CS 116x Winter 2015
Craig S. Kaplan

Module 08
Image Processing

Topics

- High-level image operations: cropping, resizing, rotating
- Reading individual pixels, halftoning
- Processing individual pixels, colour manipulations
- Processing pixel neighbourhoods
- Working with webcams

Readings

- Shiffman's online notes about Images and Pixels
- *Getting Started*, Pages 78–82
- *Learning Processing*, Chapter 15

Introduction

Last term you learned a little bit about how to load images and examine their contents, though it was nearly at the end of the course and mostly for demonstration purposes. I'd like to spend a bit more time on the subject of image processing, and get into some detail on the programming techniques that underlie typical image manipulations you'd do every day in tools like Photoshop.

Full disclosure: most of the time a photo editing tool is a perfectly fine way to use techniques like these, without having to write a line of code. But once in a while it's really useful to express image transformations as small programs. This is especially true when you want to do the same thing to a large set of photos, or where your image manipulation

depends upon some numerical feature of an image such as its dimensions.

Cropping images

Let's assume that the goal is to write a sketch that reads an image from a file, performs some transformation on it, and writes the resulting image to a new file. If the transformation of the day is simply to crop an image, you've already got all the know-how you need to make this work. The key is to realize that the built-in `save()` function writes whatever happens to be in the sketch window out to a file. To crop, then, just create a sketch window whose size is the desired cropped image size, then use the `image()` function to draw the source image so the part you want is visible within the window.

```
void setup()
{
  PImage img = loadImage( "otis.jpg" );
  size( 200, 200 );
  image( img, -100, -300 );
  save( "otis_cropped.jpg" );
  exit();
}
```

This is a bit silly. It relies on the interaction between the sketch window and the image, and forces you think about moving the image opposite to how you want to crop. Plus, it's no good if you're actually using the sketch window to do something else. It would be more sensible to be able to ask an image to give you a block of pixels. And `PImage` has a method that does exactly that. It's a four-parameter version of the `get()` method that takes parameters similar to `rect()` (x, y, width, height) and returns a new `PImage` containing the extracted rectangle from the original image.

```
void setup()
{
  PImage img = loadImage( "otis.jpg" );
  PImage cr = img.get(
    100, 300, 200, 200 );
  cr.save( "otis_cropped_2.jpg" );
  exit();
}
```

Of course, if we were writing a more fully featured sketch, we might consider allowing the user to drag out a rectangle interactively (using direct manipulation) that lets them define where the crop will take place.

Example sketch: LiveCrop

The ability to extract portions of an image can lead to some interesting ideas for sketches, such one which an image is broken into a grid of tiles and tiles swap with each other randomly. A more refined version of the same underlying idea would be a simulation of a classic Sliding 15 puzzle.

Example sketch: Swap

Example sketch: Sliding15

Scaling images

As with cropping, there's a means of scaling images that we've seen before. The built-in function `image()` has a five-parameter version that takes as input not just the image and where to place it, but also the

dimensions of a rectangle. The image will be scaled to fit that rectangle.

```
void setup()
{
  PImage img = loadImage( "otis.jpg" );
  size( 400, 200 );
  image( img, 0, 0, width, height );
  save( "otis_scaled.jpg" );
  exit();
}
```

But once again, this is a convoluted way to scale an image. Fortunately, Processing gives us a built-in method of `PImage` that does the same thing, and more. The resize() method will change an image to have the dimensions passed in as parameters. Note that it overwrites the unscaled image; if you need both the original and rescaled versions, make a copy first. (If you just need the scaled copy, you can just overwrite the original and skip a few lines of code.)

```
void setup()
{
  PImage img = loadImage( "otis.jpg" );
  // This version of get() just
  // clones an image.
  PImage img_copy = img.get();
  img_copy.resize( 400, 200 );
  img_copy.save( "otis_scaled.jpg" );
  exit();
}
```

Rotating images

When it comes to general image rotation, I really don't know of a built-in technique that doesn't just use generic geometric context functions. If you want to rotate an image by some arbitrary amount, draw it in a

geometric context that's rotated and then take a snapshot:

```
void setup()
{
  PImage img = loadImage( "otis.jpg" );
  size( 200, 200 );
  rotate( 0.25*PI );
  image( img, -200, -300 );
  save( "otis_rotated_cropped.jpg" );
  exit();
}
```

It would be natural at this point to question the *quality* of these rotated images. Drawing a rotated image is a difficult problem, and many algorithms for it do a poor job. We don't know what sort of filter Processing is using, and whether the quality will be comparable to Photoshop or other image editing software. For the purposes of this course, we'll accept the output Processing gives us and move on.

There are exactly three special cases of rotation that we can attack directly: rotations by 90, 180, and 270 degrees. These can be done by nested loops over arrays, because rotations by these amounts map the pixel grid right back onto itself. We can trust these rotations, because they're perfectly precise.

Example sketch: Rotate90

Image pixels

Processing gives us two ways to access an image pixel by pixel. The first, and probably easiest, is to use the two-parameter `get()` method of a `PImage`. That method takes the x and y coordinates of a pixel as

input, and returns a `color` telling you what's at that pixel.

```
void setup()
{
  PImage img = loadImage( "otis.jpg" );
  int r = 0;
  int g = 0;
  int b = 0;
  float sz = img.width * img.height;

  for( int y = 0; y < img.height; ++y ) {
    for( int x = 0; x < img.width; ++x ) {
      color c = img.get( x, y );
      r += red( c );
      g += green( c );
      b += blue( c );
    }
  }

  color avg = color(
    r/sz, g/sz, b/sz );
  background( avg );
}
```

The `get()` method works, but it can be slow. It's best to reserve it for cases where you're accessing only a few pixels at a time. For example, it's fairly easy to write a sketch that gives you an "eyedropper", a visualization of the colour of every pixel in a larger image.

Example sketch: PerPixel

If you know you'll be walking over all the pixels in an image systematically, it's better to use the image's `pixels[]` array. The trick is that you must then do a bit of extra work to make sure the array is valid, in the form of `loadPixels()`:

```
void setup()
{
  PImage img = loadImage( "otis.jpg" );
  int r = 0;
  int g = 0;
  int b = 0;
  int sz = img.width * img.height;
  img.loadPixels();

  for ( int idx = 0; idx < sz; ++idx ) {
    color c = img.pixels[idx];
    r += red( c );
    g += green( c );
    b += blue( c );
  }

  color avg = color(
  r/float(sz),
  g/float(sz),
  b/float(sz) );
  background( avg );
}
```

The array is *declared* in `PImage`, but Processing doesn't guarantee that it's actually storing the image data in that array—it might be using some hidden resource in the computer that's more efficient. Calling `loadPixels()` forces Processing to synchronize the `pixels[]` array with whatever internal representation it's using of the image. If you also *modify* the pixels (the example above doesn't), then you should call `img.updatePixels()` after all processing is finished in order to force Processing to "remember" your changes by synchronizing back with its internal representation of the image.

It's possible to use image content to drive all sorts of interesting and compelling visualization processes. We've already played with a few this term, including Unknown Pleasures in Assignment 1 and Impressionist in Assignment 2. We can expect that most of these techniques will involve reading every pixel in an image and doing something with the colour. We can use the following code as a template. Here I've moved pixel-level processing to the `draw()` function, in case we want to revise this later to do something different each frame.

```
PImage img;

void setup()
{
  img = loadImage( "otis.jpg" );
}

void draw()
{
  img.loadPixels();

  for( int y = 0; y < img.height; ++y ) {
    for( int x = 0; x < img.width; ++x ) {
      color c = img.pixels[ y*width+x ];
      // OK, we've got the colour of one
      // pixel, now do something with it.
    }
  }

  noLoop();
}
```

My favourite examples of pixel-level visualizations tend to be based on *halftoning*: the representation of continuous tone (i.e., greyscale values) using only black on white. It's pretty easy to scale an input image down to a small size, and then represent each of its pixels using a suitably sized object (say, a square or circle) in a grid cell. There's one small trick required to get this right: as an object gets bigger, its area goes up with the *square* of its size. You need to compensate for that in order to get "pixels" that cover the correct proportions of their cells.

Example sketch: Halftoning

There are many situations in which we want to process every pixel in isolation, turning an old image into a new one of the same resolution. Typical examples include converting an image to greyscale, adjusting the brightness or contrast, or any of a thousand Instagram-like colour space effects. For most of these effects, we can use the same general structure for the code, and focus our attention on

designing a helper function that transforms an input colour into an output colour.

```
// See the rest of the TransformColour sketch
// for the code that uses this function.

color processColour( color c )
{
  // The "trivial" colour processing function:
  // just return the input colour.
  return c;
}
```

For example, we can use the built-in brightness() function to convert a colour image to greyscale:

```
color processColour( color c )
{
  float b = brightness( c );
  return color( b );
}
```

Ah, but how is brightness actually computed? For the record, you can't just use the amount of red, green or blue, or even the average. The problem is that different primary colours have different perceived brightnesses: full green contributes more to brightness than pure blue. The usual formula for brightness takes these differences into account:

```
color processColour( color c )
{
  float b = 0.2126*red(c)
    + 0.7152*green(c)
    + 0.0722*blue(c);
  return color( b );
}
```

It's possible to write all kinds of other alternatives to `processColour()` that shift hue around, adjust saturation or brightness, map between different colour ranges, perform gamma correction, and so on. Many colour space transformations can be expressed using a "weighted sum" like the one above, in which

each of the outgoing R, G, and B is a combination of the incoming R, G, and B.

```
// Define nine "weights" that control the
// relative amounts of input R, G, and B
// mixed in to the output R, G and B.

float[] weights = {
   0.6, 0.3, 0.1,
   0.0, 0.0, 1.0,
   0.9, 0.0, 0.1 };

color processColour( color c )
{
  float r = red(c);
  float g = green(c);
  float b = blue(c);

  return color(
    weights[0]*r + weights[1]*g + weights[2]*b,
    weights[3]*r + weights[4]*g + weights[5]*b,
    weights[6]*r + weights[7]*g + weights[8]*b );
}
```

A lot of specific transformations are special cases of this one, in which you don't need *quite* as much math.

In some image transformations, we combine a pixel's colour content with its *location* (i.e, its coordinates) to produce novel effects. For example, in "vignetting", we multiply a pixel's brightness by some function of that pixel's distance from the centre of the image. That way, pixels can retain their original brightness near the centre of the image, but the brightness fades towards the periphery, simulating the imperfect light transport of old lenses. You can imagine modifying the `processColour()` function so that it takes three parameters instead of just one: the colour itself, and the x and y coordinates of the pixel being processed.

Combining images

Some operations involve combining two or more images (say, all of the same resolution) into a single result. These sorts of image operations can easily be seen in the form of the Layer Blend Modes in Photoshop—each mode is a way to combine a layer with the layer underneath it.

It isn't too hard to, say, multiply the contents of two images together pixel-by-pixel by walking over the two arrays in sync. But let's skip the question of how to code that ourselves from scratch, and note that Processing has a lot of these operations built in: have a look at the `blend()` method of `PImage`.

Neighbourhoods of pixels

A final class of operations I'll mention are the ones in which the value of every pixel in the result image depends on a *neighbourhood* of pixels in the initial image. A classic example is blurring: every pixel in the result is a weighted average of that pixel with its neighbours out to some distance. Conceptually, this isn't too much harder to implement in code than the other transformations we've seen so far. But there are annoying cases that require a fair amount of additional coding. Most obviously, these algorithms often try to read pixels from off the edge of the image. We need a "policy" for what to do in such cases. There's no right answer, but a few standard options. We can assume all off-image pixels are black, or that they're copies of the pixels at the edge of the image, or that the image wraps around, etc.

Example sketch: ManualBlur

Blurring is an important enough operation that it's also built in to Processing. `PImage` supports a `filter()` method that can apply one of a few different useful image processing filters, the most important of which is almost certainly Gaussian Blur.

Example sketch: FilterBlur

As with other functions we've seen dealing with pixels, the methods inside of `PImage` are mostly also available at the global level, in which case they operate directly on the sketch window itself. This can be used to produce some interesting visual effects, like a ghostly tail that follows the mouse.

Example sketch: CRTMouse

Working with the camera

One of the first things that attracted me to Processing was how easy it was to access my laptop's webcam and process the images that it was reading from the outside world. The camera plugs in naturally to the rest of Processing through the video library. You create an instance of the `Capture` class to obtain frames from the camera. If the camera tells you that a new frame is available (via the `available()` method) then you can ask it to download the next frame (via the `read()` method). Alternatively, you can write a `captureEvent()` function, which the library will call when a new camera frame is available. Either way, at that point the camera behaves just like any other `PImage`. The simplest demonstration of using the webcam is a "virtual mirror". Note that you have to flip

the camera's image horizontally or it's very hard to interact with your image!

Example sketch: Mirror

At this point, we can combine a virtual mirror with many of the other effects we looked at in this module already, for example halftoning.

Example sketch: HalftoningMirror