Processing and Images

Processing provides the `PImage` class to abstract away the details of working with the four supported image formats.

`PImage` provides methods that implement many of the image manipulation algorithms we’ll be studying. This means we’ll have something to test against.

Still, our main focus will be on implementing these algorithms ourselves.

Of course Processing provides methods that provide functionality beyond what we’ll do, so play around and see what they do.

The `PImage` pixel array

`PImage` includes a `public` instance variable `pixels` defined as a one-dimensional array of type `color: color pixels[]`

This array can be quite large!

For example, an image which has width 640 and height 320 is made up of 204,800 pixels, so an algorithm which processes each pixel in the array must perform its operations 204,800 times.

Image processing can be slow...

Modifying the pixel array

Many of the algorithms we’ll study modify the `pixels` array directly. There is a three step process to this:

- Call the `loadPixels` method to initialize the array
- Use a loop (typically) to access the values in each pixel, and apply some modification.
- Call the `updatePixels` method to copy the newly modified `pixels` array to the image.
Example: modifying pixels

In this example, we change a single pixel to its *inverse*:

```java
PImage img = loadImage("myImage.jpg");
img.loadPixels();
color c = img.pixels[0]; // copy the color of the upper-leftmost pixel
img.pixels[0] = color(255-red(c), 255-green(c), 255-blue(c));
img.updatePixels(); // changes now part of image
```

Note: updatePixels does not display the new image...use the `image` function for that.

Algorithm: Making a Negative

Our first image processing algorithm constructs the negative, or *inverse*, of an entire image:

- White (255,255,255) becomes black (0,0,0)
- Black (0,0,0) becomes white (255, 255, 255)
- Each color component becomes its inverse in the range 0-255

To do this, apply the single-pixel inverse calculation to the entire image. We'll look at three different techniques, representing different tradeoffs.

Access pixel array directly

For this and many other algorithms, we can use a loop to visit every value in the array from start to end. Access pixel data directly just like any other array. For example:

```java
img.loadPixels();
for (int i = 0; i < img.pixels.length; i++) {
    color c = img.pixels[i];
    c = color(255-red(c), 255-green(c), 255-blue(c));
    img.pixels[i] = c;
}
img.updatePixels();
```

Access pixel array with nested loops

It is natural to want to treat the pixel array as being two-dimensional, since the image is. In this alternative direct access method, we use visit the pixel at each (row, column) pair using nested loops.

For indices $x$ and $y$ where $(x,y)$ is the coordinate of some pixel, the following formula will give us the index into the singly-dimensioned pixel array:

```java
color c = pixels[y*img.width + x];
```

This method is not ideal for computing the inverse, but it works fine and is natural for algorithms we’ll study later.
Nested loops example

Computing the inverse using nested loops with the coordinate equation:

```java
img.loadPixels();
for (int y = 0; y < img.height; y++) { // each row
    for (int x = 0; x < img.width; x++) { // each column
        int i = y*img.width + x; // compute index
        color c = pixels[i];
        c = color(255-red(c), 255-green(c), 255-blue(c));
        img.pixels[i] = c;
    }
}
img.updatePixels();
```

Using methods to access pixels

Another way to access pixel values is using the `PImage` methods `get()` and `set()` with the row and column coordinates.

```java
img.loadPixels();
for (int y = 0; y < img.height; y++) { // each row
    for (int x = 0; x < img.width; x++) { // each column
        color c = img.get(x, y);
        c = color(255-red(c), 255-green(c), 255-blue(c));
        img.set(x, y, c);
    }
}
img.updatePixels();
```

Algorithm: Color to Grayscale

For any color `c` a gray value occurs when the following is true:

\[ \text{red}(c) == \text{blue}(c) \&\& \text{blue}(c) == \text{green}(c) \]

The intensity of the gray color is given by the average of the RGB values. For a single pixel:

```java
color c = img.pixels[0];
float r = red(c); float g = green(c); float b = blue(c);
float avg = (r + g + b) / 3.0;
img.pixels[0] = color(avg, avg, avg);
```

Grayscale conversion using nested loops

```java
img.loadPixels();
for (int y = 0; y < img.height; y++) { // each row
    for (int x = 0; x < img.width; x++) { // each column
        int i = y*img.width + x; // compute index
        color c = pixels[i];
        float r = red(c); float g = green(c); float b = blue(c);
        float avg = (r + g + b) / 3.0;
        img.pixels[i] = color(avg, avg, avg);
    }
}
img.updatePixels();
```