15-122 : Principles of Imperative Computation

Fall 2011

Assignment 1: Image Manipulation

(Programming Part)

Due: Thursday, September 15, 2011 by 23:59

For the programming portion of this week's homework, you'll review how images are stored in the computer using C_0 (described in Section 1.1), and then you'll write four C_0 files: `imageutil.c0` (described in Section 1.2), `quantize.c0` (described in Section 1.3), `rotate.c0` (described in Section 1.4), and `blur.c0` (described in Section 1.5). You'll also have an opportunity to design your own image processing function (described in Section 1.6).

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

1 Assignment: Image Manipulation (25 points)

Starter code. Download the file `hw1-starter.zip` from the course website. When you unzip it, you will find a number of files including these three C_0 files— `quantize- main.c0`, `rotate-main.c0`, and `blur-main.c0` —corresponding to the three required image manipulation problems below. Each file has a `main()` function that will read an image from disk, call your code on its representation, and then write the result image back to disk. **Do not submit these files when you hand in your code, and the files you submit should not include `main()` functions.**

In addition, you will find a sample manipulation `remove-red.c0`, which removes the red channel from each pixel of an image, and its associated main file `remove-red-main.c0`. This sample provides a complete program that you can compile and execute, and you may pattern your code after the code in `remove-red.c0` if you find it convenient to do so. (The code for the `remove_red` function also appears in Appendix A.)

Finally, you will also see an `images/` directory with some sample input images and some sample outputs for some of the manipulations. On a Linux cluster machine, there are several programs you can use to view the images, including `display`, `gpicview`, `qiv`, `eog`, and `gthumb`. Play around and find one you like.

Compiling and running. To compile one of your completed exercises just specify the file(s) on the command line in the order you want them compiled. You can compile your files on any Andrew system by running the command

```
cc0 <file1>.c0 <file2>.c0 <file3>.c0 -o <executablefilename>
```

from the directory where your c0 files reside. This will will place the compiled binary in the file `<file>` rather than the usual default `a.out`.

Once you've compiled `<file>` in this way, you can run it with the command

```
./<executablefilename>
```

The file so produced will expect some options of its own, at the very least an option `-i <input file>` specifying the input image to manipulate. If you run one of the programs without any arguments, you will get a short usage message explaining the options particular to that program.

As a concrete example, you can compile the `remove-red` filter with dynamic checking and run it on the sample image `g5.png` in the `images/` directory by running the following commands in sequence:

```
cc0 -d remove-red.c0 remove-red-main.c0 -o remove-red

./remove-red -i images/g5.png -o images/g5nored.png
```

If you have any problems compiling or running your code as described here, you should contact the course staff.

Submitting. Once you've completed some files, you can submit them by running the command

```
handin -a hw1 <file1>.c0 ... <fileN>.c0
```

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include `//@requires`, `//@ensures`, and `//@loop_invariant` annotations in your program. You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct.

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. If you find yourself writing the same code over and over, you should write a separate function to handle that computation and call it whenever you need it. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

1.1 Image Manipulation Overview

The three short programming problems you have for this assignment deal with manipulating images. An image will be stored in a one-dimensional array of integers, where each integer is a 32-bit value representing one pixel of the image. Pixels are stored in the array row by row, left to right starting at the top left of the image. For example, if a 5×5 image has the following pixel "values":

$$\begin{array}{ccccc} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{array}$$

then these values would be stored in the array in this order:

$$a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p\ q\ r\ s\ t\ u\ v\ w\ x\ y$$

In the 5×5 image, the pixel i is in row 1, column 3 (rows and columns are indexed starting with 0) but is stored in the one-dimensional array at index 8. An image must have at least one pixel.

Each pixel in the array is a 32-bit integer that can be broken up into 4 components with 8 bits each:

$$a_1a_2a_3a_4a_5a_6a_7a_8\ r_1r_2r_3r_4r_5r_6r_7r_8\ g_1g_2g_3g_4g_5g_6g_7g_8\ b_1b_2b_3b_4b_5b_6b_7b_8$$

where:

$a_1a_2a_3a_4a_5a_6a_7a_8$	represents the alpha value (how opaque the pixel is)
$r_1r_2r_3r_4r_5r_6r_7r_8$	represents the intensity of the red component of the pixel
$g_1g_2g_3g_4g_5g_6g_7g_8$	represents the intensity of the green component of the pixel
$b_1b_2b_3b_4b_5b_6b_7b_8$	represents the intensity of the blue component of the pixel

Each 8-bit component can range between a minimum of 0 (binary 00000000 or hex 0x00) to a maximum of 255 (binary 11111111 or hex 0xFF).

For example, a pixel that is completely opaque with only green at its maximum intensity would be stored as the integer 0xFF00FF00. An opaque pixel that is medium gray would be 0xFF7F7F7F (equal parts red, green, and blue at medium intensity).

For the rest of the assignment, we will work under the assumption of a type definition that makes `pixel` an alias for `int`:

```
typedef int pixel;
```

Since `int`s are used for many other things (like the width and height of an image, for example), a type alias is useful for distinguishing those instances where we mean to interpret an `int` as an RGB pixel. You should include this `typedef` in your code and use the `pixel` type when appropriate.

1.2 Creating a set of Image Utility Functions

In this problem, you will complete the implementation of the functions specified in the `imageutil.c0` file. This file contains functions that may be helpful for you in the subsequent problems. Read the comments for each function to determine what each function should do.

TASK 1 (6 pts.) Complete the C_0 file `imageutil.c0` that includes a number of helpful image utility functions. For each function, in addition to completing the code, write the **strongest** precondition(s) and postcondition(s) (using `requires` and `ensures`). Include additional assertions and loop invariants as necessary. We will compile your program as follows:

```
cc0 -d imageutil.c0  imageutil-main.c0
```

using your `imageutil.c0` file. Your code must compile using these instructions with files shown in the order given. Do NOT include a main function in your `imageutil.c0` file. You can write your own `imageutil-main.c0` if you wish for testing purposes.

Figure 1: A sporty coupe with quantization level 0 (left) and level 7 (right).

1.3 Quantization of an Image

In this problem, you will implement a function that achieves a quantization effect on an image. Quantization reduces the total number of colors used in an image. You can see an example in Figure 1.

Given an ordinary image of size $w \times h$ and a quantization level q between 0 and 7, inclusive, for each pixel in the image, take each color component (red, green and blue) and clear the lowest q bits. For example, suppose the color components for a pixel are given by the bytes

```
RED      GREEN    BLUE
01101011 10111110 11010111
```

If the quantization level is 5, then the resulting pixel should have the following color components (note how the lower 5 bits are all cleared to 0):

```
RED      GREEN    BLUE
01100000 10100000 11000000
```

Note that an image processed with a quantization level of 0 should not change. For each pixel, do not change its alpha component.

TASK 2 (6 pts.) Create a C_0 file `quantize.c0` with a function `quantize` matching the following prototype:

```
pixel[] quantize(pixel[] pixels, int width, int height, int q_level);
```

This function should implement the algorithm described above, given an array `pixels` representing an image of width `width` and height `height` using a quantization level `q_level`. The returned array should be the representation of the image after quantization has occurred. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function. If the supplied quantization level is out of range, your program should abort with an annotation failure when compiled and run with the -d flag.

We will compile your program as follows:

```
cc0 -d imageutil.c0 quantize.c0 quantize-main.c0
```

using your `imageutil.c0` and `quantize.c0` files. Your code must compile using these instructions with files shown in the order given. Do NOT include a main function in your `quantize.c0` file.

Figure 2: Original image (left); Image after "rotation effect"

1.4 Rotation Effect

In this problem, you will create a rotation effect on an image.

Your task here is to implement a function that takes as input an image of size $w \times h$ and creates a "Rotation" image of size $2w \times 2h$ that contains the same image repeated four times, the top right image containing the original image, the top left containing the original image rotated 90 degrees counterclockwise, the bottom left containing the original image rotated 180 degrees, and the bottom right containing the original image rotated 90 degrees clockwise. **Note that the original image must have the same width and height in order to do the "rotation" effect.** A sample image is shown in Figure 2.

TASK 3 (6 pts.) Create a C_0 file `rotate.c0` implementing a function `rotate` matching the following prototype:

```
pixel[] rotate(pixel[] pixels, int width, int height);
```

where `width` and `height` represent the width and height of the original input image.

The returned array should be the array representation of the "Rotation" image. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function. If the supplied image is not "square" (i.e. its width does not equal its height), your program should abort with an annotation failure when compiled and run with the -d flag.

We will compile your program as follows:

```
cc0 -d imageutil.c0 rotate.c0 rotate-main.c0
```

using your `imageutil.c0` and `rotate.c0` files. Your code must compile using these instructions with files shown in the order given. Do NOT include a main function in your `rotate.c0` file.

1.5 Blurring an Image

In this problem, you will write a function to blur an image. This is done by using a "mask". A mask is an $n \times n$ array of non-negative integers representing *weights*. (Note: although weights can be 0, the weight in the center position of the mask cannot be zero.) For our purposes, n must be odd. The *origin* of the mask is its center position. For each pixel in the input image, think of the mask as being placed on top of the image so its origin is on the pixel we wish to alter. The original intensity value of each pixel under the mask is multiplied by the corresponding value in the mask that covers it. These products are added together and then we divide by the total of the weights in the mask to get the new intensity of the mask. Always use the original values for each pixel for each mask calculation, not the new values you compute as you process the image.

For example, refer to Figure 3, which shows a 3×3 mask and an image that we want to blur. Suppose we want to compute the new intensity value for pixel `e`. Imagine overlaying the mask so its center position is on `e`. We would compute the new intensity for the pixel e as:

```
(a + 3b + c + 3d + 5e + 3f + g + 3h + i)  / 21
```

This calculation is done three times for each pixel, once for its red channel, once for its green channel, and once for its blue channel. Do not change the alpha channel of the pixel.

Note that sometimes when you center the mask over a pixel you want to blur, the mask will hang over the edge of the image. In this case, compute the weighted sum of only those pixels the mask covers. Remember that you must divide by the sum of only those weights that you use from the mask. For the example shown in Figure 4, the new intensity for the pixel `e` is given by:

```
(3b + c + 5e + 3f + 3h + i)  / 16
```

Figure 5 shows a sample image blurred using the following masks:

```
1 3 1          1 2 3 2 1
3 5 3          2 3 4 3 2
1 3 1          3 4 5 4 3
               2 3 4 3 2
               1 2 3 2 1
```

TASK 4 (7 pts.) Create a C_0 file `blur.c0` with a function `blur` matching the following prototype:

```
pixel[] blur(pixel[] pixels, int width, int height,
             int[] mask, int maskwidth);
```

This function should implement the blur algorithm described above, given an array `pixels` representing an image of width `width` and height `height`, using an array of weights `mask` with width (and height) `maskwidth`. The returned array should be the representation of the blurred image. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function. If the supplied image does not match the size

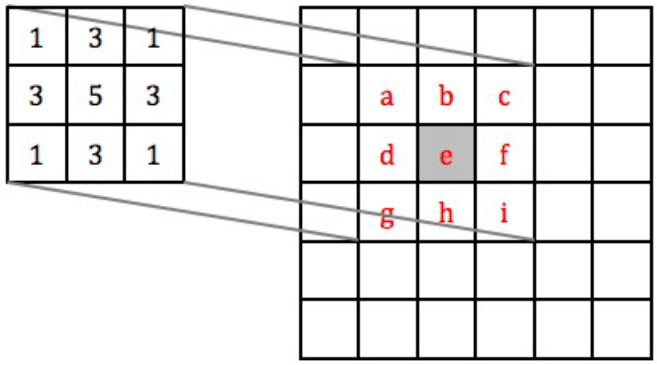

Figure 3: Overlay the 3 X 3 mask over the image so it is centered on pixel e to compute the new value for pixel e.

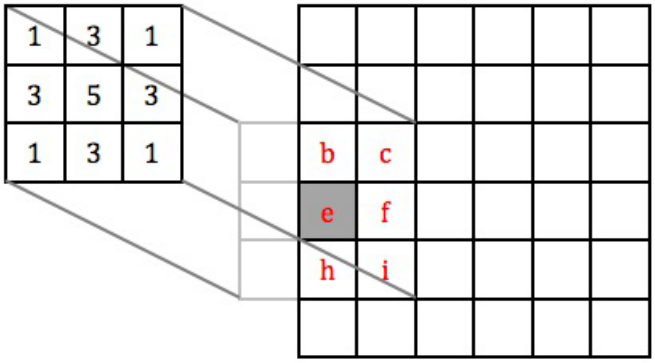

Figure 4: If the mask hangs over the edge of the image, use only those mask values that cover the image in the weighted sum.

given by width and height, or if the mask is not square or does not match the width given by maskwidth or if the mask has non-positive integers or if the maskwidth is not odd, your program should abort with an annotation failure when compiled and run with the -d flag.

We will compile your program as follows:

```
cc0 -d imageutil.c0 blur.c0 blur-main.c0
```

using your `imageutil.c0` and `blur.c0` files. Your code must compile using these instructions with files shown in the order given. Do NOT include a main function in your `blur.c0` file.

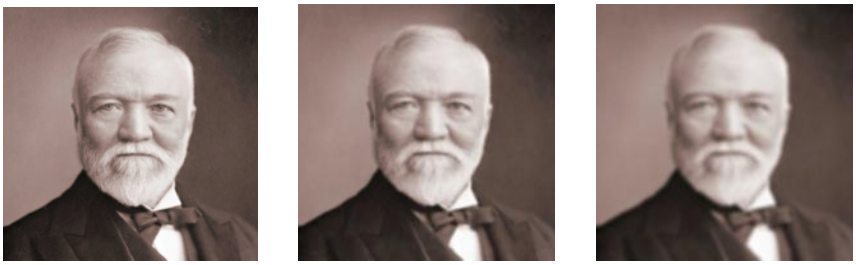

Figure 5: Andrew Carnegie: original image (left), blurred with a 3×3 mask (middle), and a 5×5 mask (right). See text for mask values.

1.6 Your own image processing algorithm (Optional)

TASK 5 (Optional)

Write a function `manipulate` that performs an image manipulation of your choice matching the following prototype:

```
pixel[] manipulate(pixel[] pixels, int width, int height);
```

You will also have to write two small functions that express the width and height of the result of your manipulation in terms of the width and height of the input image:

```
int result_width(int width, int height);
int result_height(int width, int height);
```

The starter code archive contains a file `manipulate-starter.c0` with empty stubs for these functions and a main file `manipulate-main.c0` that you can compile against to get a binary that runs your manipulation.

If you choose to do this task, be creative! A "judges' prize" will be awarded to the student whose submission "impresses" the course staff the most. (Of course, we reserve the right to decide for ourselves what that means!)

A Sample Code: Remove Red Channel from an Image

```
/* make pixel a type alias for int */
typedef int pixel;

pixel[] remove_red (pixel[] A, int width, int height)
//@requires \length(A) >= width*height;
//@ensures \length(\result) == width*height;
{
  int i;
  int j;
  pixel[] B = alloc_array(pixel, width*height);

  for (j = 0; j < height; j++)
  //@loop_invariant 0 <= j && j <= height;
  {
      for (i = 0; i < width; i++)
      //@loop_invariant 0 <= i && i <= width;
      {
        // Clear the bits corresponding to the red component
        B[j*width+i] = A[j*width+i] & 0xFF00FFFF;
      }
  }
  return B;
}
```