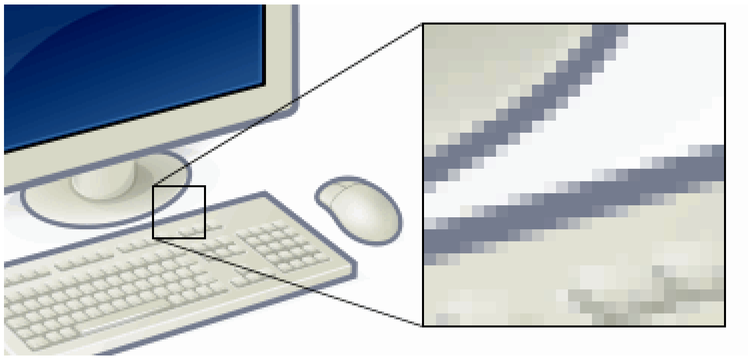
Homework 1: Image Processing

Due Wednesday Sep. 14 at 11:59 pm

Introduction

The goal of this assignment is to introduce you to the basic features of JavaScript, such as functions, variables, conditionals, and loops. You should already be familiar with these concepts from other programming languages (e.g., Java). You will use these features to write several *image processing* functions (e.g., changing image colors). You will also work with higher-order functions and should be able to see first-hand how, once developed, they facilitate code development and reuse.

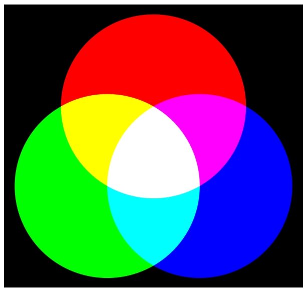
Colors, Pixels, and Images



Any image you see on your computer screen consists of tiny dots known as *pixels*. On many screens, individual pixels are too small to see with the naked eye, but if you look very closely, you may be able to discern the pattern in which pixels are arranged, as illustrated in the figure to the right.

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,2)
(0,2)	(1,2)	(2,2)

Pixels are arranged in a grid and each pixel has *x- and y- coordinates* that identify its position in the grid. All coordinates are non-negative integers and the top-left corner has the coordinates (0, 0). Therefore, the x-coordinate increases as you move right and the y-coordinate increases as you move down.[1] For example, if we had an image with just nine pixels arranged in a 3-by-3 grid, their coordinates would appear as shown on the left.



Finally, every pixel has a color that is represented using the three *primary colors* (i.e., red, green, and blue). Therefore, to set the color of a pixel, you have to specify how much red, green, and blue to use. Each of these primary colors has an intensity between 0.0 and 1.0. For example, to get a black pixel, we can set the intensity of the three primary colors to 0.0, thus the black pixel value is [0., 0., 0.], and to get a white pixel we can set the intensity of the three primary colors to 1.0, thus the white color pixel value is [1., 1., 1.].

In this project, you will learn to use a very simple library of image manipulation functions that let you load images from the web, set the color of each pixel and read the color of each pixel. These are the only primitive functions you need to build sophisticated image processing functions yourself.

[1] Note that this is not the same as coordinates in a graph from math classes, where the y-coordinate *decreases* as we move down. In contrast, on the screen, the y-coordinate *increases* as we move down.

Image Manipulation

Our JavaScript programming environment includes a simple image manipulation library that has just a handful of functions. You can load an image from a URL using the `lib220.loadImageFromURL` function. For example, the following statement loads an image:

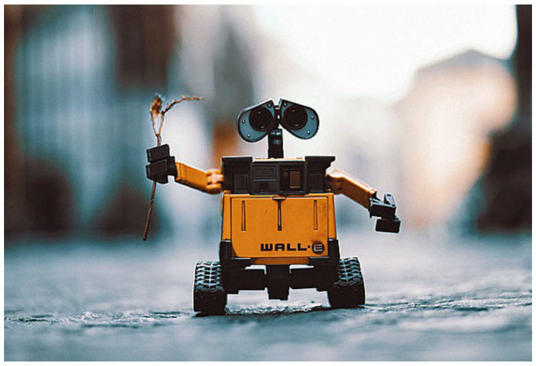
```
let robot =
lib220.loadImageFromURL('https://people.cs.umass.edu/~joydeepb/robot.jpg');
```

Note that loading an image does not show the image. If you enter `robot` at the console prompt, you'll see that the image is an object with several fields and methods:

```
>> robot
<< {width: 600, height: 400, show: function show(), setPixel: function setPixel(),
 getPixel: function getPixel()}
```

We can use the method `robot.show()` to actually see the image on screen:

```
robot.show()
```

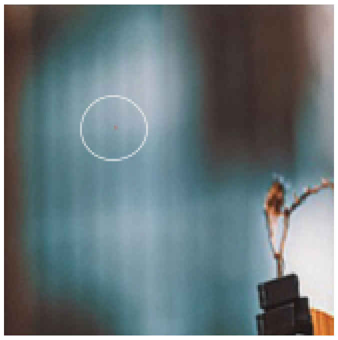


Images have two methods that allow you to get and set the color of individual pixels. The method `.getPixel(x, y)` takes the *x*- and *y*- coordinates of a pixel as arguments and returns an array of three numbers, which contain the intensity of red, green, and blue in that pixel. For example, in this image, the pixel at coordinate (72, 72) has the color:

```
>> robot.getPixel(72, 72)
<< [0.5529411764705883,
    0.6431372549019608,
    0.6980392156862745]
```

We can set the color of a pixel with the method `.setPixel(x, y, color)`:

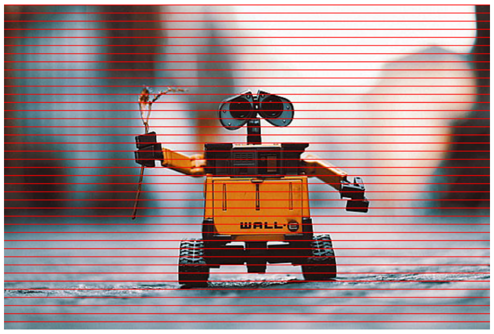
```
>> robot.setPixel(72, 72, [1.0, 0.0, 0.0])
```



How ever, this does not update the visible image. To see an image after a change, invoke the `.show()` method again. This will show the changed image and a small red dot will be visible next to the robot's arm. (You may need to zoom in considerably.)

By repeatedly using `.setPixel` and `.getPixel`, we can manipulate the image in a variety of ways. For example, the following program draws a series of red, horizontal lines that are ten pixels apart:

```
let robot =
lib220.loadImageFromURL('https://people.cs.umass.edu/~joydeepb/robot.jpg');
for (let i = 0; i < robot.width; ++i) {
  for (let j = 0; j < robot.height; j = j + 10) {
    robot.setPixel(i, j, [1.0, 0.0, 0.0]);
  }
}
robot.show();
```



Finally, images have a `.copy()` method that creates a new copy of the image. It is useful as a starting point for creating a new, modified version of an image without changing the original. This also lets us write a test case to check that the output image is different from the input image.

Programming Task

In this assignment, you will write functions that apply several effects to an image. Specifically:

1. Write a function called `removeRed` that takes an image as an argument and returns a new image, where each pixel has the red color channel removed. If the color of a pixel is $(r,\ g,\ b)$ in the input image, its color in the output must be $(0,\ g,\ b)$.
2. Write a function called `flipColors` that takes an image as an argument and returns a new image, where each pixel has each color channel set to the average of the other two channels in the original pixel.

If you have solved these two tasks, you might notice that the structure of the two functions is very similar, the difference is only in the actual processing applied. We can avoid duplication by defining functions, similar to `map`, that apply the same transformation to several or all pixels of an image.

3. Write a function called `mapLine` with the following type:
   `mapLine(img: Image, lineNo: number, func: (p: Pixel) => Pixel): void`
   The function should modify the given image in place, so that the value of each pixel in the given line is the result of applying `func` to the corresponding pixel of `img`. It does not return any value. If `lineNo` is not a valid line number, the image should not be modified.
4. Write a function called `imageMap` with the following type:
   `imageMap(img: Image, func: (p: Pixel) => Pixel): Image`
   The result must be a new image with the same dimensions as `img`. The value of each pixel in the new image should be the result of applying `func` to the corresponding pixel of `img`. Use `mapLine`.
5. Write two functions `mapToGB` and `mapFlipColors` that are equivalent to (i.e., have the same type signature and behave exactly like) `removeRed` and `flipColors` but use `imageMap`.

Testing Your Code

An important part of this project is testing your code thoroughly. Without appropriate unit tests, you may not catch bugs in your code, and hence will score poorly. To help you get started, we have provided a few test cases here. It is up to you to define additional tests to check your solution for correctness.

- Check to ensure that the function definition is correct. If it is correct, then this test will run without producing any errors. For example, there are no assertions in the following test since it does not test the output -- rather, it just confirms that the code runs to completion without run-time errors.

```
test('removeRed function definition is correct', function() {
  const white = lib220.createImage(10, 10, [1,1,1]);
  removeRed(white).getPixel(0,0);
  // Checks that code runs. Need to use assert to check properties.
});
```

- Check to ensure that the center pixel in the image returned by `removeRed` has no red channel values.

```
test('No red in removeRed result', function() {
  // Create a test image, of size 10 pixels x 10 pixels, and set it to all white.
  const white = lib220.createImage(10, 10, [1,1,1]);
  // Get the result of the function.
  const shouldBeBG = removeRed(white);
  // Read the center pixel.
  const pixelValue = shouldBeBG.getPixel(5, 5);
  // The red channel should be 0.
  assert(pixelValue[0] === 0);
  // The green channel should be unchanged.
  assert(pixelValue[1] === 1);
  // The blue channel should be unchanged.
  assert(pixelValue[2] === 1);
});
```

  - Your unit tests will need to check pixel values. However, in general, when dealing with fractional values, direct comparisons will not work, due to the limited precision of the internal representation of image pixel values. Internally, channel values are stored as 8-bit integers from 0 to 255. A value from [0, 1] is internally mapped to the closest rational number of the form k/255, which can differ by up to 1/510 < 0.002 in actual value. To overcome this issue, you should check if a value is within a small threshold (epsilon) of the expected one. The following unit test shows how to do this.

```
function pixelEq (p1, p2) {
  const epsilon = 0.002;
  for (let i = 0; i < 3; ++i) {
    if (Math.abs(p1[i] - p2[i]) > epsilon) {
      return false;
    }
  }
  return true;
};

test('Check pixel equality', function() {
  const inputPixel = [0.5, 0.5, 0.5]
  // Create a test image, of size 10 pixels x 10 pixels, and set it to the inputPixel
  const image = lib220.createImage(10, 10, inputPixel);

  // Process the image.
  const outputImage = removeRed(image);

  // Check the center pixel.
  const centerPixel = outputImage.getPixel(5, 5);
  assert(pixelEq(centerPixel, [0, 0.5, 0.5]));

  // Check the top-left corner pixel.
  const cornerPixel = outputImage.getPixel(0, 0);
  assert(pixelEq(cornerPixel, [0, 0.5, 0.5]));
});
```