Programming in Java

Assignment 5

Edge Detection Tool

Marks available: 100

Date issued: 29/11/19

Deadline: 11/12/19, 17:30

You may not use the JavaFX Scene Builder tool (or equivalent) for this assignment.

You may not use any image processing libraries that are either part of the Java API or are from elsewhere apart from the ones listed in this document. You must implement the image processing component of this tool 'manually' yourself.

This assignment is worth 12.5% of your final module mark.

Exercise 1 is worth 25% of the assignment marks. You will receive all of those marks if your GUI has the GUI elements specified below (and you have added them in a reasonable fashion to the GUI - like the example in the lecture). You may add other elements to your GUI as you see fit, however. These additional elements will not receive extra marks.

Exercise 2 is worth 75% of the assignment marks. This will be assessed 100% based on the functionality. The lack of 'quality' marks does not mean that you should ignore everything that you have learned so far about layout, naming, design, etc.

Submission instructions

Submit your work by the deadline above as a zip archive. Use only 'zip' format and no other. Submit only .java source files in your archive. Do NOT submit .class files or any type of file. Please do not submit entire Eclipse (or other IDE) projects. In your zip you must submit all Java files necessary for the assignment, including any that were provided for you. You do not need to submit any test files that were provided.

To submit, gather all of the Java files into a folder, zip that folder, and then submit the zip. Do this using your operating system functionality. Do not use third-party tools or websites to compress your files.

In your programs, do not output anything to the console unless explicitly asked to. If you add `System.out.println()` calls in your code (e.g. for debugging), remove them before you submit.

Each Java file must have a package declaration at the topic of it (these declarations must be on the very first line of the file). The package declaration this assignment is:

com.bham.pij.assignments.edgedetector;

All classes that you write for this assignment must have this package declaration. **Do not create new packages**. Do not forget to add the semicolon at the end of this line. All characters in a package name should be lower case.

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 put any code that gets user input in the required methods below. If you do, your work can't be tested. This means that you should not use `Scanner` in any required method or in any method called by a required method, etc.

Please follow these instructions. If you do not you could get a lower mark than you would have done for following them or a mark of zero. Please ask if you are unsure.

Background

For this assignment you will create an image processing tool that can perform *edge detection* on images. The edge detection applied is a simple filtering of the image using a filter.

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 grey. 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. If you have done any CSS then you are probably already familiar with this concept.

For the purposes of this assignment, you can assume that all images are **square**, i.e. *image width = image height*.

The pixel data will actually be represented by objects of type `Color`, as discussed below.

Applying an image filter

Image filtering 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 this set of exercises you will be applying a *filter*[1] to images.

A filter is a 2D array of floating point 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:

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

As an example, consider pixel c_{22}. We compute the new value of c_{22} after applying the filter 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 convolution.

Essentially, we consider the filter as being placed centrally on the pixel for which we want to compute the new value. Thus, the filter element [1,1] (the central element in a 3x3 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 result of 27 multiplications (in the case of a 3x3 filter).

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

[1] Also called a *kernel.*

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

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

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. Testing your filtering code with the identity filter is a good idea. If it leaves the image unchanged then the filtering code might be working.

To perform edge detection using a filter, the following filter should be applied to the image data. This is the filter you will use for for edge detection:

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

Edge cases

How do we compute the new, filtered 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 but will be 'off the edge' of the image.

This is literally an 'edge case'. What should we do? For this assignment you must take the following approach.

The overall approach will be to augment the image with a one-pixel border that is set to the color black. This is like a one-pixel 'frame' around the image that is black in color.

The image that you load will be a square image, as stated above. This means that if its width is n pixels then its dimensions are $n \times n$ pixels. Before you attempt any operations on the image data, you must create a new array that is of dimensions $n+2 \times n+2$ pixels. This array is one pixel bigger than the original image array at the left, right, top and bottom of the image. This border must be set to the color black. The rest of the array is a copy of the original image. This means that the original image data is offset by one pixel in the x and y dimensions in the new array.

Your image processsing code will work on this larger image until it is ready to return the pixel data for the filtered image. The final image data must be the **same size** as for the original image.

Greyscale

The second process you will apply, after creating the bordered image, will be to convert the whole image to greyscale. This does not require a filter. This can be done on a pixel-by-pixel basis as follows.

For greyscale, each new pixel's color value is computed according to the following formula:

$$red^{new} = green^{new} = blue^{new} = (red^{original} + green^{original} + blue^{original}) \div 3 \tag{2}$$

Note that this simply sets all three color channels for a pixel to same value - the mean of the red, green and blue values.

Implementation details

The `Color` class

The pixel data will actually be represented by objects of type `Color`. 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 for a pixel 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);
```

Once this new `Color` object has been created, you can add it your array.

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.

Saving an image file

You may wish to save the pixel data you have create as an image file so you can look at it more carefully in an image tool, etc. The following code can be used to do that. Note that `pixels` is the color data created by you, and `filename` is the name you want to give to the output image file. Make sure it has the extension `.png`.

```
private void saveImage(Color[][] pixels, String filename) {

  WritableImage wimg = new WritableImage(pixels.length, pixels.length);

    PixelWriter pw = wimg.getPixelWriter();

    for (int i = 0; i < pixels.length; i++) {
        for (int j = 0; j < pixels.length; j++) {
            pw.setColor(i, j, pixels[i][j]);
        }
    }

  BufferedImage bImage = SwingFXUtils.fromFXImage(wimg, null);

  try {
     ImageIO.write(bImage, "png", new File(filename));
  } catch (IOException e) {
     e.printStackTrace();
  }
```

```
}
```

Exercise 1: The GUI [25 marks]

Make sure you have read and understood the lecture on GUI creation with JavaFX. That covers virtually everything you need for this assignment.

You need to create a GUI for this tool that has the following minimal requirements.

1. The stage dimensions must be: width = 500, height = 500.
2. The stage must have an appropriate title.
3. The GUI must have two menus: one called *File* and one called *Tools*.
4. The *File* menu must have two menu items: one called *Open* and one called *Close*.
5. The *Tools* menu must have two menu items: one called *Edge Detection* and one called *Revert*.
6. Menu item operations that are not possible given the current state of the tool should not be allowed.

The GUI elements will invoke the functionality that is created in Exercise 2 (the marks for this part of the assignment are purely for the GUI, however).

When the *Open* menu item is selected, the user should be presented with a file chooser for selecting an image file. Once the file has been selected, the image should be displayed in the GUI. When the *Close* menu item is selected, the image should be removed from the GUI.

When the *Edge Detection* menu item is selected, the selected image should be filtered as required by the assignment, and then the new image displayed. When the *Revert* menu item is selected, the original image should be displayed.

You can create whichever classes you feel you need for your GUI but note that they should be distinct from the class described in Exercise 2.

Exercise 2: Edge Detection [75 marks]

Do not put any GUI code in this class.

You must create a class called `EdgeDetector` that has the methods specified below.

```
public Image filterImage(Image image)
```

This is the most important method (although the others below must also exist and work).

This method receives an `Image` as a parameter and applies the edge detection filter to it. It then returns an `Image` with the same dimensions as the original image in the parameter but containing the filtered pixel data.

This method needs to:

1. Create a new `Color` array in which the image pixel data has been augmented with the one-pixel border (by calling the method `getPixelDataExtended()`).

2. Create a greyscale version of the image (including the border) (by calling the method `applyGreyscale()`).

3. Apply the edge detection filter to the greyscale image (by calling the method `createFilter()` to create the filter and the method `applyFilter()` to apply the filter).

4. Return the new, filtered image data in a `Color[][]` array with the same dimensions as the original input image.

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

This method will create a 3x3 float array containing the values from the edge detection filter as specified above.

```
public Color[][] getPixelDataExtended(Image image)
```

This method receives an image as a parameter. It reads the pixel color data from the image and adds it to a new `Color` array that has a one-pixel border (around the original image data) that has been set to black. It then returns this new, larger array.

```
public Color[][] getPixelData(Image image)
```

This method does the same as the previous one except that it returns the whole image data as a `Color` array (i.e. this method ignores the existence or otherwise of the border).

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

This method applies the greyscale operation to the pixel data in the parameter and returns it as an array of `Color` values.

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

This method applies the filter in the parameter `filter` to the pixel data in the parameter `pixels` and returns the new data as a `Color` array. You an assume that `filter` is always a 3x3 array.

You need to provide all of these methods. You may not see the need for them all but they are needed so that your code can be tested.

Implementation hints

1. You may wish to create the GUI first. That is covered mostly in the relevant lecture and should not take too long. You can include all funtionality that is not to do with edge detection. For example, you can add the code that opens the file chooser and allows the user to select an image file.

2. You should then create the `EdgeDetector` class. Add all of the required methods to it but don't complete them yet. Add an instance of class `EdgeDetector` to (one of) your GUI class(es).

3. Now add the code required to call your `EdgeDetector` methods to the GUI class. For example, when the menu item *Edge Detector* is selected, you will need to call the method that actually performs the edge

detection on the selected image. At first you could simply add some `System.out.println()` calls to show that the correct methods are being called.

4. Then complete the rest of the methods. I suggest creating them in the following order, if you are not sure which order to do them in. This order is not compulsory.:

1. createFilter()
2. getPixelData()
3. getPixelDataExtended()
4. applyGreyscale()
5. applyFilter()
6. filterImage()

5. I suggest that early on you add a method to save a new image file, based on the code given above. This will allow you to save any of the intermediate images you create (e.g. with the border, the greyscale image, etc.) and actually look at them in an image editor or photo application to see what they look like.

6. You might try setting the border to a more obvious color like red, for testing purposes. You must change it back to black to submit, however, but red will be easier to see.

7. Two images have been uploaded. One is an image of a Rubik's Cube that will be used for the actual testing. The other is a very small single color image that you can use to test your code without having to process an entire, larger image.