13/Extend

This book focuses on using Processing for interactive graphics, because that's the core of what Processing does. However, the software can do much more and is often part of projects that move beyond a single computer screen. For example, Processing has been used to control machines, create images used in feature films, and export models for 3D printing.

Over the last decade, Processing has been used to make music videos for Radiohead and R.E.M., to create illustrations for publications such as *Nature* and the *New York Times*, to output sculptures for gallery exhibitions, to control huge video walls, to knit sweaters, and much more. Processing has this flexibility because of its system of libraries.

A Processing *library* is a collection of code that extends the software beyond its core functions and classes. Libraries have been important to the growth of the project, because they let developers add new features quickly. As smaller, self-contained projects, libraries are easier to manage than if these features were integrated into the main software.

To use a library, select Import Library from the Sketch menu and select the library you want to use from the list. Choosing a library adds a line of code that indicates that the library will be used with the current sketch.

For instance, when the PDF Export Library (pdf) is added, this line of code is added to the top of the sketch:

```
import processing.pdf.*;
```

In addition to the libraries included with Processing (these are called the *core* libraries), there are over 100 *contributed* libraries that are linked from the Processing website. All libraries are listed online at *http://processing.org/reference/libraries/*.

Before a contributed library can be imported through the Sketch menu, it must be added through the *Library Manager*. Select the Import Library option from the Sketchbook menu and then select Add Library to open the Library Manager interface. Click a library description and then click the Install button to download it to your computer.

The downloaded files are saved to the *libraries* folder that is located in your sketchbook. You can find the location of your sketchbook by opening the Preferences. The Library Manager can also be used to update and remove libraries.

As mentioned before, there are more than 100 Processing libraries, so they clearly can't all be discussed here. We've selected a few that we think are fun and useful to introduce in this chapter.

Sound

The *Sound audio library* introduced with Processing 3.0 has the ability to play, analyze, and generate (synthesize) sound. This library needs to be downloaded with the Library Manager as described earlier. (It's not included with the main Processing download because of its size.)

Like the images, shape files, and fonts introduced in Chapter 7, a *sound file* is another type of media to augment a Processing sketch. Processing's Sound library can load a range of file formats including WAV, AIFF, and MP3. Once a sound file is loaded, it can be played, stopped, and looped, or even distorted using different "effects" classes.

Example 13-1: Play a Sample

The most common use of the Sound library is to play a sound as background music or when an event happens on screen. The following example builds on Example 8-5 on page 107 to play a sound when the shape hits the edges of the screen. The *blip.wav* file is included in the *media* folder that you downloaded in Chapter 7 from *http://www.processing.org/learning/books/media.zip*.

As with other media, the `SoundFile` object is defined at the top of the sketch, it's loaded within `setup()`, and after that, it can be used anywhere in the program:

```
import processing.sound.*;

SoundFile blip;

int radius = 120;
float x = 0;
float speed = 1.0;
int direction = 1;

void setup() {
  size(440, 440);
  ellipseMode(RADIUS);
  blip = new SoundFile(this, "blip.wav");
  x = width/2; // Start in the center
}

void draw() {
  background(0);
  x += speed * direction;
  if ((x > width-radius) || (x < radius)) {
    direction = -direction; // Flip direction
    blip.play();
  }
  if (direction == 1) {
    arc(x, 220, radius, radius, 0.52, 5.76); // Face right
  } else {
    arc(x, 220, radius, radius, 3.67, 8.9); // Face left
  }
}
```

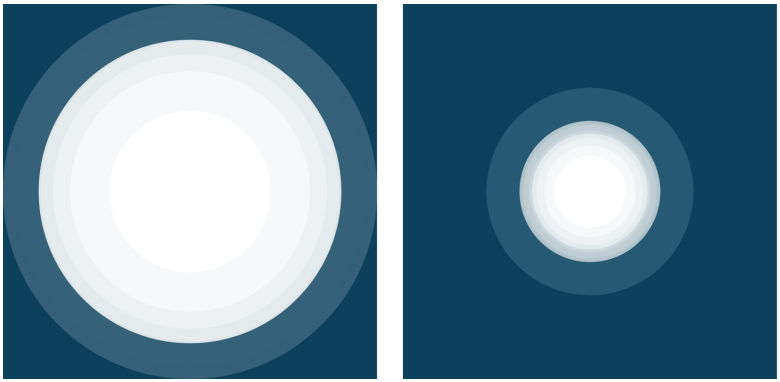
The sound is triggered each time its `play()` method is run. This example works well because the sound is only played when the

value of the `x` variable is at the edges of the screen. If the sound were played each time through `draw()`, the sound would restart 60 times each second and wouldn't have time to finish playing. The result is a rapid clipping sound. To play a longer sample while a program runs, call the `play()` or `loop()` method for that sound inside `setup()` so the sound is triggered only a single time.

The `SoundFile class` has many methods to control how a sound is played. The most essential are `play()` to play the sample a single time, `loop()` to play it from beginning to end over and over, `stop()` to halt the playback, and `jump()` to move to a specific moment within the file.

Example 13-2: Listen to a Microphone

In addition to playing a sound, Processing can *listen*. If your computer has a microphone, the Sound library can read live audio through it. Sounds from the mic can be analyzed, modified, and played:



```
import processing.sound.*;

AudioIn mic;
Amplitude amp;

void setup() {
```

```
  size(440, 440);
  background(0);
  // Create an audio input and start it
  mic = new AudioIn(this, 0);
  mic.start();
  // Create a new amplitude analyzer and patch into input
  amp = new Amplitude(this);
  amp.input(mic);
}


void draw() {
  // Draw a background that fades to black
  noStroke();
  fill(26, 76, 102, 10);
  rect(0, 0, width, height);
  // The analyze() method returns values between 0 and 1,
  // so map() is used to convert the values to larger numbers
  float diameter = map(amp.analyze(), 0, 1, 10, width);
  // Draw the circle based on the volume
  fill(255);
  ellipse(width/2, height/2, diameter, diameter);
}
```

There are two parts to getting the *amplitude* (volume) from an attached microphone. The `AudioIn class` is used to get the signal data from the mic and the `Amplitude class` is used to measure the signal. Objects from both classes are defined at the top of the code and created inside `setup()`.

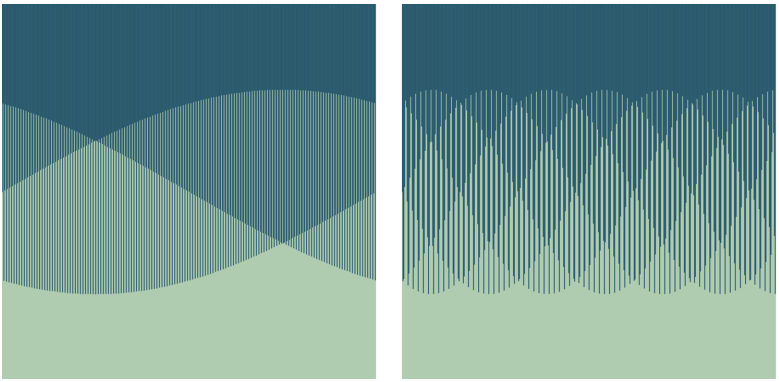
After the `Amplitude` object (named `amp` here) is made, the `AudioIn` object (named `mic`) is patched into the `amp` object with the `input()` method. After that, the `analyze()` method of the `amp` object can be run at any time to read the amplitude of the microphone data within the program. In this example, that is done each time through `draw()` and that value is then used to set the size of the circle.

In addition to playing a sound and analyzing sound as demonstrated in the last two examples, Processing can synthesize sound directly. The fundamentals of sound synthesis are waveforms that include the sine wave, triangle wave, and square wave.

A sine wave sounds smooth, a square wave is harsh, and a triangle wave is somewhere between. Each wave has a number of properties. The *frequency*, measured in hertz, determines the pitch, the highness or lowness of the tone. The *amplitude* of the wave determines the volume, the degree of loudness.

Example 13-3: Create a Sine Wave

In the following example, the value of `mouseX` determines the frequency of a sine wave. As the mouse moves left and right, the audible frequency and corresponding wave visualization increase and decrease:



```
import processing.sound.*;

SinOsc sine;

float freq = 400;

void setup() {
  size(440, 440);
  // Create and start the sine oscillator
  sine = new SinOsc(this);
  sine.play();
}

void draw() {
  background(176, 204, 176);
  // Map the mouseX value from 20Hz to 440Hz for frequency
  float hertz = map(mouseX, 0, width, 20.0, 440.0);
  sine.freq(hertz);
```

```
    // Draw a wave to visualize the frequency of the sound
    stroke(26, 76, 102);
    for (int x = 0; x < width; x++) {
      float angle = map(x, 0, width, 0, TWO_PI * hertz);
      float sinValue = sin(angle) * 120;
      line(x, 0, x, height/2 + sinValue);
    }
  }
```

The `sine` object, created from the `SinOsc class`, is defined at the top of the code and then created inside `setup()`. Like working with a sample, the wave needs to be played with the `play()` method to start generating the sound. Within `draw()`, the `freq()` method continuously sets the frequency of the waveform based on the left-right position of the mouse.

Image and PDF Export

The animated images created by a Processing program can be turned into a file sequence with the `saveFrame()` function. When `saveFrame()` appears at the end of `draw()`, it saves a numbered sequence of TIFF-format images of the program's output named *screen-0001.tif*, *screen-0002.tif*, and so on, to the sketch's folder.

These files can be imported into a video or animation program and saved as a movie file. You can also specify your own filename and image file format with a line of code like this:

```
saveFrame("output-####.png");
```

Use the **#** (hash mark) symbol to show where the numbers will appear in the filename. They are replaced with the actual frame numbers when the files are saved. You can also specify a subfolder to save the images into, which is helpful when working with many image frames:

```
saveFrame("frames/output-####.png");
```

When using `saveFrame()` inside `draw()`, a new file is saved each frame—so watch out, as this can quickly fill your *sketch* folder with thousands of files.

Example 13-4: Saving Images

This example shows how to save images by storing enough frames for a two-second animation. It loads and moves the robot file from "Robot 5: Media" on page 101. See Chapter 7 for instructions for downloading the file *robot1.svg* and adding it to the sketch.

The example runs the program at 30 frames per second and then exits after 60 frames:

```
PShape bot;
float x = 0;

void setup() {
  size(720, 480);
  bot = loadShape("robot1.svg");
  frameRate(30);
}

void draw() {
  background(0, 153, 204);
  translate(x, 0);
  shape(bot, 0, 80);
  saveFrame("frames/SaveExample-####.tif");
  x += 12;

  if (frameCount > 60) {
    exit();
  }
}
```

Processing will write an image based on the file extension that you use (*.png*, *.jpg*, or *.tif* are all built in, and some platforms may support others). To retrieve the saved files, go to Sketch → Show Sketch Folder.

A *.tif* image is saved uncompressed, which is fast but takes up a lot of disk space. Both *.png* and *.jpg* will create smaller files, but because of the compression they will usually require more time to save, making the sketch run slowly.

If your desired output is vector graphics, you can write the output to PDF files for higher resolution. The PDF Export library makes it possible to write PDF files directly from a sketch. These vector graphics files can be scaled to any size without losing resolution, which makes them ideal for print output—from posters and banners to entire books.

Example 13-5: Draw to a PDF

This example builds on Example 13-4 on page 190 to draw more robots, but it removes the motion. The PDF library is imported at the top of the sketch to extend Processing to be able to write PDF files.

This sketch creates a PDF file called *Ex-13-5.pdf* because of the third and fourth parameters to `size()`:

```
import processing.pdf.*;

PShape bot;

void setup() {
  size(600, 800, PDF, "Ex-13-5.pdf");
  bot = loadShape("robot1.svg");
}

void draw() {
  background(0, 153, 204);
  for (int i = 0; i < 100; i++) {
    float rx = random(-bot.width, width);
    float ry = random(-bot.height, height);
    shape(bot, rx, ry);
  }
  exit();
}
```

The geometry is not drawn on the screen; it is written directly into the PDF file, which is saved into the sketch's folder. The code in this example runs once and then exits at the end of `draw()`. The resulting output is shown in Figure 13-1.

There are more PDF Export examples included with the Processing software. Look in the *PDF Export* (pdf) section of the Processing examples to see more techniques.

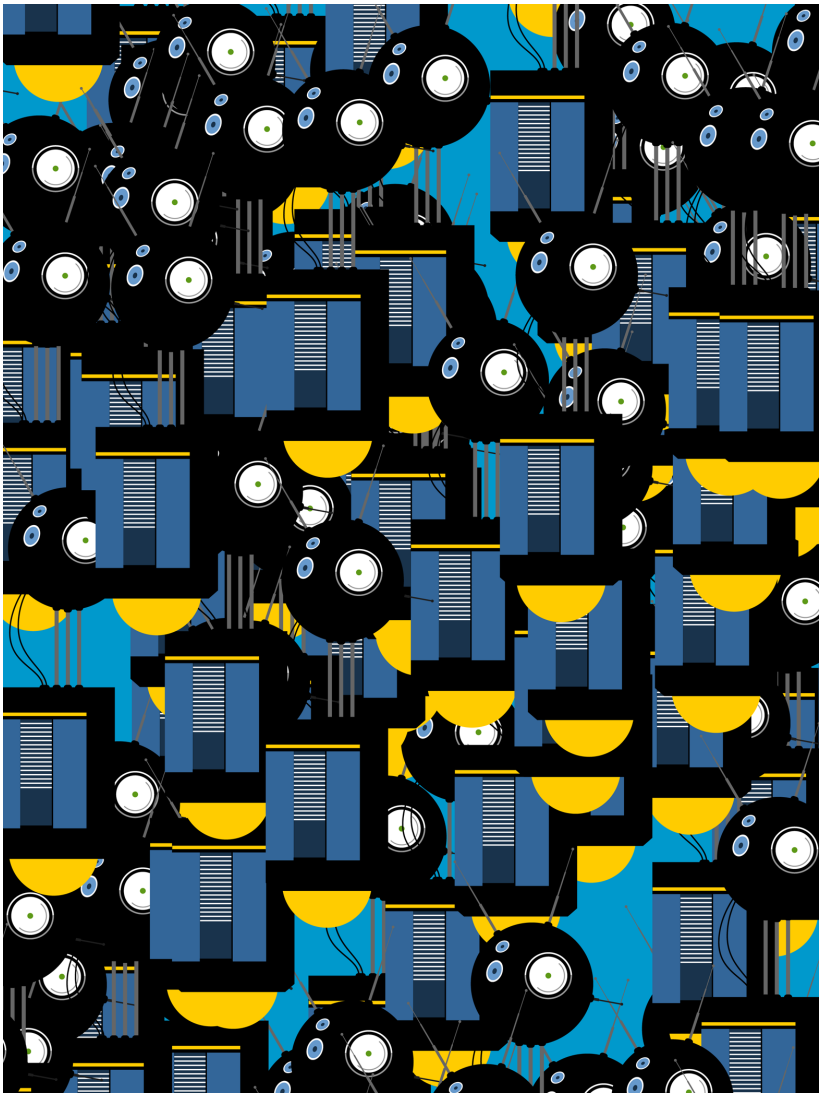


Figure 13-1. *PDF export from Example 3-5*

Hello, Arduino

Arduino is an electronics prototyping platform with a series of microcontroller boards and the software to program them. Processing and Arduino share a long history together; they are sister projects with many similar ideas and goals, though they address separate domains. Because they share the same editor and programming environment and a similar syntax, it's easy to move between them and to transfer knowledge about one into the other.

In this section, we focus on reading data into Processing from an Arduino board and then visualize that data on screen. This makes it possible to use new inputs into Processing programs and to allow Arduino programmers to see their sensor input as graphics. These new inputs can be anything that attaches to an Arduino board. These devices range from a distance sensor to a compass or a mesh network of temperature sensors.

This section assumes that you have an Arduino board and that you already have a basic working knowledge of how to use it. If not, you can learn more online at *http://www.arduino.cc* and in the excellent book *Getting Started with Arduino* by Massimo Banzi (Maker Media). Once you've covered the basics, you can learn more about sending data between Processing and Arduino in another outstanding book, *Making Things Talk* by Tom Igoe (Maker Media).

Data can be transferred between a Processing sketch and an Arduino board with some help from the Processing Serial Library. *Serial* is a data format that sends one *byte* at a time. In the world of Arduino, a `byte` is a data type that can store values between 0 and 255; it works like an `int`, but with a much smaller range. Larger numbers are sent by breaking them into a list of bytes and then reassembling them later.

In the following examples, we focus on the Processing side of the relationship and keep the Arduino code simple. We visualize the data coming in from the Arduino board one `byte` at a time. With the techniques covered in this book and the hundreds of Arduino examples online, we hope this will be enough to get you started.

Example 13-6: Read a Sensor

The following Arduino code is used with the next three Processing examples:

```
// Note: This is code for an Arduino board, not Processing

int sensorPin = 0;  // Select input pin
int val = 0;

void setup() {
  Serial.begin(9600);  // Open serial port
}

void loop() {
  val = analogRead(sensorPin) / 4;  // Read value from sensor
  Serial.write((byte)val);  // Print variable to serial port
  delay(100);  // Wait 100 milliseconds
}
```

There are two important details to note about this Arduino example. First, it requires attaching a sensor into the analog input on pin 0 on the Arduino board. You might use a light sensor (also called a photo resistor, photocell, or light-dependent resistor) or another analog resistor such as a thermistor (temperature-sensitive resistor), flex sensor, or pressure sensor (force-sensitive resistor). The circuit diagram and drawing of the breadboard with components are shown in Figure 13-2. Next, notice that the value returned by the `analogRead()` function is divided by 4 before it's assigned to `val`. The values from `analogRead()` are between 0 and 1023, so we divide by 4 to convert them to the range of 0 to 255 so that the data can be sent in a single `byte`.

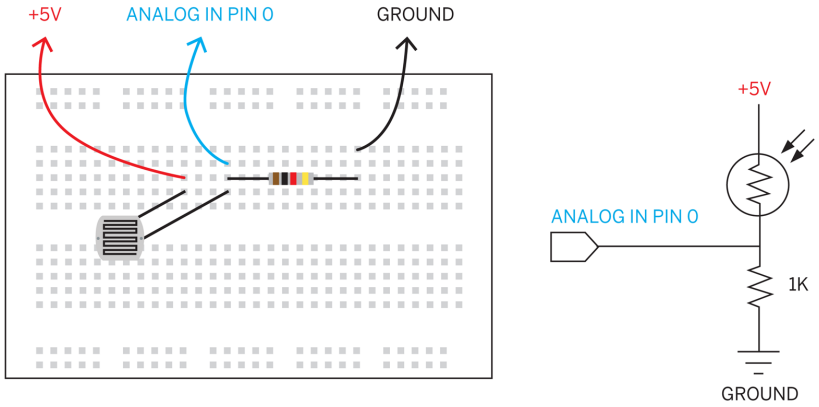


Figure 13-2. *Attaching a light sensor (photo resistor) to analog in pin 0*

Example 13-7: Read Data from the Serial Port

The first visualization example shows how to read the serial data in from the Arduino board and how to convert that data into the values that fit to the screen dimensions:

```
import processing.serial.*;

Serial port;  // Create object from Serial class
float val;  // Data received from the serial port

void setup() {
  size(440, 220);
  // IMPORTANT NOTE:
  // The first serial port retrieved by Serial.list()
  // should be your Arduino. If not, uncomment the next
  // line by deleting the // before it. Run the sketch
  // again to see a list of serial ports. Then, change
  // the 0 in between [ and ] to the number of the port
  // that your Arduino is connected to.
  //printArray(Serial.list());
  String arduinoPort = Serial.list()[0];
  port = new Serial(this, arduinoPort, 9600);
}
```

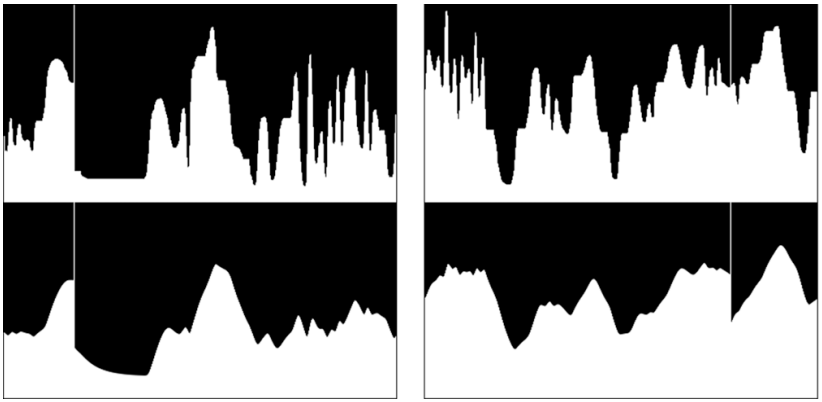
```
void draw() {
  if (port.available() > 0) {       // If data is available,
    val = port.read();         // read it and store it in val
    val = map(val, 0, 255, 0, height); // Convert the value
  }
  rect(40, val-10, 360, 20);
}
```

The Serial library is imported on the first line and the serial port is opened in `setup()`. It may or may not be easy to get your Processing sketch to talk with the Arduino board; it depends on your hardware setup. There is often more than one device that the Processing sketch might try to communicate with. If the code doesn't work the first time, read the comment in `setup()` carefully and follow the instructions.

Within `draw()`, the value is brought into the program with the `read()` method of the Serial object. The program reads the data from the serial port only when a new `byte` is available. The `available()` method checks to see if a new `byte` is ready and returns the number of bytes available. This program is written so that a single new `byte` will be read each time through `draw()`. The `map()` function converts the incoming value from its initial range from 0 to 255 to a range from 0 to the height of the screen; in this program, it's from 0 to 220.

Example 13-8: Visualizing the Data Stream

Now that the data is coming through, we'll visualize it in a more interesting format. The values coming in directly from a sensor are often erratic, and it's useful to smooth them out by averaging them. Here, we present the raw signal from the light sensor illustrated in the top half of the example and the smoothed signal in the bottom half:



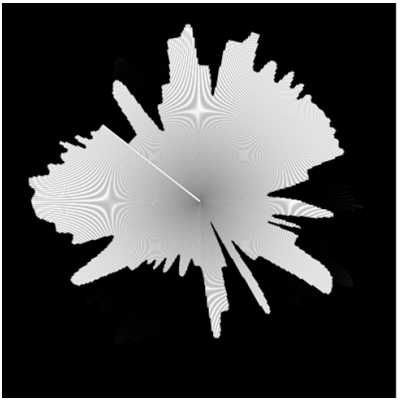
```
import processing.serial.*;

Serial port;  // Create object from Serial class
float val;    // Data received from the serial port
int x;
float easing = 0.05;
float easedVal;

void setup() {
  size(440, 440);
  frameRate(30);
  String arduinoPort = Serial.list()[0];
  port = new Serial(this, arduinoPort, 9600);
  background(0);
}

void draw() {
  if ( port.available() > 0) { // If data is available,
    val = port.read();         // read it and store it in val
    val = map(val, 0, 255, 0, height/2); // Convert the values
  }
  float targetVal = val;
  easedVal += (targetVal - easedVal) * easing;

  stroke(0);
  line(x, 0, x, height);            // Black line
  stroke(255);
  line(x+1, 0, x+1, height);        // White line
  line(x, 220, x, val);             // Raw value
  line(x, 440, x, easedVal + 220);  // Averaged value

  x++;
```

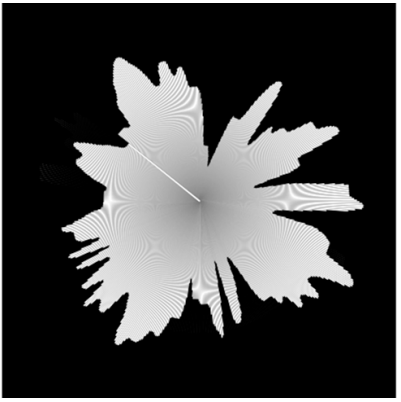
```
    if (x > width) {
      x = 0;
    }
  }
```

Similar to Example 5-8 on page 54 and Example 5-9 on page 55, this sketch uses the easing technique. Each new `byte` from the Arduino board is set as the target value, the difference between the current value and the target value is calculated, and the current value is moved closer to the target. Adjust the `easing` variable to affect the amount of smoothing applied to the incoming values.

Example 13-9: Another Way to Look at the Data

This example is inspired by radar display screens. The values are read in the same way from the Arduino board, but they are visualized in a circular pattern using the `sin()` and `cos()` functions introduced earlier in Example 8-12 on page 115, Example 8-13 on page 115, and Example 8-15 on page 116:





```
import processing.serial.*;

Serial port;  // Create object from Serial class
float val;    // Data received from the serial port
float angle;
float radius;

void setup() {
```

```
  size(440, 440);
  frameRate(30);
  strokeWeight(2);
  String arduinoPort = Serial.list()[0];
  port = new Serial(this, arduinoPort, 9600);
  background(0);
}

void draw() {
  if ( port.available() > 0) {  // If data is available,
    val = port.read();          // read it and store it in val
    // Convert the values to set the radius
    radius = map(val, 0, 255, 0, height * 0.45);
  }

  int middleX = width/2;
  int middleY = height/2;
  float x = middleX + cos(angle) * height/2;
  float y = middleY + sin(angle) * height/2;
  stroke(0);
  line(middleX, middleY, x, y);

  x = middleX + cos(angle) * radius;
  y = middleY + sin(angle) * radius;
  stroke(255);
  line(middleX, middleY, x, y);

  angle += 0.01;
}
```

The `angle` variable is updated continuously to move the line drawing the current value around the circle, and the `val` variable scales the length of the moving line to set its distance from the center of the screen. After one time around the circle, the values begin to write on top of the previous data.

We're excited about the potential of using Processing and Arduino together to bridge the world of software and electronics. Unlike the examples printed here, the communication can be bidirectional. Elements on screen can also affect what's happening on the Arduino board. This means you can use a Processing program as an interface between your computer and motors, speakers, lights, cameras, sensors, and almost anything else that can be controlled with an electrical signal. Again, check out *http://www.arduino.cc* for more information about Arduino.