# EECS 10: Assignment 7

Prepared by: Wenrui Lin, Prof. Rainer Dömer

November 14, 2019

Due Wednesday December 4, 2019 at noon, 12:00pm

## 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.

(a) Demo image "Iceland.ppm" (by R. Dömer).

Figure 1: Landscape photo provided for this assignment.

### 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, i.e. red, green, and blue; each color is represented by an intensity value between 0 and 255. In this assignment, you will work on image files with a fixed size, $480 \times 360$ pixels, and fixed format, 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 specifies

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 follows the image data, arranged as RGBRGBRGB..., pixel by pixel in binary representation.

Here is an example of a PPM image file:

```
P6
480 360
255
RGBRGBRGB...
```

## 1.2 Initial Setup

Before you start working on the assignment, do the following:

```
mkdir hw7
cd hw7
cp ~eecs10/hw7/PhotoLab.c ./
cp ~eecs10/hw7/Iceland.ppm ./
cp ~eecs10/hw7/TestImage.ppm ./
mkdir ~/public_html/
chmod 755 ~/public_html/
cp ~eecs10/hw7/index.html ~/public_html/
```

**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, test automation, as well as the DIP function prototypes and some variables. Please do not change the provided function prototypes or variable definitions. You are free to add more variables and functions to the program at the places indicated by provided comments.

The file `Iceland.ppm` is the PPM image that we will use 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 *name.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 view your images in a web browser at: `http://newport.eecs.uci.edu/~yourUCInetID/index.html`, where `yourUCInetID` is your UCI network ID. For a specific image, you may either click on the icon, or access it directly at this URL: `http://newport.eecs.uci.edu/~yourUCInetID/name.jpg`, where `name` is the basename of the image file.

**NOTE**: Each file you put into your `public_html` directory will be publicly accessible! Make sure you do not 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 load and save image files. To let you concentrate on DIP operations, the functions for file loading and saving are provided. These functions are implemented to catch many file reading and writing errors, and show corresponding error messages if problems occur.

Your program should be a menu driven program. The user should be able to select DIP operations from a menu as the one shown below:

```
-------------------------------------------
 1: Load a PPM image
 2: Save the image in PPM and JPEG format
 3: Change the color image to black and white
 4: Make a negative of the image
 5: Flip the image horizontally
 6: Mirror the image horizontally
 7: Zoom into the image
 8: Sharpen the image
 9: Exchange the red and green color channels
10: Add noise to an image
11: Add overlay to an image
12: Add border to an image
13: Test all functions
14: Exit
-------------------------------------------
Please make your choice:
```

Note: options '11: Overlay' and '12: Border' are bonus questions (worth 10 extra points each). If you decide to skip these two options, you still need to implement the options 13 and 14.

### 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, namely `LoadImage`. When option 1 is selected and the user enters our demo image name "Iceland", the following should be shown:

```
Please input the file name to load: Iceland
Iceland.ppm was read successfully!
```

Specifically, after the file basename is entered, the program will load the file `Iceland.ppm`. Note that in this assignment we always enter file names without their extension when you load or save a file (i.e. enter `Iceland`, *not* `Iceland.ppm`).
If there is a file reading error, for example the file name is entered incorrectly or the file does not exist, the following error message should be printed:

```
Please make your choice: 1
Please input the file name to load: Iceland.ppm
Cannot open file "Iceland.ppm" for reading!
```

In this case, the user should try option 1 again with a correct file name.

### 1.3.2 Save the image in PPM and 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; again, just use the provided one, `SaveImage`. When option 2 is selected, the program output should look like this:

```
Please make your choice: 2
Please input the file name to save: bw
bw.ppm was saved successfully.
bw.jpg was stored for viewing.
```

The saved image will be automatically converted to a JPEG image and sent to the folder *public_html*. You then are able to see the image at: `http://newport.eecs.uci.edu/~yourUCInetID/index.html`. Note that you may need to hit Refresh in your web browser.

### 1.3.3 Change the color image to black and white

(a) Color image.

(b) Black and white image.

Figure 2: A color image and its black and white counterpart.

A black and white image is one where the intensity values are the same for all three color channels, red, green, and blue, at each pixel. To change a color image to such a grey-scale image, assign the average intensity, which is given by $(R + G + B)/3$, to all the color channels of a pixel. Here, $R, G, B$ are the old intensity values for the red, the green, and the blue channels, respectively.

Define and implement the following function to do this job.

```
/* Change a color image to black and white */
void  BlackNWhite(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);
```

Figure 2 shows an example of this operation. Your program's output for this option should look like this:

```
Please make your choice: 3
"Black & White" operation is done!
```

For demonstrating your program, save the image with name **bw** after this step.

### 1.3.4 Make a negative of the 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.

(a) Original image.

(b) Negative image.

Figure 3: An image and its negative counterpart.

You need to define and implement the following function to do this DIP.

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

Figure 3 shows an example of this operation. Your program's output for this option should be like:

```
Please make your choice: 4
"Negative" operation is done!
```

Save the image with name **negative** after this step, so that it shows at the correct place in your web browser.

### 1.3.5 Flip the image horizontally

To flip the 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 the following function to do this DIP.

(a) Original image.

(b) Horizontally flipped image.

Figure 4: An image and its horizontally flipped counterpart.

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

Figure 4 shows an example of this operation. Your program's output for this option should be like this:

```
Please make your choice: 5
"HFlip" operation is done!
```

Save the image with name **hflip** after this step.

### 1.3.6 Mirror the image horizontally

To mirror an image horizontally, the intensity values at the left should be reversed and copied to the right. The following shows an example.

```
                            1 2 3 4 5 6                             1 2 3 3 2 1
  Before horizontal mirror: 7 8 9 0 1 2    After horizontal mirror: 7 8 9 9 8 7
                            3 4 5 6 7 8                             3 4 5 5 4 3
```

You need to define and implement the following function to do this DIP.

```
/* Mirror an image horizontally */
void  HMirror(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);
```

Figure 5 shows an example of this operation. Your program's output for this option should be like this:

```
Please make your choice: 6
"HMirror" operation is done!
```

Save the image with name **hmirror** after this step.

(a) Original image.

(b) Horizontally mirrored image.

Figure 5: An image and its horizontally mirrored counterpart.

### 1.3.7 Zoom into the image

(a) Original image.

(b) 2x zoom-in image.

Figure 6: An image and its 2x zoom-in counterpart.

In this section, you need to implement a simple zoom-in function, which enlarges the interesting part of the image by a factor of two.
To zoom in, the intensity values in the center of the picture are distributed over the whole picture. The following shows an example:

```
                1 |5  4| 10                      5  5  4  4
                4 |2  6| 11                      5  5  4  4
Before zoom-in: 3  8  7  14      After zoom-in: 2  2  6  6
                2  9  2  12                      2  2  6  6
```

You need to define and implement the following function to do this DIP.

```
/* Zoom in an image */
void  Zoomin(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);
```

For our demo image, the interesting part of the image you want to zoom in is within the area of (120, 0), (359, 0), (120, 179) and (359, 179). You may hand-code this region in your program.

Figure 6 shows an example of this operation. Your program's output for this option should be like this:

```
Please make your choice: 7
"Zoom" operation is done!
```

Save the image with name **zoomin** after this step.

### 1.3.8 Sharpen the image

The sharpening works this way: the intensity value at each pixel is mapped to a new value, which is the sum of itself and its 8 neighbour pixels, with different **weighted factors**. To sharpen the image, we intensify the value of each pixel while reducing the intesity of its neighbors. The sum of all factors is 1, which will result in an image with the same overall brightness as the original, but look sharper.

The following shows an example of the filter and the applied pixel $E$:

```
Filter :           Original Pixels
X  X  X  X  X       X  X  X  X  X
X -1 -1 -1  X       X  A  B  C  X
X -1  9 -1  X       X  D  E  F  X
X -1 -1 -1  X       X  G  H  I  X
X  X  X  X  X       X  X  X  X  X
```

To sharpen the edges of the image, the intensity of the center pixel $E$ is changed to $(-A-B-C-D+9*E-F-G-H-I)$. Repeat this calculation for every pixel and for every color channel (red, green, and blue) of the image. Note that you have to adjust the boundaries for the newly generated pixel values, i.e., the value must be within the range of [0,255]. To achieve this, we will use a technique called *saturated arithmetic*. Specifically, you calculate the intensity values first with full integer precision (use a temporary variable of type `int`). Next, you change any value larger than 255 to 255, and any value smaller than 0 to 0. Finally, you assign the resulting *saturated* value to the target pixel intensity (which is an `unsigned char`).

Note also that special care has to be taken for pixels located at the image boundaries. For ease of implementation, you may choose to ignore the pixels at the border of the image where no neighbor pixels exist.

You need to define and implement the following function to do this DIP.

```
/* Sharpen an image */
void Sharpen(unsigned char R[WIDTH][HEIGHT],
```

(a) Original Image

(b) Sharpened Image

Figure 7: An image and its sharpened counterpart.

```
unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);
```

The sharpened image should look like the right image in Figure 7 and your program output should look like this:

```
Please make your choice: 8
"Sharpen" operation is done!
```

Save the image with name **sharpen** after this step.

#### 1.3.9 Exchange the color channels of the image

An interesting effect to colorful images is achieved by swapping the color channels, which accidentally can happen easily when composite video cables are plugged into a color TV incorrectly. We will apply such exchanged color channels in the following three DIP operations.

```
 9: Exchange the red and green color channels
```

In each this DIP operations, the intensity values of the pixels are swapped, red becomes green and vice versa.

You need to define the following functions to implement these DIP operations.

```
/* exchange R and G color channels */
void ExRG(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
          unsigned char B[WIDTH][HEIGHT]);
```

Your program's output for these options should like this:

```
Please make your choice: 9
"Exchange RG" operation is done!
```

Save the image with name **xRG** after this step, so that it shows at the correct place in your web browser.

(a) Original image.

(b) Red-Green color channel swap.

Figure 8: Example images with color channel exchanges.

### 1.3.10 Add noise to the image

(a) Original image.

(b) Noise setting: n=20.

Figure 9: An image and its noise (salt-and-pepper) corrupted counterpart.

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 generation from the previous assignments. Figure 9 shows an example of this operation with n set to 20%.

You need to define and implement the following function to do this DIP.

```
/* Add noise to an image */
```

```
void  AddNoise(unsigned int percentage, unsigned char R[WIDTH][HEIGHT],
       unsigned char G[WIDTH][HEIGHT] ,unsigned char B[WIDTH][HEIGHT]);
```

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

```
please make your choice: 10
"Add noise" operation is done!
```

Save the image with name **noise** after this step.
**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 coordinates) of each noise pixel. Create a white or black pixel by setting the intensity value at each color channel to its maximum (255) or minimum (0) respectively. Pick the pixel color randomly but note that the number of white pixels should be approximately equal to the number of black pixels.

### 1.3.11 Add overlay to the image [10 Points Bonus]

By adding an overlay image, you can add logo, watermark or text to your original image. In this part, you add an overlay image of anteater to the original image. Figure 10 shows an example of this operation.

(a) Anteater running on the ground.

(b) Overlay anteater with original image

Figure 10: An overlay image and result of the overlaying

You need to define and implement the following function to do this DIP.

```
/* Add overlay image */
void Overlay(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);
```

In this function, you need to read the overlay image "anteater.ppm" using LoadImage and then create the result image by picking either a pixel from the original image or a pixel from the overlay image, depending on whether the pixel in the overlay image has all-white background color. Once user chooses this option, your program's output should be like:

(a) Original image

(b) Image with borders, border color = black, width = 20 pixels

Figure 11: An image and its counterpart when borders are added.

```
please make your choice: 1
"Add overlay" operation is done!
```

Save the image with name **overlay** after this step.

#### 1.3.12 Add borders to an image [10 Points Bonus]

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 11 shows an example of adding borders to an image.

You need to define and implement the following function to do this DIP.

```
/* add a border to the image */
void AddBorder(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
     unsigned char B[WIDTH][HEIGHT], int r, int g, int b, int bwidth);
```

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

```
Please make your choice: 12
Enter the R value of the border color(0 to 255): 0
Enter the G value of the border color(0 to 255): 0
Enter the B value of the border color(0 to 255): 0
Enter the width of the border: 20
"Add Border" operation is done!
```

Save the image with name **border** after this step.

### 1.3.13 Test all functions

Finally, you are asked to write a function to test all your implemented functions. In this test function, you are going to call your DIP functions one by one and observe the result on the original `Iceland` image. The function is for the designer to quickly test the program, so you should supply all necessary parameters without user interaction.

Specifically, the test function should be implemented as follows:

```
void AutoTest(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT])
{

    LoadImage("Iceland", R, G, B);
    BlackNWhite(R, G, B);
    SaveImage("bw", R, G, B);
    printf("Black & White tested!\n\n");

    LoadImage("Iceland", R, G, B);
    Negative(R, G, B);
    SaveImage("negative", R, G, B);
    printf("Negative tested!\n\n");

...

    LoadImage("Iceland", R, G, B);
    AddBorder(R, G, B, 0, 0, 0, 20);
    SaveImage("border", R, G, B);
    printf("AddBorder tested!\n\n");

}
```

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

```
Please make your choice: 13
Iceland.ppm was read successfully!
bw.ppm was saved successfully.
bw.jpg was stored for viewing.
Black & White tested!

Iceland.ppm was read successfully!
negative.ppm was saved successfully.
negative.jpg was stored for viewing.
Negative tested!

...
```

## 1.4 Implementation

In the following subsections, we provide some more specific information for your implementation.

### 1.4.1 Function Prototypes

For this assignment, you need the following functions. These function prototypes/declarations are already provided in the template file `PhotoLab.c`. Please do not change them.

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

/* Load an image from a file */
int LoadImage(const char fname[SLEN], unsigned char R[WIDTH][HEIGHT],
        unsigned char G[WIDTH][HEIGHT], unsigned char B[WIDTH][HEIGHT]);

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

/* Change a color image to black & white */
void BlackNWhite(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

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

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

/* Mirror an image horizontally */
void HMirror(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

/* Zoom into an image */
void  Zoomin(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

/* Sharpen an image */
void Sharpen(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

/* exchange R and G color channels */
void ExRG(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT], int percentage);
```

```
/* Add salt-and-pepper noise to image */
void AddNoise(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT], int percentage);

/* Add overlay image */
void Overlay(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

/* add a border to the image */
void Addborder(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT], int r, int g, int b, int bwidth);

/* Example: aging the photo */
void Aging(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);

/* Test all functions */
void AutoTest(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
        unsigned char B[WIDTH][HEIGHT]);
```

You may define any other functions as needed.

NOTE: The `LoadImage()`, `SaveImage()` functions are already defined in the provided `PhotoLab.c` start file. The `AutoTest()` function is partially defined. You need to complete the definition of its body. Also, the `AutoTest()` function is initially called in the `main()` function without the menu. This is just an example to show how to call the `AutoTest()` function. You will need to add the code for the menu of this program and use proper function calls for the different DIP operations.

### 1.4.2 Global Constants

The following global constants are also declared in the provided `PhotoLab.c`. Please do not change their names, nor their values.

```
#define WIDTH  480 /* image width */
#define HEIGHT 360 /* image height */
#define SLEN    80 /* maximum length of file names */
```

### 1.4.3 Passing arrays by reference

In the main function, three two-dimensional arrays are defined. These are used to store the RGB information for the current image:

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

```
    unsigned char G[WIDTH][HEIGHT];
    unsigned char B[WIDTH][HEIGHT];
}
```

When any of the DIP operations are called from the main function, those three arrays, R, G, and B are the arguments passed into the DIP functions. Since arrays are passed by reference, any changes to R, G, and B in the DIP functions will be applied to those referenced objects in the main function. This way the current image can be updated by DIP functions without defining any global variables.

In your DIP function implementation, there are two ways to save the target image information in R, G, and B. Generally both options work. However, you will need to decide which option is better for a specific DIP manipulation function at hand.

**Option 1: Using temporary variables** You can define local variables to save temporary image information. For example:

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

Then, at the end of each DIP function implementation, you may copy the data in RT, GT, and BT over to R, G, and B, respectively.

**Option 2: In-place manipulation** Sometimes you do not have to create any additional variables. Instead, you can directly manipulate the pixels in R, G, and B. For example, in the implementation of the `Negative` function, you can perform the manipulation of each intensity value in a single assignment.

# 2 Script File

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

1. Start the script by typing the command: *script*.
2. Compile and run your program.
3. Choose 'Test all functions'.
4. Exit the program.
5. Stop the script by typing the command: *exit*.
6. Rename the script file to *PhotoLab.script*.

NOTE: Make sure you use exactly the same names as shown in the above steps when saving the resulting images! The script file is important and will be checked for grading, so be sure to follow the above steps exactly to create your script file.

# 3 Submission

Use the standard submission procedure, namely `~eecs10/bin/turnin.sh`, to submit the following files:

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

Finally, please leave the images generated by your program in your *public_html* directory, so that we may consider them when grading. You don't have to submit any images.