15-122 : Principles of Imperative Computation

Fall 2012

Assignment 1: Image Manipulation - UPDATE 1

(Programming Part)

Due: Thursday, September 13, 2012 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), `invert.c0` (described in Section 1.3), `reflect.c0` (described in Section 1.4), and `mask.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 `handout-hw1.tar` from the course website or Autolab. When you unpack it, you will find a number of files.

Four of the files are `imageutil.c0`, `invert.c0`, `reflect.c0`, and `mask.c0`, which are where you will write your solution to the required image manipulation problems below. These four files are the only files that you should submit for these problems. Note that you should not write a `main()` function in any of these files. You will also see several `*-main.c0` files, which contain the `main()` functions for each task. These can be used to compile and test your code. Finally, there is `manipulate.c0` and `manipulate-main.c0`, for the optional manipulation described in Section 1.6.

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 to Autolab. There are two ways to do this:

From the terminal on Andrew Linux (via cluster or ssh) type:

```
handin hw1 imageutil.c0 invert.c0 reflect.c0 mask.c0 manipulate.c0
```

Your score will then be available on the Autolab website.

Your files can also be submitted to the web interface of Autolab. To do so, please `tar` them, for example:

```
tar -cvf hw1_solutions.tar imageutil.c0 invert.c0 reflect.c0 mask.c0 manipulate.c0
```

Then, go to `https://autolab.cs.cmu.edu/15122-f12` and submit them as your solution to homework 1.

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

Testing. You are encouraged to use the provided `*-main.c0` files to help you test your code. Feel free to write additional testing code of your own before submitting to Autolab– don't rely on the grader's output for debugging. For this assignment, we are providing a program, `imagediff` to help you compare your output images to the sample images in the handout, optionally saving an image that shows you exactly where the two images differ. It is in the course directory on `afs`, so it is available on any cluster machine or when you are connected via ssh. For example:

```
imagediff -i images/sample-image.png -j images/my-image.png -o images/diff.png
```

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. Proper use of annotations will be tested by Autolab and may be considered in the style grade.

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 Piazza if you're unsure of what constitutes good style.

Task 0 (5 points) 5 points on this assignment will be given for 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:

$$a_0a_1a_2a_3a_4a_5a_6a_7\, r_0r_1r_2r_3r_4r_5r_6r_7\, g_0g_1g_2g_3g_4g_5g_6g_7\, b_0b_1b_2b_3b_4b_5b_6b_7$$

where:

$a_0a_1a_2a_3a_4a_5a_6a_7$	represents the alpha value (how opaque the pixel is)
$r_0r_1r_2r_3r_4r_5r_6r_7$	represents the intensity of the red component of the pixel
$g_0g_1g_2g_3g_4g_5g_6g_7$	represents the intensity of the green component of the pixel
$b_0b_1b_2b_3b_4b_5b_6b_7$	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. The specification (expected input and output) for each function is written in a comment above the function declaration.

TASK 1 (5 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. [**UPDATED**] We did not distribute a `imageutil-main.c0` file; you can check that `imageutil.c0` compiles using `coin`. You are encouraged to write your own `imageutil-main.c0` file that tests `imageutil.c0`. Do NOT include a main function in your `imageutil.c0` file.

1.3 Selectively Inverting an Image

You may be familiar with the "invert colors" or "photo negative" effect commonly available in image editors. In this problem, we will invert the colors of only a part of an image. The part that we choose to invert will be based on a target color and a tolerance level. You can see an example in Figure 1.

Figure 1: [**UPDATED**] A sporty coupe with no inversion (left); partial inversion, color=0xffcf0000, tolerance=100 (middle); and full inversion, color=0xffcf0000, tolerance=255 (right).

Given an ordinary image of size $w \times h$ along with a target color and a tolerance, examine each pixel. If the difference between the starting image and the target color is less than or equal to the tolerance for the red, green, and blue channels, then that pixel should be inverted. You should not look at the alpha value. The tolerance will be specified as an integer between 0 and 255. Essentially, if the target color is close enough to the color of a particular pixel, that pixel should be inverted. By inverting, we mean flipping each bit. For example, suppose the color components for a pixel are given by the bytes:

```
COLOR    RED       GREEN     BLUE
BINARY   01111101  10100001  10010111
DECIMAL  125       161       151
HEX      0x7d      0xa1      0x97
```

And our target color is:

```
COLOR    RED       GREEN     BLUE
DECIMAL  143       178       158
HEX      0x8f      0xb5      0x9e
```

If our tolerance is 20, then we have a match and should invert the pixel to the following value:

```
COLOR    RED       GREEN     BLUE
BINARY   10000010  01011110  01101000
DECIMAL  130       94        104
HEX      0x82      0x5e      0x68
```

Note that an image processed with a tolerance of 0 will only invert an exact match and a tolerance of 255 will invert the entire image. For each pixel, do not change its alpha component.

TASK 2 (4 pts.) In the C_0 file `invert.c0`, complete the `invert` function:

```
pixel[] invert(pixel[] pixels, int width, int height, int color, int tolerance);
```

This function should implement the algorithm described above, given an array `pixels` representing an image of width `width` and height `height` using a target color of `color` and a tolerance of `tolerance`. The returned array should be the representation of the image after inversion has occurred. It should **not** be destructive - that is, you should make your changes in a copy of the array, and not in the original array. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function. If the supplied tolerance level is out of range or if `width` and `height` do not agree with the size of the array `pixels`, 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 invert.c0 invert-main.c0 -o invert
```

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

After compiling, here is an example of how to call your program:

```
./invert -i images/g5.png -o images/g5-invert.png -c 0xffcf0000 -t 100
```

Note that the target color is specified in hexadecimal, beginning with "0x".

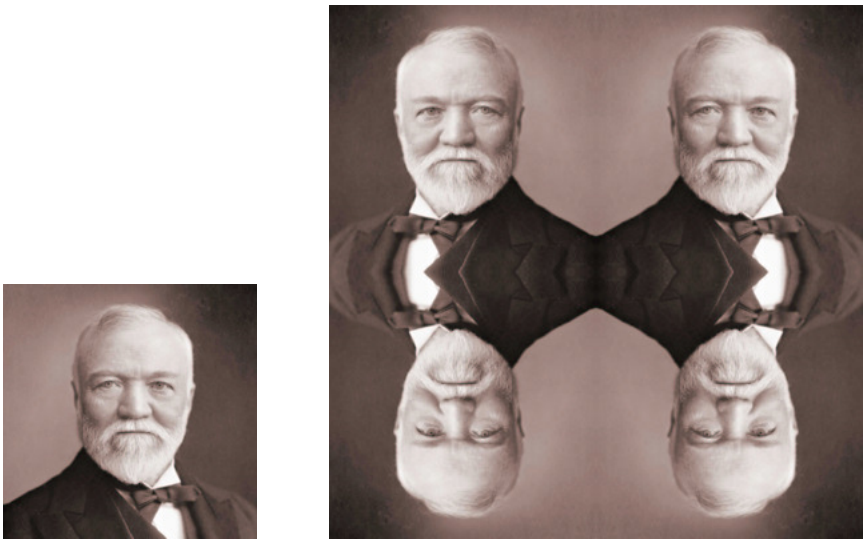

Figure 2: Original image (left); Image after "reflection effect"

1.4 Reflection Effect

In this problem, you will create a reflection 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 "Reflection" 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 image reflected across the y-axis, the bottom right containing the image reflected across the x-axis, and the bottom left containing the image reflected across both axes. A sample image is shown in Figure 2.

TASK 3 (5 pts.) In the C_0 file `reflect.c0`, complete the `reflect` function:

```
pixel[] reflect(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 "Reflected" image. It should **not** be destructive - that is, you should make your changes in a copy of the array, and not in the original array. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function. If `width` and `height` do not agree with the size of the array `pixels`, 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 reflect.c0 reflect-main.c0 -o reflect
```

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

```
./reflect -i images/carnegie.png -o images/carnegie-reflect.png
```

1.5 Applying Masks to an Image

In this problem, you will write a function that will apply a "mask" to an image. A mask is an $n \times n$ array of integers representing *weights*. 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 examine. The 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. Always use the original values for each pixel for each mask calculation, not the new values you compute as you process the image. Your function will return an array of integers the same size as the image, which contains the result of applying the mask.

Suppose we want to apply the following mask to the image:

$$\begin{bmatrix} 1 & 3 & 1 \\ 3 & 5 & 3 \\ 1 & 3 & 1 \end{bmatrix}$$

Refer to Figure 3 to see how this process works. Suppose we want to compute the value for pixel `e`. Imagine overlaying the mask so its center position is on `e`. We would compute the result for pixel e as:

```
a + 3b + c + 3d + 5e + 3f + g + 3h + i
```

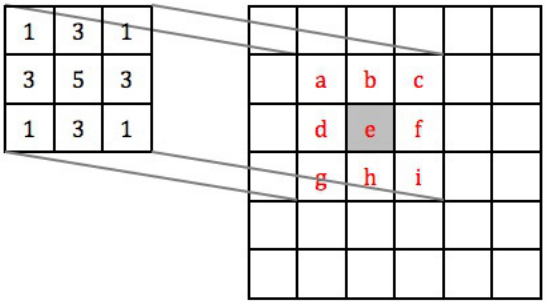

Figure 3: Overlay the 3 X 3 mask over the image so it is centered on pixel e to compute the new value for pixel e.

[UPDATED] Instead of doing this calculation for each channel individually, use the average value of the red, green, and blue channels. Ignore the alpha channel. For example, if the pixel is given by $(a, r, g, b) = (255, 107, 9, 217)$, then use $(107 + 9 + 217)/3 = 111$.

Note that sometimes when you center the mask over a pixel you want to operate on, the mask will hang over the edge of the image. In this case, compute the weighted sum of only those pixels the mask covers. For the example shown in Figure 4, the result for the pixel `e` is given by:

```
3b + c + 5e + 3f + 3h + i
```

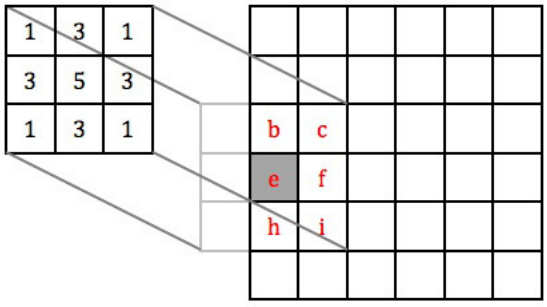

Figure 4: If the mask hangs over the edge of the image, use only those mask values that cover the image in the weighted sum.

TASK 4 (6 pts.) In the C_0 file `mask.c0`, complete the `apply_mask` function:

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

This function should implement the masking algorithm described above, given an array `pixels` representing an image of width `width` and height `height`, using the mask specified by `mask` and `maskwidth`. The returned array should contain the values after the mask is applied. 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 given by width and height, if the size of the mask does not agree with `maskwidth`, or if `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 mask.c0 mask-main1.c0 -o mask1
cc0 -d imageutil.c0 mask.c0 mask-main2.c0 -o mask2
```

using your `imageutil.c0` and `mask.c0` files. Your code must compile using these instructions with files shown in the order given. Do NOT include a main function in your `mask.c0` file. The transformations done by these main function is described in the next section. Sample usage is:

```
./mask1 -i images/cmu.png -o images/cmu-blur.png -m blur1.txt
./mask1 -i images/cmu.png -o images/cmu-blur.png -m blurmask.txt
./mask2 -i images/cmu.png -o images/cmu-edge.png
```

[**UPDATED**] The sample file `images/cmu-gaussian.png` distibuted with the assignment was created with `-m blurmask.txt`.

1.5.1 Applications

The first main function you are given to test your code, `mask-main1.c0`, reads a mask from a text file, specified by the `-m` option. The mask is read in from the file and passed along to `apply_mask`. Then, the data returned from `apply_mask` is used to calculate new intensity values for the pixels. This is done by summing all of the weights of the mask and dividing by it. Note that this will cause the edge of the image to have a lower intensity than it should, since we're not considering the part of the mask that hangs off of the image, but this is an acceptable simplification of the problem. Since we're allowing our masks to have negative values, this creates the possible issue of having an intensity greater than 255. If this is the case, the intensities are scaled appropriately. Since we're returning just one value instead of when per channel, this has the effect of converting the image to grayscale.

One application of masks is blurring an image, which would be the effect created by the examples shown in Figure 3 and Figure 4.

The other main function you are given to test your code, `mask-main2.c0`, implements an edge detection algorithm, which is another application of masks. The algorithm described here is an implementation of Canny Edge Detection, using Sobel Operators. In this case, the function `apply_mask` will be called three times. The first call will be to blur the image. For this purpose, the following mask will be used:

$$\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

After getting the resulting grayscale image, two more filters (the Sobel operators) are applied to it. These filters determine the change in intensity, which approximates the horizontal and vertical derivatives.

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

After these two calls to `apply_mask`, the values obtained are used to search for edges based on the magnitude and direction of the change in intensity. An example of the final result is shown in Figure 5.

You can even see the intermediate results of the X and Y filters individually by trying:

```
./mask1 -i images/cmu.png -m sobelX.txt -o images/cmu-edgeX.png
./mask1 -i images/cmu.png -m sobelY.txt -o images/cmu-edgeY.png
```

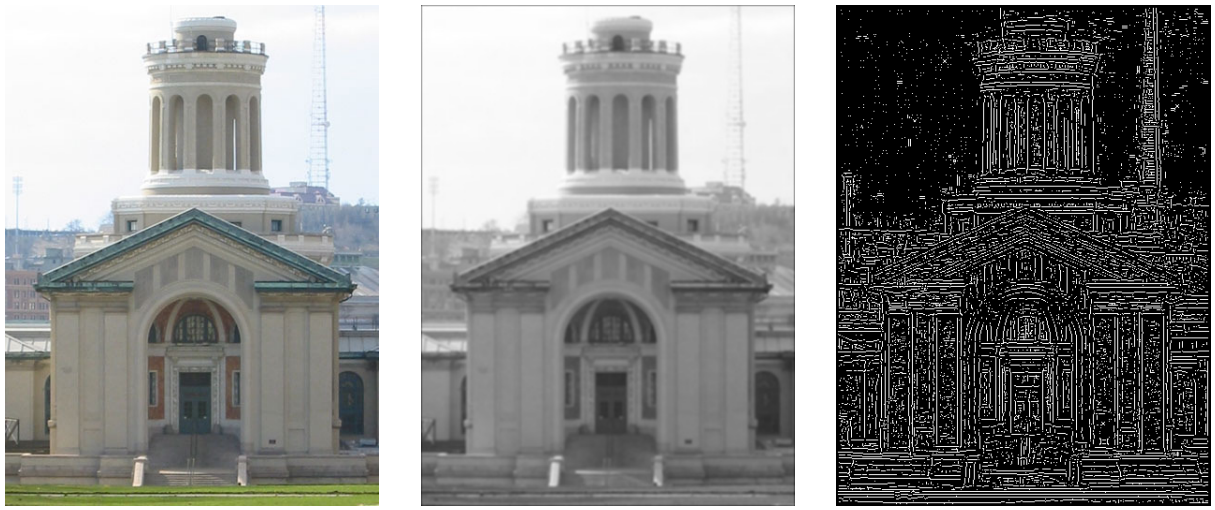

Figure 5: Hammerschlag Hall: original image (left), blurred with the mask (middle), and after running edge detection (right). See text for mask values.

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.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.

[**UPDATED**] If you choose to do this task, be creative! Submissions will be displayed on the Autolab scoreboard and bonus points may be awarded for exemplary submissions. To include your image in the scoreboard, use the `handin` script and include the file `output.png`. We suggest generating `output.png` from one of the sample files, but if your `output.png` file is generated from some other (small!) file, include it as `manipulate.png` with your final submission.

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;
}
```