Final Project: Mars Images

The Assignment

This project asks you to work with with image data, and learn two simple techniques for enhancement of images defined by pixel shades. Additionally, you will learn how secret images can be encoded in an image, and explore the power of colorization of images.

You will read images data files of NASA images of the planet Mars, display the images as received from the Viking Explorer, and by manipulating this data generate enhanced images.

The Images

The data in the files **mg20s002**, **mg20s007**, etc. came from a NASA jukebox of planetary images. Each file starts with several lines of text (a label) that identifies the image - the location on Mars, the resolution (how large an area is represented by one pixel), what spacecraft took the image, and the information about the size of the image data (number of lines and the number of pixels per line). After the label is histogram data - specifying how many pixels there are of each shade (gray shade, just like the color shades, has values in the range from 0 to 255). The last part of the file contains the image data: each pixel is represented as one byte.

Your program that manipulates the images should use the given library class `MarsReader`. You will use the following functionality of the class `MarsReader`:

- The constructor for `MarsReader` looks for the original Mars image file, reads the file labels and stores them in the field `labels`.
- The constructor then initializes the field `BufferedInputStream bytestream` to deliver the bytes of the selected image.

To create new images and save then as *.png* files, use the given class `ImageBuilder`. It works as follows:

- The constructor also initializes the field `BufferedImage image` that is ready to receive the data needed to represent the resulting image. You need to supply the height and the width of the image.
- The method `setColorPixel (x, y, r, g, b)` sets the color of the given pixel in the textttimage to the specified RGB shade.

- The method `public void saveImage(String filename)` saves the image you have created in the *.png* format — it adds the *.png* to the filename you specify.

Image Processing

Each pixel shade is represented as one byte. You can read one byte of data from the `bytestream` using the method

```
int read()
```

The integer will be in the range from 0 to 255.

All images in this collection have the same size: 320 lines of 306 pixels in each line. You can set the color of each individual pixel in the `BufferedImage image` using the method

```
void setColorPixel(int x, int y, int r, int g, int b)
```

A 'black and white' image is represented by pixels of different shade of gray. By choosing `setColorPixel (x, y, s, s, s)` with values of `s` ranging from 0 to 255 we can represent 256 different shades of gray.

In pictures of low quality the range of the shades is often much smaller than 256. For example, in the images from Mars most of the shades are in the range between about 70 and 170, leaving more than half of the shades unused. Image enhancement methods take advantage of this deficiency.

Linear scaling.

The first method uses linear scaling to modify the shade of each pixel. It starts with computing the minimum and maximum of the existing shades. It then scales each shade so that the range of shades is expanded to 256 values. The scaling formula is:

```
newshade = (oldshade - min) * (255 / (max - min));
```

That means that in our example (the range between 70 and 170), `oldshade=70` would be represented as `newshade=0`, similarly, `oldshade=170` would be represented as `255`, and finally, `oldshade=100` would be represented as `(100 - 70) * (255 / (170 - 70)) = 30 * 2.55 = 76.5`, or `newshade=76`.

We do not want to do this computation for every pixel over and over again. Instead, we should save the computed values in a table indexed by the `oldshade` values with the `newshade` values in the table.

Implement the linear scaling image processing and observe the impact on the original image.

Histogram equalization.

The second method is called histogram equalization. Histogram equalization is simply a transformation of the original distribution of pixels such that the resulting histogram is more evenly distributed from black to white.

We start by computing the distribution of the pixel shades (a frequency array or a histogram H). Histogram is a simple count of the number of occurrences of each pixel shade (a frequency chart). (For example a histogram of rolling a die 100 times may tell us that we rolled 1 15 times, 2 18 times, 3 17 times, 4 12 times, 5 15 times and 6 13 times.)

We start by reading all pixel data and building the histogram. You have actually computed a histogram of the Hamlet play in an earlier assignment. Let us assume that h_i is the number of pixels of the shade i and that h is the count of all pixels in the image.

We compute the *scaling factor* s_i of each pixel initially at gray level i as:
$s_i = (1/h) * sum(h_0, h_1, h_2, ..., h_i)$

where h is the total number of pixels and hi is the number of pixels at gray level i (i.e. the histogram data).

Once we have the *scaling factor*s, we compute $newshade_i = 255 * s_i$

Of course, again we do not want to keep recomputing these values and store them in a lookup table instead.

Include in your program a visual display of the histogram you have computed to verify that our assumptions about the color distribution are correct.

Note: You will need to read the Mars data file twice - first just to compute the histogram and set up the mapping of old shades to new ones, the second time, reading the old shades and writing the new shades into the output file.

Steganography — Omit this part

Note: Java is very strict about not mixing data types and makes conversion between data types quite difficult. While this is good for making sure programs work correctly and programmers do not misuse the language, converting bits into characters is not a trivial task. We decided that the work needed to accomplish this task is too difficult without a support of appropriate library. We plan to provide such library in the future.

————-

One of the images provided to you contains a hidden message. Looking at the picture, it is hard to tell the difference between the pixel shade 78 and pixel shade 79, a malicious intruder encoded a secret message from Mars in the image.

Your job is to decipher this message. Our intelligence tells us that their

message is encoded in the last bit of every pixel. By collecting the values of last bits and converting them to characters using the ASCII encoding, you should be able to recover the hidden message.

To accomplish this task you need to learn a bit about ASCII encodings, about how to find the last bit of a number (*hint:* think of the difference between odd and even numbers), and how to combine eight bits into a byte. None of this is difficult, with just a bit of thinking.

Colorization

Explore what happens to your image when you add a bit of coloring to it. One way to do it is by replacing the gray shade color

```
new Color(shade, shade, shade);
```

by

```
new Color(shade, 255 - shade, 255 - shade);
```

Hidden images

Another image among those given contains a hidden images interposed upon the Mars image. All pixels that comprise the hidden image have their last bit equal to 0. Your job is to recover the hidden image from the original.

Note:

We do not provide such images. Instead, create such an image ourself, then run another program that will reveal the hidden image.

Color processing

The class `ImageReader` allows you to read any *.bmp* or *.png* file and analyze the individual pixels. The constructor expects the name of the file to be read. It reads the file and initializes the value of the `width` and `height` fields for the given image.

The method `Color getColorPixel(int x, int y)` returns the color value of the pixel at the given location. You can extract the red, blue, and green component of the color as integers using the methods

`c.getRed()`, `c.getGreen()` and `c.getBlue()`

Create a negative of the given *Flowers.png* image. Explore other ways of manipulating the images and document your exploration.

Sonification — Omit this part

Note:

Again, we hope to provide support for this type of processing in the future — currently we do not have the library available.

The software support for this will be available next week.

———————

Often the naked eye has a hard time recognizing subtle changes in the color that may represent the underlying structure of the surface. One place

where this is crucial is in analyzing medical scan images.

To help the observer recognize the structure of the image, one can use *sonification* or sound representation of the image colors.

Your program will respond to the *mouse* movement over the image by playing a tune that represents the color of the underlying shade.

Mosaic of images

One can combine several images together, for example overlay one image in the middle of another one, place several images in different locations in a larger image, etc.

Experiment with this and create something interesting.

You can also try to *shrink* an image by replacing every four adjacent pixels with one pixel that has the color that is the average of the four pixels. This reduces the size of your picture by half in both directions.

Summary

Your project should collect the various features of this image processing suite and allow the user to select one at a time. Your documentation should include a brief explanation of each of its part — feel free to quote or copy parts of this assignment for inclusion in your document.

References

The idea for this lab came from the book by Robert S. Wolff and Larry Yaeger, *Visualization of Natural Phenomena*, Springer Verlag 1993 (TELOS Series)

Thanks also to Peter Ford from MIT who helped us locate the original image data file.

In 1999, The Viking Orbiter and other planetary data files could be found at

```
ftp://pdsimage.wr.usgs.gov/cdroms/
```

We suggest using the files in vo_2002 that start with *mg*. For example:

```
ftp://pdsimage.wr.usgs.gov/cdroms/vo_2002/mg25sxxx/mg25s022.img
```

These files are relatively small, about 100K and contain images that are approximately 300 by 300 pixels. See

```
ftp://pdsimage.wr.usgs.gov/cdroms/vo_2002/volinfo.txt
```

for a description of the file format.