In assignment A5, you saw how images could be manipulated by rearranging the two-dimensional array of pixels that make up the image. In this assignment, you learn how the pixels themselves can be changed to provide various effects —light or darken a picture, provide better contrast, change from a color image to a grayscale image, etc.

This assignment has more programming in it than the others. Further, it will require more thinking about the problem and how to go about it than the others. Please start on this assignment early. Again, finish one part completely before going on to the next! And read this handout carefully.

This introduction to image processing is necessarily brief and even simplistic. You can find 800-page books in image processing, and there is still lots of research on the topic. Much of image processing is a mixture of mathematics and understanding human perception. This assignment will make you more aware of what goes on in applications like Adobe Photoshop and may spark your interest in the field —perhaps you will take a course on graphics or image processing and will end up working the field. Cornell has just about the best academic graphics program in the world. This is one of the many interesting areas in which computer scientists are working at Cornell.

Much of the material in this handout comes from the text *Digital Image Processing*, by Gonzalez and Woods (Pearson Education, 2002). We also drew from several websites, including:

http://www.videoessentials.com/glossary.php contains a huge glossary of imaging terminology.

http://www.poynton.com/index.html. This web page of Charles Poynton answers many questions about "gamma" and color —click on the appropriate links. From the "gamma" link, you can find definitions of brightness, intensity, luminance, lightness, and other terms used in image processing.

Download the files for the following classes from the course website (assignment A6): `ImageJFrame`, `ImageMaintainer`, `ImageMap`, and `ImagePanel`. The website also has some images that you can use.

2. Conversion to grayscale.

In the grayscale —in a black-and-white picture— there is no color, just different shades of gray. As you will see (literally), we can simulate a grayscale in the RGB color system by making the red, green, and blue components the same —changing them to what is called the pixels *luminancy value*.

One obvious way to do this is to compute the average of the red, green, and blue components:

gray= (red+ green + blue)/3

and then to set each of the red, green, and blue components to this average. A number of other formulas have been proposed. The most "accurate" is the ITU (International Telecommunications Union) standard:

gray = (222*red + 707*green + 71*blue) / 1000

Here is what to do for conversion to grayscale

1. In class `ImageMaintainer`, implement two methods to change the image to grayscale: one using the averaging method and the other using the ITU standard formula. In class `ImageJFrame`, add two methods (like `hreflect`) to call the two methods you just implemented. You can use these to test your methods from the interactions pane.

2. Add buttons to the GUI for the two methods you just wrote, as follows. Do the following in class `ImageJFrame` (it will help to compile and test, if there is anything to test, after each step):

(1) Near the top of the class, under "// Buttons", add two buttons for these two conversions to grayscale.

(2) In method `setUp`, insert instructions to add the two buttons to `buttonBox`.

(3) Add **`this`** as an `actionListener` for the two buttons (mimic how this is done for button `buttonrestore`).

(4) In method `actionPerformed`, which is called when a button is clicked, for each of the two new buttons, add code to test for `e.getSource()` being the button and call the appropriate method if it is (mimic the code that is already there).

(5) At the top of class `ImageMaintainer`, indicate in a comment which of the two conversions to grayscale works better, in your opinion, and explain why. You can determine this by making creating two `JFrames` with the same picture, converting both to grayscale, and looking at the two side by side.

2. Power-law transformations.

In a CRT (Cathode Ray Tube, a kind of monitor), voltage is applied to a dot on a screen to get light there with a certain intensity. The higher the voltage, the higher the intensity. But it is not a linear scale —the intensity does not rise proportionally with the voltage. Instead, a "power-law" governs how the intensity depends on the voltage. Here is the basic formula:

`intensity = voltage`$^{\texttt{gamma}}$ (for some value of `gamma`).

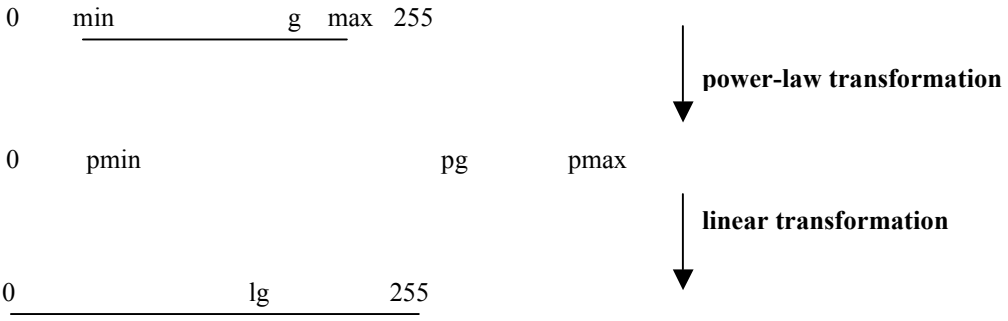
For the traditional CRT, `gamma` = 2.5; for television, Wikipedia says that for some TV's, it is 2.2, and for a Game Boy Advance, it is between 3 and 4.

We tell you about this power-law transformation because it is also useful for providing for contrast manipulation and lightening and darkening an image. Consider a grayscale pixel value `g`, and consider transforming the value to

`c * g`$^{\texttt{gamma}}$ (for some real numbers `c` and `gamma`)

If an image is washed out, meaning there is too much light, transform the pixels using the above formula with `c = 1` and `gamma < 1`—perhaps .95, or .90. On the other hand, if the image seems too dark and detail cannot be seen, transform all the pixels using the above formula with `c = 1` and `gamma > 1`.

Normalization: the grayscale component `g` is supposed to be in the range 0..255. However, performing the power-law transformation with `gamma > 1` may make g greater than 255. Below, let `min` (and `max`) be the minimum (and maximum) grayscale value in the image, and let `g` be some pixel value in the image. The power-law transformation changes `min`, `g`, and `max` into `pmin`, `pg`, and `pmax` (below). The larger pixel values get pushed farther out (get much larger) than the smaller ones, and the larger ones may get > 255.



We want to change the range `pmin..pmax` to the scale 0..255, this time with a linear transformation. This transformation is explained by the following formula, where `pmin` is transformed to 0, `pg` to `lg`, and `pmax` to 255.

```
(255 – 0) / (pmax – pmin) =  (lg – 0) / (pg – pmin)
```

Solving for `lg`, we have:

```
lg = 255 * (pg – pmin) / (pmax – pmin)
```

To do a power-law transformation on an RGB image, just do it on each of the red, green, and blue components separately. Because of the way we change an image from the RGB system to grayscale (see task 1 above), you can to a do power-law transformation on a grayscale image in the same way that you do it on an RGB image: by applying the transformation on all three components of each pixel.

Combining the power-law transformation with normalization.

To find a formula that combines the power-law transformation with the linear transformation, substitute for `pg`, `pmin`, and `pmax` in the formula for `lg`:

```
    lg = 255 * (pg – pmin) / (pmax – pmin)
=       <substitute>
    lg = 255 * (c*g^gamma – c*min^gamma) / (c*max^gamma – c*min^gamma)
=       <arithmetic>
    lg = 255 * (g^gamma – min^gamma) / (max^gamma – min^gamma)
```

Note that `c` disappears from the calculation.

Suppose `gamma > 1`. Then the power-law transformation spreads out the pixel values, with the larger values moving further away from the smaller ones. Normalization then pushes the values closer together, but in a linear fashion. Suppose also that the minimum pixel value is 0 (black) and the maximum pixel value is 255 (white). Then the formula for `lg` reduces to:

```
lg = 255 * (g^gamma / 255^gamma)
```

This means that, with `gamma > 1` and `g < 255`, `lg < g` —so the transformation *darkens* the pixel.

Here is what to do for a power-law transformation

Step 1. In class `ImageMaintainer`, implement a method that does a power-law transformation using values `c` and `gamma` that are in the GUI fields. They can be retrieved using these function calls:

```
getFrame().getFieldC()  and  getFrame().getFieldGamma()
```

Each of the red, green, and blue components should be transformed only if the corresponding checkbox in the GUI is checked (the default is for them to be checked). Whether a checkbox is checked can be found out using these function calls:

```
getFrame().redIsChecked()
getFrame().greenIsChecked()
getFrame().blueIsChecked ()
```

Note that before you can transform the red component (for example) of all pixels, you have to know what the minimum and maximum values of the red component are. This means that you must process the pixel array *twice*: the first time to find the min and max values for the red, green, and blue components and the second time to convert the red, green, and blue components.

Step 2. Add a button to the GUI that causes your method to be called. See the end of the grayscale conversion discussion for an explanation of how to do this.

Step 3. Now that the method has been tested and you believe it to be correct, experiment with it and write us a few paragraphs about your findings. Answer these questions and tell us anything else you find interesting –put your comments as a comment at the top of class `ImageMaintainer`.

(1) Make a few transformations with `gamma` less than 1. (Remember, you can always hit the restore button to get the original image back.) What is a good value for `gamma`, so that the picture doesn't get too light too quickly?

(2) Do the same thing as (1) but with `gamma > 1`.

(3) Try to make some transformations with `gamma = 1` and with different values of c. What happens?

(4) Perform a transformation on an original picture with c = 1 and gamma = .2. Now perform the reverse transformation on the same picture, with c = 1 and gamma = 1/.2. Do you get the same picture back? Explain what is happening in a comment at the top of class `ImageMaintainer`.

(5) Deselect two of the red-green-blue checkboxes in the GUI and perform a transformation like `c = 1` and `gamma = 1.5`. What happens? Try it on the original picture `with c = 1` and `gamma = 1/.5`. Try the other of the two colors and see how the image changes. Tell us about your findings in a comment at the top of class `ImageMaintainer`.

3. Enlarging an image.

Let us assume a grayscale image —to enlarge an RGB image, just perform the grayscale image stuff shown below to each of the three color components.

To enlarge an image, you have to insert rows and columns. Consider an image with `r` rows and `c` columns, and think of it as in an array `b[0..r–1][0..c–1]`. We will enlarge the image by inserting one row between each pair of rows and one column between each pair of columns, to arrive at an array: `c[0..2r–2][0..2c–2]`. In this situation, `b[i][j]` is placed in `c[2i][2j]`. We do it this way so that the first and last rows and columns of `c` get their values from image b. We show the first few rows of `c` below, with 1 denoting a value that comes from array `b` and 0 denoting a value that has not yet been calculated.

Row c[0] 1 0 1 0 1 0 …
Row c[1] 0 0 0 0 0 0 …
Row c[2] 1 0 1 0 1 0 …
Row c[3] 0 0 0 0 0 0 …
Row c[4] 1 0 1 0 1 0 …

Now we have to consider filling in the zeros with grayscale pixel values. We do this using a nearest-neighbor strategy. First, calculate the pixel values for elements of `c` that have four diagonal neighbors whose values come from `b` as the average of those four. Below, we have marked each of these elements with a boldfaced **`1`**:

Row c[0] 1 0 1 0 1 0 …
Row c[1] 0 **1** 0 **1** 0 **1** …
Row c[2] 1 0 1 0 1 0 …
Row c[3] 0 **1** 0 **1** 0 **1** …
Row c[4] 1 0 1 0 1 0 …

Now note that each of the 0's (which represents an element to be calculated) has four neighbors that are pixel values (except for those on the boundary, which have only 3!!!). Calculate the value for each of these elements as the average of these four (or three) neighbors.

What to do to enlarge an image

Step 1. This programming is a bit tricky. So we suggest first writing a method to enlarge a double array `b[0..r–1][0..c–1]` and testing it thoroughly. It may help to write a method to print out such an array in a form that you can see it well.

Step 2. Implement a method in class `ImageMaintainer` to enlarge an image, as just shown. Insert a method in class ImageJFrame so that you can easily test it —and test it.

To help you out, class `ImageMaintainer` contains a method `average(p1, p2, p3)` that produces a pixel that is the average of the three pixels `p1`, `p2`, and `p3`. You will also need a method (it can have the same name) to compute the average of four pixels.

Step 3. Add a button to the GUI so that enlarging can be done by clicking this button. Adding a button is explained at the end of the discussion in the grayscale section.

Your enlargement algorithm is perhaps the simplest and most obvious one. However, when you look at the results, you will not be impressed. The enlarged image looks quite fuzzy. More sophisticated algorithms are required to get a better enlargement. Some of these use more points to average. Others attempt to look for and make use of edges or paths of similar color.

4. Submitting your solution.

First, make sure that all the methods you wrote have a clear and precise Javadoc specification. Verify this by clicking the javadoc button in DrJava and looking at the generated method specifications to see whether you can understand what the method does (and what each parameter is for) from the specification.

Second, submit files `ImageJFrame.java` and `ImageMaintainer.java` on the CMS, before the due date, Tuesday, 29 November (time 23:59).