Fundamentals of Programming in Java

Exercises 2

Digital Image Processor

Marks available: 100

Date issued: 01/10/2018

Deadline: 1200, 19/11/2018

Submission instructions

Submit your work by the deadline above. The accepted submission format is zip. Submit only .java source files in your archive. Do NOT submit .class files. All of your Java files must be in the same zip file. Please do not submit Eclipse (or other IDE) projects.

Do not output anything to the console unless explicitly asked to.

Your Java file must have the following declaration at the top of it.

```
package com.bham.pij.exercises.e2a;
```

Since all of your methods will be in the same file, you will only need this precise package declaration and no others.

In each of these exercises, you are not restricted to only creating the *required* methods. If you want to, you can create others. However, you must create **at least** the required methods.

Do not import any libraries other than those already imported. Do not use code from other libraries in any other way. You must implement your methods using only the built-in arrays. Do not copy code from elsewhere that uses other techniques. If in doubt, please ask Ian.

Introduction

For this set of exercises you will complete some methods in an image processing tool. You do not need to create the tool itself. That is provided for you on Canvas as the file `ImageProcessor.java`. You are simply completing some methods so that the functionality is fully present.

Digital images

A digital image is essentially a 2D array of color values. Each color element of the 2D array is called a 'pixel' (short for 'picture element'). Each pixel has a color. This color can be represented in a number of ways. One of the most common ways is to represent each pixel color as a combination of the colors red (R), green (G) and blue (B). This is the RGB color model. Using this model, each pixel usually has three bytes of data associated with it: one for red, one for green and one for blue. (Another byte of data called the 'transparency' is also usually associated with each pixel but we aren't going to worry about that value).

If using an integer type, we can see that one byte of data per color per pixel leads to a range of 0 to 255 different values. If the value of that color (red, green or blue) is set to zero it means that none of that color is represented in that pixel. If it is set to 255 it means that the maximum level of that color is represented in that pixel. Since each pixel has a value for each of the RGB values, this means that we can mix the colors together to get the colors we want.

If we set all three RGB values to zero we obtain the color black. If we set them all to 255 we get the color white. If we set them all to the same value, somewhere between 0 and 255, we get gray. If we set them to different values we then get other colors. There are many online tools for playing around with colors in this way and I recommend doing that if you are unfamiliar with this topic.

Image filters

Image processing involves amending the color values of the pixels in an image in some way, or creating a new image from an old one having processed the old one in some way. In these set of exercises you will be largely applying *filters*[1] to images (but there are some slightly different techniques to be used - described below).

A filter is a 2D array of values. The filter is usually a small array, for example 3 rows by 3 columns (3x3). The values in the filter are used to modify each pixel in the source image in a specific way, to be described below. Each single pixel in the source image is modified by the *entire* filter.

The color value of each pixel to be filtered is computed as follows.

Consider a filter matrix f of dimension 3x3:

$$\begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix}$$

Consider a digital image img, of dimension 8x8[2], represented as a matrix of colors:

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & \dots & c_{07} \\ c_{10} & c_{11} & c_{12} & \dots & c_{17} \\ c_{20} & c_{21} & c_{22} & \dots & c_{27} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{70} & c_{71} & c_{72} \dots & \dots & c_{77} \end{bmatrix}$$

[1] Also called *kernals.*

[2] This is just an example. This would be an incredibly small image.

As an example, consider pixel c_{22}. The color of pixel c_{22} after applying the filter is computed as follows:

$$\begin{aligned} c_{22}^{new} = \quad & x_{00} \times c_{11} + x_{01} \times c_{12} + x_{02} \times c_{13} \\ & + x_{10} \times c_{21} + x_{11} \times c_{22} + x_{12} \times c_{23} \\ & + x_{20} \times c_{31} + x_{21} \times c_{32} + x_{22} \times c_{33} \end{aligned} \tag{1}$$

Note that this is not a regular matrix multiplication. It is a matrix convolution.

Essentially, we consider the filter as being placed centrally on the pixel for which we want to compute the new value. Thus, the element [1,1] in the filter is positioned 'over' the pixel to be computed.

This computation is applied to all of the pixels in the image. This means that the filter is 'moved' across the image from the top-left corner, pixel by pixel. Thus every new pixel value is the sum of nine multiplications.

Consider the identity filter. The identity filter leaves the source image unchanged. The 3x3 identity filter is shown below:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

If you consider this filter, and the equation for the computation of the new pixel value shown above, you will see that this matrix will leave the color of the pixel unchanged. Only the element [1,1] will contribute to the result (since all other terms will be zero), and element [1,1] is positioned over the original pixel value.

Edge cases

How do we compute the value of pixel [0,0]? If we position the filter over this pixel, it means that parts of the filter will **not** be over the image since the filter element [1,1] will be placed over the target pixel. This means, for example, that filter element [0,0] will not be over the image.

This is literally an 'edge case'. What should we do? Well, you can read up about the various techniques that are used to overcome this issue, but the solution used for the tool you are working on is a very simple one and is explained below under 'Implementation details'.

Implementation details

As you will see below under the specific exercises below, you need to implement four methods. The methods you will implement are largely about producing a new color array from an original color array. For the filtering techniques, this requires the specification of a filter. For sepia and grayscale, the process is slightly different, as shown below.

Dealing with the edge problem

For the `applyFilter()` method, the `Color` pixel array that will be passed to your method will have been augmented as described here.

To overcome the edge case issue described above, the array that your method receives has been increased in size compared to the original image array. A one-pixel border (set to the color white) has been added around the image. This is so that you can apply your filter without having to worry about the edge of the image. This does mean that, at the edges of your filtered image, there will be a slightly odd color for each pixel because it will have been mixed with varying degrees of white (apart from for the identity filter).

Thus, an image of dimensions $n \times m$ pixels will be passed to your method as an array of $n+2 \times m+2$ pixels.

However, the array returned by your `applyFilter()` method must be the same size as the original image, so you do not include this border in your result. In the example above, your returned array would be be $n \times m$.

The `Color` class

Spend some time looking at the Java `Color` class API. You don't need to know too much about it but please note the following.

Although we described the RGB values above as being in the range [0,255], you will actually specify them using `double` values in the range [0,1]. These are exactly equivalent. The value 0 maps to 0 and 1 maps to 255.

To compute a `Color` value you must compute the red, green and blue components. Note that this means that equation (1) must be used three times: once for each component to create a new red, green *and* blue value.

To create a new `Color` value from three `double` values called `red`, `green` and `blue` you do the following:

```
Color color = new Color(red,green,blue,1.0);
```

Note that the final parameter is the transparency value and you should always set that to 1.0.

One other thing to consider: when you compute the new red, green and blue values that you want for a pixel, you must ensure that the values are still in the range [0,1] otherwise you will cause an Exception.

Sepia

The creation of a sepia image requires a different process. It is not filtered, as such. For sepia, each pixel's color value is simply changed according to the following formula:

$$red^{new} = red^{original} \times 0.393 + green^{original} \times 0.769 + blue^{original} \times 0.189 \quad (2)$$

$$green^{new} = red^{original} \times 0.349 + green^{original} \times 0.686 + blue^{original} \times 0.168 \quad (3)$$

$$blue^{new} = red^{original} \times 0.272 + green^{original} \times 0.534 + blue^{original} \times 0.131 \quad (4)$$

Grayscale

The creation of a grayscale image also requires a different process that has more in common with the process for a sepia image than with filtering.

For grayscale, each pixel's color value is simply changed according to the following formula:

$$red^{new} = green^{new} = blue^{new} = (red^{original} + green^{original} + blue^{original}) \div 3 \quad (5)$$

The Exercises

Exercise 2a: `createFilter()` **[10 marks]**

For this task you should create a method with the following signature:

```
public float[][] createFilter(String filterType)
```

As you can see, this method receives a `String` as a parameter and returns a 2D array of `float`s.

The parameter will have one of the following values:

```
{"IDENTITY","BLUR", "SHARPEN", "EMBOSS", "EDGE"}
```

The method should return a 3x3 array containing the appropriate values for the parameter. These are:

Identity {{0,0,0},{0,1,0},{0,0,0}}

Blur {{0.0625f,0.125f,0.0625f},{0.125f,0.25f,0.125f},{0.0625f,0.125f,0.0625f}}

Sharpen {{0,-1,0},{-1,5,-1},{0,-1,0}}

Edge {{-1,-1,-1},{-1,8,-1},{-1,-1,-1}}

Emboss {{-2,-1,0},{-1,0,1},{0,1,2}}

Therefore, for example, for the Identity filter, your method must return a 2D array of dimensions 3X3 with all elements set to zero except for element [1][1] which will have the value 1.

For these filters you must use these exact values. Do not use other values that you have looked up elsewhere. Of course, if you want to experiment with other filters, you can set those values to anything you wish. Just make sure you implement the ones required here.

Exercise 2b: `applyFilter()` [40 marks]

For this task you must create a method with the following signature.

```
public Color[][] applyFilter(Color[][] pixels, float[][] filter)
```

This method receives as parameters one of the filters (created by your method from Exercise 2a) and a 2D array of `Color` values. Remember that the pixel array that your method receives will have been augmented by a one-pixel border, as described above. Your method will apply the filter to the `Color` array and return a 2D `Color` array *minus* the one-pixel border.

Exercise 2c: `applySepia()` [10 marks]

For this task you must create a method with the following signature.

```
public Color[][] applySepia(Color[][] pixels)
```

The `Color` array passed to this method will **not** have the one-pixel border. To create the sepia image you must apply the equations (2), (3) and (4) to the colors in the input array,

Exercise 2d: `applyGrayscale()` [10 marks]

For this task you must create a method with the following signature.

```
public Color[][] applyGrayscale(Color[][] pixels)
```

The `Color` array passed to this method will **not** have the one-pixel border. To create the grayscale image you must apply the equation (5) to the colors in the input array,