You Can Make a Difference!

Due: Wed. 4/13 at 11PM (for Mon. lab), Thurs. 4/14 at 5PM (for Mon. evening), or Thurs. 4/14 at 11 (for Tues. aft.)

Implementation plans due at the start of your lab period (see page 7 for details!)

In last week’s lab, we introduced some of the basic mechanisms used to manipulate images in Java programs. In this week’s lab, we ask you to use your knowledge of arrays to:

• explore additional image processing operations, and
• implement several common algorithms for processing arrays.

Once again, we encourage (though don’t require) you to work with a partner on this lab.

The screen shot above shows the main window that the program you write this week should display. You should recognize the basic layout of the program’s window from last week’s lab. In fact, we will use the ImageViewer and AdjustLevels classes we asked you to write last week as a starting point for this program. The “Load Image” and “Take Selfie” buttons will function exactly as they were supposed to function last week. We will discuss the roles of the other buttons shortly.
The slider at the bottom of the window controls the color resolution with which an image is displayed. That is, it functions like the “Levels” slider you implemented in the last lab. In the sample window shown above, the image is displayed using 256 brightness values, the standard resolution for grayscale images. The illustration on the left shows the same image displayed using only 3 distinct brightness values.

The “Zoom In” and “Zoom Out” buttons perform simple versions of image scaling. Pressing “Zoom Out” replaces the image displayed with an image whose dimensions are half the dimensions of the original image and that looks like a reduced version of the original. One way to accomplish this is to set the brightness values for the pixel at each x,y position in the new image equal to the brightness values for the pixel at position 2*x, 2*y in the original image. The image below shows what would be displayed if one pressed “Zoom Out” after loading our sample image.

For the purpose zooming out, you should interpret “half” in a very specific way. If an image’s width is 2N, then the width of a half is obviously N. If the width is 2N+1, then your program should produce a “half” whose width is N+1. That is, when dividing the width of the image by 2 you should round up (even though the default behavior of the Java division operation is to round down). This way, you will not lose the information in the image’s last row or column of pixels when zooming. The height of the new image should be determined in a similar way.

“Zoom in” should perform the opposite transformation. It should produce an image twice as large as the image your program is currently displaying. The pixel at position x,y in the new image should have the same brightness values.
as the pixel at position \(x/2,y/2\) in the source image. As a result, the zoomed image will be composed of 2x2 blocks of pixels with identical colors. The image to the right shows a portion of the image produced by clicking “Zoom In” twice after loading our sample image. Applying the transformation twice leads to 4x4 blocks of pixels of identical color.

Three additional buttons are provided in the program’s interface. The “Show Histogram” button creates a new window containing histograms of the brightness values associated with the red, green, and blue components of the displayed image’s pixels. An example of such a histogram window is shown below. The graph in the left third of the window shows the distribution of redness values in the image, the graph in the middle shows the greenness values, and the rightmost graph shows the blueness values. This particular set of histograms was created using an image showing pinkish flowers with quite a bit of green foliage.

The “Expand Range” button adjusts the brightness values of an image’s color components linearly so that they range all the way from 0 to 255. This operation has no effect on an image with a histogram like the one shown above since the pixel values already stretch from 0 to 255. This operation is valuable, however, on images that are either too dark (the largest brightness value is much less than 255) or too bright.

This week’s ImageViewer class will save a copy of the last image placed in the display using either the “Load Image” or “Take Selfie” buttons. This saved image may be different from the image actually displayed in the window if the user adjusts the image resolution using the slider or presses the “Expand Range” or either of the “Zoom” button. To implement this program correctly, it is important to understand when an operation should be performed using the saved image and when it should be performed using the currently displayed image. As explained in last week’s lab handout, whenever the slider that controls the number of brightness levels used is moved, the program should use the brightness values from the original image to determine the image that should be displayed. On the other hand, when the user presses either of the “Zoom” buttons or the “Expand Range” button, the operation should start with the brightness
values of the displayed image to compute a new image to display. This is particularly important for zooming because it makes it possible to repeatedly increase or decrease the size of an image by clicking on one of the “Zoom” buttons several times.

This functionality will have one odd result. If you zoom an image and then adjust the slider, the image will return to its original size! Try to think of this as a feature.

The final button in the program’s interface is “Show Difference”. When the user presses the “Show Difference” button, the program creates an image with brightness values determined by the differences between the pixel brightness values of the image currently displayed in the window and those of the image that was originally loaded. This image is then displayed in a new, independent image viewer window like the one shown on the right. This window shows the differences between the original image shown on the first page of this handout and the version of the image produced by setting the Levels slider to 3 (as shown on the second page). Images produced using “Show Difference” often become more interesting when “Expand Range” is applied to them.

If the “Show Difference” button is pressed after either of the “Zoom” buttons, the dimensions of the images involved in the difference operation will not be the same. In this case, the width of the difference image should equal the smaller of the widths of the two images involved and the height of the result should equal the smaller of their heights. In any case, the difference value found in position x,y of the result image should be obtained by subtracting the brightness values for the pixels at position x,y in each of the two source images.

**Class Structure**

The program you will write this week will be composed of 5 distinct classes. The good news is that we will give you working versions of three of them (ImageViewer, AdjustLevels, and DisplayHistograms). The versions of ImageViewer and AdjustLevels we provide are very similar to what you completed last week. They differ from what you had to construct last week in ways that make it easier for them to support the larger collection of image operations you will implement this week. To complete this program, you will:

- revise the ImageViewer class we provide in the starter folder,
- define a new class named DisplayDifference that extends ImageViewer, and
- complete the definition of a class named Histogram that will provide the information about an image that our DisplayHistograms class needs to draw pictures of an image’s color histograms.
**Extending ImageViewer**

The most obvious change you will make to the `ImageViewer` class is to add a lot of new buttons. In last week's lab, the class displayed just two buttons: “Load Image” and “Take Selfie”. This week you will add 5 more buttons. In addition, you will add private methods to the `ImageViewer` class to implement the “Zoom” and “Expand Range” buttons. The code for “Show Difference” and “Show Histogram” will be implemented primarily in two of the other classes that will be included in your program.

**Using a GridLayout**

To avoid having the new buttons stretch all the way across your computer’s screen, our `ImageViewer` class uses a new layout manager within the `JPanel` that holds the program’s buttons. The layout manager is called a `GridLayout`. As its name suggests, it divides the space within a panel into a two-dimensional grid of cells and places one component in each cell. Components are placed in cells from left to right and from top to bottom as they are added to the panel. If you look back at the screenshots included above, you can see that we used a 2 by 4 grid to hold the buttons in our `ImageViewer`. The “Load Image” button was added first while the “Expand Range” button was added last.

The `GridLayout` constructor expects two `int` parameters specifying the number of rows and columns in the grid. Therefore, the code for `ImageViewer` in our starter project includes the instruction

```java
controlPane.setLayout( new GridLayout( 2, 4 ) );
```

**Expanding an Image’s Brightness Range**

The code that handles the “Expand Range” button will have to determine the smallest and largest brightness values used for each of the color components of the image and then linearly scale the brightness values so that they extend from 0 to 255. To do this you will probably want to include at least four private methods in the class. One of these methods will determine the largest value in a two dimensional array of values. Another will determine the smallest value.

The other two methods will expand the range of the pixel values. The first, which we will call `expand`, should take an `SImage` as a parameter and return an `SImage` with all of its brightness values appropriately expanded. This method will depend on a second method (`expandPixels`) that takes an `SImage` and an `int` specifying which color component to process (`SImage.RED`, `SImage.GREEN`, or `SImage.BLUE`). This second method will return an `int` array containing the expanded brightness values for just the specified color component. We suggest using similar pairs of methods for all of the image transformations in this lab.

To help you understand the approach we have in mind, we have used it in the version of `AdjustLevels` provided in this lab’s starter folder. In Lab 7, we suggested that you define an `adjustPixels` method that takes three parameters: the number of brightness levels to be used, an `SImage`, and an `int` specifying a color layer. This method returned a 2-dimensional (2-D) array of `int`s describing the adjusted brightness values for one layer of the image. We then told you to invoke this method three times within your `sliderChanged` method in a construction of the form:

```java
new SImage( adjustPixels( sliderValue, getImage(), SImage.RED ),
           adjustPixels( sliderValue, getImage(), SImage.GREEN ),
           adjustPixels( sliderValue, getImage(), SImage.BLUE ) )
```

In the version of `AdjustLevels` in the starter folder, `requantize` plays a role similar to the `expand` method we described above. It produces a new version of an image given just two parameters: the number of brightness levels to be used and the `SImage` to be processed. The `requantize` method includes three invocations of `adjustPixels`, the method from last week that processes the pixel array for one color component. This approach makes our `sliderChanged` method a bit shorter since a single invocation of `requantize` replaces the complex construction shown above.
Honestly, the `sliderChanged` method you wrote last week was already concise. By the end of lab, however, the `buttonClicked` method in your `ImageViewer` class will handle seven separate buttons. If several lines of code are required to handle each button, `buttonClicked` will get large and difficult to read. It is in this method, therefore, that the technique we have illustrated in our `AdjustLevels` will pay off.

**Maintaining the Original Image**

Our `ImageViewer` class has two `SImage` instance variables named `original` and `updated`. The variable `original` should be associated with the `SImage` most recently placed in the `ImageViewer` by pressing the “Load Image” or “Take Selfie” button, while `updated` should refer to the `SImage` that is currently being displayed. These two images will frequently be different since the displayed image can be modified by clicking the “Expand Range” button, either of the “Zoom” buttons or by adjusting the slider at the bottom of an `AdjustLevels` window. These actions should change `updated` but not `original`.

When asked to display a histogram, expand the range of an image, or “Zoom”, your code should use the displayed image rather than the original. That is, the code for these operations should start with the value associated with the variable `updated`. When the slider at the bottom of the window is adjusted, a new image to be displayed should be computed starting with the original image. Therefore, the `getImage` method provided by our `ImageViewer` returns the image associated with `original`.

**The Histogram Class**

In the starter project for this lab, you will find a skeletal definition of a class named `Histogram`. You should complete the definition of this class. The constructor for the `Histogram` class will take two parameters. The first parameter should be an `SImage`. The second parameter should be an `int` specifying the layer of the `SImage` from which the histogram should be constructed. That is, the second parameter’s value will either be `SImage.RED`, `SImage.BLUE`, or `SImage.GREEN`. The constructor should create a 1-dimensional (1-D) array of 256 integer values. It should include a loop that will examine all the elements in the pixel array for the specified layer, setting each of the 256 elements of the histogram array so that the element at index `b` is equal to the number of entries in the pixel array that are equal to `b`.

The histogram class should provide one public method named `frequency`. This method should take a brightness value, `b`, and return the count stored at position `b` in the histogram array. If the parameter value is out of range (i.e., less than 0 or greater than 255), your `frequency` method should return `-1`.

Our `DisplayHistogram` class will use your `Histogram` class to display histograms for the three color layers of an `SImage`. The constructor for the `DisplayHistogram` class takes an `SImage` as a parameter. It creates three histograms, one for each color layer, and then creates a new window displaying graphs of the brightness distributions for these three color layers. Your `ImageViewer` class should create a new `DisplayHistogram` window when the “Show Histogram” button is pressed. The histograms displayed should be based on the image currently displayed in the `ImageViewer`.

**The DisplayDifference Class**

When the “Show Difference” button is clicked, your `ImageViewer` should create a new `DisplayDifference` object. The constructor for the `DisplayDifference` class should take two `SImages` as parameters. It should compute the differences between the brightness values of corresponding pixels of its parameter images and display an image using these difference values as brightness values.

The `DisplayDifference` class should be defined to extend `ImageViewer`. This provides a convenient way to display the difference image and to examine its histogram and/or expand its brightness range. Our `ImageViewer` class provides a public method named `setImage` that can be used to set the “original” image displayed within an `ImageViewer`. The `DisplayDifference` constructor should invoke `setImage`
age to display the image it has computed. Of course, several of the buttons included in ImageViewer (like “Load Image”) will be pointless in a window created to display a difference image.

We suggest that you use a pair of private methods to compute the difference image. The first method will take two SImage s and return an SImage based on their differences. The other method will take two SImage s and an int specifying a color layer and return a 2-D array of brightness values for one layer of the difference image. The first method will be defined by invoking the second method once for each color layer and then combining the results.

The difference between corresponding pixels of two images can range from -255 to 255. Though the SImage class adjusts for values outside of the range 0 to 255, we would like you to take the absolute value when computing the difference between corresponding pixels. Java provides a handy method named Math.abs that makes this easy. For example, if \( x \) and \( y \) are integer values, \( \text{Math.abs}(x - y) \) produces the absolute value of their differences.

Remember that to handle images of different sizes, you should determine the smaller of the widths of the two images being compared and the smaller of their heights. The dimensions of the pixel arrays your method produces should be the smaller of the widths and heights of the original images. Java again provides some help with this task. If \( x \) and \( y \) are integer values, the expression \( \text{Math.min}(x, y) \) produces the smaller (minimum) of the two and \( \text{Math.max}(x, y) \) returns the larger of the two values.

**Implementation Plan?**
This lab handout is missing one very familiar feature. There is no “Implementation Plan.”

This week, we want you to take a stab at making your own step-by-step plan for completing this program similar to the implementation plans we have presented in previous weeks. This is a dry run! We will collect your implementation plans at the beginning of the lab period, and we will provide you with copies of our implementation plan once you have turned in your own.

Your implementation plan should be about 1 or 2 TYPED pages. **Bring 2 copies** since we will collect one at the start of the lab. In preparing your plan, consider how you will test the correctness of the code you write in each step. That is, remember that one of your main goals in developing your plan is to ensure that as you implement the program there is a way to test the correctness of each addition you make before moving on to the next step.

Before writing your implementation plan, **study the java code for the ImageViewer, AdjustLevels, and Histogram classes we have provided.** Understanding the implementations of these classes will allow you to produce a more useful implementation plan and have a more productive experience in lab. In addition, you will be much better prepared to make good decisions about which code should go in which class. Finally, these classes provide good examples of clean, concise coding.

**Getting Started**
To start this lab, you should download a copy of the starter project described above.

- Launch Safari (you can use another browser, but these instructions are specific to Safari) and go to the “Labs” section of the CS 134 web site (http://www.cs.williams.edu/~cs134).
- Find the link that indicates it can be used to download the Lab8Starter program.
- Point at the link. Hold down the control key and depress the mouse button to make a menu appear.
- Find and select the “Download Linked File As ...” item in the menu.
- Using the dialog box that appears, navigate to your “Documents” folder and save the Lab8Starter.zip file in that folder.
• Return to the Finder, locate the Lab8Starter.zip file in your Documents folder, and double-click on it to create a Lab8Starter folder.
• Rename the folder using a name including “Lab8” and your name(s) (e.g., FloydLab8). Remember not to include any blanks in the new folder’s name.
• Launch BlueJ and use the “Open Project” item in the “File” menu to open your Lab 8 project.

Extra! Extra!
If you have extra time, there are a number of features you could add to this program to increase its usefulness (or entertainment value).

First, if you make an image that you like, it might be nice to add a “Save” button so that you can keep it for posterity. The SImage class makes this relatively easy because it includes a method named saveAs that takes a file name String as a parameter and saves the image to which it is applied in PNG (portable networks graphics) format. You can use the same JFileChooser you use to choose a file to load to let the user pick a file name after the Save button is clicked. Just use the showSaveDialog method in place of the showOpenDialog method.

A feature you can try to add if the “imaginary rapids” image surprised you is a mechanism to make pictures like that one yourself. The idea is fairly simple. You start with two images where you want to hide one of them (the secret image) in a copy of the other (the cover image). If you are lucky and they have the same width and height, then you make a copy of the cover image using only 16 levels of brightness. Then you take the brightness values of the secret image, divide them all by 16 and add them to the corresponding pixels of the cover image. If the images have different widths and heights, the simplest approach is probably to produce a result whose width is equal to the smaller of the widths of the cover and secret image and whose height is equal to the smaller of their heights. As far as a user interface goes, you could add a “Hide image” button that would be disabled unless your program was already displaying some image. When this button is clicked, you should first show the file dialog box to let the user choose an image file. Then, you would treat the image already being displayed as your cover image and the image file just chosen as the secret image. The result should then replace the image displayed.

There are many other image transformations you could implement. One simple transformation is to blur the difference between the pixels in an image to produce a soft-focus effect. To accomplish this, replace the value of each pixel with the average of the values of its four (north, south, east, west) neighbors. You will have to treat the pixels in the top and bottom rows and the leftmost and rightmost columns specially since they won’t have all the expected neighbors. You can either leave the values of such pixels unchanged, or replace them with the average of the neighbors they do have. You should add a “Blur” button to your interface if you add this feature. Repeating the transformation increases the effect, so you should make “Blur” use and replace the image displayed.

Another interesting option is to implement a more sophisticated approach to zooming out. Following the approach we described above, zooming out effectively replaces each 2x2 block of pixels in the original image with the pixel in the upper-left corner of the block of pixels. The new approach would make each pixel in the reduced size image equal to the average brightness of the pixels found in the corresponding 2x2 box of pixels from the original image. In general, this will do a better job of preserving the details of the original than the technique suggested above which completely ignores the colors of three quarters of the pixels in the image. (Similarly, there are more sophisticated approaches than copying that one could use when zooming in, but they are probably beyond the scope of this lab exercise.)

Another variation on the zooming transformation is to allow for magnification/reduction factors other than 2. That is, one might have a slider whose values might ranges from 2 to 10 and when you pressed the zoom in or zoom out button the image’s size would be increased or decreased by a factor equal to the setting of the slider.
There are many other possibilities (rotation of the image by an angle determined by a slider, for example). If you don’t like our suggestions, feel free to be creative.

**Clean Up**

Be sure to take a final look through your code, checking its correctness and style. Check over the style guide accessible through the course web page and make sure you have followed its guidelines. Make sure you have included your name(s) and lab section in a comment in each class definition.

You can find instructions describing how to submit your program on the “Labs” page of our web site at

http://www.cs.williams.edu/~cs134/Labs.html