CS 101 – Problem Solving & Programming I
Fall 2012
Program 6 – PPM Image Editor
Algorithm due Oct 18
Program due Oct 21

Images are the heart of multimedia presentations, and there are many applications to manipulate and edit image files. For this assignment, we're going to do some basic image processing. We're not writing the next Instagram, but we'll see the basics of how manipulating digital images is done—and along the way, get some practice in file manipulation.

There are many image formats, of course. One of the simplest is the PPM format. PPM stands for *portable pixel map*. It is a 'lowest-common-denominator' format that stores the image data as a text file. Although this is not very compact, it is very easy to read and write, and is often used as an intermediate format by image editing software. Your program will read files in this format and will output a transformed image in the same format.

Unfortunately, many common image tools won't display PPM format files as an image. There are some options, though. Photoshop will read such files. So will GIMP, the GNU Image Manipulation Program. GIMP is an open-source image editing package available for download at www.gimp.org. GIMP is also installed on all Flarsheim labs. A more compact option is ImageJ, a Java program that provides support for a variety of formats. Best of all, ImageJ can be installed anywhere (provided Java is present), and is compact.

If you've already got software that will read PPM files, great. Use it. Otherwise, you can copy this zipfile and unzip the folder onto your Q: drive. The program will run on any computer that already has Java installed. The first time you run the program it will save a configuration file recording where it found Java. Also, if you run it on a campus computer (such as Remote Labs) you may get a popup that the campus firewall has blocked some features of the program. That's not a problem; it tries to phone home to check for updates when it's started, and we don't care if it's not able to do that. All we need for this project is to be able to read and display PPM files.

The details of the file format are very simple. The first thing in the file is a 'magic number' identifying what type of file this is. This will always be 'P3'. There is an optional one-line comment that begins with a # in the first column; this comment will *not* be present in the files we use. Then there is a line with two integers, the width of the image in pixels, and the height (that is, the number of pixels per row, then the number of rows). Then there's a number which is the highest value any one color element can have; this will be 255 for our images. This is followed by the pixel data. Pixel data is stored in row-major order; this means that the data begins with all the pixels in the first row, then the pixels in the second row, etc. Each pixel will be on a line by itself, and is represented by three integers in the range 0-255: the red, green, and blue values for that pixel. An example will make this clearer. Suppose a PPM file begins as folows:

```
P3
650 466
255
66 59 67
68 61 69
53 48 54
55 50 56
```

This PPM image (the P3 flag) is 650 pixels wide and 466 pixels high. Each pixel has a maximum intensity of 255 for each color element. The first 4 pixels in the first row have the pixel values shown.

Incidentally, why 255? Because 0-255 can be expressed in one byte. By using 8 bits for each color, we create 24-bit color images. This is the "millions of colors" option on Windows. 24-bit color is capable of displaying approximately 16 million distinct shades. While the eye can distinguish more colors than that, our display devices can't display more than that (except for some high-end graphics workstations).

Your program will ask the user for the name of a PPM file, which you can assume will be formatted correctly. You will ask the user whether to perform one of two transformations, and the name of the output file, and will output the transformed image in correct PPM format. The transforms you will support are:

- convert to grayscale (R, G, B values are all the same for each pixel, found by averaging the pixel values) or
- convert to photographic negative (each pixel value X is replaced by 255-X).

***Extra credit*:** There are two optional features you can add for extra credit. You can do either or both of these, independently (you do not have to do one in order to do the other):

- Flip image left-to-right (5 points); or
- Apply a blur filter (10 points)

Examples of each of these transformations is shown below.

Development notes:

- Develop one part of the program at a time.
- Start by making sure you can read, parse, and write the header portions of the file. Then add code to read the pixels. Write the filters one at a time, each in its own function.
- Remember that all input from files arrives as strings, one string per line.
- Remember that all output to files has to be as strings, even if it's numeric data. The str() function will be handy.
- Writing output to a file won't automatically go the next line unless you include the '\n' character with the output.
- The grayscale and negative filters each deal with one pixel at a time.
- Remember that even a moderate sized image can take up multiple megabytes in this format. Thus, the program may seem to be 'locked up' when it's actually just busy.

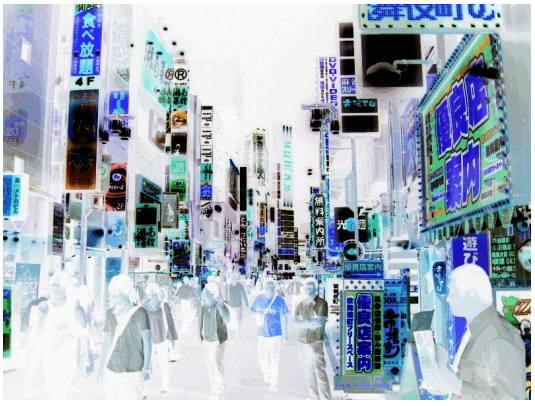
Samples of the various filters:

Original Image - Christchurch College, Oxford. Charles Dodgson was a lecturer there, and his niece Alice played in the courtyard. He wrote some stories for her, which he published as Lewis Carroll. Also, Christchurch was used in filming the Harry Potter movies, including the dining hall, which became the dining hall at Hogwarts. (None of that has anything to do with the assignment, I just thought you might be interested.)

Grayscale image:

Street scene, Shinjuku Ward, Tokyo:

Negative:



Flip left/right:

Blur: