EECS 22: Assignment 3

Prepared by: Weiwei Chen, Prof. Rainer Doemer

October 13, 2011

Due on Monday 10/31/2011 11:59pm. Note: this is a two-week assignment.

1 Digital Image Processing [100 points + 20 bonus points]

In this assignment you will learn how to break a program into multiple translation units, and compile them into one program. Based on the program *PhotoLab* for assignment2, you will be asked to develop some advanced digital image processing (DIP) operations, make them in separate translation units, manipulate bit operations, and develop an appropriate **Makefile** to compile your program with DEBUG mode on or off.

1.1 Introduction

In assignment2, you were asked to develop an image manipulation program *PhotoLab* by using DIP techniques. 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 by using the *PhotoLab*. This assignment will be based on assignment2.

1.2 Initial Setup

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

```
cd  ~/eecs22
mkdir hw3
cd hw3
cp ~eecs22/hw3/sailing.ppm ./
```

We will extend the *PhotoLab* program based on assignment2. Please reuse your **PhotoLab.c** file as the starting point for this assignment. You may need to copy **PhotoLab.c** from the *hw2* folder to *hw3* first.
NOTE:
We will use the PPM image file *sailing.ppm* for this assignment. Once a DIP operation is done, you can save the modified image as *name.ppm*, and 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 any web browser at: *http://newport.eecs.uci.edu/∼youruserid*, if required names are used. If you save images by other names, use the link *http://newport.eecs.uci.edu/∼youruserid/imagename.jpg* to access the photo.

Note that whatever you put in the *public_html* directory will be publicly accessible; 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 Decompose the program into multiple translation units

Please decompose the **PhotoLab.c** file into multiple translation units and header files:

- **PhotoLab.c**: the main module contains the *main()* function, and your menu function *PrintMenu()*.
- **FileIO.c**: the module for the function definitions of *ReadImage()* and *SaveImage()*.

- **FileIO.h**: the header file for **FileIO.c**, with the function declarations of *ReadImage()* and *SaveImage()*.
- **Constants.h**: the header file in which the constants to be used are defined.
- **DIPs.c**: the module for the DIP function definitions in assignment2, i.e. *BlackNWhite()*, *Negative*, *HFlip()*, *HMirror()*, *VFlip()*, *VMirror()*, *AutoTest()*.
- **DIPs.h**: the header file for **DIPs.c**, with the DIP function declarations.

HINT: Please refer to the slides of *Lecture7* for an example on decomposing programs into different translation units.

1.4 Compile the program with multiple translation units

The *PhotoLab* program is now modularized into three translation units: **PhotoLab.c**, **FileIO.c**, and **DIPs.c**. To compile them into one program, please use *"-c"* option for **gcc** to generate the object files for each translation unit, and then compile those object files into one program, e.g.

```
% gcc -c PhotoLab.c -o PhotoLab.o -ansi -Wall
...
% gcc PhotoLab.o FileIO.o DIPs.o -o PhotoLab -ansi -Wall
% ./PhotoLab
program executes
% _
```

1.5 Using 'make' and 'Makefile'

On the other hand, we can put the commands above into a **Makefile** and use the *make* utility to automatically build the executable program from source code. Please create your own **Makefile** with at least the following targets:

- *all*: the target to generate all the executable programs.
- *clean*: the target to clean all the intermedia files, e.g. object files, and the executable programs.
- **.o*: the target to generate the object file *.o from the C source code file *.c.
- *PhotoLab*: the target to generate the executable program *PhotoLab*.

To use your **Makefile**, please use this command:

```
% make all
```

The executable program *PhotoLab* shall be automatically generated then.

HINT: Please refer to the slides of *Lecture7* for an example on how to create a **Makefile**.

1.6 Advanced DIP operations

In this assignment, please add one more module named **Advanced.c** (**Advanced.h** as the header file) and implement the advanced DIP operations described below.

Please reuse the menu you designed for assignment2 and extend it with the advanced operations. The user should be able to select DIP operations from a menu as the one shown below:

```
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
```

(a) Original image

(b) Aged image

Figure 1: An image and its aged counterpart.

```
 6:  Mirror an image horizontally
 7:  Flip an image vertically
 8:  Mirror an image vertically
 9:  Age the image
10:  Decipher the watermark in the image
11:  Blur an image
12:  Detect edges
13:  Test all functions
14:  Exit
```

Note: options ' 11: Blur an image' and '12: Detect edges' are bonus questions (10pts each). If you decide to skip these two options, you still need to implement the option '13: Test all functions'.

1.6.1 Aging

This function ages the original photo to make it look like an old photo. For each pixel, the colored pixels are converted into gray first, then yellow color is added to each of the pixels proportionally. The formula to get the aged pixel is given below:

$p[B] = (p[R] + p[G] + p[B])/5$
$p[R] = p[B] * 1.6$
$p[G] = p[B] * 1.6$

Note that: $p[R]$, $p[G]$, $p[B]$ are the intensities of the R, G, B channel of one pixel respectively here.
Figure 1 shows an example of this operation. Once the user chooses this option, your program's output should like this:

```
Please enter your choice: 9
"Aging" operation is done!
------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Flip an image vertically
```

```
 8:  Mirror an image vertically
 9:  Age the image
10:  Decipher the watermark in the image
11:  Blur an image
12:  Detect edges
13:  Test all functions
14:  Exit
please enter your choice:
```

Save the image with name 'aging' after this step.

1.6.2 Bit Manipulations: decipher the digital watermark in the image

Steganography is the art and science of writing hidden messages in such a way that no one, apart from the sender and intended recipient, suspects the existence of the message, a form of security through obscurity (`http://en.wikipedia.org/wiki/Steganography`).

We are using bit manipulations to put digital watermark in the input file **"sailing.ppm"** this time. You can hardly notice any color changes in the image with bare eyes. However, we hid a short message as a digital watermark in **"sailing.ppm"**. We will tell you the way how we hid the message here. Please write a function to decipher the message and display it on the screen.

- **Function Prototype**
  You need to define and implement the following function to do this DIP.

  ```
  /* Decipher the Watermark */
  void WaterMarkDecipher(unsigned char R[WIDTH][HEIGHT],
  unsigned char G[WIDTH][HEIGHT],
  unsigned char B[WIDTH][HEIGHT]);
  ```

- **Description of the Cryptography**
  We have digitally watermarked a short message (20 characters) in the input image **"sailing.ppm"**.

  A message is basically a string, in other words, an array of **unsigned char** elements.
  The size of an **unsigned char** variable is 8 bits.
  We distribute each bit of the message characters as the last bits of the intensities of the R channel for the first 160 (8 * 20) pixels (R[0][0] - R[0][159]) in the image respectively.
  More specifically, the 1st character in the string is watermarked in pixel 0-7, the 2nd character is watermarked in pixel 8-15, ..., the 20th character is watermarked in pixel 152-159. For each pair of character (*ch*) and pixels (p_0, p_1, ..., p_7), the lowest bit of p_0's R channel intensity is set to be the lowest bit of *ch*, the lowest bit of p_1's R channel intensity is set to be the 2nd lowest bit of ch, ..., the lowest bit of p_7's R channel intensity is set to be the highest bit (the 8th lowest bit) of ch.

  For example: we need to watermark the letter $'A'$ (ASCII value is $0x41 = 01000001_2$) to pixels p_0, p_1, ..., p_7. Therefore,

  - 1st lowest bit of 'A' is: 1, then the lowest bit of p_0's R channel intensity is set to be 1;
  - 2nd lowest bit of 'A' is: 0, then the lowest bit of p_1's R channel intensity is set to be 0;
  - 3rd lowest bit of 'A' is: 0, then the lowest bit of p_2's R channel intensity is set to be 0;
  - 4th lowest bit of 'A' is: 0, then the lowest bit of p_3's R channel intensity is set to be 0;
  - 5th lowest bit of 'A' is: 0, then the lowest bit of p_4's R channel intensity is set to be 0;
  - 6th lowest bit of 'A' is: 0, then the lowest bit of p_5's R channel intensity is set to be 0;

(a) Image without blur

(b) Image with blur

Figure 2: The image and its blur counterpart.

- 7th lowest bit of 'A' is: 1, then the lowest bit of p_6's R channel intensity is set to be 1;
- highest bit of 'A' is: 0, then the lowest bit of p_7's R channel intensity is set to be '0'.

Since only the lowest bit of the intensity value is changed, the color difference can be barely noticed.

HINT: You can test your function by using a test input image **sailinghw.ppm**. The hidden message in this image is *****Hello World******. The test image can be accessed by the following commands:

```
% cd hw3
% cp  eecs22/hw3/sailinghw.ppm ./
```

You will need to use bitwise operators, e.g. **'&'**, **'<<'**, **'>>'**, **'|'** to extract the bits from or set the bits to a variable. If you use **sailinghw.ppm** as the input image, Your program's output for this option should be like:

```
please make your choice: 10
The watermark message is "****Hello World****".
"Decipher the Watermark Message" operation is done!

--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Flip an image vertically
 8:  Mirror an image vertically
 9:  Age the image
10:  Decipher the watermark in the image
11:  Blur an image
12:  Detect edges
13:  Test all functions
14:  Exit
please make your choice:
```

1.6.3 Blur (bonus points: 10pt)

After color aging, the image should to be blurred to make it look more like an old photo. The blurring works this way: the intensity value at each pixel is mapped to a new value, which is the average of itself and its 8 neighbors. The following shows an example:

```
X X X X X
X 4 1 2 X
X 2 0 1 X
X 4 1 3 X
X X X X X
```

To blur the image, the intensity of the center pixel with the value of 0 is changed to $(4+1+2+2+0+1+4+1+3)/9 = 2$. Repeat this for every pixel, and for every color channel (red, green, and blue) of the image. You need to define and implement a function to do this DIP.
Note 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.
After the two process, the aged image should look like the figure shown in Figure 2(b):

```
Please enter your choice: 11
"Blur" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Flip an image vertically
 8:  Mirror an image vertically
 9:  Age the image
10:  Decipher the watermark in the image
11:  Blur an image
12:  Detect edges
13:  Test all functions
14:  Exit
please enter your choice:
```

Save the image with name 'blur' after this step.

1.6.4 Edge Detection (bonus points: 10 pts)

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 comparisons 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 3 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. Let's 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:

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

(a) Original image

(b) Edge detection with k = 60

Figure 3: An image and its edge detection counterpart.

So, here is the algorithm for edge detection in pseudo code.

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) are greater than a given threshold K, then we have an edge pixel and turn this pixel to white, otherwise we turn this pixel to black.*

Note that you can avoid the need for the square-root function in your program (making it much faster!) by comparing the squares of the distances with K^2.
Note also that you should skip the pixels at the very bottom and the very right side since there are no neighbor pixels to compute.

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

```
Please make your choice: 12
Enter the difference threshold: 60
"Edge Detection" operation is done!
--------------------------------
 1:  Load a PPM image
 2:  Save an image in PPM and JPEG format
 3:  Change a color image to black and white
 4:  Make a negative of an image
 5:  Flip an image horizontally
 6:  Mirror an image horizontally
 7:  Flip an image vertically
 8:  Mirror an image vertically
 9:  Age the image
10:  Decipher the watermark in the image
11:  Blur an image
12:  Detect edges
13:  Test all functions
14:  Exit
please enter your choice:
```

Save the image with name 'edge' after this step.

1.7 Test all functions

Finally, you are going to complete the *AutoTest()* function to test all previous functions as assignment2. In this function, you are going to call DIP functions one by one and observe the results. The function is for the designer to quickly test the program, so you should supply all necessary parameters when testing. The function should look like:

```
void AutoTest(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
 unsigned char B[WIDTH][HEIGHT])
{
 char fname[SLEN] = "sailing";
 char sname[SLEN];

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

 ...

 ReadImage(fname, R, G, B);
 WaterMarkDecipher(R, G, B);
 printf("Watermark Deciphering tested!\n\n");

 ...
 ReadImage(fname, R, G, B);
 EdgeDetection(60, R, G, B);
 SaveImage("edge", R, G, B);
 printf("EdgeDetection tested!\n\n");

 ...
}
```

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

```
Please make your choice: 13

sailing.ppm was read successfully!
bw.ppm was saved successfully.
bw.jpg was stored for viewing.
Black & White tested!

...

sailing.ppm was read successfully!
The watermark message is "the hidden message".
Watermark Deciphering tested!

...

sailing.ppm was read successfully!
edge.ppm was saved successfully.
edge.jpg was stored for viewing.
EdgeDetection tested!
```

Please move this *AutoTest()* function from **DIPs.h** and implement it in **Advanced.c**. Make sure you change the header files accordingly. Since the *AutoTest()* function will call the functions in the **DIPs.c** unit, please include the header files properly. Also, be sure to adjust your **Makefile** for proper dependencies.

1.8 Support for the DEBUG mode

In C program, *macros* can be defined as preprocessing directives. Please define a macro named **"DEBUG"** in your source code to enable / disable the messages shown in the *AutoTest()* function.

When the macro is defined, the *AutoTest()* function will output as in Section 1.7. Otherwise, the output will be like:

```
Please make your choice: 13
The watermark message is "the hidden message".
```

Please decide in which function and in which module this **"DEBUG"** macro needs to be added.

1.9 Extend the Makefile

For the **Makefile**, please

- extend it properly with the targets for your program with the new translation unit: **Advanced.c**.
- generate two executable programs
  1. *PhotoLab* with the user interactive menu and the **DEBUG** mode off.
  2. *Test* without the user menu, but just call the *AutoTest()* function for testing, and turn the **DEBUG** mode on. The needed file **Test.c** will be very simple (only the *main()* function calling *AutoTest()*). We can thus use the same modules to generate different programs.

  Define two targets to generate these two programs respectively. Please use the *"-D"* option for gcc to enable / disable the DEBUG mode instead of defining the "DEBUG" macro in the source code. You may need to define more targets to generate the object files with different DEBUG modes.

2 Implementation Details

2.1 Function Prototypes

For this assignment, you need to define the following functions in **Advanced.h**:

```
/*** function declarations ***/

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

/* Decipher the Watermark */
void WaterMarkDecipher(unsigned char R[WIDTH][HEIGHT], unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);

/* blur the image */
void Blur(  unsigned char R[WIDTH][HEIGHT],  unsigned char G[WIDTH][HEIGHT],
unsigned char B[WIDTH][HEIGHT]);

/* detect the edge of the image */
void EdgeDetection( double K,  unsigned char R[WIDTH][HEIGHT],
```

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

You may want to define other functions as needed.

2.2 Global constants

The following global constants should be defined in **Constants.h**(please don't change their names):

```
#define WIDTH  640 /* Image width */
#define HEIGHT 425 /* image height */
#define SLEN    80 /* maximum length of file names */
#define MLEN    20       /*message length */
```

Please make sure that you properly include this header file when necessary.

2.3 Pass in arrays by reference

In the main function, three two-dimensional 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 updated by 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 specific DIP manipulation function at hand.

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];
}
```

Then, 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 Sometimes 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 directly back to this pixel entry.

3 Budgeting your time

You have two weeks to complete this assignment, but we encourage you to get started early as there are more work than assignment2. We suggest you budget your time as follows:

- Week 1:
  1. Decompose the program into different translation units, i.e. **PhotoLab.c**, **FileIO.c**, **FileIO.h**, **Constants.h**, **DIPs.c**, **DIPs.h**.
  2. Create your own **Makefile** and use it to compile the program.
  3. Create translation unit **Advanced.c**, **Advanced.h**, and implement an initial advanced DIP function.
- Week 2:
  1. Implement all the advanced DIP functions.
  2. Implement the *AutoTest()* function.
  3. Implement the **Test.c** file.
  4. Figure out how to enable disable the **DEBUG** mode in the source code and add targets to the **Makefile** accordingly.
  5. Script the result of your programs and submit your work.

4 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 *PhotoLab* by using your **Makefile**.
3. Choose 'Test all functions' (The file names must be 'bw', ..., 'aging', 'blur', 'edge' for the corresponding function).
4. Exit the program.
5. Compile and run *Test* by using your **Makefile**.
6. Clean all the object files and executable programs by using your **Makefile**.
7. Stop the script by typing the command: *exit*.
8. Rename 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. ***Please don't open any text editor while scripting !!!***

5 Submission

Use the standard submission procedure to submit the following files as the whole package of your program:

- *PhotoLab.c*
- *PhotoLab.script*
- *FileIO.c*
- *FileIO.h*

- *Constants.h*
- *DIPs.c*
- *DIPs.h*
- *Advanced.c*
- *Advanced.h*
- *Test.c*
- *Makefile*

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