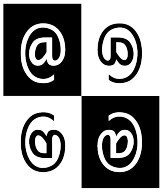
COMP 211: Principles of Imperative Computation, Fall 2014

Programming 2: Images



Due: Friday, Sept 26, 2014 by 11:59pm

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

1 Getting the code

The code handout for this assignment is available from the course web page. The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in.

To download and unzip the code, do the following:

1.1 Mac OS

1. On the assignments page of the course web site, right-click the Programming 2 `Code` link, and then click "Save Link as...", and choose to save it in `Documents/comp211/`.

2. Start a new terminal window and do the following:

   ```
   cd Documents
   cd comp211
   tar -xzvkf hw3-handout.tgz
   ```

   This will make a directory called `Documents/comp211/hw3-handout` and extract all of the code for the assignment.

3. The file `hw3-handout/README.txt` contains a description of the various files, and instructions on running them.

1.2 Windows

1. On the assignments page of the course web site, right-click the Programming 2 `Code` link, and then click "Save Link as...", and choose to save it in your `comp211/`. To find this directory in your web browser, look in

   ```
   C:\cygwin\home\YOURNAME\comp211\
   ```

2. Start a new terminal window and do the following:

```
cd comp211
tar -xzkvf hw3-handout.tgz
```

This will make a directory called `comp211/hw3-handout` and extract all of the code for the assignment.

3. The file `hw3-handout/README.txt` contains a description of the various files, and instructions on running them.

2 Representation of Pixels

An image is represented by many little dots called *pixels*. To represent a single pixel, we need to know two things: how opaque or transparent it is, and what color it is.

One common way to do this is called *ARGB*.[1] The transparency is stored as an integer in the range $[0, 256)$ (this means 0,1,...,255—the 0 is included but the 256 is excluded), where 0 is completely transparent and 255 is completely opaque. This is called the *alpha (A)* value. The color is stored as three other integers, each also in the range $[0, 256)$, which respectively describe the intensity of the *red (R)*, *green (G)*, and *blue (B)* color in the pixel.

In this scheme, a pixel is described by four integers between 0 (inclusive) and 256 (exclusive)—i.e. four 8-bit integers. We could represent this as four values of type `int`. However, since `int`s are 32-bits, and each of the alpha,red,green,blue are only 8 bits, we would be wasting 24 bits of each of the four words. This is bad, because it would make the images take up more space on your hard drive, take longer to send over the internet, and use more of your monthly data limit on your phone.

Since four 8-bit numbers are only 32 bits total, we can be a little bit clever and *pack* the four numbers into a single 32-bit word. This means we will use *one* `int` to represent *four different numbers*, which uses only one quarter of the bits of the simple solution proposed above[2].

Thus, we will think of the bits of a single 32-bit word as describing the alpha, red, green, and blue, by breaking it 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). Note that the alpha is the leftmost

[1] `http://en.wikipedia.org/wiki/RGBA_color_space`

[2] Caveat: images are "compressed" by other means when they are stored or transmitted, so it wouldn't actually take four times as much space to have all the 0s, but there is still some savings

(most significant) bits of the pixel, and the blue is the rightmost (least significant) bits of the pixel. For example, the pixel $(alpha = 255, red = 0, green = 0, blue = 128)$ would be represented in binary by

$$(alpha = 11111111, red = 00000000, green = 00000000, blue = 10000000)$$

Putting these together into a single word results in

11111111000000000000000010000000

This word can also be written in hexadecimal as

`0xFF000080`

Note that the decimal values of these words is pretty meaningless; e.g. the above pixel happens to be the decimal integer -16777088, but you will want to use hex and bitwise operations.

Figure 1 shows the color of a bunch of different pixels. Make sure you understand why each hex word gives the corresponding color.

In the file `pixel.c0`, right at the top we announce that we will be working with a type `pixel` that is actually represented as a single integer by writing a *type definition*:

```
typedef int pixel;
```

You may find it helpful to use the function `printhex` (defined in `pixel.c0`) for testing; this function prints an `int` in hexadecimal.

3 Image manipulation

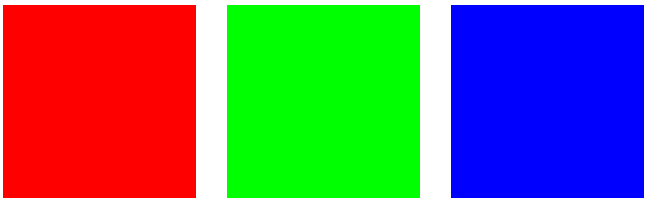
The programming problems you have for this assignment deal with manipulating images. An image will be stored in an array of pixels. Pixels are stored in the array row by row, left to right starting at the top left of the image. For example, if a 5×5 image has the following pixel "values":

	col 0	*col* 1	*col* 2	*col* 3	*col* 4
row 0	a	b	c	d	e
row 1	f	g	h	i	j
row 2	k	l	m	n	o
row 3	p	q	r	s	t
row 4	u	v	w	x	y

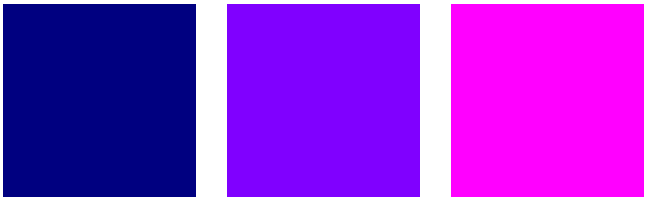
then these values would be stored in the array in the following positions:

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
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

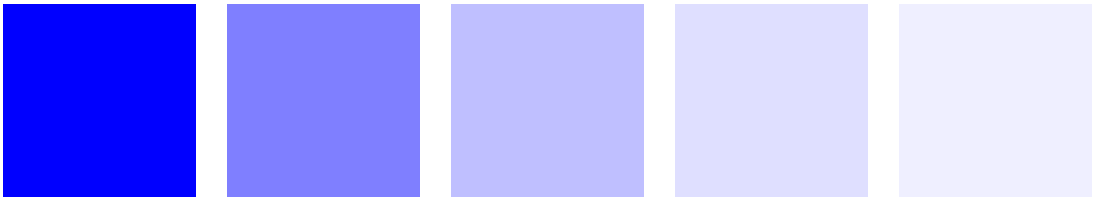
In the 5×5 image, the pixel i is in row 1, column 3 (rows and columns are indexed starting with 0) but is stored in the array at index 8. An image must have at least one pixel.



0xFFFF0000 0xFF00FF00 0xFF0000FF



0xFF000080 0xFF8000FF 0xFFFF00FF



0xFF0000FF 0x800000FF 0x400000FF 0x200000FF 0x100000FF

Figure 1: Colors by Pixel Value

Task 1 (0pts) *Given a row number i and the column number j, express the position in the array of the pixel at (i, j) in terms of i and j.*

Task 2 (0pts) *Given a position in the array, express the row and column numbers in terms of that position.*

Task 3 (0pts) *Generalize your above answers to an image with dimensions* `width` *and* `height` *that can be different than 5.*

You do not need to hand anything in for these tasks; they will help you with the code below.

4 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 create a new pixel array with different contents, and not make any changes to the original array.

Task 4 (2pts) *You should write* `//@requires` *and* `//@ensures` *annotations on every function you write in this assignment:*

- *Each function should have a precondition that the given width and height are positive and match the length of the pixel array passed to the function.*
- *Each function should have a postcondition that the result array has the correct length for the specific transformation.*
- *Some of the tasks have additional specifications that are specific to them.*

Testing. The easiest way to test these tasks will be to try the image transformations on specific images and look at the results: it is fairly easy to see the correctness of these transformations visually. You can open the image files by going to

1. Mac: `Documents/comp211/hw3-handout/images`
2. Windows: `C:\cygwin\home\YOURNAME\comp211\images`

in Finder/Windows explorer and double-clicking them.

You can also use the commands

1. Mac: `open images/gargoyle.png`
2. Windows: `cygstart images/gargoyle.png`

to open the image named `gargoyle.png`.

Figure 2: A gargoyle before and after red removal.

4.1 Removing red

Task 5 (3pts) *As a first example of image manipulation, you will implement a transformation that removes the red component of each pixel in an image, in the file* `remove-red.c0`.

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

An example of this transformation is given in Figure 2.

The variables `width` and `height` are the dimensions of the input image (you should add a `@requires` that states this.)

Hint: write a loop over all of the pixels in the image, and use word operations (at least one of `&` , `|`, `<<`, `>>`) to compute the new color value for each pixel.

See the `README.txt` file for how to test `remove_red` on an image. In particular, we will be using a program called `cc0` instead of `coin`. `cc0` *compiles* the code you give it, producing another new program that you can run in the Unix shell.

You basic workflow should be:

1. Edit your C0 files as usual in sublime/emacs.

2. After you make changes, use `cc0` to compile your code. You can do this by typing

```
$ cc0 -d -o transform pixel.c0 remove-red.c0 quantize.c0 rotate90.c0 rotate.c0 manipulate.c0 images-main.c0
```

   into the terminal. This says "make the files `pixel.c0`, `remove-red.c0`, ... into a new program named `transform`". However, you don't need to type that every time; there is a shorthand of just typing

```
$ make
```

which does the same thing.

3. To run remove red on a file `images/gargoyle.png`, type the following into the Unix shell, from your `hw3-handout` directory:

   ```
   $ ./transform -t remove_red -i images/gargoyle.png
   ```

   This says "run the `transform` program that I produced in step 2, in particular with the remove red transform, on the indicated imaged file."

4.2 Quantization of an Image

In this task, you will implement *quantization*, a transformation that can be used to reduce the amount of space it takes to store an image. The idea with quantization is that you reduce the total number of colors used in the image, by turning some bits in each pixel into 0 (the way images are stored on disk then represents these pixels using less space).

Given a pixel and a quantization level q in the range $[0, 8)$, we quantize by taking each color component (red, green and blue) and clearing the lowest q bits (setting the q leftmost bits to 0). For example, suppose we have a pixel with red intensity $R = 107$, green intensity $G = 190$, and blue intensity $B = 215$. The color components of this pixel are represented by these 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
```

A pixel processed with a quantization level of 0 should not change. For each pixel, do not change its alpha component.

You can see an example in Figure 3.

Task 6 (4pts) *In the file* `quantize.c0`*, implement the function*

```
pixel quantize_pixel(pixel p, int q)
//@requires 0 <= q && q < 8;
```

that quantizes the given pixel `p` *by* `q` *bits.*

You can test this function in `coin` by running the function `test_quantize()`, which tries your quantize function on a few examples:

Figure 3: A gargoyle original image (left) and quantized at level 6 (right).

```
$ coin -d pixel.c0 quantize.c0
C0 interpreter (coin) 0.3.2 'Nickel' (r349, Wed Aug 28 18:31:41 EDT 2013)
Type '#help' for help or '#quit' to exit.
--> test_quantize();
```

Task 7 (2pts) *Write a function*

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

Given an array ***pixels*** *representing an image of width* ***width*** *and height* ***height****, this function should create a new pixel array where each pixel in it is the quantization of the corresponding position in* `pixels` *by* `q`. Use your `quantize_pixel` function from the previous task.

In addition to the preconditions for all of these functions, `quantize` should `require` that `q` is a number in the range $[0, 8)$.

See the `README.txt` file for how to test quantize on an image.

Task 8 (1pts) *Run the transformations* `quantize1` *through* `quantize7` *on* `gargoyle.png`.

Make a table listing the file sizes of each image. You can do this in the shell by running the following command:

```
ls -lh images/gargoyle_quantize*.png
```

from your `hw3-handout` *directory. When you see a line*

```
-rw------- 1 dlicata 410922006 2.3M Sep 17 17:50 images/gargoyle.png
```

the `2.3M` *is the size of the file* `images/gargoyle.png` *(2.3 megabytes, or 2.3 million bytes).*

At what quantization level can you notice a difference in the image?

Figure 4: Gargoyle before and after rotate 90.

4.3 Rotate90

In this task, you will implement a "clockwise rotation by 90 degrees" transform, as depicted in Figure 4. This transformation assumes that the image is square—that its width is equal to its height.

For each pixel p' in the new image, think about the coordinates of the pixel p in the old image goes in that spot. It may help to work through a small example by hand, like if the original image had the following 9 pixels:

$$\begin{matrix} a & b & c \\ d & e & f \\ g & h & i \end{matrix}$$

Hint: you may wish to use a nested `for` loop like this:

```
for (int row = 0; row < height; row = row + 1)
  {
    for (int col = 0; col < width; col = col + 1)
      {
        ...
      }
  }
```

to loop over all of the rows and columns in the image.

Task 9 (3pts) *Edit the file* `rotate90.c0` *and implement the function* `rotate90`*. In addition to the requirements for all transformations, you should add a* `@requires` *that states the input is square.*

You should look at `README.txt` to see how to compile and run this transformation.

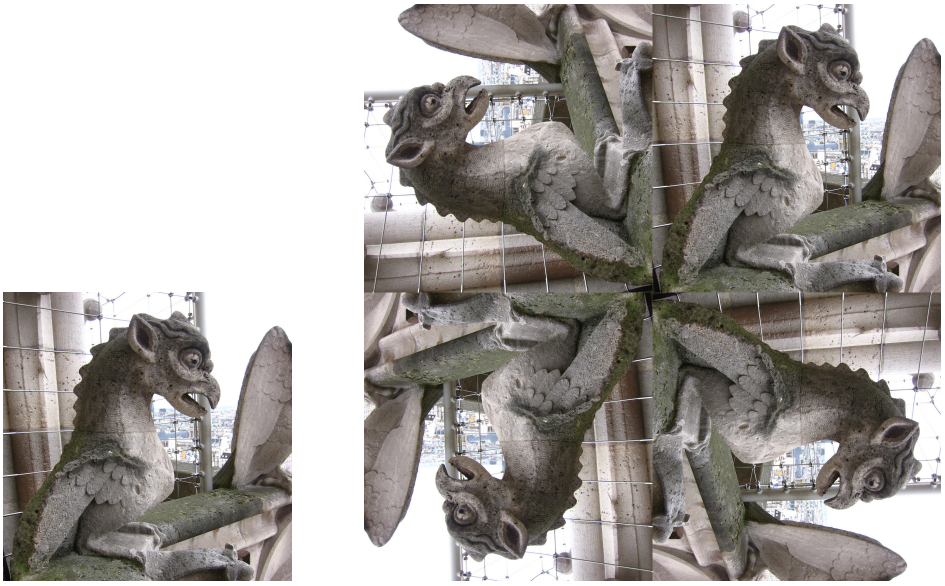


Figure 5: Original image (left); Image after "rotation effect"

4.4 Rotation Effect

In this task, you will create a more complicated 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 rotated image. An example of this transformation is given in Figure 5.

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 be square the same width and height in order to do the "rotation" effect; you should include a `requires` stating this.

Task 10 (5pts) *Edit the file* `rotate.c0` *and implement a function* `rotate`*.*

You should look at `README.txt` to see how to compile and run this transformation.

Task 11 (1pt) *Download a* `PNG` *of your choice (make sure it is square, or use a photo editing program to crop it), then run* `rotate` *on it, and post the results in the Rotate Thread on Piazza.*

4.5 Bonus Task

For extra credit, write some other image transformation of your choice. Put your code in the file `manipulate.c0`, and post an example in the Bonus Task thread on Piazza. See the

`README.txt` file for how to run manipulate. Note that if your transformation changes the width and height of the image, you will need to edit the functions `expected_result_width` and `expected_result_height` in `images-main.c0`.