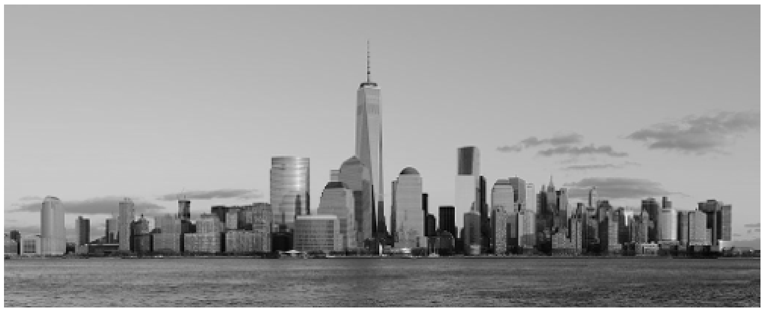
5.1.1 Greyscale

A simple operation is to convert a color image into a greyscale image. A greyscale image is composed only of shades of grey (if the red, green and blue values are the same, it is a shade of grey). There are many ways to convert a color image into greyscale. A simple way is to use the following color transformation: $r' = g' = b' = 0.2126r + 0.7152g + 0.0722b$. This is a standard formula to compute the "luma" for high-definition television (read more about it at `https://en.wikipedia.org/wiki/Rec._709`).This can be expressed as a color transformation in matrix form as:

$$\begin{bmatrix} r' \\ g' \\ b' \end{bmatrix} = \begin{bmatrix} 0.2126 & 0.7152 & 0.0722 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.2126 & 0.7152 & 0.0722 \end{bmatrix} * \begin{bmatrix} r \\ g \\ b \end{bmatrix}$$

The effect of this conversion is illustrated below:

Original Image


Greyscale Image

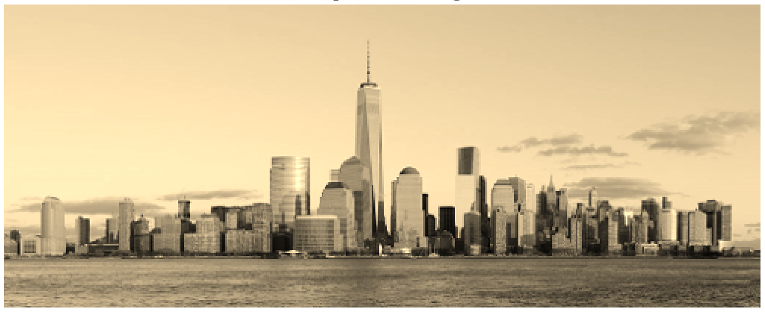
5.1.2 Sepia tone

Photographs taken in the 19th and early 20th century had a characteristic reddish brown tone. This is referred to as a sepia tone (see the origin of this term at `https://en.wikipedia.org/wiki/Sepia_(color)`). Most image processing programs allow an operation to convert a normal color image into a sepia-toned image. This conversion can be done using the following color transformation:

$$\begin{bmatrix} r' \\ g' \\ b' \end{bmatrix} = \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix} * \begin{bmatrix} r \\ g \\ b \end{bmatrix}$$

The effect of this conversion is illustrated below:

Original Image



Sepia-toned Image

6 What to do

In this assignment you must design and implement the **model** of this image processing application. Your model must represent images suitably and should offer the above operations. Instead of writing a formal controller you will write a simple "driver" main method that suitably loads and saves images. In this way your code can function as a standalone working program.

Keep in mind that this is a multi-part project, so design accordingly. Try not to cater your design to only these requirements, but attempt to generalize, abstract to make it possible to add features to your program later on.

This assignment does not require you to test using JUnit, but rather try out the program and verify results visually. However it may help to write tests for certain parts of your program.

6.1 Required features

Your program must be able to load and save images. It should be possible to blur and sharpen images loaded into your program. It should also be able to generate images with horizontal and vertical rainbow stripes, and a checkerboard pattern (you must generate these images in your code, not simply submit these images). For the rainbow, the image should be of a user-specified size. All stripes must be of the same thickness/width (depending on whether the stripes are horizontal or vertical). If this is not possible for the provided dimensions, the last strip may be up to 7 pixels thinner/shorter. For the checkerboard pattern the squares must be of a user-specified size.

6.2 Extra credit

For extra credit, your program should be able to convert images to greyscale and sepia. This will be a required feature of your program later, so you can earn extra credit for implementing future features now. Think of it as delivering on features initially promised to the user in Version 2.0!

For smaller extra credit, your program should offer ways to create the flags of France, Greece and Switzerland of a user-specified size, but of the prescribed proportions (look for the proportions in the link in Section 3). Again, you must generate these images, not simply submit them as files. The design should strive to support adding more flags later if needed.

7 Criteria for grading

You will be graded on:

1. Your design (interfaces, classes, method signatures in them, etc.)
2. Whether your code looks well-structured and clean (i.e. not unnecessarily complicating things, or using unwieldy logic).
3. Correctness of your implementations, evidenced in part by the images you submit.
4. Whether you have written enough comments for your classes and methods, and whether they are in proper Javadoc style.
5. Whether you have used access modifiers correctly: `public` and `private`.
6. Whether your code is formatted correctly (according to the style grader).

8 What to submit

Your zip file should contain three folders: `src`, `test`, and `res` (even if empty). The `res` folder should contain at least two images and their blurred, sharpened, greyscaled and sepia-toned results, along with generated images. Your submission should have a main method that illustrates how all of this can be done (parts of it may be commented out). Your submission must also include a `README.md` file that documents how to use your program, which parts of the program are complete and includes citations for each image you have used. It is your responsibility to ensure that you are legally allowed to use any images, and your citation should be detailed enough for us to access the image and read its terms of usage. If an image is your own photograph or other original work, specify that you own it and are authorizing its use in the project.