CS 100 Project Five – Fall 2018

1. Overview

This goal of this project is to convert a color image into a grayscale (black and white) image. Specifically, the program will read in a color image in the PPM (Portable PixMap) format and use different algorithms to convert it into a grayscale image in the PGM (Portable GreyMap) format.

2. Background on PPM and PGM Image Files

An image can be viewed as a two-dimensional array with each element representing a pixel (picture element). A pixel in a color image contains color information such as red, green and blue intensities whereas a pixel in a grayscale image contains just grayscale intensity. Images can be stored in many formats. You probably have heard of the more common ones, such as **jpeg** (from the Joint Photographic Experts Group) and **gif** (Graphics Interchange Format). The vast majority of these formats store the image data in a binary file. In this project, we are interested in formats in which an image is stored in a text file (a file that can be edited by a text editor such as **vim**). Specifically, color images will be stored in **ppm** format, and grayscale images will be stored in **pgm** format.

A **ppm** image file consists of header information and then a long list of integers representing the **red**, **green** and **blue** intensities of each pixel in the image, as shown below.

```
P3
width-in-pixels     height-in-pixels
maximum-value
pixel-1-1-red pixel-1-1-green pixel-1-1-blue     pixel-1-2-red pixel-1-2-green pixel-1-2-blue ...
pixel-2-1-red pixel-2-1-green pixel-2-1-blue     pixel-2-2-red pixel-2-2-green pixel-2-2-blue ...
...
pixel N-1-red pixel-N-1-green pixel-N-1-blue     pixel-N-2-red pixel-N-2-green pixel-N-2-blue ....
```

The very first line of the file is the magic identifier that identifies the file type. For the **ppm** image file to be used in this project, it is always **P3**. After that, you have three integers (`width`, `height`, `maximum-value`) that can be on a single line or separate lines. Finally, you see the actual RGB intensities (three integers) for each pixel in the image. You see a list of `3*width*height` integers. `maximum-value` is the maximum value that can appear in the list. Usually it is 255 because eight (8) bits are used to store the red, green and blue intensities. Sometimes, `maximum-value` is the actual maximum value of the list. A very tiny **ppm** image file that is four (4) pixels wide and six (6) pixels high, with the top two rows being red, the middle two rows being green, and the bottom two rows being blue, is shown below.

```
P3
4  6  255
255 0 0   255 0 0   255 0 0   255 0 0
255 0 0   255 0 0   255 0 0   255 0 0
0 255 0   0 255 0   0 255 0   0 255 0
0 255 0   0 255 0   0 255 0   0 255 0
0 0 255   0 0 255   0 0 255   0 0 255
0 0 255   0 0 255   0 0 255   0 0 255
```

Your job is to convert a color image in **ppm** format into a grayscale image in **pgm** format. The **pgm** format is similar to the **ppm** format. As shown below, the magic identifier for the **ppm** image file to be used in this project is always **P2**. After the magic identifier, you also have three integers (`width`, `height`, `maximum-value`) that can be on a single line or separate lines. Finally, you see a total of `width*height` integers that represent the grayscale intensities of all the pixels in the image. For a grayscale image, each pixel has only one grayscale intensity that usually ranges from 0 through 255. A pixel with a value of 0 is black (absence of light) and a pixel with a value of 255 is white (all light). For this project, you can simply use the `maximum-value` of the **ppm** input file as the `maximum-value` of the **pgm** output file.

```
P2
width-in-pixels
height-in-pixels
maximum-value
pixel-1-1-intensity  pixel-1-2-intensity  pixel-1-3-intensity ...
pixel-2-1-intensity  pixel-2-2-intensity  pixel-2-3-intensity ...
...
pixel-N-1-intensity  pixel-N-2-intensity  pixel-N-3-intensity ...
```

An example of a **pgm** image file is shown below. This image is seven (7) columns wide and six (6) rows high. It has 255 as the maximum grayscale intensity that can appear in the image pixel data. This image has a black (intensity=0) border of one pixel wide and a white (intensity=255) interior.

```
P2
7  6  255
0  0   0   0   0   0  0
0 255 255 255 255 255 0
0 255 255 255 255 255 0
0 255 255 255 255 255 0
0 255 255 255 255 255 0
0  0   0   0   0   0  0
```

3. Project Details

As mentioned previously, this project is to convert a color image in **ppm** format into a grayscale image in **pgm** format. You are asked to use one of the following six methods to convert a color image into a grayscale image.

1. ***red*** (use the red intensity as the grayscale intensity)
2. ***green*** (use the green intensity as the grayscale intensity)
3. ***blue*** (use the blue intensity as the grayscale intensity)
4. ***average*** (use the average of red, green and blue intensities as the grayscale intensity)
5. ***ligntness*** (compute the lightness from red, green and blue intensities, and use the lightness as the grayscale intensity)
6. ***luminosity*** (compute the luminosity from red, green and blue intensities, and use the luminosity as the grayscale intensity)

Please see https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale for details on the average, lightness, and luminosity methods.

This project consists of three files, `main.c`, `image.h` and `image.c`, for its source code. `main.c` and `image.h` can be downloaded from Blackboard. In `image.h`, two structures are defined to represent a color image in **ppm** format, as shown below.

```
typedef struct _pixel {
    int red;
    int green;
    int blue;
} Pixel;

typedef struct _imagePPM {
    char magic[3];  // magic identifier, "P3" for PPM
    int width;      // number of columns
    int height;     // number of rows
    int max_value;  // maximum intensity of RGB components
    Pixel **pixels; // the actual color pixel data
} ImagePPM;
```

One structure is defined to represent a grayscale image in **pgm** format.

```
 typedef struct _imagePGM {
    char magic[3];  // magic identifier, "P2" for PGM
    int width;      // number of columns
    int height;     // number of rows
    int max_value;  // maximum grayscale intensity
    int **pixels;   // the actual grayscale pixel data
} ImagePGM;
```

The `image.h` file also lists the signatures of the ten (10) functions you need to implement in the `image.c` file, as shown below.

- `ImagePPM *readPPM(char *filename);` Given a filename of a **ppm** image, read in the image and store it in the `ImagePPM` structure. Return the address of the `ImagePPM` structure if the file can be opened or NULL otherwise. You can assume the file will be in the **ppm** format if it can be opened.
- `int writePGM(ImagePGM *pImagePPM, char *filename);` Write out a **pgm** image stored in a `ImagePGM` structure into the specified file. Return 1 if writing is successful or 0 otherwise.
- `ImagePGM *extractRed(ImagePPM *pImagePPM);` Convert a **ppm** image into a **pgm** image using grayscale = R.
- `ImagePGM *extractGreen(ImagePPM *pImagePPM);` Convert a **ppm** image into a **pgm** image using grayscale = G.
- `ImagePGM *extractBlue(ImagePPM *pImagePPM);` Convert a **ppm** image into a **pgm** image using grayscale = B.

- `ImagePGM *computeAverage(ImagePPM *pImagePPM);` Convert a **ppm** image into a **pgm** image using grayscale = (R + G + B) / 3.
- `ImagePGM *computeLightness(ImagePPM *pImagePPM);` Convert a **ppm** image into a **pgm** image using grayscale = (max(R, G, B) + min(R, G, B)) / 2.
- `ImagePGM *computeLuminosity(ImagePPM *pImagePPM);` Convert a **ppm** image into a **pgm** image using grayscale = 0.21 R + 0.72 G + 0.07 B.
- `void freeSpacePPM(ImagePPM *pImagePPM);` Free the space used by a **ppm** image.
- `void freeSpacePGM(ImagePGM *pImagePGM);` Free the space used by a **pgm** image.

Note that all the intensities (red, green, blue, and grayscale) are integers. However, the real number computation should be done in the `computeAverage`, `computeLightness`, and `computeLuminosity` functions. **When the computation results in a real number, you shall use the `round` function in `math.h` to convert a real number to an integer.**

The main function in main.c will use the 10 functions listed in `image.h`, and you need to implement the 10 functions in the `image.c` file. When implementing the `readPPM` function, you need to allocate the space for a **ppm** image, and populate the space using the data read from the corresponding **ppm** file. In order to help you understand how these data structures work, we walk through the steps in reading a **ppm** image file into the `ImagePPM` structure.

First, to allocate space for the image, you need to allocate a new **ImagePPM** object.

```
ImagePPM *pImagePPM=malloc(sizeof(ImagePPM));
```

Second, once you have that object, you read the first few items (magic identifier, width, height, maximum value) from the file and store them in the corresponding fields of `pImagePPM`.

Third, since you know the height (the number of rows) and the width (the number of columns), you can allocate space for the actual two-dimensional array of pixels. Assume you have ten (10) rows and four (4) pixels on each row. Allocate ten (10) pointers that point to the first Pixel on each row.

```
pImagePPM->pixels = malloc(sizeof(Pixel *) * 10);
```

Then allocate space for the four (4) Pixels that exist on each row. Since you are doing this for each row, the statement below is inside a loop that executes ten (10) times.

```
pImagePPM->pixels[i] = malloc(sizeof(Pixel) * 4);
```

After you have allocated the necessary space, you can read pixel values into the structures. For each pixel in your image, you are reading three values (red/green/blue). The following shows how to read the red value.

```
fscanf(fp, "%d", &pImagePPM->pixels[i][j].red);
```

The `freeSpacePPM` function is to free all the space you have allocated in the `readPPM` function.

Similarly, in each of six converting functions, you need to allocate the space for a **pgm** image and the `freeSpacePGM` function is to free all the space you have allocated in one of the six converting functions.

4. Compiling and Debugging

This project requires three files, `main.c`, `image.h` and `image.c`. You can download `main.c` and `image.h` from Blackboard. You shall not modify anything in `main.c` and `image.h`. You implement the ten (10) functions in the `image.c` file you have created. Please make sure to include `image.h` in your `image.c` by inserting the following line.

```
#include "image.h"
```

To compile this project, use the following command.

```
gcc -Wall -std=c99 main.c image.c
```

To test the program, type in a command in the following format

```
./a.out method ppm_filename pgm_filename
```

Valid methods include `red`, `green`, `blue`, `average`, `lightness`, `luminosity`.

When debugging, it is probably a good idea to use a text editor such as **vim** to create a very small image as an input file, then examine the output file by hand to see whether the output file is correct. Once your program is working properly with small files, test your program with large files.

To test your program with large files, we recommend you download the free program **GIMP** (GNU Image Manipulation Program) from http://www.gimp.org/downloads. You can use **GIMP** to convert images into **ppm** format. Simply load an existing image that you have into GIMP, then select the **File→Export As** option and use **ppm** as the file extension of the exported file. Make sure to click **ASCII** when asked how to export. Please note when **GIMP** converts an image into a **ppm** file, it puts a comment line (a line starting with #) after the magic identifier. You need to delete that comment line, as you are not asked to skip comments in this project. Once your program has converted a **ppm** image into a **pgm** image, you can use **GIMP** again to view the **pgm** image.

5. Checklist for Completing this Project

- Create a directory named **`project5`** on your machine. Download **`main.c`** and **`image.h`** to that directory, and create a file named **`image.c`** under that directory.
- In **`image.c`**, implement the required functions, make sure there is a header block of comments that includes your name and a brief overview of your task.
- When you are ready to submit your project, first **DELETE ALL THE IMAGE FILES** under the **`project5`** directory, then compress your **`project5`** directory into a single (compressed) zip file, **`project5.zip`**. Make sure **`project5.zip`** contains the `project5` directory as well as `image.c` under it. (`main.c` and `image.h` are not required.)

- Once you have a compressed zip file named **`project5.zip`**, submit that file to Blackboard.

Project Five is due at 5:00pm on Friday, November 9. Late projects are not accepted.

**A project shall be completed individually, with no sharing of code or solutions.
All submissions will go through MOSS (Measure Of Software Similarity)
for similarity check.
The University of Alabama's Code of Academic Conduct will be rigorously enforced.**