LEARNING PROCESSING

A Beginner's Guide to Programming Images, Animation, and Interaction

Daniel Shiffman

MORGAN KAUFMANN

Learning Processing

A Beginner's Guide to Programming Images, Animation, and Interaction

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, we 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

By now, we are quite comfortable with the idea of data types. We specify them often—a float variable called *speed*, an int named *x*, perhaps even a char entitled *letterGrade*. These are all *primitive* data types, bits sitting in the computer's memory ready for our use. Though perhaps a bit trickier, we are also beginning to feel at ease with objects, *complex* data types that store multiple pieces of data (along with functionality)—our Zoog class, for example, included floating point variables for location, size, and speed as well as methods to move, display itself, and so on. Zoog, of course, is a user-defined class; we brought Zoog into this programming world, defining what it means to be a Zoog, and defining the data and functions associated with a Zoog object.

In addition to user-defined objects, *Processing* has a bunch of handy classes all ready to go without us writing any code. (Later, in Chapter 23, we will find out that we also have access to a vast library of Java classes.) The first *Processing*-defined class we will examine is PImage, a class for loading and displaying an image such as the one shown in Figure 15.1.

fig. 15.1

Example 15-1: "Hello World" images

```
// Declaring a variable of type PImage
PImage img;

void setup() {
  size(320,240);
  // Make a new instance of a PImage by loading an image file
  img = loadImage("mysummervacation.jpg");
}

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

Declaring a variable of type PImage, a class available to us from the *Processing* core library.

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

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* are explored in greater detail in Chapter 17) indicating a file name, and loads the 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?

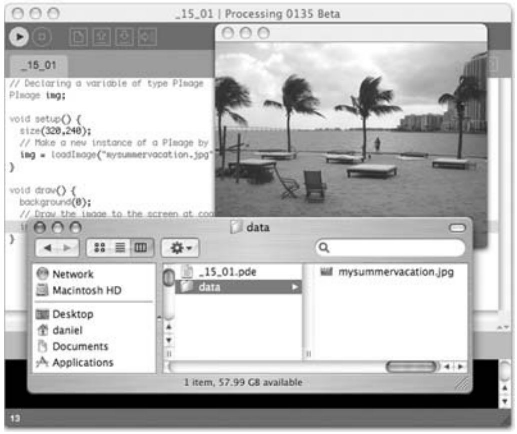
Images can be added to the data folder automatically via:

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.



_15_01 | Processing 0135 Beta

```
// Declaring a variable of type PImage
PImage img;

void setup() {
  size(320,240);
  // Make a new instance of a PImage by
  img = loadImage("mysummervacation.jpg"
}

void draw() {
  background(0);
  // Draw the image to the screen at coo
}
```

fig. 15.2

In Example 15-1, it may seem a bit peculiar that we never called a "constructor" to instantiate the PImage object, saying **"new PImage()"**. After all, in all 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:

```
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. We 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, 200X200 pixels with RGB color
PImage img = createImage(200,200,RGB);
```

We should also note that the process of loading the image from the hard drive into memory is a slow one, and we should make sure our program only has to do it once, in ***setup()***. Loading images in ***draw()*** may result in slow performance, as well as "Out of Memory" errors.

Once the image is loaded, it is 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 is easy to see how you can use images to further develop examples from previous chapters.

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 = 0.0f;
  y = width/2.0f;
  rot = 0.0f;
}

void draw() {
  background(255);

  // Translate and rotate
  translate(x,y);
  rotate(rot);
  image(head,0,0); // Draw image
```

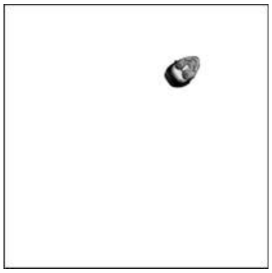


fig. 15.3

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

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

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 {
  ________  // Avariable 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() {
    ________________________________________
    ________________________________________
    ________________________________________
    ________________________________________
    ________________________________________
    ________________________________________
  }
}
```

String is also a class we get for free and will be explored further in Chapter 17.

15.3 My Very First Image Processing Filter

Every now and then, when displaying an image, we choose to alter its appearance. Perhaps we 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, we will assume that two images (a sunflower and a dog) have been loaded and the dog is displayed as the background (which will allow us 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);
```

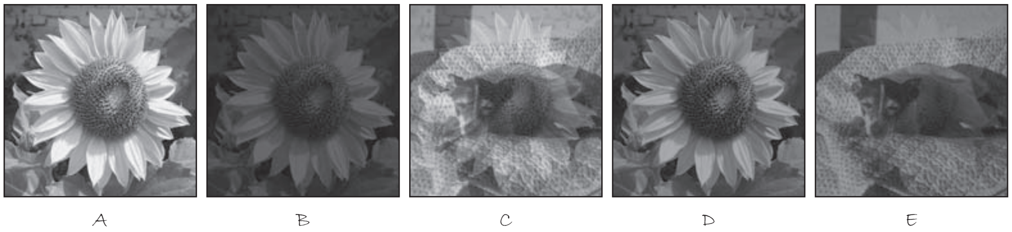


fig. 15.4

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, we 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 we have five images and want to display a new background image each time the user clicks the mouse.

First, we set up an array of images, as a global variable.

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

Second, we 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 is 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.

"Happy" + " Trails" = "Happy Trails"

"2" + "2" = "22"

See more about *Strings* in Chapter 17.

Even better, if we just took a little time out of our hectic schedules to plan ahead, numbering the image files ("animal1.jpg", "animal2.jpg", "animal3.jpg", etc.), we can 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, we 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. We 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.

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 us 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 we say ***line()***, it appears because we color all the pixels along a linear path between two points. Fortunately, we do not have to manage this lower-level-pixel-setting on a day-to-day basis. We 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, we do want to break out of our mundane shape drawing existence and deal with the pixels on the screen directly. *Processing* provides this functionality via the *pixels* array.

We are familiar with the idea of each pixel on the screen having an *X* and *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.

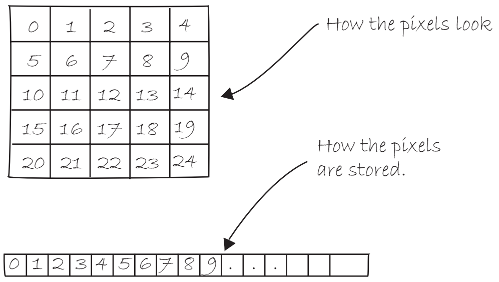


fig. 15.5

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 we do not have to declare it since it is a *Processing* built-in variable.

Example 15-5: Setting pixels

```
size(200,200);
// Before we 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);
```

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

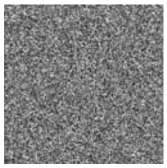


fig. 15.6

```
    // Create a grayscale color based on random number
    color c = color(rand);
    // Set pixel at that location to random color
    pixels[i] = c;
  }
  // When we are finished dealing with pixels
  updatePixels();
```

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

First, we should point out something important in the above example. Whenever you are 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, we did not have to worry about where the pixels are onscreen as we access them, since we are simply setting all the pixels with no regard to their relative location. However, in many image processing applications, the *XY* 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, we 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 is made available to us). We can do this via the following formula:

1. Assume a window or image with a given WIDTH and HEIGHT.
2. We 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 our one-dimensional pixel array is:

   LOCATION = X + Y * WIDTH

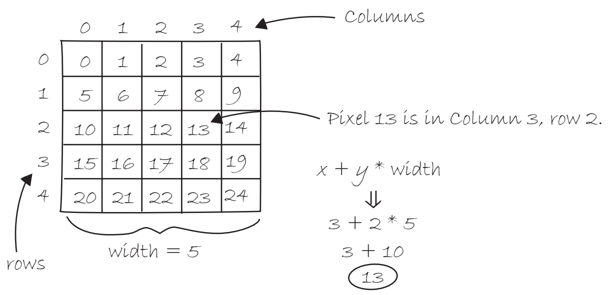


fig. 15.7

This may remind you of two-dimensional arrays in Chapter 13. In fact, we will need to use the same nested ***for*** loop technique. The difference is that, although we want to use ***for*** loops to think about the pixels in two dimensions, when we go to actually access the pixels, they live in a one-dimensional array, and we have to apply the formula from Figure 15.7.

Let's look at how it is 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++){


    // Use the formula to find the 1D location
    int loc = x + y * width;


    if (x % 2 == 0){ // If we are an even column
      pixels[loc] = color(255);
    } else {          // If we are an odd column
      pixels[loc] = color(0);
    }
  }
}
updatePixels();
```

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

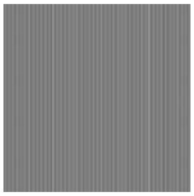
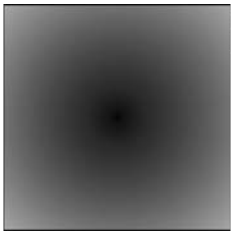


fig. 15.8

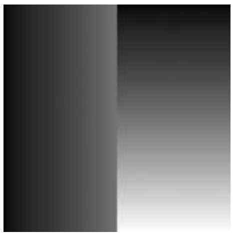
The location in the pixel array is calculated via our formula: 1D pixel location = *x* + *y* * width

We use the column number (*x*) to determine whether the color should be black or white.

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. We will now look at how we 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 PImage, 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 our user-defined classes, we 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 us 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();
  // Since we are going to access the image's pixels too
  img.loadPixels();
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      int loc = x + y*width;


      // Image Processing Algorithm would go here
      float r = red  (img.pixels [loc]);
      float g = green(img.pixels[loc]);
      float b = blue (imq.pixels[loc];


      // Image Processing would go here

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

fig. 15.9

We must also call ***loadPixels()*** on the PImage since we are going to read its pixels.

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

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

Now, we 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 us 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 (*XY* location); ultimately, this will allow us to develop more advanced image processing algorithms.

Before we 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 Our Second Image *Processing* Filter, Making Our Own *Tint( )*

Just a few paragraphs ago, we 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 us 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 we can alter the brightness of an image by increasing or decreasing the color components of each pixel. In the next example, we 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 R,G,B values from image
    float r = red (img.pixels[loc]);
    float g = green (img.pixels[loc]);
    float b = blue (img.pixels[loc]);
    // Change brightness according to the mouse here
    float adjustBrightness = ((float) mouseX / width) * 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 and set pixel in the window
    color c = color(r,g,b);
    pixels[loc] = c;
  }
}
```

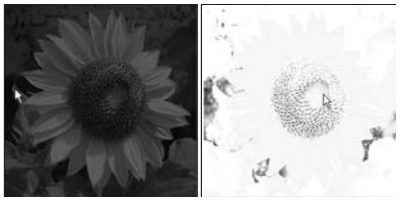


fig. 15.10

We calculate a multiplier ranging from 0.0 to 8.0 based on ***mouseX*** position. 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 we are altering the image on a per pixel basis, all pixels need not be treated equally. For example, we 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 R,G,B values from image
    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 adjustBrightness = (50-distance)/50;
    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,o,255);
    // Make a new color and set pixel in the window
    color c = color(r,g,b);
    pixels[loc] = c;
  }
}
```

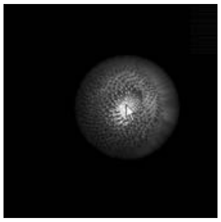


fig. 15.11

The closer the pixel is to the mouse, the lower the value of "distance" is. We want closer pixels to be brighter, however, so we invert the value with the formula:

adjustBrightness = (50-distance)/50

Pixels with a distance of 50 (or greater) have a brightness of 0.0 (or negative which is equivalent to 0 here) and pixels with a distance of 0 have a brightness of 1.0.

*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 our image processing examples have read every pixel from a source image and written a new pixel to the *Processing* window directly. However, it is often more convenient to write the new pixels to a destination image (that you then display using the ***image()*** function). We 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, we color the pixel white, less than, black. Example 15-10 uses an arbitrary threshold of 100.

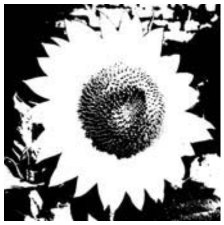


fig. 15.12

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;

  // We are 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
      }
    }
  }

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

We need two images, 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.

We have to display the destination image!

Exercise 15-9: Tie the threshold to mouse location.

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 filter():

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

The ***filter()*** function offers a set of prepackaged filters for the display window. It is 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.

15.9 Level II: Pixel Group Processing

In previous examples, we have seen a one-to-one relationship between source pixels and destination pixels. To increase an image's brightness, we 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, we 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 we know the 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];
```

We 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 we are looking at left neighbors
// We 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];
```

Reading the pixel to the left.

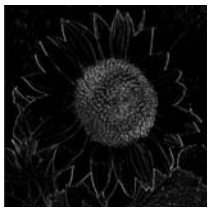


fig. 15.13

```
      // New color is difference between pixel and left neighbor
      float diff = abs(brightness(pix) - brightness(leftPix));
      pixels[loc] = color(diff);
    }
  }
```

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, we look 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.

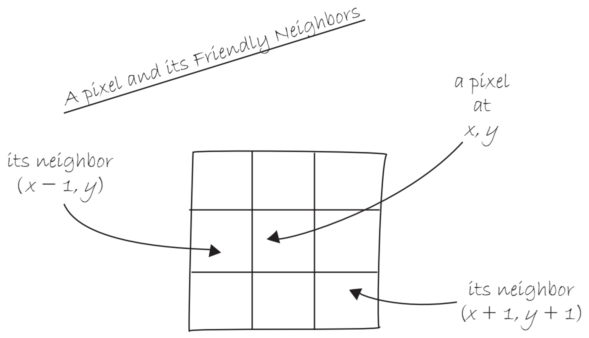

A pixel and its Friendly Neighbors
a pixel at x, y
its neighbor (x − 1, y)
its neighbor (x + 1, y + 1)

fig. 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, we "sharpen" an image 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 we have encountered in this book so far, since it involves so many elements (nested loops, 2D arrays, PImage pixels, etc.).

Example 15-13: Sharpen with convolution

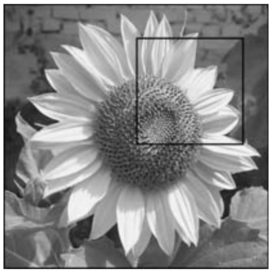


fig. 15.15

```
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() {
  // We're only going to process a portion of the image
  // so let's set the whole image as the background first
  image(img,0,0);
  // Where is the small rectangle we will process
  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;
  loadPixels();
  // Begin our loop 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 are we testing
```

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

In this example we are only processing a section of the image—an 80 × 80 rectangle around the mouse location.

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

```
        int xloc = x+i-offset;
        int yloc = y+j-offset;
        int loc = xloc + img.width*yloc;
        // Make sure we haven't walked off the edge of the pixel array
        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);
  }
}
```

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

We 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 our 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 we have achieved 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 us 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 we have in the past, we select colors from the pixels of a PImage object. The image itself is never displayed; rather, it serves as a database of information that we can exploit for our own creative pursuits.

In this first example, for every cycle through ***draw()***, we 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);
  smooth();
}

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
  loadPixels();
  float r = red(img.pixels[loc]);
  float g = green(img.pixels[loc]);
  float b = blue(img.pixels[loc]);
  noStroke();

  // Draw an ellipse at that location with that color
  fill(r,g,b,100);
  ellipse(x,y,pointillize,pointillize);
}
```

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

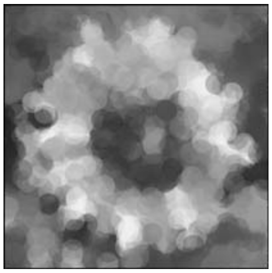


fig. 15.16

In this next example, we 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* location is determined by the brightness of the color. Brighter colors appear closer to the viewer and darker ones further away.

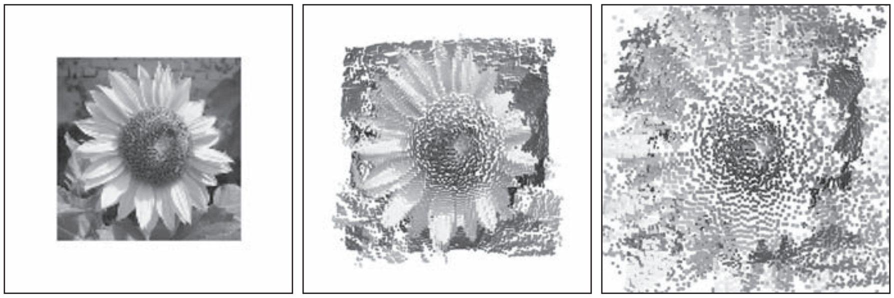


fig. 15.17

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 our 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(0);
  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 = (mouseX/(float)width) * brightness(img.pixels[loc])- 100.0;
  // Translate to the location, set fill and stroke, and draw the rect
  pushMatrix();
    translate(x,y,z);
    fill(c);
    noStroke();
    rectMode(CENTER);
    rect(0,0,cellsize,cellsize);
    popMatrix();
    }
  }
}
```

A ***z*** value for 3D translation is calculated as a function of the pixel's brightness as well as mouse 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.