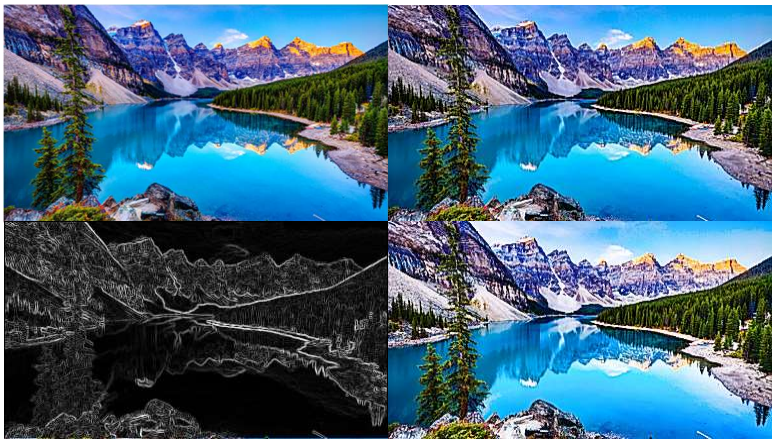
ECE220: Computer Systems & Programming **Fall 2020 ZJUI**
Machine Problem 6 **due: Saturday 6 November, 11:59:59 p.m.**

Processing Images as Arrays of Pixels



Your task this week is to implement a few basic image processing techniques on arrays of pixels. Specifically, you must perform conversion from RGB (red, green, and blue) pixels to HSL (hue, saturation, and luma/luminance) and back, edge detection using convolution with Sobel kernels, and histogram equalization to balance the distribution of brightness across the image.

Code provided to you makes use of your functions to process PNG image files. Starting with the image in the upper left, for example, edge detection produces the image shown in the lower left. Sharpening those edges produces the image in the upper right, and following up with histogram equalization produces the image in the lower right.

The objective for this week is for you to gain experience with using multi-dimensional arrays to represent information in C.

Background

Image Representations: Images are often stored and represented using red, green, and blue (RGB) intensities, which map naturally into displays. For example, image data stored as a Portable Network Graphics (PNG) file, the pixels are represented in this way. When we want to process those images, however, other representations are useful. The HSL (hue, saturation, luma) representation was developed by computer graphics researchers to better match human perception. The luma component represents luminosity, also known as intensity, and corresponds roughly to the brightness of a given pixel. Hue represents the color of the pixel, and saturation represents how much of that color is present in the pixel (a saturation of 0 is white, for example).

For this assignment, you must write C functions that convert between the RGB and HSL representations and back. Specifically, for each pixel in an image, its RGB representation is stored in a fairly common way as three separate bytes, with 8 bits of red (0-255), 8 bits of green (0-255), and 8 bits of blue (0-255). The HSL representation is defined for the purposes of this assignment: while it is similar conceptually to those that you can read about, the details do not match precisely. To maintain more information about the original RGB color, we make use of 16-bit unsigned values for H, S, and L. **All equations below correspond to the use of integer arithmetic, so be careful about the order of operations.**

To obtain H, S, and L from RGB, we start by finding the minimum and maximum RGB values, as follows:

$$M = \max(R, G, B) \qquad N = \min(R, G, B)$$

The luma L is then given by the sum of M and N, and we compute the chromaticity C as the difference between M and N (C is used to compute H and S):

$$L = M + N \qquad C = M - N$$

Using C, we can then find the pixel's saturation S:

$$S = \begin{cases} 0 & \text{if } M = 0 \text{ or } N = 255 \\ \dfrac{0\text{x}8000\ C}{L} & \text{if } 0 < L \leq 255 \\ \dfrac{0\text{x}8000\ C}{510 - L} & \text{otherwise} \end{cases}$$

Be sure that your implementation obeys the order of operations in the equations above (multiplication before division) to avoid differences in integer arithmetic results.

Finally, we compute H using the following equation:

$$H = \begin{cases} 0 & \text{if } C = 0 \\ \dfrac{0\text{x}2000\ (C + G - B)}{C} & \text{if } C > 0 \text{ and } M = R \\ \dfrac{0\text{x}2000\ (3C + G - B)}{C} & \text{if } C > 0 \text{ and } M > R \text{ and } M = G \\ \dfrac{0\text{x}2000\ (5C + G - B)}{C} & \text{otherwise} \end{cases}$$

Again, be sure to perform the division last in each case, and to use large enough integers to avoid overflow in your computations.

The meaning of hue is somewhat arbitrary, and is related to the mixing of display color components. The definition above is also rotated for simplicity of calculation, and thus does not match any standard definitions. For interest, the colors magenta, red, yellow, green, cyan, and blue correspond to H values 0x0000, 0x2000, 0x4000, 0x6000, 0x8000, and 0xA000, respectively.

Converting from HSL back to RGB is slightly more complicated, but is also done on a per-pixel basis. Again, your code must use the definitions and equations given in this specification.

We start by calculating chromaticity C from S and L as follows:

$$C = \begin{cases} \dfrac{S\ L}{0\text{x}8000} & \text{if } L \leq 255 \\ \dfrac{S\ (510 - L)}{0\text{x}8000} & \text{otherwise} \end{cases}$$

As before, be careful about operation order. The minimum RGB component N and maximum RGB component M can then be recovered from C and L:

$$N = (L - C)/2 \qquad M = N + C$$

Then we need to break apart the distinct regions of hue H. To do so, we define a major and a minor part of the hue as follows:

$$H_{\text{major}} = \frac{H}{0x2000} \qquad H_{\text{minor}} = H \ \&\ 0\text{x}3\text{FFF}$$

Remember that both values are integers. For simplicity, we have used the C bitwise AND operator in the definition of H_{minor}.

The third RGB component, which we call T, can be found using H_{minor} as follows:

$$T = \begin{cases} N + \dfrac{C\ (H_{\text{minor}} - 0\text{x}2000)}{0\text{x}2000} & H_{\text{minor}} \geq 0\text{x}2000 \\ N + \dfrac{C\ (0\text{x}2000 - H_{\text{minor}})}{0\text{x}2000} & \text{otherwise} \end{cases}$$

The major hue ranges from 0 to 5, and the order of RGB components depends on this value as follows:

$$(R, G, B) = \begin{cases} (M, N, T) & \text{if } H_{\text{major}} = 0 \\ (M, T, N) & \text{if } H_{\text{major}} = 1 \\ (T, M, N) & \text{if } H_{\text{major}} = 2 \\ (N, M, T) & \text{if } H_{\text{major}} = 3 \\ (N, T, M) & \text{if } H_{\text{major}} = 4 \\ (T, N, M) & \text{otherwise} \end{cases}$$

Edge Detection: Once we have an image in an HSL representation, we can use the L channel (the luma values for the pixels) to identify edges in the image. Edge detection is useful for image segmentation and computer vision applications, and the kinds of operations that you will implement often form the first step in image processing pipelines for any type of recognition. The mathematical operations, specifically convolution, are also used widely in implementing convolutional neural network layers, an important component in most AI/machine learning applications.

For this assignment, we make use of Sobel edge detection, which uses two 3×3 kernels (matrices) to identify changes in intensity in the horizontal and vertical directions in an image. The two kernels are shown here:

$$K_{\text{X}} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & \mathbf{0} & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad K_{\text{Y}} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & \mathbf{0} & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

In the kernels, X values increase to the right, and Y values increase as one moves down. These axes directions match those commonly used with images.

To perform edge detection, we convolve the kernels with the image. Consider the horizontal kernel, K_{X}. For each pixel in the image, we logically align the center value (the blue one) in the kernel with the pixel, multiply each kernel value by the corresponding image luma (one pixel's L value), and sum up the products. For example, to compute the value of the convolution of K_{X} with the blue pixel highlighted in the example L data to the right, we multiply (from the first row) 9 by 1, 8 by 0, and 4 by -1—these sum to 5. From the second row, we multiply 2 by 2, 3 by 0, and 4 by -2—these sum to -4. And, from the third row, we multiply 7 by 1, 8 by 0, and 7 by -1—these sum to 0. The total sum is thus 1, which is the result of the convolution for that pixel.

4	6	3	7	5	9
4	7	9	8	4	3
9	2	2	3	4	9
7	8	7	8	7	7
8	7	8	7	2	3
4	5	3	4	9	3

1	0	-1
2	0	-2
1	0	-1

Denoting the two-dimensional convolution operation by *, we compute $G_{\text{X}} = K_{\text{X}} * L$ and $G_{\text{Y}} = K_{\text{Y}} * L$. These values can then be used to find the magnitude and direction of the gradient (the rate of change) at each point in the image. You might wonder what we should do at the image boundaries, where some values in the kernel do not correspond to pixels (choose any edge pixel as the blue pixel in L). For simplicity, we do not compute anything for those pixels.

Histogram Equalization: Operations to balance color and lighting in an image are routinely performed by modern digital cameras. You may have noticed, for example, how much better each new smartphone captures images in twilight than did the last generation. For the last operation in this assignment, we make use of a simple technique to evenly distribute the luminosity of pixels in an image.

Starting again with the L channel for an image with P pixels, we begin by computing a histogram of L values over the entire image. In this assignment, our HSL definition restricts L values to the range [0,510], so the histogram is fairly small. Let's call that histogram H, then compute the cumulative histogram K as follows:

$$K(i) = \sum_{j=0}^{i} H(j)$$

If the pixels have a uniform distribution of luma values, we expect to find

$$H(i) = \frac{L}{511} \qquad \text{and} \qquad K(i) = \frac{(i+1)\,L}{511}$$

However, most images are not so uniform. Fortunately, we can make the lighting more uniform by remapping each of the 511 luma values to one that produces a more uniform distribution. Given a specific luma value X, K(X) corresponds to the number of pixels with luma value at or below that level, which we want to be equal to (X+1) L / 511. To make lighting uniform, let's choose another value, X', such that

$$K(X) = \frac{(X'+1)\,L}{511}.$$

Solving for X', we obtain

$$X' = \frac{511\,K(X)}{L} - 1$$

In other words, we should replace the luma value for every pixel that has L=X with the new value X', as given above. Somewhat arbitrarily, but following convention, we choose to round the division up rather than down, and, since the numbers may overflow 32 bits for large images, we make use of 64-bit values using the following, more C-like expression:

$$X' = (511 * (\text{int64\_t})K(X) + L - 1) \,/\, L \; - 1$$

By replacing all luma values in the image, we make lighting more uniform.

Pieces

Your program will consist of a total of three files:

`mp6.h`	This header file provides type definitions, function declarations, and brief descriptions of the subroutines that you must write for this assignment.
`mp6.c`	The main source file for your code. A version has been provided to you with full headers for each of your subroutines; **be sure to read the function headers before you begin coding.** You need merely fill in the body of each subroutine.

A third source file is also provided to you:

`main.c`	A source file that interprets commands, reads and writes PNG files, calls your subroutines, and sharpens edges in an image. You need not read this file, although you are welcome to do so.

A **Makefile** and several sample images have also been provided for convenience. The **Makefile** allows you to build the **mp6** executable by simply typing **make** at the command line. The images are in a subdirectory. We have also included a compiled version of a solution, called **gold**, that you can use to generate additional results for comparison with your own solution. The outputs of your program should match the gold version exactly in all cases.

The Task

You must write four C subroutines in this assignment. Don't panic: the total amount of code needed in my version was only about 130 extra lines! I recommend implementing the functions in the order described in this document, which allows you to perform some amount of debugging as you implement rather than trying to debug everything at once.

Step 1: Implement RGB to HSL conversion.

Converting RGB to HSL is slightly simpler than the reverse conversion. Implement the C function

```
void convert_RGB_to_HSL (int32_t height, int32_t width,
    const uint8_t* red, const uint8_t* green, const uint8_t* blue,
    uint16_t* H, uint16_t* S, uint16_t* L);
```

in **mp6.c**. Use the equations presented earlier in this specification. Each of the six arrays is a two-dimensional image laid out in row-major order in memory. Be sure that you understand the discussion in lecture and know how to compute the linear index for any (x,y) coordinate in the image. Note that the image width (and height) are given to you as parameters.

Once you have finished this function, compile the program and use the HSL output option to make sure that you have produced the correct answers. Note that the output file in this case is human-readable, and is not a PNG image. You can use the sample images provided as inputs or use your own (in PNG format), and can use the **gold** executable to produce correct answers for any input image.

Step 2: Implement HSL to RGB conversion.

Next, implement the reverse conversion by completing the C function

```
void convert_HSL_to_RGB (int32_t height, int32_t width,
    const uint16_t* H, const uint16_t* S, const uint16_t* L,
    uint8_t* red, uint8_t* green, uint8_t* blue);
```

in **mp6.c**. As with the first step, use the equations presented earlier in this specification. To keep the computations reasonably simple, the equations are slightly lossy, so converting RGB to HSL and back does not always produce the original RGB value. Feel free to improve the design on your own if you are interested, but be sure to turn in a solution that matches this specification exactly.

Again, each of the six arrays is a two-dimensional image laid out in row-major order in memory.

Once you have finished this function, compile the program and use the RGB→HSL→RGB output option to make sure that you have produced the correct answers. This option (and all others except the first option) produces a PNG image as output. Visual comparison of the original and final images should be enough for initial testing, but you can also use image comparison tools to compare your program's output with that of the **gold** executable provided to you.

Step 3: Implement Sobel edge detection.

You are now ready to implement Sobel edge detection on the L channel. To do so, complete the C function

```
void compute_sobel_kernels (int32_t height, int32_t width,
    const uint16_t* L, int32_t* Gx, int32_t* Gy);
```

in **`mp6.c`**. Sobel edge detection is fairly standard, but we still suggest following the specification given earlier, as both gradient direction and handling of boundaries is somewhat arbitrary, and yours must match ours exactly for credit.

As with the earlier functions, each of the three arrays is a two-dimensional image laid out in row-major order in memory. Although you need not produce any values for the boundary pixels **in the Gx and Gy output arrays, these boundaries are included in the arrays**. In other words, all three arrays have size **`height * width`**. The boundary pixels will be ignored—your code need write nothing into them.

Once you have finished this function, compile the program and use the edge detection output option to make sure that you have produced the correct answers. This option produces a PNG image as output. Visual comparison of your output with that of **`gold`** should be enough for initial testing, but a more thorough comparison with image comparison tools is recommended.

The edge detection output uses only the magnitude of the gradient (the Gx and Gy arrays). You can check that you didn't flip the direction by using the edge sharpening output option, which also uses the gradient direction to sharpen the edges in an image. Note that the code for doing so has been provided to you, so you need do nothing other than run the program with a different option to check that your program's output matches that of **`gold`**.

Step 4: Implement histogram equalization.

Finally, implement the C function

```
void equalize_intensities (int32_t height, int32_t width, uint16_t* L);
```

in **`mp6.c`**. Here, again, the implementation is fairly specific to our HSL definition, so follow the description earlier in this specification exactly, and be sure to use the final expression given for re-mapping intensities. **For full credit, build a lookup table for intensity remapping rather than re-calculating the new intensity for each pixel.**

Only the L channel array is provided to this function, both as input and as output. The image height and width are also given, of course.

Once you have finished this function, compile the program and use the all + equalization output option to make sure that you have produced the correct answers. This option produces a PNG image as output. Visual comparison of your output with that of **`gold`** should be enough for initial testing, but a more thorough comparison with image comparison tools is again recommended.

Specifics

Be sure that you have read the function headers and other information in the code before you begin coding.

- Your code must be written in C and must be contained in the **`mp6.c`** file in the **`mp/mp6`** subdirectory of your repository. We will NOT grade any other files. **Changes made to any other files WILL BE IGNORED during grading.** If your code does not work properly without such changes, you are likely to receive 0 credit.
- You must implement the **`convert_RGB_to_HSL`**, **`convert_HSL_to_RGB`**, **`compute_sobel_kernels`**, and **`equalize_intensities`** functions correctly.
- You may assume that all parameter values are valid when your functions are called, provided that you also produce only valid parameters in your own functions (incorrect values produced by your functions may be passed back to your other functions). In particular,
  - arrays will be sized as specified and laid out in row-major order,
  - red, green, and blue values will be in the range 0-255, and
  - luma values will be in the range 0-510.
- We will not test on images larger than a few million pixels, although your code should not have any explicit checks on the provided values of image height and width.
- Your routines' outputs must be correct.
- Do not make any assumptions about the order or number of times that your functions are called. For testing, we will make use of additional code, and you will lose points if your functions do not work repeatedly as specified, whether or not other functions are called before or after them.
- Your code must be well-commented. Follow the commenting style of the code examples provided in class and in the textbook.

Compiling and Executing Your Program

When you are ready to compile, type

```
make
```

The Makefile describes the dependences between your source files and the mp6 executable, so whenever you make changes to a source, the command above should automatically rebuild your code. The "**`-g`**" argument is included to tell the compiler to include debugging information so that you can use **`gdb`** to find your bugs (you will have some).

The "**`-Wall`**" argument is also included to tell the compiler to give you warning messages for any code that it thinks likely to be a bug. Track down and fix all such issues, as they are usually bugs. Also note that if your code generates any warnings, you will lose points.

If compilation succeeds, you can then execute the program by typing, "**`./mp6`**" (no quotes) to get instructions on how to use the program, including how to specify input file, output file, and choice of processing option.

Remember that testing is your responsibility. Follow the suggested ordering and guidelines for developing your code and comparing with the output of the **`gold`** program.

Grading Rubric

Functionality (70%)

- 20% - **`convert_RGB_to_HSL`** function works correctly
- 20% - **`convert_HSL_to_RGB`** function works correctly
- 15% - **`compute_sobel_kernels`** function works correctly
- 15% - **`equalize_intensities`** function works correctly

Style (15%)

- 10% - conversion code (both **`convert_RGB_to_HSL and convert_HSL_to_RGB`**) uses a variable to store pixel index rather than re-computing linear array indices on every array access
- 5% - **`equalize_intensities`** uses a lookup table rather than re-computing a new intensity value for each pixel

Comments, Clarity, and Write-up (15%)

- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 10% - code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if you code does not compile, you will receive no functionality points. As always, your functions must be able to be called many times and produce the correct results, so we suggest that you avoid using any static storage (or you may lose most/all of your functionality points).