ECE198KL: Introduction to Computer Engineering II **Spring 2013**
Program 6 **due: 10 p.m. on Monday 4 March**

Image Convolution

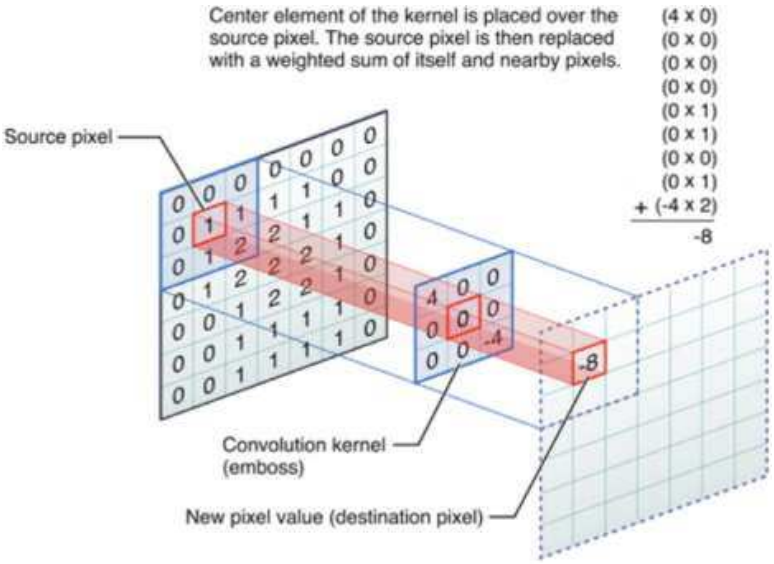
Your task this week is to implement the Gaussian blur technique used for blurring images using image convolution. Gaussian blur is a type of image blurring that uses a Gaussian function for calculating the filter to apply to each pixel in the image.

The objective for this week is for you to gain some experience with arrays and pointers, to implement code using subroutines, and to solve a problem using programming knowledge gained from the class.

As always, routine details such as how to obtain the code and how hand in your program can be found in the specification for Program 1. Instructions on targeting and making the Linux and Android versions can be found in the specification for Program 4.

Background

An image is composed of an array of pixels. Each pixel has four bytes of information in the RGBA format representing the red, blue, green and alpha channels. The values for each of the channels range from 0 to 255. Alpha channel is reserved for transparency information and may not be present in all images. Images can be modified by a process called filtering. A filter in image processing is an array that is applied to the entire image by a process shown to the right:



You overlay the filter over the source pixel and multiply each overlapping value in the filter by the corresponding value in the image, then sum all of these values (as shown in the upper-right of the figure). This process is called convolution.

The resulting sum of products should never be out of bounds (< 0 or > 255). You should clamp the values should they go out of bounds. In other words, replace values larger than 255 with 255, and values smaller than 0 with 0.

The convolution process is performed for each pixel of each color array (red, green, blue). The alpha channel may or may not be included in the process and is dependent on the type of image processing done. It will be made clear before the filter process whether to include or not include the alpha channel.

You may notice that if the filter is placed over a pixel at the edge of the image, it will not have enough information to process. Therefore, you can consider the pixels outside of the image to be zero-valued pixels (equivalently, ignore them). In other words, you will apply the filter only to the available pixels. *Do not access values beyond the array bounds!*

In order to generate the filter for Gaussian blur, a Gaussian function is used. The equation of the Gaussian function in two dimensions is given as

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{x^2 + y^2}{2\sigma^2}\right] \tag{1}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution. x and y go from $-radius$ to $radius$. The meaning of $radius$ will be explained soon.

In theory, the Gaussian function at every point on the image will be non-zero, meaning that the entire image would need to be included in the calculations for each pixel. In practice, when computing a discrete approximation of the Gaussian function, pixels at a distance of more than 3σ are small enough to be considered effectively zero. Thus contributions from pixels outside that range can be ignored.

Hence, σ determines the size of the filter. The radius of the filter which is the number of elements extending in the either direction from the center can be determined by the following equation

$$radius = ceil(3 * \sigma) \tag{2}$$

For example, if $\sigma = 2.5$, then $radius = ceil(7.5) = 8$. This means that the filter array is 17×17 (indices from 0 to 16 in the filter, applied from -8 to 8 around each pixel in the image). Note that the `ceil` function is a standard C math library function.

Understanding Arrays

Your functions will be given arrays for each of the four channels: red, green, blue and alpha. These arrays will be passed as pointers to 8-bit pixels (that is, as `uint8_t*`). However, they represent two-dimensional arrays of specified `height` and `width` (these values are also provided to your functions). To make use of the arrays, you will have to calculate the appropriate offset into the one-dimensional C array based on the two indices of the pixel that you want to access. For example, the 2×3 array of integers shown to the right could be defined in C as:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
int A[6] = { 1, 2, 3, 4, 5, 6};
```

In memory, the array will be laid out as (assuming 4 bytes for ints):

Memory location	1000	1004	1008	1012	1016	1020
Contents	1	2	3	4	5	6

To access individual elements we use the index operator. For example, As you notice, in this case the difference in offset from one column to the next is 1 and from one row to the next is 3 (the width of the two-dimensional array). The linear offset from the beginning of the array to any given element $A_{\text{row,column}}$ can be computed as:

$offset = row * width + column$

In the example, the width is 3. Hence, if you have a pointer as `int* arr_ptr = A;`, then you can access an element at row r and column c as:
`arr_ptr[r*NUMCOLS + c]`

Pieces

We are using the same package to allow cross-compilation on different platforms as well as support for the graphical user interface. With a number of image files and more sample tests, this week's distribution is almost 150 files. Again, however, you again need only look at three of those files—the rest serve to incorporate your code into the graphical game and to allow you to build the game as a text version on the lab machines or as a graphical version on Android, Linux, Windows, or WebOS.

The three files that you should examine are a header file and the source file that you must complete.

Let's discuss each of these in a little more detail:

`jni/prog6.h`	This header file provides function declarations and descriptions of the functions that you must write for this assignment.
`jni/prog6.c`	The main source file for your code. Function headers for all regular and challenge functions are provided to help you get started.
`jni/test.cpp`	The main source file for testing your code. It compares your solution with the gold solution for all the functions you have written. This program is written in C++.

The Interface

The screenshot of the program is shown to the right. The text box at the lower left corner represents the value for σ. The + and − buttons can be used to increase and decrease σ in steps of 0.5. The other buttons are used as follows:
Blur - Gaussian blur
Pixelate - Pixelate the image (challenge)
Rotate - Rotate the image by 90 degrees clockwise (challenge)
Pencil - Pencil sketch effect (challenge)

Details

You should read the descriptions of the functions in the header file and peruse the function headers in the source file before you begin coding.

Below are the tasks you need to complete for this assignment.

1 Create Gaussian filter based on σ

2 Apply Gaussian filter to the image to obtain Gaussian blur effect

Creating the Gaussian Filter

To create the Gaussian filter, you will use Equation (1). In the program, you will create the Gaussian filter in the function:

```
void calculateGFilter (double *gFilter, double sigma);
```

Here, *gFilter* represents a pointer which points to the Gaussian filter which has already been allocated enough memory. You need not allocate any memory. You just have to calculate the Gaussian values.

The radius of the filter is calculated using Equation (2). You need to implement this in the function and use it to get the radius value:

```
int getRadius(double sigma);
```

Once the Gaussian values for the filter are calculated, you need to *normalize* the filter. To normalize the Gaussian filter, you first add up the values of each element in the filter array. This sum is known as the 'weight'. You then divide the value of each element in the filter array by the weight. After normalization, the numbers in your filter should sum to exactly 1.

Applying the Gaussian Filter to the Image

To apply the Gaussian filter to the image you will use the Gaussian filter created in the above function and the convolution process described in the Background section. You need to apply convolution for all four channels. Note that if the radius of the filter is less than one, the input image should not be altered. In the program, you will implement your logic in the function:

```
void blurImage (uint8_t *in_red, uint8_t *in_green, uint8_t *in_blue, uint8_t *in_alpha,
                uint8_t *out_red, uint8_t *out_green, uint8_t *out_blue, uint8_t *out_alpha,
                double *gFilter, double sigma, int height, int width);
```

Here, *gFilter* is the pointer to the Gaussian kernel created in the previous function, σ is the standard deviation of the Gaussian function, *height* and *width* are the height and width of the image.

In your program you should make use of the following math functions and constant which are already defined and included:

```
double ceil (double x );
double exp (double x );
double sqrt (double x );
M_PI - π constant
```

Challenges for Program 6

Here are some challenges for this week. You can find the function signatures in the header or the source file. You can test correct behavior in the Android demo package (see the web page). For the challenges, you must either write your own tests or use the graphical version to test on the lab machines.

(9 points) Implement Pencil sketch effect. To do this, you need to implement the following functions.
`invertColors` - inverts the value of the pixel p, in each channel to $255 - p$. Alpha channel is not inverted but just copied from the input channel. (3 points)

`calculateGFilter` - calculates the Gaussian filter (part of the regular assignment).
`blurImage` - Gaussian blur (part of the regular assignment).
`linearDodge` - is a blending mode for two images. The effect is for each pixel(say pix_a) in image one, the relating pixel(say pix_c) in result picture would be pix_a plus the relating pixel in picture two(say pix_b) multiplied by a float number which determines the amount of dodge. (pix_c = pix_a + pix_b * amount). (3 points)
`convertToGray` - converts the image to gray scale. Each pixel will be set to the weighted average according to the formula $0.299R + 0.587G + 0.114B$. Alpha channel will be retained from the input image. (3 points)
You just have to implement these functions. We have written the code to call each of these functions serially to obtain the pencil sketch effect.

(6 points) Implement `rotate90Clockwise`, which rotates the image by 90 degrees clockwise.

(5 points) Implement `pixelateImage`, which pixelates the image. An input block size is given. Apply the block repeatedly from the upper left corner of the image (starting at (0,0)). Set the color of each block to the average color of all pixels in the block including the alpha channel. At the boundaries, use only the available pixels.

Specifics

- Your code must be written in C and and must be contained in the `prog6.c` file provided to you — we will NOT grade files with any other name.
- You must implement `calculateGFilter` and `blurImage` correctly.
- Your routine's return values and outputs must be correct.
- Your code must be well-commented. You may use either C-style (`/* can span multiple lines */`) or C++-style (`// comment to end of line`) comments, as you prefer. Follow the commenting style of the code examples provided in class and in the textbook.

Building and Testing

We suggest that you begin by developing your code on the Linux machines in the lab. As with earlier programs, all operations mentioned here should be performed from your `prog6` directory, not from the `jni` subdirectory that contains the file with your code.

You should test your program thoroughly before handing in your solution. You should get in the habit of writing your own tests.

We have provided a test program that individually checks each of the functions implemented. It checks the output of your functions with the output from the the gold solution. If the outputs do not match, it will point out the channel and the location of the mismatching pixel along with the expected value.

In order to compile the test program, run the script `compile_test` from the `prog6` directory as:

```
./compile_tests
```

Once compiled, run the tests from `prog6` directory as:

```
./run_tests
```

Note that the tests can only be run on the EWS machines.

Grading Rubric

Functionality (65%)
- 20% - `calculateGFilter` function works correctly
- 40% - `blurImage` function works correctly
- 5% - `getRadius` function works correctly

Style (15%)
- 5% - compilation generates no warnings (note: any warning means 0 points here)
- 10% - indentation and variable names are appropriate and reasonably meaningful

Comments, clarity, and write-up (20%)
- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 15% - code is clear and well-commented

Some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.

Note on performance: The image convolution process is compute intensive. The performance with high sigma values may not be that great on the Android device. We recommend using sigma values less that 3.0 on Android device.