EECS 10: Assignment 7

November 16, 2007

Due on Monday 12/03/2007 12:00pm. Note: this is a two-week assignment.

1 Digital Image Processing [80 points + 20 bonus points]

In this assignment you will learn some basic digital image processing (DIP) techniques by developing an image manipulation program called *PhotoLab*. Using the *PhotoLab*, the user can load an image from a file, apply a set of DIP operations to the image, and save the processed image in a file.

1.1 Introduction

A digital image is essentially a two-dimensional matrix, which can be represented in C by a two-dimensional array of pixels. A pixel is the smallest unit of an image. The color of each pixel is composed of three primary colors, red, green, and blue; each color is represented by an intensity value between 0 and 255. In this assignment, you will work on images with a fixed size, 800×450, and type, Portable Pixel Map (PPM).

The structure of a PPM file consists of two parts, a header and image data. In the header, the first line gives the type of the image, P6; the next line shows the width and height of the image; the last line is the maximum intensity value. After the header is the image data, arranged as RGBRGBRGB..., pixel by pixel.

Here is an example of a PPM image file:

```
P6
800 450
255
RGBRGBRGB...
```

1.2 Initial Setup

Before start working on the assignment, do the following:

```
cd ~
mkdir -p hw7
cd hw7
cp ~eecs10/hw7/PhotoLab.c .
cp ~eecs10/hw7/imgColor.ppm .
cp ~eecs10/hw7/text.ppm .
```

NOTE: Please execute the above setup commands only **ONCE** before you start working on the assignment! Do not execute them after you start the implementation, otherwise your code will be overwritten!

The file, *PhotoLab.c*, is the template file where you get started. It provides the functions for image file reading and saving, as well as the DIP function prototypes and some variables (do not change those function prototypes or variable definitions). You are free to add more variables to the program.

imgColor.ppm and *text.ppm* are the PPM images used to test the DIP operations. Once a DIP operation is done, you can save the modified image. You will be prompted for a name of the image. The saved image *image.ppm* will be automatically converted to a JPEG image and sent to the folder, *public_html* in your home directory. You are then able to see the image in a web browser at: *http://east.eecs.uci.edu/∼userid*

Note that whatever you put in the *public_html* directory will be publicly accessable; make sure you don't put files there that you don't want to share, i.e. do not put your source code into that directory.

1.3 Program Specification

In this assignment, your program should be able to read and save image files. To let you concentrate on DIP operations, the functions for file reading and saving are provided. These functions are able to catch many file reading and saving errors, and show corresponding error messages.

Your program is a menu driven program (like the previous assignment). The user should be able to select DIP operations from a menu similar to the one shown below:

```
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image quantization
 9:  Add borders to an image
10:  Image Overlay
11:  Exit
please make your choice:
```

1.3.1 Load a PPM Image

This option prompts the user for the name of an image file. You don't have to implement a file reading function; just use the provided one, *ReadImage*. Once option 1 is selected, the following is shown:

```
Please input the file name to load: imgColor.ppm
```

After a name, for example *imgColor.ppm*, is entered, the *PhotoLab* will load the file. If it is read correctly, the following is shown:

```
imgColor.ppm was read successfully!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image quantization
 9:  Add borders to an image
10:  Image Overlay
11:  Exit
please make your choice:
```

In this case, you can select other options. If there is a reading error, for example the file name is entered incorrectly or the file does not exist, the following message is shown:

```
Cannot open file "imgColor.ppm" for reading!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image quantization
 9:  Add borders to an image
10:  Image Overlay
11:  Exit
please make your choice:
```

In this case, try option 1 again.

1.3.2 Save a PPM Image in JPEG format

This option prompts the user for the name of the target image file. You don't have to implement a file saving function; just use the provided one, *SaveImage*. Once option 2 is selected, the following is shown:

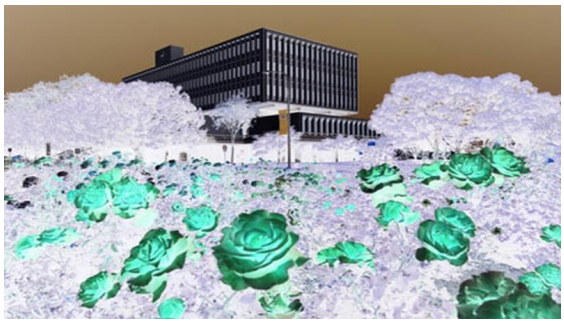
```
please make your choice: 2
Please input the file name to save: negative
Save the processed image? (y or n): y
negative was saved successfully.
negative.jpg was stored for viewing.
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image Quantization
 9:  Add Borders to an image
10:  Image Overlay picture
11:  Exit
please make your choice:
```

The saved image will be automatically converted to a JPEG image and sent to the folder, *public_html*. You are able to see the image at: *http://east.eecs.uci.edu/~userid*

1.3.3 Make a negative of an image

A negative image is an image in which all the intensity values have been inverted. To achieve this, each intensity value at a pixel is subtracted from the maximum value, 255, and the result is assigned to the pixel as a new intensity. You need to define and implement a function to do the job. Figure 1 shows an example of this operation.
Your program's output for this option should be like:

(a) Original image



(b) Negative image

Figure 1: An image and its negative counterpart.

```
please make your choice: 3
"Negative" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image Quantization
 9:  Add Borders to an image
10:  Image Overlay picture
11:  Exit
please make your choice:
```

1.3.4 Flip Image Horizontally

To flip an image horizontally, the intensity values in horizontal direction should be reversed. The following shows an example.

```
                          1 2 3 4 5                                5 4 3 2 1
  before horizontal flip: 0 1 2 3 4     after horizontal flip:    4 3 2 1 0
                          3 4 5 6 7                                7 6 5 4 3
```

You need to define and implement a function to do the job. Figure 2 shows an example of this operation. Your program's output for this option should be like:

```
please make your choice: 4
"HFlip" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
```

(a) Original image

(b) Horizontally flipped image

Figure 2: An image and its horizontally flipped counterpart.

(a) Original image

(b) Vertically flipped image

Figure 3: An image and its vertically flipped counterpart.

```
 7:  Add noise to an image
 8:  Image Quantization
 9:  Add Borders to an image
10:  Image Overlay picture
11:  Exit
please make your choice:
```

1.3.5 Flip Image Vertically

To flip an image vertically, the intensity values in vertical direction should be reversed. The following shows an example:

```
                        1 0 5                          5 4 9
                        2 1 6                          4 3 8
  before vertical flip: 3 2 7    after vertical flip:  3 2 7
                        4 3 8                          2 1 6
                        5 4 9                          1 0 5
```

You need to define and implement a function to do the job. Figure 3 shows an example of this operation.

```
please make your choice: 5
"VFlip" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image Quantization
 9:  Add Borders to an image
10:  Image Overlay picture
11:  Exit
please make your choice:
```

1.3.6 Edge Detection

The aim of edge detection is to determine the edge of shapes in a picture and to be able to draw a result image where edges are in white on black background. The idea is very simple; we go through the image pixel by pixel and compare the color of each pixel to its right neighbor, and to its bottom neighbor. If one of these comparison is greater than a threshold K the pixel studied is part of an edge and should be turned to white (RGB code = 255, 255, 255), otherwise it is turned to black (RGB code = 0, 0, 0). Figure 4 shows an example of this operation when k is set to 50.

To compare two colors, we can quantify the "difference" between two colors by computing the geometric distance between the vectors representing those two colors. Lets consider two color pixels C1 = (R1,G1,B1) and C2 = (R2,B2,G2), the difference between the two color pixels is given by the formula:

$$Difference(C1, C2) = \sqrt{(R1 - R2)^2 + (B1 - B2)^2 + (G1 - G2)^2}$$

So here is the algorithm in symbolic language.

For every pixel (i , j) on the source image

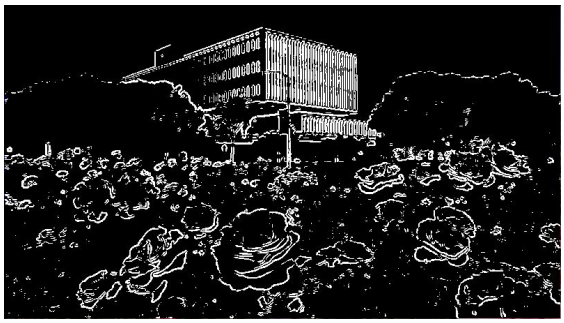
- *Extract the (R,G,B) components of this pixel C, its right neighbor C1(R1,G1,B1), and its bottom neighbor C2(R2,G2,B2)*
- *Compute D(C,C1) and D(C,C2)*
- *If D(C,C1) OR D(C,C2) superior to a given threshold K, then we have an edge pixel and turn this pixel to white, otherwise turn this pixel to black.*

Note: no need to perform difference calculation for the pixels at the bottom row since there are no pixels below them.

Once user chooses this option, your program's output should be like:

```
please make your choice: 6
Enter the difference threshold: 50
"Edge Detection" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
```

(a) Original image



(b) Edge detection with k = 50

Figure 4: An image and its edge detection counterpart.

```
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image Quantization
 9:  Add Borders to an image
10:  Image Overlay picture
11:  Exit
please make your choice:
```

1.3.7 Add Noise to Image

In this operation, you add noise to an image. The noise added is a special kind, called salt-and-pepper noise, which means the noise is either black or white. You need to define and implement a function to do the job. If the percentage of noise is n, then the number of noise added to the image is given by *n*WIDTH*HEIGHT/100*, where *WIDTH* and *HEIGHT* are the image size. You need the knowledge of random number generator from the previous assignments. Figure 5 shows an example of this operation with n set to 20%.

Once user chooses this option, your program's output should be like:

```
please make your choice: 7
Enter the percentage of noise: 20
"AddNoise" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image Quantization
 9:  Add Borders to an image
10:  Image Overlay picture
11:  Exit
please make your choice:
```

(a) Original image

(b) Noise setting: n=20

Figure 5: An image and its noise (salt-and-pepper) corrupted counterpart.

Implementation hint: calculate first the number of noise pixels depending on the noise percentage; then, use the random number generator to determine the position (x and y coordinate) of each noise pixel. Create a white or a black pixel by setting the intensity value at each color channel to its maximum (255) or minimum (0) respectively. Note: the number of white pixels should be approximately equal to the number of black pixels.

1.3.8 Image Quantization

Quantization is a way of reducing the dynamic range of an image. It means that the image information is slotted into specified quanta(s) or categories. In this assignment, we will practice quantizing an image to four categories (64-color since $4(red) \times 4(green) \times 4(blue)$).

To quantize an image to four categories, the following steps need to be performed:

For the R, G, B channels of each pixel, if the value of this pixel is between 0 and 63, set the value to 0, if the value is between 64 and 127 then set it to 64, if the value is between 128 and 191, set it to 128, finally if the value between 192 and 255, set it to 192.

When you see the effect of this transformation in the image, it should appear blocky since now the number of colors values have been reduced to 4. Figure 6 shows an example of image quantization.

Once user chooses this option, your program's output should be like:

```
please make your choice: 8
"Quantize" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image Quantization
 9:  Add Borders to an image
10:  Image Overlay picture
```

(a) Original image

(b) Quantized image

Figure 6: An image and its quantized counterpart.

```
11:  Exit
please make your choice:
```

1.3.9 Add borders to an image

This operation will add borders to the current image. The border color and width (in pixels) of the borders are parameters given by users. Figure 7 shows an example of adding borders to an image.

Once user chooses this option, your program's output should be like:

```
please make your choice: 9
Enter the R value of the border color(0 to 255): 255
Enter the G value of the border color(0 to 255): 255
Enter the B value of the border color(0 to 255): 0
Enter the width of the border: 10
"Add Border" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image Quantization
 9:  Add Borders to an image
10:  Image Overlay picture
11:  Exit
please make your choice:
```

1.3.10 Image Overlay (bonus points: 20pt)

This function overlies the current image with a second image. In our program, we will use an image with the text "EECS10" at the top right corner on a white background as the second image.

For your convenience, we have already prepared the overlay image, text.ppm, which is of the same size as the original photo, 800 by 450 pixels. In other words, both of these images have the same number of pixels,

(a) Original image

(b) Image with borders, border color = yellow, width = 10 pixels

Figure 7: An image and its counterpart when borders are added.

and each pixel in the overlay picture has its corresponding pixel in the original image (the one with the same "width" and "height" values).

To achieve the overlay effect, we will treat the background in the second image as transparent color. That is, each of the non-background pixels in text.ppm will be overlayed to its corresponding pixel in the original image, whereas background pixels will stay as in the original image. Whether or not a pixel in text.ppm is a background pixel can be decided by the RGB values of this pixel. More specifically, if the RGB values of a pixel are 255/255/255, which represents white color, then this pixel is a background pixel.

When the non-background pixel (p2) in the second image is overlayed to the correspondent pixel (p1) in current image, users may specify how transparent they want p1 to be. For example, if the transparency percentage is set to 10%, then the new pixel (p3) in the target image should be calculated using the following formula:

$$p3 = p1 \times 10\% + p2$$

The function first needs to load the second image file (text.ppm) and read in the transparency percentage, then overlay the pixels in the original image with their corresponding non-background pixels in the second image.

Once user chooses this option, your program's output should be like:

```
please make your choice: 10
Please input the file name for the second image: text.ppm
Enter the transparency percentage: 5
text.ppm was read successfully!
"Overlay" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in JPEG format
 3:  Make a negative of an image
 4:  Flip an image horizontally
 5:  Flip an image vertically
 6:  Detect image edges
 7:  Add noise to an image
 8:  Image Quantization
```

(a) An image

(b) Overlay of those two image

Figure 8: A image and the overlayed image.

```
 9:  Add Borders to an image
10:  Image Overlay picture
11:  Exit
please make your choice:
```

For example, if the transparency percentage is set to 5%, after the successful overlay operation, the overlayed image should like the figure shown in Figure 8:

1.4 Implementation

1.4.1 Function Prototypes

For this assignment, you need to define the following functions (those function prototypes are already provided in PhotoLab.c and please do not change them):

```
/* print a menu */
void PrintMenu();

/* read image from a file */
int  ReadImage(char fname[SLEN], unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]));

/* save a processed image */
void SaveImage(char fname[SLEN], unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);

/* reverse image color */
void Negative(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);

/* flip image horizontally */
void HFlip(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);

/* flip image vertically */
void VFlip(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);
```

```
/* detect the edges of the objects in the image */
/* threshold is the parameter to specify the difference_threshold k */
void EdgeDetect(double threshold, unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);

/* add salt-and-pepper noise to image */
/* percentage is the parameter to specify the percentage of noise added to the image */
void AddNoise( int percentage, unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);

/* quantize the picture */
void Quantize(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);

/* Add a border to the image */
/* borderR, borderG, borderB are the parameters to specify the border color */
/* Width is the border width (in pixels) */
void AddBorder( int borderR, int borderG, int borderB, int width, unsigned char R[WIDTH][HEIGHT],
unsigned char G[WIDTH][HEIGHT], unsigned char B[WIDTH][HEIGHT]);

/* Load the second image and overlay it on the current image, bonus points (20 pt)*/
/* fname is the filename of the second image */
/* transparency is the parameter for transparency percentage*/
void Overlay(char fname[SLEN], int transparency, unsigned char R[WIDTH][HEIGHT],
unsigned char G[WIDTH][HEIGHT], unsigned char B[WIDTH][HEIGHT]);
```

You may want to define other functions as needed.

1.4.2 Global constants

You also need the following global constants (they are also declared in PhotoLab.c and please don't change their names):

```
#define WIDTH 800   /* image width */
#define HEIGHT 450  /* image height */
#define SLEN 80     /* maximum file name length */
```

1.4.3 Pass in arrays by reference

In the main function, three two dimentional arrays are defined. They are used to save the RGB information for the current image:

```
int main()
{
unsigned char R[WIDTH][HEIGHT];  /* for image data */
unsigned char G[WIDTH][HEIGHT];
unsigned char B[WIDTH][HEIGHT];
}
```

When any of the DIP operations is called in the main function, those three arrays: R[WIDTH][HEIGHT], G[WIDTH][HEIGHT], B[WIDTH][HEIGHT] are the parameters passed into the DIP functions. Since arrays are passed by reference, any changes to R[ ][ ], G[ ][ ], B[ ][ ] in the DIP functions will be applied to those variables in the main function. In this way the current image can be udpated in DIP functions without

defining global variables.

In your DIP function implementation, there are two ways to save the target image information in R[ ][ ], G[ ][ ], B[ ][ ]. Both options work and you should decide which option is better based on the DIP manipulation function.

Option 1: using local variables You can define local variables to save the target image information. For example:

```
void DIP_function_name()
{
unsigned char RT[WIDTH][HEIGHT];  /* for target image data */
unsigned char GT[WIDTH][HEIGHT];
unsigned char BT[WIDTH][HEIGHT];
}
```

And at the end of each DIP function implementation, you should copy the data in RT[ ][ ], GT[ ][ ], BT[ ][ ] over to R[ ][ ], G[ ][ ], B[ ][ ].

Option 2: in place manipulation You do not have to create new local array variables to save the target image information. Instead, you can just manipulate on R[ ][ ], G[ ][ ], B[ ][ ] directly. For example, in the implementation of Negative() function, you can assign the result of 255 minus each pixel value and back to this pixel entry as descibed below:

$$R[x][y] = 255 - R[x][y];$$

1.4.4 Compile your programs

If you use any functions from "math.h" (such as sqrt function), you will need to include an extra option "-lm" when compiling programs. This option tells the linker where to find the libraries required to use the math functions.

```
gcc PhotoLab.c -o PhotoLab -lm -ansi -Wall
```

2 Script File

To demonstrate that your program works correctly, perform the following steps and submit them as your script file:

1. Start the script by typing the command: *script*
2. Compile and run your program
3. Perform the following operations
   - Load *imgColor.ppm*
   - Make a negative of the current image and save it as *negative*
   - Reload the original picture *imgColor.ppm*
   - Flip the current image horizontally and save it as *hflip*
   - Flip the current image vertically and save it as *vflip*
   - Reload the original picture *imgColor.ppm*
   - Detect the edges of the current image using $difference\_threshold = 60$ and save it as *edges*
   - Reload the original picture *imgColor.ppm*

- Generate a noisy image with setting, n=10, and save it as *noise*
- Reload the original picture *imgColor.ppm*
- Quantize the image and save it as *quantize*
- Add borders to the current the image with blue color (RGB code = 0, 0, 255),width = 10 pixels and save it as *borders*
- Reload the original picture *imgColor.ppm* (only if you do the bonus points)
- Overlay the current image with with transparency percentage 2% and save it as *overlay* (only if you do the bonus points)

4. Exit the program
5. Stop the script by typing the command: *exit*
6. Change the script file to *PhotoLab.script*

NOTE: make sure use exactly the same names as shown in the above steps when saving modified images! The script file is important, and will be checked in grading; you must follow the above steps to create the script file.

3 Submission

Use the standard submission procedure to submit the following files:

- PhotoLab.c (with your code filled in!)
- PhotoLab.script

Please leave the images generated by your program in your *public_html* directory. Don't delete them as they will be checked when grading! You don't have to submit any images.