15 Images

Politics will eventually be replaced by imagery. The politician will be only too happy to abdicate in favor of his image, because the image will be much more powerful than he could ever be.
—Marshall McLuhan

When it comes to pixels, I think I've had my fill. There are enough pixels in my fingers and brains that I probably need a few decades to digest all of them.
—John Maeda

In this chapter:
- The `PImage` class
- Displaying images
- Changing image color
- The pixels of an image
- Simple image processing
- Interactive image processing

A digital image is nothing more than data — numbers indicating variations of red, green, and blue at a particular location on a grid of pixels. Most of the time, we view these pixels as miniature rectangles sandwiched together on a computer screen. With a little creative thinking and some lower level manipulation of pixels with code, however, you can display that information in a myriad of ways. This chapter is dedicated to breaking out of simple shape drawing in Processing and using images (and their pixels) as the building blocks of Processing graphics.

15-1 Getting started with images

As you just learned in Section 14-10 on page 294, Processing has a bunch of handy classes all ready to go for your use. (Later, in Chapter 23, you will discover that you also have access to a vast library of Java classes.) While I only briefly examined `PShape` objects in Chapter 14, this chapter will be entirely dedicated to another Processing-defined class: `PImage`, a class for loading and displaying an image such as the one shown in Figure 15-1.

Figure 15-1

Example 15-1. "Hello World" images

```
PImage img;

void setup() {
  size(320, 240);
  img = loadImage("runde_bird_cliffs.jpg");
}

void draw() {
  background(0);
  image(img, 0, 0);
}
```

Declare a variable of type `PImage`, a class available from the Processing core library.

Make a new instance of a `PImage` by loading an image file.

The `image()` function displays the image at a location – in this case the point (0,0).

Just as with `PShape`, using an instance of a `PImage` object is no different than using a *user-defined* class. First, a variable of type `PImage`, named `img`, is declared. Second, a `new` instance of a `PImage` object is created via the `loadImage()` method. `loadImage()` takes one argument, a `String` (strings of text are explored in greater detail in Chapter 17) indicating a file name, and loads that file into memory. `loadImage()` looks for image files stored in your Processing sketch's data folder.

The data folder: How do I get there?

Images can be added to the data folder automatically by dragging a file into your Processing window. You can also add files via:

Sketch → Add File...

or manually:

Sketch → Show Sketch Folder

This will open up the sketch folder as shown in Figure 15-2. If there is no data directory, create one. Otherwise, place your image files inside. Processing accepts the following file formats for images: GIF, JPG, TGA, and PNG. You'll want to make sure you include the file extension when referencing the file name as well, e.g., "file.jpg".

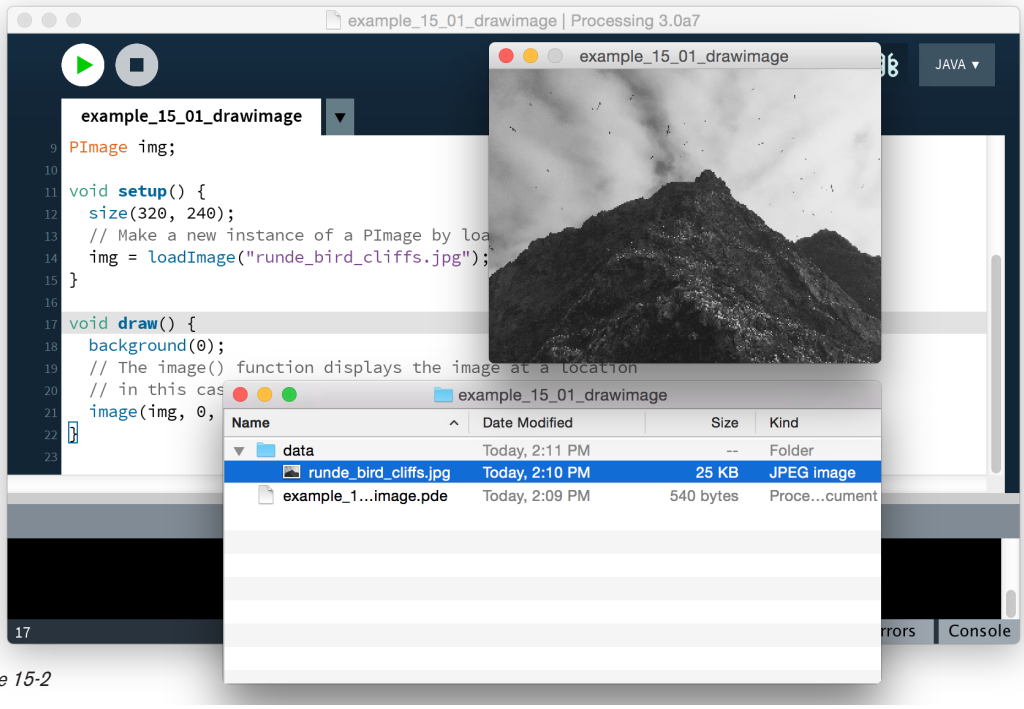


Figure 15-2

In Example 15-1, it may seem a bit peculiar that I never called a "constructor" to instantiate the `PImage` object, saying **`new PImage()`**. After all, in each of the object-related examples to date, a constructor is a must for producing an object instance.

```
Spaceship ss = new Spaceship();
Flower flr = new Flower(25);
```

And yet a `PImage` is created without `new`, using `loadImage()`:

```
PImage img = loadImage("file.jpg");
```

In fact, the `loadImage()` function performs the work of a constructor, returning a brand new instance of a `PImage` object generated from the specified filename. You can think of it as the `PImage` constructor for loading images from a file. For creating a blank image, the `createImage()` function is used.

```
// Create a blank image, 200 X 200 pixels with RGB color
PImage img = createImage(200, 200, RGB);
```

I should also note that the process of loading the image from the hard drive into memory is a slow one, and you should make sure your sketches only have to do it once, in `setup()`. Loading images in `draw()` may result in slow performance, as well as "Out of Memory" errors. You should also avoid calling `loadImage()` above `setup()` as Processing will not yet know the location of the "data" folder and will report an error.

Once the image is loaded, it's displayed with the `image()` function. The `image()` function must include three arguments — the image to be displayed, the x location, and the y location. Optionally, two arguments can be added to resize the image to a certain width and height.

```
image(img, 10, 20, 90, 60);
```

Exercise 15-1: Load and display an image. Control the image's width and height with the mouse.

15-2 Animation with an image

From here, it's easy to see how you can use images to further develop examples from previous chapters. Note in the following example how I draw the image relative to its center using `imageMode()` which works similarly to `rectMode()` but is applied to images.

Example 15-2. Image "sprite"

```
PImage head; // A variable for the image file
float x, y;  // Variables for image location
float rot;   // A variable for image rotation

void setup() {
  size(200, 200);
  // Load image, initialize variables
  head = loadImage("face.jpg");
  x = 0;
  y = width/2;
  rot = 0;
}

void draw() {
  background(255);

  translate(x, y);
  rotate(rot);
  imageMode(CENTER);
  image(head, 0, 0);

  // Adjust variables for animation
  x += 1.0;
  rot += 0.01;
  if (x > width) {
    x = 0;
  }
}
```

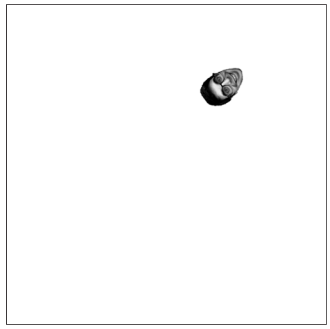


Figure 15-3

Images can be animated just like regular shapes using variables, `translate()`, `rotate()`, and so on.

Exercise 15-2: Rewrite this example in an object-oriented fashion where the data for the image, location, size, rotation, and so on is contained in a class. Can you have the class swap images when it hits the edge of the screen?

```
class Head {

  ________ // A variable for the image file

  ________ // Variables for image location

  ________ // A variable for image rotation


  Head(String filename, _________, _________) {
    // Load image, initialize variables

    _____________ = loadImage(_________________);

    ___________________________________________

    ___________________________________________

    ___________________________________________
  }

  void display() {

    ___________________________________________

    ___________________________________________

    ___________________________________________
  }

  void move() {

    ___________________________________________

    ___________________________________________

    ___________________________________________

    ___________________________________________

    ___________________________________________

    ___________________________________________
  }
}
```

The `String` class will be explored in detail in Chapter 17.

15-3 My very first image processing filter

Every now and then, when displaying an image, you might like to alter its appearance. Perhaps you would like the image to appear darker, transparent, bluish, and so on. This type of simple image filtering is achieved with Processing's `tint()` function. `tint()` is essentially the image equivalent of shape's `fill()`, setting the color and alpha transparency for displaying an image on screen. An image, nevertheless, is not usually all one color. The arguments for `tint()` simply specify how much of a given color to use for every pixel of that image, as well as how transparent those pixels should appear.

For the following examples, assume that two images (a sunflower and a dog) have been loaded and the dog is displayed as the background (which will allow me to demonstrate transparency). See Figure 15.4. For color versions of these images visit *http://www.learningprocessing.com*.

```
PImage sunflower = loadImage("sunflower.jpg");
PImage dog = loadImage("dog.jpg");
background(dog);
```

Figure 15-4

If `tint()` receives one argument, only the brightness of the image is affected.

```
tint(255);
image(sunflower, 0, 0);
```

A: The image retains its original state.

```
tint(100);
image(sunflower, 0, 0);
```

B: The image appears darker.

A second argument will change the image's alpha transparency.

```
tint(255, 127);
image(sunflower, 0, 0);
```

C: The image is at 50% opacity.

Three arguments affect the brightness of the red, green, and blue components of each color.

```
tint(0, 200, 255)
image(sunflower, 0, 0);
```

D: None of it is red, most of it is green, and all of it is blue.

Finally, adding a fourth argument to the method manipulates the alpha (same as with two arguments). Incidentally, the range of values for `tint()` can be specified with `colorMode()` (see Chapter 1).

```
tint(255, 0, 0, 100);
image(sunflower, 0, 0);
```

E: The image is tinted red and transparent.

Exercise 15-3: Display an image using `tint()`. *Use the mouse location to control the amount of red, green, and blue tint. Also try using the distance of the mouse from the corners or center.*

Exercise 15-4: Using `tint()`, *create a montage of blended images. What happens when you layer a large number of images, each with different alpha transparency, on top of each other? Can you make it interactive so that different images fade in and out?*

15-4 An array of images

One image is nice, a good place to start. It will not be long, however, until the temptation of using many images takes over. Yes, you could keep track of multiple images with multiple variables, but here is a magnificent opportunity to rediscover the power of the array. Let's assume I have five images and want to display a new background image each time the user clicks the mouse.

First, I'll set up an array of images, as a global variable.

```
// Image array
PImage[] images = new PImage[5];
```

Second, I'll load each image file into the appropriate location in the array. This happens in `setup()`.

```
// Loading images into an array
images[0] = loadImage("cat.jpg");
images[1] = loadImage("mouse.jpg");
images[2] = loadImage("dog.jpg");
images[3] = loadImage("kangaroo.jpg");
images[4] = loadImage("porcupine.jpg");
```

Of course, this is somewhat awkward. Loading each image individually is not terribly elegant. With five images, sure, it's manageable, but imagine writing the above code with 100 images. One solution is to store the filenames in a `String` array and use a `for` statement to initialize all the array elements.

```
// Loading images into an array from an array of filenames
String[] filenames = {"cat.jpg", "mouse.jpg", "dog.jpg", "kangaroo.jpg",
"porcupine.jpg");
for (int i = 0; i < filenames.length; i++) {
  images[i] = loadImage(filenames[i]);
}
```

Concatenation: a new kind of addition

Usually, a plus sign (+) means, add. 2 + 2 = 4, right?

With text (as stored in a *string*, enclosed in quotes), + means *concatenate,* that is, join two strings together.

"cow" + "bell" → "cowbell"

"2" + "2" → "22"

See more about strings in Chapter 17.

Even better, if I just took a little time out of my hectic schedule to plan ahead, numbering the image files ("animal0.jpg", "animal1.jpg", "animal2.jpg", etc.), I could really simplify the code:

```
// Loading images with numbered files
for (int i = 0; i < images.length; i++) {
  images[i] = loadImage("animal" + i + ".jpg");
}
```

Once the images are loaded, it's on to `draw()`. There, I'll choose to display one particular image, picking from the array by referencing an index ("0" below).

```
image(images[0], 0, 0);
```

Of course, hard-coding the index value is foolish. I'll need a variable in order to dynamically display a different image at any given moment in time.

```
image(images[imageIndex], 0, 0);
```

The `imageIndex` variable should be declared as a global variable (of type integer). Its value can be changed throughout the course of the program. The full version is shown in Example 15-3.

Example 15-3. Swapping images

```
int maxImages = 10; // Total # of images
int imageIndex = 0; // Initial image to be displayed is the first
PImage[] images = new PImage[maxImages]; // The image array

void setup() {
  size(200, 200);
  // Loading the images into the array
  // Don't forget to put the JPG files in the data folder!
  for (int i = 0; i < images.length; i++) {
    images[i] = loadImage("animal" + i + ".jpg");
  }
}

void draw() {
  image(images[imageIndex], 0, 0); // Displaying one image
}

void mousePressed() {
  // A new image is picked randomly when the mouse is clicked
  // Note the index to the array must be an integer!
  imageIndex = int(random(images.length));
}
```

Declaring an array of images.

Loading an array of images.

Displaying one image from the array.

Picking a new image to display by changing the index variable!

To play the images in sequence as an animation, follow Example 15-4. (Only the new `draw()` function is included below.)

Example 15-4. Image sequence

```
void draw() {
  background(0);
  image(images[imageIndex], 0, 0);
  // Increment image index by one each cycle
  // use modulo "%" to return to 0 once the size
  // of the array is reached
  imageIndex = (imageIndex + 1) % images.length;
}
```

Remember modulus? The % sign? It allows you to cycle a counter back to 0. See Chapter 13 for a review.

Exercise 15-5: Create multiple instances of an image sequence onscreen. Have them start at different times within the sequence so that they are out of sync. Hint: Use object-oriented programming to place the image sequence in a class.

15-5 Pixels, pixels, and more pixels

If you have been diligently reading this book in precisely the prescribed order, you will notice that so far, the only offered means for drawing to the screen is through a function call. "Draw a line between these points," or "Fill an ellipse with red," or "Load this JPG image and place it on the screen here." But somewhere, somehow, someone had to write code that translates these function calls into setting the individual pixels on the screen to reflect the requested shape. A line does not appear because you say `line()`, it appears because all the pixels along a linear path between two points change color. Fortunately,

you do not have to manage this lower-level-pixel-setting on a day-to-day basis. You have the developers of Processing (and Java) to thank for the many drawing functions that take care of this business.

Nevertheless, from time to time, you may want to break out of a mundane shape drawing existence and deal with the pixels on the screen directly. Processing provides this functionality via the `pixels` array.

You are familiar with the idea of each pixel on the screen having an (x,y) position in a two-dimensional window. However, the array `pixels` has only one dimension, storing color values in linear sequence. See Figure 15-5.

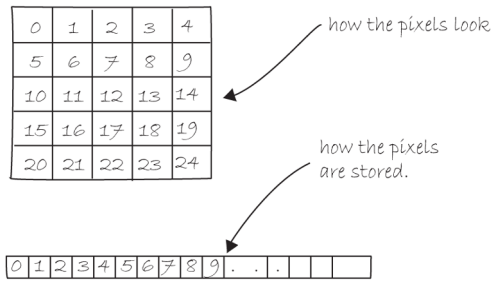


Figure 15-5

Take the following example. This sketch sets each pixel in a window to a random grayscale value. The *pixels* array is just like an other array, the only difference is that you do not have to declare it since it is a Processing built-in variable.

Example 15-5. Setting pixels

```
size(200, 200);
// Before I deal with pixels
loadPixels();

// Loop through every pixel
for (int i = 0; i < pixels.length; i++) {

  // Pick a random number, 0 to 255
  float rand = random(255);
  // Create a grayscale color based on random number
  color c = color(rand);
  // Set pixel at that location to random color
  pixels[i] = c;
}

// When you are finished dealing with pixels
updatePixels();
```

You can get the length of the pixels array just like with any array.

You can access individual elements of the pixels array via an index, just like with any other array.

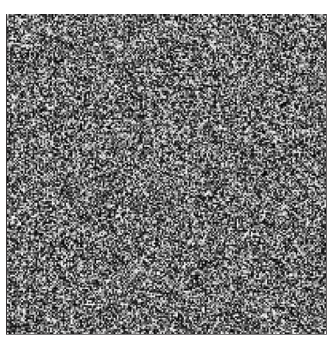


Figure 15-6

First, I should point out something important in the above example. Whenever you're accessing the pixels of a Processing window, you must alert Processing to this activity. This is accomplished with two functions:

- `loadPixels()` — This function is called *before* you access the pixel array, saying "load the pixels, I would like to speak with them!"
- `updatePixels()` — This function is called *after* you finish with the pixel array, saying "Go ahead and update the pixels, I'm all done!"

In Example 15-5, because the colors are set randomly, I did not have to worry about where the pixels are onscreen as I accessed them, since I was simply setting all the pixels with no regard to their relative location. However, in many image processing applications, the (x,y) location of the pixels themselves is crucial information. A simple example of this might be: set every even column of pixels to white and every odd to black. How could you do this with a one-dimensional pixel array? How do you know what column or row any given pixel is in?

When programming with pixels, you need to be able to think of every pixel as living in a two-dimensional world, but continue to access the data in one dimension (since that is how it's made available to us). You can do this via the following formula:

1. Assume a window or image with a given width and height.
2. You then know the pixel array has a total number of elements equaling width × height.
3. For any given (x,y) point in the window, the location in the one-dimensional pixel array is:

   pixel array location = x + (y × width);

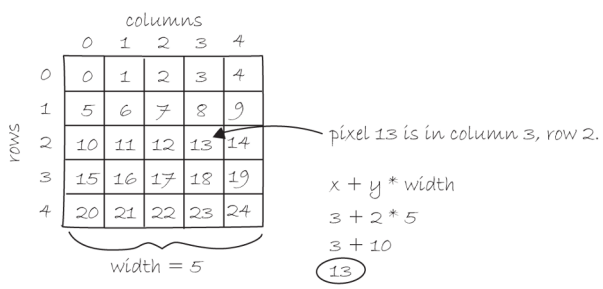


Figure 15-7

This may remind you of two-dimensional arrays in Chapter 13. In fact, you will need to use the same nested `for` loop technique. The difference is that, although you want to use `for` loops to think about the pixels in two dimensions, when you go to actually access the pixels, they live in a one-dimensional array, and you have to apply the formula from Figure 15-7.

Let's look at how it's done, completing the even/odd column problem. See Figure 15-8.

Example 15-6. Setting pixels according to their 2D location

```
size(200, 200);
loadPixels();

// Loop through every pixel column
for (int x = 0; x < width; x++) {

  // Loop through every pixel row
  for (int y = 0; y < height; y++) {

    int loc = x + y * width;

    if (x % 2 == 0) {
      pixels[loc] = color(255);
    } else {
      pixels[loc] = color(0);
    }
  }
}
updatePixels();
```

Two loops allow you to visit every column (x) and every row (y).

The location in the pixel array is calculated via the formula: 1D pixel location = x + y * width

Using the column number (x) to determine whether the color should be black or white

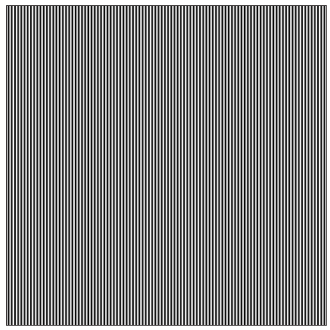


Figure 15-8

Pixel density revisited

In Section 14-2 on page 271, I briefly mentioned the `pixelDensity()` function which can be used for higher quality rendering on high pixel density displays (like the Apple "Retina"). Setting `pixelDensity(2)` actually quadruples the number of pixels used for the sketch window; the number of horizontal and vertical pixels are each doubled. When drawing shapes, everything is handled behind the scenes but in the case of working with the `pixels` array directly you have to account for the actual pixel width and height (which is now different than sketch `width` and `height`). Processing includes convenience variables `pixelWidth` and `pixelHeight` for this very scenario. For example, here's a version of Example 15-5 with `pixelDensity(2)`.

```
size(200, 200);
pixelDensity(2);
loadPixels();
for (int x = 0; x < pixelWidth x++) {
  for (int y = 0; y < pixelHeight; y++) {
    int loc = x + y * pixelWidth;
    pixels[loc] = color(random(255));
  }
}
updatePixels();
```

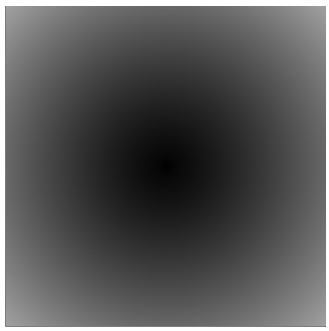
The pixel density is set to 2.

`pixelWidth` and `pixelHeight` are used instead of `width` and `height`

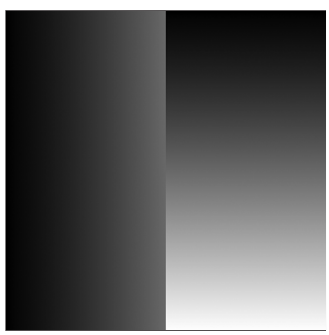
For the rest of this chapter, a pixel density of one will be assumed.

Exercise 15-6: Complete the code to match the corresponding screenshots.

```
size(255, 255);

____________________;
for (int x = 0; x < width; x++) {
  for (int y = 0; y < height; y++) {

    int loc = ____________________;

    float distance = ____________________);

    pixels[loc] = ____________________;
  }
}

____________________;
```



```
size(255, 255);

____________________;
for (int x = 0; x < width; x++) {
  for (int y = 0; y < height; y++) {

    ____________________;

    if (____________________) {

      ____________________;

    } else {

      ____________________;
    }
  }
}

____________________;
}
```



15-6 Intro to image processing

The previous section looked at examples that set pixel values according to an arbitrary calculation. I will now look at how you might set pixels according to those found in an existing `PImage` object. Here is some pseudocode.

1. Load the image file into a `PImage` object.
2. For each pixel in the image, retrieve the pixel's color and set the display pixel to that color.

The `PImage` class includes some useful fields that store data related to the image — `width`, `height`, and `pixels`. Just as with user-defined classes, you can access these fields via the dot syntax.

```
PImage img = createImage(320, 240, RGB); // Make a PImage object
println(img.width);  // Yields 320
println(img.height); // Yields 240
img.pixels[0] = color(255, 0, 0); // Sets the first pixel of the image to red
```

Access to these fields allows you to loop through all the pixels of an image and display them onscreen.

Example 15-7. Displaying the pixels of an image

```
PImage img;

void setup() {
  size(200, 200);
  img = loadImage("sunflower.jpg");
}

void draw() {
  loadPixels();
  img.loadPixels();

  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
     int loc = x + y * width;
     float r = red  (img.pixels[loc]);
     float g = green(img.pixels[loc]);
     float b = blue (img.pixels[loc]);


     // image processing!
     // image processing!
     // image processing!


     // Set the display pixel
     pixels[loc] = color(r, g, b);
    }
  }
  updatePixels();
}
```

You must also call `loadPixels()` on the `PImage`.

The functions `red()`, `green()`, and `blue()` pull out the three color components from a pixel.

If the RGB values were to change, it would happen here, before setting the pixel in the display window.

Figure 15-9

Now, you could certainly come up with simplifications in order to merely display the image (e.g., the nested loop is not required, not to mention that using the `image()` function would allow you to skip all this pixel work entirely). However, Example 15-7 provides a basic framework for getting the red, green, and blue values for each pixel based on its spatial orientation — (x,y) location; ultimately, this will allow you to develop more advanced image processing algorithms.

Before you move on, I should stress that this example works because the display area has the same dimensions as the source image. If this were not the case, you would simply need to have two pixel location calculations, one for the source image and one for the display area.

```
int imageLoc   = x + y * img.width;
int displayLoc = x + y * width;
```

Exercise 15-7: Using Example 15-7, change the values of r, g, and b before displaying them.

15-7 A second image processing filter, making your own `tint()`

Just a few paragraphs ago, you were enjoying a relaxing coding session, colorizing images and adding alpha transparency with the friendly `tint()` method. For basic filtering, this method did the trick. The pixel by pixel method, however, will allow you to develop custom algorithms for mathematically altering the colors of an image. Consider brightness — brighter colors have higher values for their red, green, and blue components. It follows naturally that you can alter the brightness of an image by increasing or decreasing the color components of each pixel. In the next example, I will dynamically increase or decrease those values based on the mouse's horizontal location. (Note that the next two examples include only the image processing loop itself, the rest of the code is assumed.)

Example 15-8. Adjusting image brightness

```
for (int x = 0; x < img.width; x++) {
  for (int y = 0; y < img.height; y++) {
    // Calculate the 1D pixel location
    int loc = x + y * img.width;
    // Get the red, green, blue values
    float r = red  (img.pixels[loc]);
    float g = green(img.pixels[loc]);
    float b = blue (img.pixels[loc]);


    // Adjust brightness with mouseX
    float adjustBright
        = map(mouseX, 0, width, 0, 8);
    r *= adjustBright;
    g *= adjustBright;
    b *= adjustBright;

    r = constrain(r, 0, 255);
    g = constrain(g, 0, 255);
    b = constrain(b, 0, 255);

    // Make a new color
    color c = color(r, g, b);
    pixels[loc] = c;
  }
}
```

Figure 15-10

I calculate a multiplier ranging from 0.0 to 8.0 based on `mouseX` position using `map()`. That multiplier changes the RGB value of each pixel.

The RGB values are constrained between 0 and 255 before being set as a new color.

Since I am altering the image on a per pixel basis, all pixels need not be treated equally. For example, I can alter the brightness of each pixel according to its distance from the mouse.

Example 15-9. Adjusting image brightness based on pixel location

```
for (int x = 0; x < img.width; x++) {
  for (int y = 0; y < img.height; y++) {
    // Calculate the 1D pixel location
    int loc = x + y * img.width;
    // Get the red, green, blue values from pixel
    float r = red  (img.pixels[loc]);
    float g = green(img.pixels[loc]);
    float b = blue (img.pixels[loc]);

    // Calculate an amount to change brightness
    // based on proximity to the mouse
    float distance = dist(x, y, mouseX, mouseY);
    float adjustBright = map(distance, 0, 50, 8, 0);
    r *= adjustBrightness;
    g *= adjustBrightness;
    b *= adjustBrightness;
    // Constrain RGB to between 0-255
    r = constrain(r, 0, 255);
    g = constrain(g, 0, 255);
    b = constrain(b, 0, 255);
    // Make a new color
    color c = color(r, g, b);
    pixels[loc] = c;
  }
}
```

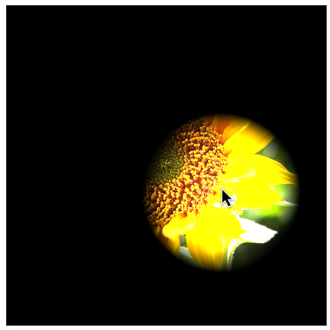


Figure 15-11

The closer the pixel is to the mouse, the lower the value of `distance` is. I want closer pixels to be brighter, however, so I invert the `adjustBrightness` factor using `map()`. Pixels with a distance of 50 (or greater) have their brightness multiplied by 0.0 (resulting in now brightness) and brightness for pixels with a distance of 0 is multiplied by a factor of 8.

Exercise 15-8: Adjust the brightness of the red, green, and blue color components separately according to mouse interaction. For example, let `mouseX` *control red,* `mouseY` *green, distance blue, and so on.*

15-8 Writing to another `PImage` object's pixels

All of the image processing examples have read every pixel from a source image and written a new pixel to the Processing window directly. However, it's often more convenient to write the new pixels to a destination image (that you then display using the `image()` function). I will demonstrate this technique while looking at another simple pixel operation: *threshold*.

A *threshold* filter displays each pixel of an image in only one of two states, black or white. That state is set according to a particular threshold value. If the pixel's brightness is greater than the threshold, I color the pixel white, less than, black. Example 15-10 uses an arbitrary threshold of 100.

Example 15-10. Brightness threshold

```
PImage source;      // Source image
PImage destination; // Destination image

void setup() {
  size(200, 200);
  source = loadImage("sunflower.jpg");
  destination = createImage(source.width,
                  source.height, RGB);
}

void draw() {
  float threshold = 127;

  // The sketch is going to look at both image's pixels
  source.loadPixels();
  destination.loadPixels();

  for (int x = 0; x < source.width; x++) {
    for (int y = 0; y < source.height; y++) {
      int loc = x + y*source.width;
      // Test the brightness against the threshold
      if (brightness(source.pixels[loc]) > threshold){
        destination.pixels[loc] = color(255); // White
      } else {
        destination.pixels[loc] = color(0);   // Black
      }
    }
  }

  // The pixels in destination changed
  destination.updatePixels();
  // Display the destination
  image(destination, 0, 0);
}
```

Two images are required, a source (original file) and destination (to be displayed) image.

The destination image is created as a blank image the same size as the source.

`brightness()` returns a value between 0 and 255, the overall brightness of the pixel's color. If it is more than 100, make it white, less than 100, make it black.

Writing to the destination image's pixels.

I have to display the destination image!

Figure 15-12

Exercise 15-9: Change the threshold according to `mouseX` *using* `map()`.

This particular functionality is available without per pixel processing as part of Processing's `filter()` function. Understanding the lower level code, however, is crucial if you want to implement your own image processing algorithms, not available with `filter()`.

If all you want to do is threshold, Example 15-11 is much simpler.

Example 15-11. Brightness threshold with filter

```
// Draw the image
image(img, 0, 0);
// Filter the window with a threshold effect
// 0.5 means threshold is 50% brightness
filter(THRESHOLD, 0.5);
```

More on the `filter()` function

```
filter(mode);
```

```
filter(mode, level);
```

The `filter()` function offers a set of prepackaged filters for the display window. It's not necessary to use a `PImage`, the filter will alter the look of whatever is drawn in the window at the time it is executed. Other available modes besides `THRESHOLD` are `GRAY`, `INVERT`, `POSTERIZE`, `BLUR`, `OPAQUE`, `ERODE`, and `DILATE`. See the Processing reference (*http://processing.org/reference/filter_.html*) for examples of each.

In addition, while beyond the scope of this book, Processing also supports *shaders* via the `PShader` class. Shaders are low-level programs written in a special language called GLSL (OpenGL Shading Language) and can be used for a variety of computer graphics effects, including image processing. `PShader` is available for use with both the `P3D` and `P2D` renderers and you can learn more by reading the processing.org `PShader` tutorial by Andres Colubri (*https://processing.org/tutorials/pshader/*).

15-9 Level II: Pixel group processing

In previous examples, you have seen a one-to-one relationship between source pixels and destination pixels. To increase an image's brightness, you take one pixel from the source image, increase the RGB values, and display one pixel in the output window. In order to perform more advanced image processing functions, however, you must move beyond the one-to-one pixel paradigm into *pixel group processing*.

Let's start by creating a new pixel out of two pixels from a source image — a pixel and its neighbor to the left.

If I know a pixel is located at (x,y):

```
int loc = x + y * img.width;
color pix = img.pixels[loc];
```

Then its left neighbor is located at (x-1,y):

```
int leftLoc = (x - 1) + y * img.width;
color leftPix = img.pixels[leftLoc];
```

I could then make a new color out of the difference between the pixel and its neighbor to the left.

```
float diff = abs(brightness(pix) - brightness(leftPix));
pixels[loc] = color(diff);
```

Example 15-12 shows the full algorithm, with the results shown in Figure 15-13.

Example 15-12. Pixel neighbor differences (edges)

```
// Since I am looking at left neighbors
// I skip the first column
for (int x = 1; x < width; x++) {
  for (int y = 0; y < height; y++) {
    // Pixel location and color
    int loc = x + y * img.width;
    color pix = img.pixels[loc];

    // Pixel to the left location and color
    int leftLoc = (x - 1) + y * img.width;
    color leftPix = img.pixels[leftLoc];
    // New color is difference between
    // pixel and left neighbor
    float diff = abs(brightness(pix)
        - brightness(leftPix));
    pixels[loc] = color(diff);
  }
}
```

Reading the pixel to the left.

Figure 15-13

Example 15-12 is a simple vertical edge detection algorithm. When pixels differ greatly from their neighbors, they are most likely "edge" pixels. For example, think of a picture of a white piece of paper on a black tabletop. The edges of that paper are where the colors are most different, where white meets black.

In Example 15-12, I looked at two pixels to find edges. More sophisticated algorithms, however, usually involve looking at many more neighboring pixels. After all, each pixel has eight immediate neighbors: top left, top, top right, right, bottom right, bottom, bottom left, and left. See Figure 15-14.

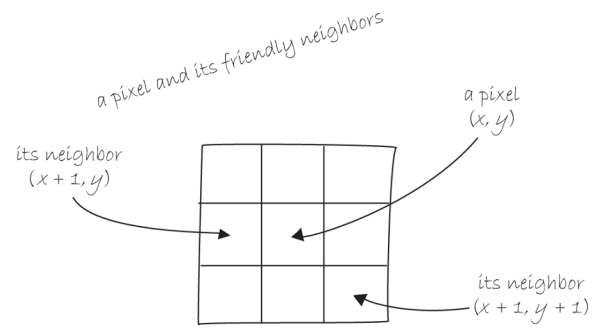


Figure 15-14

These image processing algorithms are often referred to as a "spatial convolution." The process uses a *weighted average* of an input pixel and its neighbors to calculate an output pixel. In other words, that new pixel is a function of an area of pixels. Neighboring areas of different sizes can be employed, such as a 3 × 3 matrix, 5 × 5, and so on.

Different combinations of weights for each pixel result in various effects. For example, an image can be *sharpened* by subtracting the neighboring pixel values and increasing the centerpoint pixel. A *blur* is achieved by taking the average of all neighboring pixels. (Note that the values in the convolution matrix add up to 1.)

For example,

```
Sharpen:
 -1 -1 -1
 -1  9 -1
 -1 -1 -1

Blur:
1/9 1/9 1/9
1/9 1/9 1/9
1/9 1/9 1/9
```

Example 15-13 performs a convolution using a 2D array (see Chapter 13 for a review of 2D arrays) to store the pixel weights of a 3 × 3 matrix. This example is probably the most advanced example you have encountered in this book so far, since it involves so many elements (nested loops, 2D arrays, pixels, etc.).

Example 15-13. Sharpen with convolution

```
PImage img;
int w = 80;

// it's possible to perform a convolution
// the image with different matrices

float[][] matrix = { { -1, -1, -1 },
                     { -1,  9, -1 },
                     { -1, -1, -1 } } ;
void setup() {
  size(200, 200);
  img = loadImage("sunflower.jpg");
}

void draw() {
  // The sketch is only going to process a portion of the image
  // so let's set the whole image as the background first
  image(img, 0, 0);

  int xstart = constrain(mouseX - w/2, 0, img.width);
  int ystart = constrain(mouseY - w/2, 0, img.height);
  int xend   = constrain(mouseX + w/2, 0, img.width);
  int yend   = constrain(mouseY + w/2, 0, img.height);
  int matrixsize = 3;
```

The convolution matrix for a "sharpen" effect stored as a 3 × 3 two-dimensional array.

Figure 15-15

In this example only a section of the image – an 80 × 80 rectangle around the mouse location – is processed.

```
  loadPixels();
  // Begin loops for every pixel
  for (int x = xstart; x < xend; x++) {
    for (int y = ystart; y < yend; y++) {
       color c = convolution(x, y, matrix, matrixsize, img);
       int loc = x + y*img.width;
       pixels[loc] = c;
    }
  }
  updatePixels();

  stroke(0);
  noFill();
  rect(xstart, ystart, w, w);
}
color convolution(int x, int y, float[][] matrix, int matrixsize, PImage img) {
  float rtotal = 0.0;
  float gtotal = 0.0;
  float btotal = 0.0;
  int offset = matrixsize / 2;
  // Loop through convolution matrix
  for (int i = 0; i < matrixsize; i++) {
    for (int j = 0; j < matrixsize; j++) {
      // What pixel is being examined
      int xloc = x + i - offset;
      int yloc = y + j - offset;
      int loc = xloc + img.width * yloc;

      loc = constrain(loc, 0, img.pixels.length-1);

      // Calculate the convolution
      rtotal += (red(img.pixels[loc]) * matrix[i][j]);
      gtotal += (green(img.pixels[loc]) * matrix[i][j]);
      btotal += (blue(img.pixels[loc]) * matrix[i][j]);
    }
  }
  // Make sure RGB is within range
  rtotal = constrain(rtotal, 0, 255);
  gtotal = constrain(gtotal, 0, 255);
  btotal = constrain(btotal, 0, 255);
  // Return the resulting color
  return color(rtotal, gtotal, btotal);
  }
}
```

Each pixel location (x,y) gets passed into a function called `convolution()` which returns a new color value to be displayed.

It's often good when looking at neighboring pixels to make sure you have not gone off the edge of the pixel array by accident.

Sum all the neighboring pixels multiplied by the values in the convolution matrix.

After the sums are constrained within a range of 0–255, a new color is made and returned.

Exercise 15-10: Try different values for the convolution matrix.

Exercise 15-11: Using the framework established by the image processing examples, create a filter that takes two images as input and generates one output image. In other words, each pixel displayed should be a function of the color values from two pixels, one from one image and one from another. For example, can you write the code to blend two images together (without using `tint()`*)?*

15-10 Creative visualization

You may be thinking: "Gosh, this is all very interesting, but seriously, when I want to blur an image or change its brightness, do I really need to write code? I mean, can't I use Photoshop?" Indeed, what I have covered here is merely an introductory understanding of what highly skilled programmers at Adobe do. The power of Processing, however, is the potential for real-time, interactive graphics applications. There is no need for you to live within the confines of "pixel point" and "pixel group" processing.

Following are two examples of algorithms for drawing Processing shapes. Instead of coloring the shapes randomly or with hard-coded values as I have in the past, I am going to select colors from the pixels of a `PImage` object. The image itself is never displayed; rather, it serves as a database of information that you can exploit for your own creative pursuits.

In this first example, for every cycle through `draw()`, I will fill one ellipse at a random location onscreen with a color taken from its corresponding location in the source image. The result is a "pointillist-like" effect. See Figure 15-16.

Example 15-14. "Pointillism"

```
PImage img;
int pointillize = 16;

void setup() {
  size(200, 200);
  img = loadImage("sunflower.jpg");
  background(0);
}

void draw() {
  // Pick a random point
  int x = int(random(img.width));
  int y = int(random(img.height));
  int loc = x + y * img.width;

  // Look up the RGB color in the source image
  img.loadPixels();
  float r = red(img.pixels[loc]);
  float g = green(img.pixels[loc]);
  float b = blue(img.pixels[loc]);

  noStroke();
  fill(r, g, b, 100);
  ellipse(x, y, pointillize, pointillize);
}
```

Back to shapes! Instead of setting a pixel, use the color from a pixel to draw a circle.

Figure 15-16

In this next example, I'll take the data from a two-dimensional image and, using the 3D translation techniques described in Chapter 14, render a rectangle for each pixel in three-dimensional space. The z position is determined by the brightness of the color. Brighter colors appear closer to the viewer and darker ones further away.

Figure 15-17

Example 15-15. 2D image mapped to 3D

```
PImage img;       // The source image
int cellsize = 2; // Dimensions of each cell in the grid
int cols, rows;   // Number of columns and rows in the system

void setup() {
  size(200, 200, P3D);
  img = loadImage("sunflower.jpg"); // Load the image
  cols = width / cellsize;               // Calculate # of columns
  rows = height / cellsize;              // Calculate # of rows
}

void draw() {
  background(255);
  img.loadPixels();
  // Begin loop for columns
  for (int i = 0; i < cols; i++) {
    // Begin loop for rows
    for (int j = 0; j < rows; j++) {
      int x = i*cellsize + cellsize/2; // x position
      int y = j*cellsize + cellsize/2; // y position
      int loc = x + y * width;         // Pixel array location
      color c = img.pixels[loc];       // Grab the color

      // Calculate a z position as a function of mouseX and pixel brightness
      float z = map(brightness(img.pixels[loc]), 0, 255, 0, mouseX);


      // Translate and draw!
      pushMatrix();
      translate(x, y, z);
      fill(c);
      noStroke();
      rectMode(CENTER);
      rect(0, 0, cellsize, cellsize);
      popMatrix();
    }
  }
}
```

A `z` position is calculated by mapping pixel brightness to the mouse's x location.

Exercise 15-12: Create a sketch that uses shapes to display a pattern that covers an entire window. Load an image and color the shapes according to the pixels of that image. The following image, for example, uses triangles.