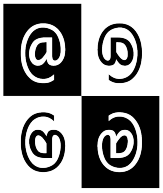
15-122: Principles of Imperative Computation, Spring 2017

Programming Homework 3: Images



Due: Thursday 9th February, 2017 by 9pm

This programming assignment will have you using arrays to represent and manipulate images.

The code handout for this assignment is on Autolab and at

`http://cs.cmu.edu/~15122/hw/images-handout.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a FIVE (5) PENALTY-FREE HANDIN LIMIT, with the idea that for each task you can test your code, hand in, and then fix any bugs found by autolab while working on and testing the next task. Make sure to leave enough submissions to work on the optional Task 5, if you wish to do that. Every additional handin will incur a small (5%) penalty.

Style Grading: With this assignment, we will begin to emphasize *programming style* more heavily. We will actually be looking at your code and evaluating it based on the criteria outlined at `http://cs.cmu.edu/~15122/misc/styleguide.pdf`. We will make comments on your code via Autolab, and will assign an overall passing or failing style grade. A failing style grade will be temporarily represented as a score of -15 points. This -15 will be reset to 0 once you:

1. fix the style issues,
2. see a member of the course staff during office hours **within 5 days** after the grades are released, and
3. briefly discuss the style issues and how they were addressed.

We will evaluate your code for style in two ways. We will use `cc0` with the `-w` flag that gives style warnings — code that raises warnings with this flag is almost certain to fail style grading. Because the `-w` flag does not check for good variable names, appropriate comments, or appropriate use of the functions defined in `pixel.c0` and `imageutil.c0`, these issues will be checked by hand.

Task 1 (3 points) In addition to using good style, be sure to include appropriate contracts, `@requires`, `@ensures`, and `@loop_invariant`. Your annotations should at least be sufficient to ensure that all your array accesses are safe, and part of your grade will be based on a visual inspection of this.

1 Image manipulation

The programming problems you have for this assignment deal with manipulating images. An image will be stored in a one-dimensional array of pixels. (The C0 image library assumes the ARGB implementation of pixels that you wrote last week.) 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.

Task 2 (4 points) Complete the C0 file `imageutil.c0`. As with the `pixel.c0` implementation from last week, you must fill in the missing code and translate the English preconditions and postconditions into `@requires` and `@ensures` statements.

We do not require you to hand in an `images-test.c0` file that tests your *imageutil* implementation the way you tested your *pixel* implementation. It would be a good idea to write one to test your own implementation, however!

2 Image Transformations

The rest of this assignment involves implementing the core part of a series of image transformations. Each function you write will take an array representation of the input image and return an array representation of the output image. These functions should *not* be destructive: you should make your changes in a copy of the array, and not make any changes to the original array. Your implementations should be relatively efficient, meaning both that they should have a reasonable big-O running time and that they should take at most a few seconds to run on our example images.

Remember that your code should have appropriate preconditions and postconditions. It is always a precondition that the given width and height are a valid image size that matches the length of the pixel array passed to the function. It is always a postcondition that the returned array is a different array from the one that was passed in, and that this resulting array has the correct length.

In order to pass style grading, you will be expected to use functions from the *pixel* interface (the type `pixel` and functions `get_red`, `get_green`, `get_blue`, `get_alpha`, and `make_pixel`) and the *imageutil* interface (the functions `is_valid_imagesize`, `get_row`, `get_column`, `is_valid_pixel`, `get_index`) in the next two tasks. On Autolab, we will

Figure 1: A sporty coupe before and after red removal.

compile your code for tasks 2 and 3 against *our* implementation of the *pixel* and the *imageutil* interfaces, so you cannot add new functions to these interfaces.

Testing. You should use the provided `*-main.c0` files to help you test your code. The use of these files is described in the `README.txt` in the code handout.

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.png -j images/my-image.png -o images/diff.png
```

This command compares the image `images/sample.png` and `images/my-image.png` and creates a visual representation of the difference in `images/diff.png`.

2.1 Removing red

As an example of image manipulation, you should take a look at `remove-red.c0`. The core of this transformation is this function:

```
pixel[] remove_red (pixel[] pixels, int width, int height)
```

An example of this transformation is given in Figure 1.

You should look at the file `remove-red.c0` to get an idea of how this transformation works, and you should look at `README.txt` to see how to compile and run this transformation against `remove-red-main.c0`. You are strongly encouraged to write some smaller test cases for your programs. An example of what this should look like is given in `remove-red-test.c0`.

Note that `remove-red.c0` doesn't use the *pixel* or *imageutil* libraries. If *your* code doesn't use the *pixel* or *imageutil* libraries, you will fail style grading! While it is not required, you might want to try your hand at modifying `remove-red.c0` to use the *pixel* and *imageutil* libraries.

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

2.2 Rotation Effect

In this task, you will create a rotation effect on an image. The core of this transformation is this function:

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

The returned array should be the array representation of the duplicated and rotated image. An example of this transformation is given in Figure 2.

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 $(w + h) \times (w + h)$ 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.

The original image must have the same width and height in order to do the "rotation" effect, i.e., $w = h$. If the supplied image is not "square" (i.e., its width does not equal its height) or does not match the size given by the given width and height, your function should abort with a precondition failure when compiled and run with the `-d` flag.

Task 3 (8 points) Create a C0 file `rotate.c0` implementing a function `rotate`. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function.

You should look at `README.txt` to see how to compile and run this transformation against `rotate-main.c0`. You are also strongly encouraged to write some test cases for your programs in `images-test.c0`.

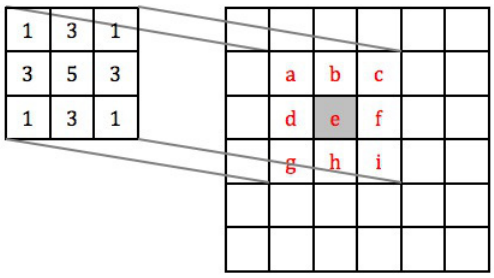


Figure 3: Overlay the 3×3 mask over the image so it is centered on pixel e to compute the new value for pixel e.

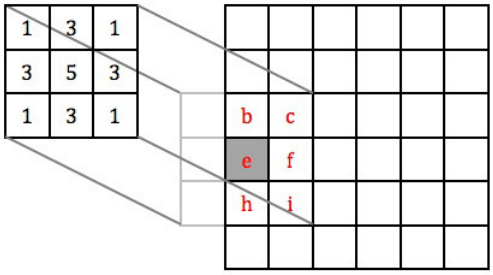


Figure 4: If the mask hangs over the edge of the image, use only those mask values that cover the image in the weighted sum.

2.3 Applying Masks to an Image

In this problem, you will write a function that will apply a "mask" to an image. The core of this transformation is this function:

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

The returned array should contain the results of running the mask computation, a weighted sum, on each pixel in the input. This is an array of *integers*, not an array of pixels; each integer in the returned array corresponds to a pixel in the given image.

Masks In addition to an input image, we pass this transformation a *mask*, an $n \times n$ array of integers representing *weights*. For our purposes, n must be odd. This means that the $n \times n$ array has a well defined center — the *origin*. The weights in the mask can be arbitrary integers — positive, negative, or zero.

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.

For example, refer to Figure 3, which shows a 3×3 mask and an image that we want to perform the mask computation on. Suppose we want to compute the result of the mask computation for pixel `e`. This result would be:

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

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

Instead of doing this calculation for each channel individually, use the average value of the red, green, and blue channels — we ignore the alpha channel. Going back to the example in Figure 3, if the pixels `a` and `e` are both given by $(a, r, g, b) = (255, 107, 9, 217)$, then use $(107 + 9 + 217)/3 = 111$ as the average intensity of those pixels. If every other pixel in that figure is given by $(a, r, g, b) = (15, 200, 120, 100)$ for an average intensity of 140, then index 14 (which corresponds to pixel `e` in Figure 3) of the returned array should store 2766:

$$111+(3\times140)+140+(3\times140)+(5\times111)+(3\times140)+140+(3\times140)+140 = 2766$$

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 stored in index 18 of the returned array, which corresponds to pixel `e`, is given by

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

where `b` is the average intensity of the pixel labeled `b` and so on.

Task 4 (10 points) Create a C0 file `mask.c0` implementing a function `apply_mask`. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function.

You should look at `README.txt` to see how to use this transformation to perform a grayscale blur (`maskblur-main.c0`) and edge detection algorithms (`maskedge-main.c0`). The next page talks a little bit about how these algorithms, especially edge detection, work. You are also strongly encouraged to write some test cases for your programs in file `images-test.c0`.

Applications

The first main function you are given to test your code, `maskblur-main.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 modified appropriately — for the blur masks, the `maskblur-main.c0` program will do division to get an average intensity that is between 0 and 255. Since we're returning just one value instead of one 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, `maskedge-main.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` is called three times. The first call will be to blur the image. For this purpose, the following mask is 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.

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

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:

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

2.4 Your own image processing algorithm (Optional)

In this task, you will perform an image manipulation of your choice. The core of this transformation are three functions:

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

If `I` is the representation of an image with width `w` and height `h`, then the result of calling `manipulate(I,w,h)` should the representation of image of width `result_width(w,h)` and height `result_height(w,h)`.

Task 5 (bonus) Create a C0 file `manipulate.c0` implementing the three functions described above: `result_width`, `result_height`, and `manipulate`. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function. You may not add arguments to `manipulate`, but you can write a separate function `my_manipulate` (or whatever) and then call your function from the `manipulate` function with some specific arguments.

You should look at `README.txt` to see how to compile and run this transformation against `manipulate-main.c0`.

If you choose to do this task, be creative! Submissions will be displayed on the Autolab scoreboard and we will make an effort to highlight exemplary submissions. If you include a (small!) file `manipulate.png`, we'll run your transformation against that image; otherwise we'll run your transformation on `g5.png`.

Figure 6: Manipulate me!