COMPSCI 220 Fall 2017
Assignment 3: Design For Reuse, Image Manipulation

Out: 19 September, 2017
Due: 3 October, 2017

General Instructions For All Assignments

Assignments in COMPSCI 220 are to be programmed in C++11, using the virtual machine image provided for the class. All assignments must follow the instructions on the handouts exactly, including instructions on input and output formatting, use of language features, libraries, file names, and function declarations. Failure to do so will automatically result in docked points. All assignments are individual: you must not discuss assignment contents, solutions, or techniques with fellow students. Please review the course policies on the course website, `https://people.cs.umass.edu/~joydeepb/220R`.

Instructions For This Assignment

In this assignment, you will be writing robust, scalable, and generalizable code to perform image filtering by convolution. Your code must be written in the file `assignment3.cpp`, and it must be accompanied by a header file `assignment3.h`. You may have helper functions in your submission, as long as there is no `main` function (see Section 4 for how to compile with provided main files). You are not allowed to use any library other than the Google Testing library, CImg, and iostream. Within the CImg library, you are not allowed to use any member function of the CImg class other than the constructors, raw pixel read / write functions (`operator() (...)`), and image property functions (`width()`, `height()`, `depth()`, `spectrum()`).

1 Image Filtering By Convolution

Convolution is a versatile technique of image filtering to produce a variety of effects such as blurring and sharpening, with variable parameters. The key to image filtering by convolution is to compute the value of each filtered pixel as a weighted sum of its neighboring pixels in the original image. Table 1 shows an example filter, and image. For each pixel in the filtered image, the filter is centered at that pixel location, and the pixel values from the original image, for every neighbor that overlaps with the filter, is weighted by the value of the filter at that location. The sum of all the values is then normalized by the sum of the weights in the filter. From the example image and filter in Table 1, the value of the center pixel of the filtered image will be,

$$
\begin{aligned}
&(1*22+1*32+1*42+\\
&\ 1*23+4*33+1*43+\\
&\ 1*24+1*34+1*44)/(1+1+1+1+4+1+1+1+1)\\
=&\ (396)/(12)\\
=&\ 33
\end{aligned}
$$

1	1	1
1	4	1
1	1	1

(a) Example filter

11	21	31	41	51
12	22	32	42	52
13	23	33	43	53
14	24	34	44	54
15	25	35	45	55

(b) Example Image

Table 1: Example filter and image for convolution.

Since the value of each pixel in the filtered image depends on the value of the neighboring pixels in the original image, you will need to appropriately handle the cases near the edges of the image when the filter boundary extends beyond the edges of the image. Exclude the pixel locations that are invalid, and only take into account the values and the weights that are within the image. For example, the value of the top left corner of the filtered image from the example in Table 1 will be,

$$\begin{aligned} & (4*11+1*21+1*12+1*22)/(4+1+1+1) \\ =\ & (99)/(7) \\ =\ & 14.143 \end{aligned}$$

2 Code Implementation

The CImg library will be used to load and input the image and filter. The filter is guaranteed to be square, with an odd integer length, and with a single color channel. Note that the values in the image are of type `unsigned char`, and can thus take values between 0 and 255 (both inclusive). The filter should have values in the range -128 to +127. Thus, you must determine the correct value by subtracting 128 from the values loaded. That is, if the CImg pixel value for a particular cell in the filter is 120, it should be interpreted as having the value $120 - 128 = -8$. You will write the image filtering function with the following signature:

```
CImg<unsigned char> FilterImage(const CImg<unsigned char>& image,
                                const CImg<unsigned char>& filter)
```

In the function signature, `image` is the input image, and `filter` is the filter. The function will return the filtered image.

3 Code Robustness and Versatility

You will have to think about how to design your program to be both robust, as well as versatile. In particular, your code should be able to handle:

1. Images of any size (width and height),
2. Images of arbitrary number of color channels (spectrum), and

3. Filters of any size (width and height) as long as width = height and width is an odd integer,

For images with multiple color channels, the filtering operation must be performed independently for each channel. Furthermore, if the resulting values overflow the range of an `unsigned char`, you will have to clip the values to 0 or 255. Feel free to create appropriate helper functions. For reference, this assignment can be written in less than 50 lines of code in C++. If you find yourself writing a significantly longer program, try re-thinking the organization of your program.

4 Example Output

To assist with the development of this assignment, we have provided you with following files:

1. `filter_main.cpp`, to compile with your code and header file and create an executable that can load an image file and a filter file.
2. `testing_main.cpp`, to compile with your code and header file and create an executable to run unit tests.
3. `blur1.png`, a 5x5 filter to blur the image,
4. `blur2.png`, a 19x19 filter to blur the image even more,
5. `sharp1.png`, a 5x5 filter to sharpen the image, and
6. `sharp2.png`, a 19x19 filter to sharpen the image even more.
7. `robot.jpg`, a sample color image, and
8. `robot-bw.jpg`, a sample grayscale image.

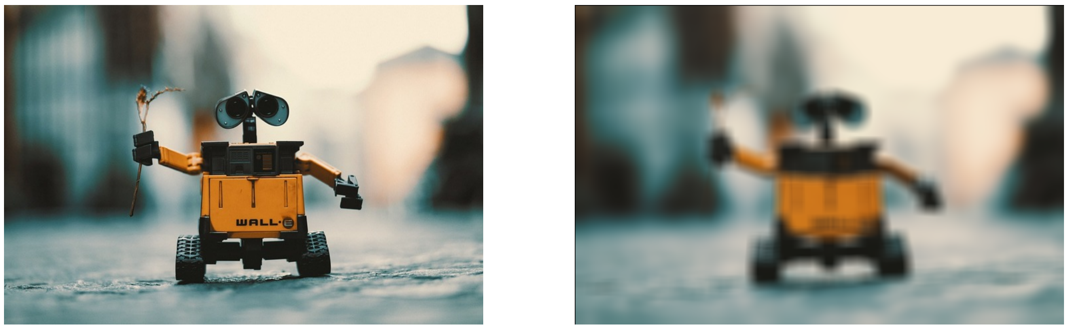
Figure 1 shows an example input image, and resulting filtered image when applying the `blur2.png` filter. To compile your code with `filter_main.cpp` and create a standalone image filtering program called `filter`, run:

```
clang++ -std=c++11 assignment3.cpp filter_main.cpp -lX11 -lpthread -o filter
```

To compile your code with `testing_main.cpp` and create a unit testing program called `run_tests`, run:

```
clang++ -std=c++11 assignment3.cpp testing_main.cpp -lX11 -lpthread -lgtest -o
↪  run_tests
```

Important: Note that the order of the parameters matters to the compiler because it resolves dependencies from left to right. The test cases in `testing_main.cpp` are not sufficient to test for correctness: it is up to you to ensure that you write adequate additional test cases to verify correctness. However, a correct submission must pass the provided test cases.



(a) Input image (b) Output image

Figure 1: Example input and output image, filtered using the blur2.png filter.

5 What To Turn In

Using the provided submission script `submit.sh`, the files `assignment3.h` and `assignment3.cpp` will be checked to see if they compile, and if successful, will be added to an archive, `assignment3.tar.gz`. You must upload this archive to Moodle.