ECE26400 Programming Assignment #4

This assignment is related to PE07. It builds on what you have done in PE07. In this assignment, you will perform Floyd-Steinberg dithering (see below) so that when you downscale the 24-bit bitmap representation to the 16-bit bitmap representation, the image continues to look reasonable.

In addition to the goals in PE07, you will learn

1. How to implement the Floyd-Steinberg dithering algorithm.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pa04_files.zip` using the following command:

```
unzip pa04_files.zip
```

The zip file `pa04_files.zip` contains a folder, named `PA04`, and two files within the folder and one subfolder containing some sample files:

1. `answer04.h`: This is a "header" file and it declares the functions you will be writing for this exercise.
2. `dithered`: This is a subfolder with valid BMP files generated using the Floyd-Steinberg dithering algorithm.
3. `idiff_tol`: An executable that helps you to identify the differences in the output files generated using your implementation of the Floyd-Steinberg algorithm and image files in the `dithered` subfolder.

In this exercise, you have to define in `answer04.c` all functions that have been declared in `answer04.h` and create a file called `pa04.c` for the `main` function. There is only one new function that you have to write. If you have completed PE07, you should copy your `answer07.c` from PE07 over, rename it as `answer04.c`, and define the new function in it. You will also find that `pa04.c` is very similar to `pe07.c` that you have written for PE07. You should also copy that file over, rename it as `pa04.c`, and make modifications to it to meet the requirements of PA04.

Please note that starting from PE07, there are changes to the `gcc` command used for the compilation of your program. **Please read Section 4**.

2 Floyd-Steinberg dithering algorithm

In PE07, the conversion of a 24-bit BMP into a 16-bit BMP file results in some loss in accuracy. When you use 32 levels (5-bit representation) to represent 256 levels (8-bit representation), the conversion suffers quantization error, which is defined as follows:

```
16_bit_value = 24_bit_value >> 3;  // 16_bit_value = 24_bit_value / 8;
equiv_24_bit_value_for_16_bit_value = (16_bit_value * 255) / 31;
quantization_error = 24_bit_value - equiv_24_bit_value_for_16_bit_value;
```

Note that although both 16-bit value and 24-bit value are non-negative, the quantization error may be positive, zero, or negative. You should recognize that in the preceding three equations, you have seen the first two, which you have used for 24-bit-to-16-bit conversion and 16-bit-to-24-bit conversion, respectively.

In PE07, we simply ignore the quantization errors at all pixels. As a consequence, the nice blue sky in the image `airplane.bmp` (see the image in PE07) became a sky with 50 shades of blue in `airplane16.bmp` and `airplane24.bmp`.

To render the quantization errors less noticeable, a dithering algorithm diffuses the quantization error of a pixel to 4 neighboring pixels, as shown below:

```
. . . . . . . . .
. . . X 7 . . .
. . 3 5 1 . . .
. . . . . . . . .
```

The quantization error of pixel `X` at location (x,y) is diffused to its right pixel $(x+1,y)$ with a weight of 7/16, to three pixels below X at locations $(x-1,y-1)$, $(x,y-1)$ and $(x+1,y-1)$ by a weight of 3/16, 5/16, and 1/16, respectively. Note that the denominator of 16 is for normalization as it is the sum of the weights 7, 3, 5, 1.

By diffusion, it means that the weighted quantization error of X is added to the pixel values of its 4 neighbors.

While a pixel contributes its quantization error to four neighbors, it also receives quantization errors from four neighbors.

Here, we assume a Cartesian coordinate system, with $(0,0)$ being the lower-left corner pixel of the image. In the BMP format, the first byte after the header information belongs to the pixel at $(0,0)$ format. The pixels are stored in the file from the bottom to the top of an image, and from the left to the right.

The following pseudo code shows that Floyd-Steinberg dithering algorithm.

```
/* FloydSteinberg dithering
 * for each y from TOP to BOTTOM
 *    for each x from LEFT to RIGHT
 *       oldpixel  := pixel[x][y]
 *       newpixel  := oldpixel scaled to fewer number of bits
 *       pixel[x][y]  := newpixel
 *       quant_error  := oldpixel - (newpixel scaled back to
 *                                     original number of bits)
 *       pixel[x+1][y  ] := pixel[x+1][y  ] + quant_error * 7/16
 *       pixel[x-1][y-1] := pixel[x-1][y-1] + quant_error * 3/16
 *       pixel[x  ][y-1] := pixel[x  ][y-1] + quant_error * 5/16
 *       pixel[x+1][y-1] := pixel[x+1][y-1] + quant_error * 1/16
 */
```

Here, `pixel[x][y]` refers to a particular color (R, G, or B) of the pixel, and the weighted quantization error is added to the corresponding color of the neighboring pixels. You have to do it for each color of RGB. Also, you diffuse the quantization error to only valid pixel locations (the pseudo-code did not check for the validity of the locations).

Lastly, you have to combine the 5 bits of each color into 16-bit as in PE07. It is important to note that addition of the quantization error may result in invalid value ($>$ 255 or $<$ 0). You have to make sure that you are dealing with valid pixel value. If the addition results in a value > 255, assign 255 to the pixel. If the addition results in a value < 0, assign 0 to the pixel.

The results of an implementation of the algorithm are shown in the `dithered` subfolder. You may compare the image `airplane16_dithered.bmp` and `airplane16.bmp` from PE07. You should observe that the 50 shades of blue have magically disappeared.

3 Functions you have to define

3.1 Function in `answer04.c`

You should copy over the `answer07.c` file that you wrote for PE07, rename it `answer04.c`, and add a new function to it.

```
// Given a BMP_image, create with dithering a new 16-bit image that is
// converted from a given 24-bit image
//
BMP_image *convert_24_to_16_BMP_image_with dithering(BMP_image *image);
```

For the given image, you may assume that it is a valid 24-bit image; otherwise, this function should not be called at all. You have to take the 24-bit representation and convert it into a 16-bit representation using the Floyd-Steinberg dithering algorithm.

As you have the experience of implementing the `convert_24_to_16_BMP_image` function, we will highlight a few differences between the two 24-bit-to-16-bit functions, and provide you some pointers that may be helpful in the completion of this new function.

First, the dithering algorithm scans the pixels from the TOP to the BOTTOM, and from LEFT to RIGHT. In the BMP file, the pixels are stored from the bottom to the top, and from left to right. If you processed pixels from bottom to top in the `convert_24_to_16_BMP_image` function, it may be helpful to copy the code from that function and modify it to perform the conversion of a 24-bit representation to a 16-bit representation from top to bottom. The correctness of conversion (without dithering) can be easily tested.

Second, you have to compute the quantization error. Pay attention that the quantization error for a color is not computed between the 8-bit value and the 5-bit value. Rather, it is between the 8-bit value and the 8-bit value converted from the 5-bit value. A static function for that may be useful.

Third, the algorithm shows that the weighted quantization error has to be added to the original 8-bit value of a color. However, as you are not allowed to make modifications to the given image (the header and the image data), you may want to store the sum of weighted quantization errors diffused to each pixel separately. Note that the algorithm involves only two rows of pixels when it scans from left to right for each row.

Fourth, the quantization error diffused to the neighbors of a pixel has to be divided by 16 (and multiplied by either 1, 3, 5, or 7). Consequently, the weighted quantization error from a single neighbor may become zero. It is therefore better to collect the quantization errors (multiplied by 1, 3, 5, or 7) from all four neighbors, and divided the sum by 16. can capture quantization error properly because the quantization error from just a neighbor may be small although the sum of quantization errors from all four neighbors is big enough to affect the pixel.

Fifth, the addition of weighted quantization errors may not result in a valid 8-bit value for a color (nonnegative and < 256). You have to make sure that your implementation does not have an underflow/overflow issue.

3.2 `main` function in `pa04.c`

Your `main` function should expect `argv[1]` as the input file and `argv[2]` as the output file.

If `argv[1]` does not provide you a valid BMP file, you should not produce a new BMP file, and you should return `EXIT_FAILURE`. Based on the format of a valid BMP file, you should perform a 24-to-16 bit conversion with dithering or a 16-to-24 bit conversion, and save the new image in `argv[2]`.

If for whatever reasons (insufficient arguments, memory allocation problem, file opening issue, format issue), the conversion should not be performed, the output file should not be produced and you should return `EXIT_FAILURE`.

Return `EXIT_SUCCESS` only if it is a successful conversion.

You are responsible for opening and closing files, allocating and deallocating memory.

3.3 Printing, helper functions and macros

All debugging, error, or log statements in `answer04.c` and `pa04.c` should be printed to `stderr`.

You may define your own helper functions in `pa04.c` and `answer04.c`. All these functions should be declared and defined as static functions. In other words, the scope of these functions are only within the files that they are found. Declaring and defining these functions as static eliminate the conflicts in function names.

You should not use macros that start with `T_` (Upper case T and underscore) in their names. We will use such macros in the evaluation of your submissions. If you use such macros, we may not be able to evaluate your submission properly.

4 Compiling your program

As you have advanced to this stage, we believe that you have the necessary experience to decide what warnings are allowed and what warnings should be eliminated. We will now remove the `-Werror` flag from the `gcc` command.

```
gcc -std=c99 -Wall -Wshadow -Wvla -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pa04.c answer04.c -o pa04
```

When we evaluate your program, we also use the optimization flag `-O3` (uppercase letter `O` and number three) instead of the `-g` flag as follows:

```
gcc -std=c99 -Wall -Wshadow -Wvla -O3 -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pa04.c answer04.c -o pa04
```

You are recommended to copy the `Makefile` from PE01 and modify it appropriately for PA04.

In fact, we will remove the `-Werror` flag for all exercises and assignments that come after PE07. However, for the re-submissions of PA01, PA02, and PA03, we will continue to use the `gcc` command as mentioned in the respective PDF files for those assignments.

5 Writing, running and testing your program

We provide a few test cases for you. You should already have some images from PE07. The `dithered` subfolder contains some images that have been converted from a 24-bit representation to a 16-bit representation with dithering.

We also included an executable called `idiff_tol` that you may use to compare two images. You may run the executable as follows:

```
./idiff_tol max_count tol bmp_file1 bmp_file2
```

Both `max_count` and `tol` should be replaced with integers. `bmp_file1` and `bmp_file2` should be replaced with the actual filenames of the BMP files you want to compare.

The comparison takes place only if the two files have identical header information. If not, the comparison stops.

A pixel from `bmp_file1` and the corresponding pixels from `bmp_file2` are checked for the RGB information. The (absolute) differences in R, G, and B values are added. If the sum is greater than the `tol`, the two pixels are considered to be different.

The output of the executable is made up of up to three parts. The optional first part reports the first `max_count` pixels that are different in the two given files. The coordinates of these pixels are given, with a pixel per line.

The second to last output is a number that corresponds to the total number of pixels in the BMP file.

The last output is a number that corresponds to the total number of pixels that are different.

If you just want to count the number of different pixels, do

```
./idiff_tol 0 0 bmp_file1 bmp_file2
```

If you allow only one of the RGB values to differ by 1, and do not want to know the locations:

```
./idiff_tol 0 1 bmp_file1 bmp_file2
```

If you allow only the sum of the (absolute) differences of RGB values to be no more than 2, and want to know the first 5 locations that are different:

```
./idiff_tol 5 2 bmp_file1 bmp_file2
```

You should also run `./pa04` with appropriate arguments under `valgrind`.

Please see PE01 description about `valgrind`.

6 Submission

You must submit a zip file called `PA04.zip`, which contains two files:

1. `answer04.c`
2. `pa04.c`

Assuming that you are in the folder that contains `answer04.c` and `pa04.c`, use the following command to zip your files:

```
zip PA04.zip answer04.c pa04.c
```

Make sure that you name the zip file as `PA04.zip`. Moreover, the zip file should not contain any folders. Submit `PA04.zip` through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

7 Grading

All debugging messages should be printed to `stderr`. It is important that if the instructor has a working version of `pa04.c`, it should be compilable with your `answer04.c` to produce an executable. For evaluation purpose, we will use different combinations of your submitted `.c` files and our `.c` files to generate different executables. If a particular combination does not allow an executable to be generated, you do not get any credit for the function(s) that the executable is supposed to evaluate. We use both `-g` and `-O3` flags (separately) to compile your submission. Your submission is evaluated only when compilation using each of the flags is successful.

It is important to note that the `.c` files from the instructor do not assume the presence of global variables that are declared by the students. Of course, the `.c` files from the instructor may use global variables that are in C libraries, such as `errno`, `stdout`, `stderr`, and so on. If your submission contains global variables that are declared by you, it is unlikely that your executable will work correctly.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The `read_BMP_image` function accounts for 10%. and the `write_BMP_image` function accounts for 5%. The `convert_24_to_16_BMP_image` function accounts for 20%. The `convert_16_to_24_BMP_image` function accounts for 10%. The `main` function accounts for 5%. Note that these functions are same as or similar to those from PE07 and they account for a total of 50% in this assignment. The only new function `convert_24_to_16_BMP_image_with_dithering` accounts for the remaining 50% of this assignment.

PA04 gives you two different ways of improving the final grade of PE07. Assume that you obtain x for the functions that are same as or similar to those from PE07 and y for the new function. Your final PE07 grade is $\max(\text{original PE07}, 2x, x+y)$.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

8 A few points to remember

All debugging messages should be printed to `stderr`. If the program creates an output file when it should not, or the output file is correct, you get 0 for that test case.

You can declare and define additional static functions that you have to use in `pa04.c` and `answer04.c`.

You should not use macros that start with `T_`.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.