1. Getting Started

This book explores the cutting-edge hardware and software features that are built into Android phones and tablets today. You'll create sophisticated graphics and user interfaces in no time, and you'll develop a range of projects that build on the hardware sensors, cameras, and networking capabilities of your Android device. You'll put them to work creatively to make your Android apps more useful, usable, and exciting. We'll consider Android phones and tablets as universal sensors, processors, and remote controls in the context of this book, and we'll create projects that go beyond the typical app. Along the way, we'll spark new app ideas for you to explore in the future.

You'll learn how to create advanced Android apps using Processing, a widely popular open source programming language and environment that is free to use and was designed for learning the fundamentals of programming. With more than 150 libraries expanding the Processing core, as well as the possibility to extend it with Java and Android classes and methods, it is a simple yet powerful language to work with. Processing comes with three modes that let you create applications for different devices and operating systems: Java mode lets us create standalone applications for GNU/Linux, Mac OS X, and Windows. Android mode in Processing enables us to create apps for Android phones and tablets---we'll use this mode throughout the book. And finally, JavaScript mode enables us to create web apps, and

those will run in all HTML5-enabled web browsers installed on smart phones, tablets, and desktop computers.

Processing's simple syntax lets you write apps whose sophisticated displays belie the straightforward, readable code in which they're written. Initially developed to serve as a software sketchbook for artists, designers, and hobbyists and to teach the fundamentals of computer programming in a more visual way, Processing is one of the most versatile production environments on the market today.

In 2010, the case for programming with Processing became even stronger with the addition of the Android mode to the Processing environment, whose intent, in the words of the Processing all-volunteer team, is to make it foolishly easy to create Android apps using the Processing API.

In this chapter, we'll begin by installing the software tools we'll need, and then we'll take a look at the basic structure of a typical Processing program, known as a sketch. We'll write our first Android sketch, one that draws figures on our desktop screen. Then we'll switch to Android mode without leaving the Processing IDE and run that same app on the built-in Android emulator. Finally, we'll load the sketch onto an actual Android device and run it there.

With an understanding of the basic development cycle in hand, we'll learn how to use the touch screen interface to add some interactivity to our sketch. We'll explore how it differs from a mouse pointer and make use of touch screen values to change the visual properties of the app, first with gradations of gray and then with color.

In addition to the traditional RGB (red, green, and blue) values that most programmers are familiar with, Processing provides additional color modes that provide greater control over hue, saturation, and brightness (HSB). As we work on our first apps, we'll take a closer look in particular at the HSB mode, which delivers all three.

Throughout the chapter we'll work with simple code that uses the Android touch screen sensor interface to change the position, color, and opacity of the 2D graphics it displays. Let's jump right in and install the software we need to develop Android apps in Processing.

Install the Required Software

Let's get started and download the software we'll need to develop Android apps. The Processing download comes in a fairly small package of approximately 165 MB. It consists of free open source software and is available from the Processing website without prior registration. For workshops, in the lab, in an office, or in a teaching environment where

multiple machines are in use, the lightweight Processing development environment (or "Processing IDE") is a good alternative to a full-featured integrated development environment (IDE) such as Eclipse.

The Processing IDE supports some of the advanced syntax highlighting and autocomplete features for which Eclipse is valued. Additionally, professional programmers appreciate the Processing IDE for its quick install. It comes with all the necessary tutorials and example sketches that allow us to explore specific programming topics right away. Processing does not require installation; just extract the application file and start.

What You Need

To implement the projects in this book, you'll need the following tools:

- Processing 3.0
- Java 8
- Android 4.0 Ice Cream Sandwich (2.3 Gingerbread is sufficient for all projects except Chapter 7, Peer-to-Peer Networking Using Bluetooth and Wi-Fi Direct, and Chapter 8, Using Near Field Communication (NFC).)

These are the minimum software requirements. If you have a newer version, you'll be just fine. Later we'll install some additional libraries that give us easier access to the features of an Android device. For now, use the following steps to build the core Processing environment we'll use throughout this book.

Install Processing for Android

Here are the steps to install Processing for the Android.

---

1. Download Processing 3.0 for your operating system (OSX, Windows, or Linux) at Processing.org/download. The Processing download includes the Processing IDE, a comprehensive set of examples and tutorials, and a language reference. The Processing package does not include the Android software development kit, which you'll need to download separately.
2. Extract the Processing application from the `.zip`file on Windows, `.dmg` file on Mac OS, or `.tar.gz` file on Linux, and move it to your preferred program folder (for example,

Applications if you are developing on OSX, `Program Files` on Windows, or your preferred `/bin` folder on Linux).

Install the Android SDK

1. Find and download the Android SDK by going to http://developer.android.com/sdk/. Choose the package that's right for your operating system and complete the installation of the Android SDK. On Windows, you may wish to download and use the installer that Android provides to guide your setup. If Java JDK is not present on your system, you will be prompted to download and install it.

2. When the Android SDK download is complete, go to the Processing wiki at http://android.processing.org/install.html and open the Android installation instructions you'll find there. Follow the instructions for your OS step by step. The wiki lists which components are required to configure Processing for Android on your operating system and tells you how to get Android installed properly. Android may have dependencies that are specific to your operating system, such as additional device drivers. If you are developing on Windows, follow the USB driver installation instructions available at http://developer.android.com/tools/extras/oem-usb.html. If you are developing on Linux, follow the instructions for setting up your device for development at http://developer.android.com/tools/device.html#setting-up.

   Now that you have installed all the necessary components to develop Android apps on your own system, let's jump right into Processing and write our first sketch.

Write Your First Android Sketch

Go ahead and launch Processing from the applications directory. The Processing IDE launches, opening an empty sketch window, as shown in Figure 1, The Processing IDE, below.

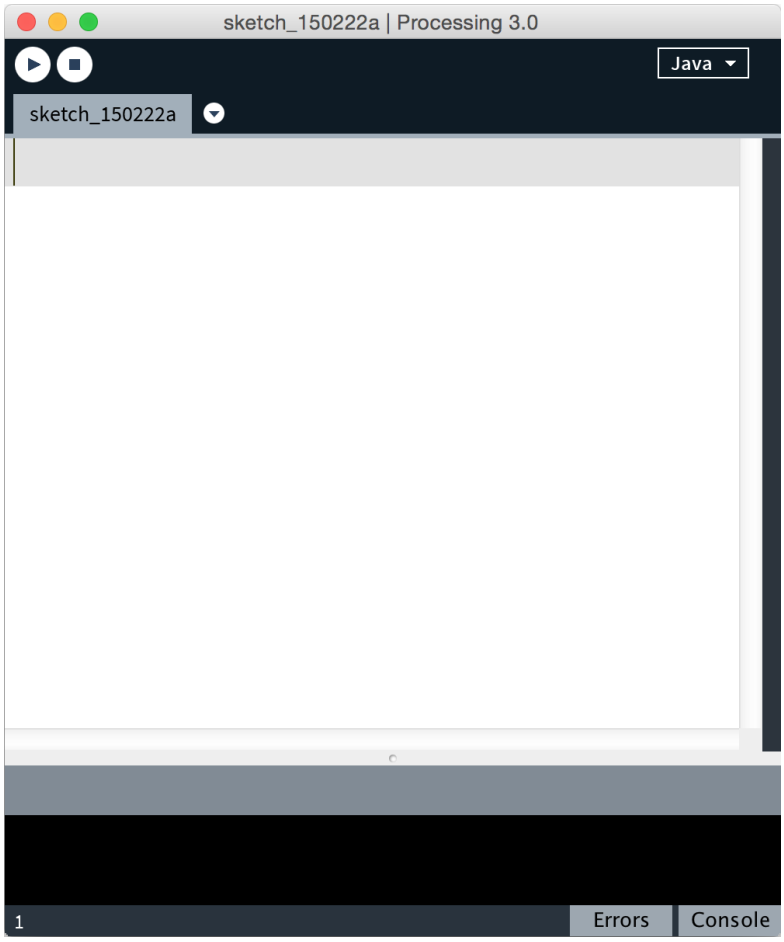


Figure 1.1 — The Processing IDE.

We edit Processing code directly within the Processing IDE sketch window, as shown here.

Since you've launched the application for the first time, Processing has just created a sketchbook folder for you, which is located in `Documents` on the hard drive, independent of the OS you are developing on. I recommend you save all your sketches to this location. Then Processing can list them for you within the IDE (click the "Open..." toolbar button). Also, when you update to future versions of Processing, the sketchbook loads up exactly the same way as before.

Explore the Processing IDE

The toolbar on top of the sketch window contains the key features of the IDE, with a Run button to launch and a Stop button to stop your apps. You can find a more detailed description of the sketchbook and the IDE in the Processing Development Environment tutorial on the Processing website.

When you start Processing for the first time, it defaults to Java mode, as indicated on the right side of the toolbar. This area also functions as a drop-down menu, allowing us to switch between the different modes the Processing IDE provides. You'll need to add the Android mode, choosing "Add mode…" from the menu. Depending on which mode you've selected, the Run button on the toolbar produce different results, which are listed next.

- Java mode "Run" displays a program window to view the sketch running on the desktop computer.
- Android mode "Run" launches the app on the Android device. "Export" creates a signed Android package for Google Play.
- JavaScript mode "Run" launches a web page in the default browser, with a Processing JavaScript canvas showing the sketch. "Export" creates a web package, including all dependent files for uploading to a web server.

A tab below the toolbar shows the current sketch name, which defaults to one containing the current date if the sketch has not been saved yet. Processing prompts us to provide another filename as soon as we save the sketch. The right-arrow button to the right of the tab allows us to add more tabs if we'd like to split the code into separate sections. As sketches grow in scope and complexity, the use of tabs can be a great way to reduce clutter by separating classes and methods for different purposes into distinct tabs. Each tab is saved as a separate Processing source file, or `pde`, in the sketch folder.
The text editor, shown in white below the tab in the image above, is the actual area where we write and edit code. The line number of our current cursor location within the code is shown at the very bottom of the sketch window.

The message area and console below the text editor provide us with valuable feedback as we develop and debug.

You can always find more information on the key IDE features discussed here, as well as a summary of the installation, on the Learning page of the Processing website.

Now that you know how to work with the Processing editor, you're almost ready to write your first sketch.

Understand the Structure of a Sketch

Any Processing sketch that will interact with users or make use of animated graphics–as is the case for all the sketches in this book–must include two methods:

- An instance of the `setup` method, which initializes key variables and settings the sketch will use and is executed only once when the app starts
- An instance of the `draw` method, which continuously updates or redraws the screen to respond to user input and real-time events

  If we redraw the screen fast enough, users will perceive individual images, or frames, as continuous movement. It's a principle of film and animation we have all experienced. A typical Processing sketch starts by defining the global variables it uses, followed by both `setup` and `draw` methods. `setup` is called exactly once when you start a sketch to initialize key parameters. For instance, we can set a particular window `size`or screen `orientation`, or we can load custom fonts and media assets. `setup` is responsible for taking care of everything we need to do once to configure a sketch.

  The `draw` method, in contrast, is called repeatedly to update the screen sixty times per second by default. We can adjust this rate using the `frameRate` method. If our drawings are complex or if they require substantial amounts of processor power to compute, Processing might not always be able to keep up with the 60 fps frame rate. We can always get some information on the current playback rate through the `frameRate`constant Processing provides to us. As a point of reference, cinema film runs at 24 fps and digital video typically at 30 fps.

  Neither `setup` nor `draw` accepts parameters. They are `void` methods and do not return values. Both are used in virtually every Processing sketch.

Write a Sketch

Let's now say "Hi" to Processing by creating a simple sketch that draws an ellipse repeatedly at the current cursor position. We'll add some complexity to its graphical output by having the ellipse expand or contract along its vertical and horizontal axes, depending on how fast the mouse moves across the screen. This basic drawing sketch, shown in Figure 2, gives us immediate visual feedback and uses your mouse as input. As you move along, experiment and play with parameter values to better understand them.

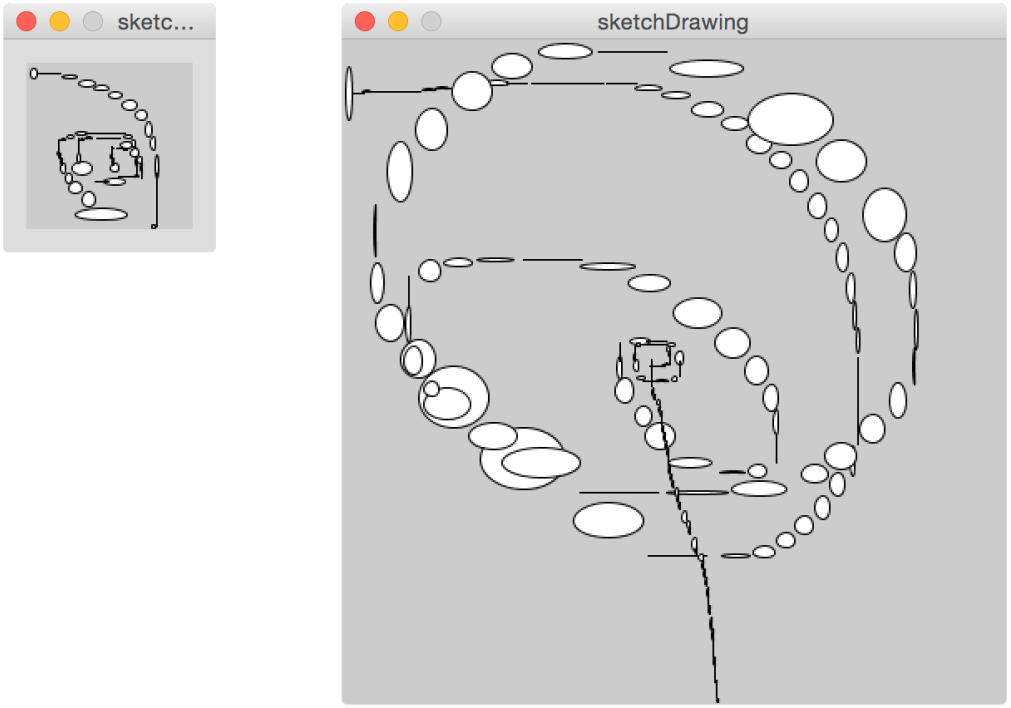


Figure 1.2 — A simple sketch.

With the ellipse-drawing primitive, Processing can generate dynamic output. On the left is a 100 x 100 pixel window; on the right, a 400 x 400 pixel window.

We use a single drawing primitive for this sketch, the `ellipse`, also used to draw circles by providing equal width and height for the ellipse. In Processing, an `ellipse(x, y, width, height)` requires four parameters:

- The horizontal `x` position of the ellipse center
- The vertical `y` position of the ellipse center
- The ellipse `width`
- The ellipse `height`

The following snippet contains the code we'll need for our sketch.

```
void setup()
{
}

void draw()
{
    ellipse(mouseX, mouseY, mouseX-pmouseX, mouseY-pmouseY);
}
```

Go ahead and type this into the text editor, as illustrated below:

We want the position of the ellipse to follow the mouse, and for this we need to know where it's located at any given moment. Processing stores this information in two system defined variables: `mouseX` and `mouseY`. The pair returns the `x` and `y` coordinates of the mouse in pixels relative to the origin of the display window, not the computer screen. In Processing, the origin of the display window (`[0, 0]`) is located at the upper left corner of the device window; `[width-1, height-1]` is located at the lower right.

We'll use `mouseX` and `mouseY` to set the horizontal and vertical position of the ellipse center. For the `width`and `height` parameters of the ellipse, we'll use two additional system defined variables: `pmouseX` and `pmouseY`. `pmouseX` and `pmouseY` store the previous mouse position from one frame ago.

If we move the mouse, we can calculate the mouse speed by subtracting the previous from the current mouse position. By subtracting `mouseX` from `pmouseX`, we determine the horizontal mouse travel distance in pixels within one frame, or for one-sixtieth of a second. We use the same approach for the vertical trajectory by subtracting `pmouseY` from `mouseY` for the vertical speed. `pmouseX` and `pmouseY` are lesser-known system defined variables and more rarely used than `mouseX` and `mouseY`, but they're very useful when we are interested in the speed of the mouse.

Run the Sketch

Go ahead and run the sketch by pressing the Play button. Processing will open a display window whose default size is 100 by 100 pixels, as shown in the image above. Alternatively, you can select Sketch ↦ Run on the Processing menu bar. When the window appears, place your mouse pointer there and move it around.

If we move the mouse quickly from left to right or up and down inside the display window, the ellipse width and height increase, depending on where we are heading. Drawing in such a small sketch window restricts our mouse movement, so let's use the `size` method to increase the window size to `[400, 400]`, as shown in Figure 2. We add the `size` method to `setup`, because we need to define the window size only once when we start up the sketch. In a typical Processing sketch, the idea is to keep everything strictly away from `draw` so the application doesn't get bogged down executing extra statements at the rate of sixty times per second.
Go ahead and add the following statements to `setup` in your Processing text editor:

```
size(400, 400);
```

Now rerun the sketch. With a bit more pixel real estate (400 x 400px), we now have the space to build up some speed.

Save the Sketch

Let's finish by saving the sketch as `basicDrawing.pde` into the `Processing` sketchbook, located in `Documents` on the hard drive. When the sketch is saved, the tab is renamed to the current sketch name. Press Open in the toolbar and see your sketch listed at the top in the sketchbook.
You've just completed your first Processing sketch in Java mode. Time now to make an Android app from the sketch and run it in the Android emulator.

Run a Sketch in the Android Emulator

 Let's now switch our basic drawing sketch Figure 2, to Android mode. Click "Java" in the upper right corner of the Processing IDE and use the drop-down menu to add the Android mode. Click "Add mode..." and install the "Android Mode" from the list. Close the Mode Manager and use the drop-down menu to switch to "Android" mode. The structure and statements of a Processing sketch are identical across modes. So there is nothing we need to change in our sketch to run it on an Android.

Run the App

To run the sketch in the emulator, select Sketch ↦ "Run in Emulator" from the Processing menu.

The following lines should appear in the console area at the bottom of the IDE when the Android emulator launches for the first time:

```
Building Android project...
Waiting for device to become available...
```

```
Installing sketch on emulator.
Stating Sketch on emulator.
Sketch launched on the emulator.
```

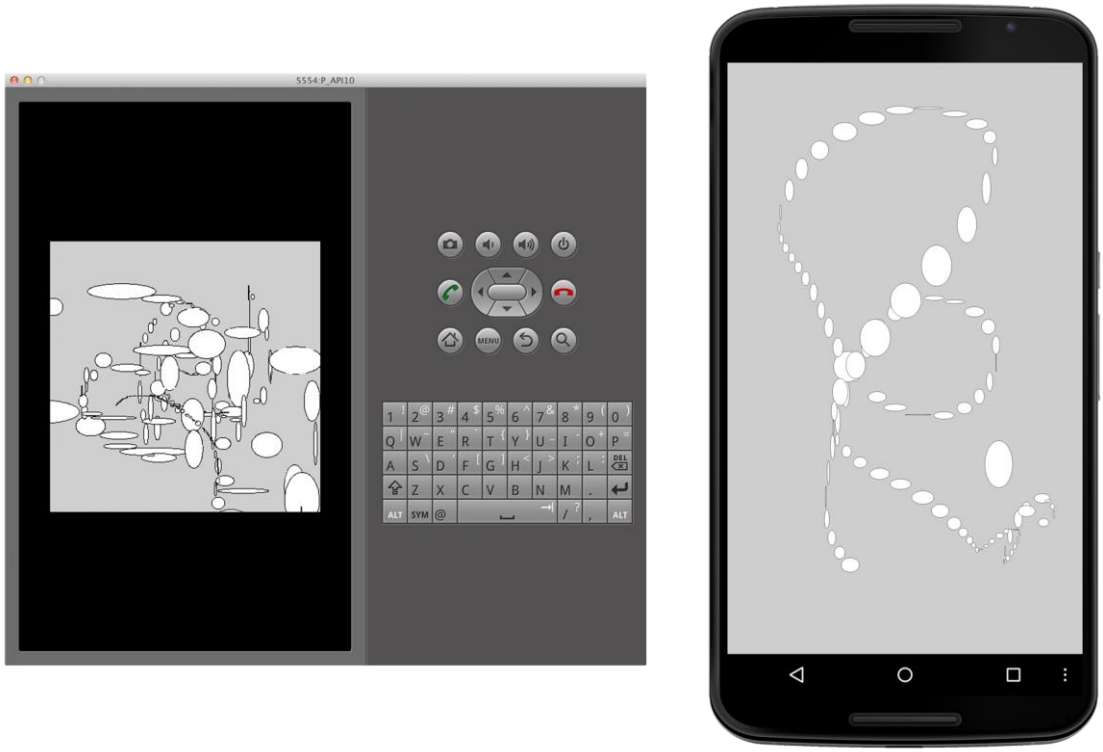


Figure 1.3 — Running the drawing sketch in Android mode.

We can run a sketch in either the Android emulator installed on our desktop computer (left) or directly on the Android device (right).

The emulator starts up the Android OS exactly as a device would, just a bit more slowly. Once Android is running, Processing then installs the Android package (`apk`) and launches the sketch. If it takes too long for the emulator to start up, Processing might time out and you might have to relaunch your sketch one more time. The sketch looks identical to the Java mode sketch illustrated in Figure 3, and if you move the mouse in the emulator window, it responds the same way it did in Java mode. The frame rate is noticeably lower, and the screen has a different aspect ratio. In the emulator, the mouse methods are a stand-in for the touch screen interface. Don't close the Android emulator as you keep testing your sketches.
Here's the first Android app for you. Congratulations! If you get an error, please jump to Troubleshooting.

As you can see, the emulator is a good way to check whether your sketch is operational. When it comes to testing responsiveness or the touch screen user experience of your app, however, we need an actual device. So next up is testing the sketch on an actual Android device.

Run a Sketch on an Android Device

Let's run the sketch on a real Android device. First you'll need to check which version of Android it's running. Then you'll need to connect the device to your computer with a USB cable and enable USB debugging.

To determine which version of Android you're running, go to the home screen, tap Settings and then tap "About phone" (or "About phone/tablet") at the bottom of the menu that appears. Look for the version of Android that is installed on your device under "Android version" and make a note. The procedure for enabling USB debugging depends on that number. Now connect your device to your desktop.

- If you're running version 4.2 Jelly Bean or a more recent version of Android, activate the hidden "Developer options" menu by tapping "Build number" at the bottom of the "About phone/tablet" menu five times. Then navigate to the Settings ↦ "Developer options" menu and check "USB Debugging." To respond to the warning that "USB debugging is intended for development purposes only," tap OK.
- If you're running version 4.1 Jelly Bean or an earlier version of Android, tap Settings and look for the "Developer options" menu under the System section at the bottom of the list. Click OK after the warning and check "USB Debugging" at the top of the list that appears.

If you are developing on Windows or Linux, you'll need a special USB device driver. You can find instructions on the Android website for downloading and installing it.

With all the software tools in place and the Android device plugged into your desktop, let's run our basic drawing example on the device. To run the sketch fullscreen, let's comment out `size` in `setup`, so the app covers the full screen of the Android device.

```
// size(400, 400);
```

Run the App

Choose Run on Device from the Sketch menu, or use the shortcut `⌘R` on a Mac or `Ctrl R` on Windows or Linux. The shortcut is a timesaver when we're testing and running a sketch frequently. Processing compiles the sketch and produces a `basicDrawing.apk` package, which it then moves onto the device and launches.

```
Building Android project...
Waiting for device to become available...
Installing sketch on DEVICE_ID.
Starting Sketch on DEVICE_ID.
Sketch launched on the device.
```

Your sketch is now up and running on your Android device. In contrast to running the sketch in the emulator, installing and launching it on an Android device goes fairly quickly. If you play with your sketch on the touch screen, you'll be able to tell how much more responsive the device is. This is because the device provides a higher frame rate. Being able to test the actual

user experience more accurately while saving some time rerunning your sketch are the main reasons why testing on the Android device is preferable.

Let's also note what we are not doing! We are not signing up as a developer. We are not installing certificates, and we haven't used a credit card. Processing and Android are open platforms. The apps we create can be shared directly with other Android users. And if we intend to further develop our project in Eclipse or collaborate with Eclipse developers, Processing provides us with an Export Android Project option, which you can find on the Processing menu toolbar under File. This command will create a folder containing all the necessary Eclipse project files.

Wrapping Up

Let's summarize. You've seen how easy it is to use Processing to create graphics for the Android. You've downloaded the Processing IDE and the Android SDK, and you've installed the software tools to develop Android apps in Processing. You've written your first Processing sketch using the mouse as input, and you are now able to switch to Android mode and write basic Android apps in Processing. The Android emulator that comes with the Android SDK and the Android device you've used in this chapter will help you test the apps you'll write throughout the book.

In the next chapter, we'll work with the touch screen interface and device display. You'll learn how to work with color, use fingertip pressure on the screen surface, and work with multitouch gestures such as tap and pinchto manipulate the graphical elements on the device display.

2. Working with the Touch Screen Display

Now that we've completed our first Android app, let's explore a device feature that has become particularly popular with mobile phones and tablets—multitouch. Virtually all Android devices ship today with a capacitive touch screen panel. It's a device we've gotten so accustomed to that we hardly "see" it as the hardware sensor that it is.

User interaction (UI) with Android touch screens differs somewhat from that of a mouse on a traditional computer display. First of all, we don't have one omnipresent mouse pointer for interacting with UI elements via rollovers, clicks, right-clicks, and double-clicks. In fact, we don't have a rollover or a physical "click" on the touch screen panel at all, hence UI interactions often require adjustments for the touch screen. Typically the Android device uses audiovisual cues such as click sounds or small device vibrations for user feedback.

There are a number of advantages to the multitouch screen interface to point out. First and foremost, the capacitive touch screen panel affords us more than one mouse pointer. We can work with two, five, even ten fingers on the Android, although more than three are rarely used. Multitouch allows us a variety of distinct finger gestures compared to the mouse, which we can only use to interact with the UI elements and other components displayed on the screen. The two most common multitouch gestures are the pinch and rotate gestures, typically used for scaling and rotating objects on the screen.

In this chapter, we'll get started by learning to use the mouse callback methods available in Processing for Android. Then we'll dive into the different color modes Processing has to offer, an essential topic that we need to address to work with graphics and images throughout the book. Building on the basic drawing sketch, we'll use the mouse speed to manipulate the hues of the ellipses we draw.

Finally, we'll dedicate the second part of the chapter to the multitouch features of the Android touch screen and create a sketch that showcases the most common gestures, including the tap, double-tap, long press, flick, pinch, and rotate gestures. In the sketch we'll develop, we'll manipulate the scale, position, rotation, and color of a rectangle using multitouch gestures.

To make working with multitouch gestures easy, we'll use the Ketai library for Processing, which greatly simplifies the process. We'll work with Ketai throughout the book, as it also simplifies working with sensors, cameras, location, and networking—all the

hardware features that are typically difficult to work with. We'll download and install the library step by step and take a quick look at the main Ketai classes.

Let's take a look at how the touch screen panel works.

Introducing the Android Touch Screen

The capacitive touch screen panel of an Android device consists of a glass insulator coated with a transparent conductor. When we interact with the touch screen surface, our fingertips act as electrical conductors—not very good ones, but good enough to be detected. A touch on the screen surface distorts the electrostatic field, causing a change in its electric capacitance, which can be located relative to the screen surface. The horizontal and vertical position of the fingertip relative to the screen is then made available to us through the Android OS; it is updated only when we touch or move a fingertip across the screen.

The apps we write in Processing have a flexible screen orientation by default, which means our app switches orientation automatically from portrait to landscape depending on how we are holding the phone or tablet–this is detected by the accelerometer sensor we'll get to know in Display Values from the Accelerometer. We can lock the orientation using Processing's `orientation()` method using either the `PORTRAIT` or the `LANDSCAPE`parameter.
For compatibility, Processing uses the constants `mouseX` and `mouseY` when it's running in Android mode, corresponding in this case to the position of a user's fingertip relative to the upper left corner of the device touch screen rather than the position of the mouse cursor on a desktop screen. This allows us to use the same code across modes. When using `mouseX` in Android mode, we refer to the horizontal position of the fingertip on the touch screen panel, and when we use `mouseY`, we refer the fingertip's vertical position. Both are measured relative to the coordinate system's origin in the upper left corner of the touch screen. Moving the finger to the right on the screen will increase `mouseX` values; moving the finger down will increase `mouseY`.
In Android mode, we can also use the following mouse methods, which are available in all Processing modes. The Android touch screen gestures correspond to the following mouse events:

- `mousePressed()` This callback method is called every time a finger touches the screen panel. It corresponds to a mouse-pressed event on the desktop when the mouse button is pressed down.
- `mouseReleased()` This callback method is called every time a finger lifts off the touch screen surface, but only if its position has changed since first touching the panel. It corresponds to a mouse-up event on the desktop.

- `mouseDragged()` This callback method is called every time a new finger position is detected by the touch screen panel compared to the previously detected position. It corresponds to a mouse-dragged event on the desktop when the mouse moves while the button is pressed.
  All three methods respond only to one finger's touch. When you use more than one finger on the multitouch surface, the finger that triggers callback events is the first one that touches the screen panel—the second, third, or more are ignored. If you hold down one finger on the screen surface, add another one on, and remove the first, then the second finger one will now be first in line and take over mouse events. We will work with multiple fingers and multitouch gestures in just a bit in Detect Multitouch Gestures.

  Let's put the mouse callback methods to the test with a simple sketch that prints the mouse position and events into the Processing console. We'll need `draw()` to indicate that this sketch is running and listening to the mouse continuously. Then we add our callback methods and have each print a brief text string indicating which mouse method has been called at what finger position.
  Create a new Android sketch by choosing File ↦ New from the Processing menu. If your new sketch window is not yet in Android mode, switch it to Android using the drop-down menu in the upper right corner. Add a few lines of code to the sketch window:

```
void draw()
{
    // no display output, so nothing to do here
}

void mousePressed ()
{
    println("PRESSED x:" + mouseX + " y: " + mouseY);
}

void mouseReleased ()
{
    println("RELEASED x:" + mouseX + " y: " + mouseY);
}

void mouseDragged ()
{
    println("DRAGGED x:" + mouseX + " y: " + mouseY);
}
```

  Let's go ahead and test the touch screen panel of an Android device.

Run the App

With your Android device connected to your desktop via a USB cable, run the sketch on the device by pressing the "Run on Device" button in the sketch window. When the sketch is installed and launched on the device, we don't need to pay attention to the screen output of the touch screen panel, but keep an eye on the Processing console at the bottom of the sketch window.

Hold your device in one hand and get ready to touch the screen surface with the other. Take a look at the console and tap the screen. In the console, you'll see output similar to this:

```
PRESSED x:123 y:214
```

Lift your finger and see what happens. If you see no additional mouse event, don't be surprised. Although we might expect a `RELEASED` here, we shouldn't get this event if we just tap the screen and lift the finger. The `mouseX` and `mouseY` constants always store and maintain the last mouse position. To get a mouse-released event, touch the screen, move your finger a bit, and release. Now you should see something like this:

```
PRESSED x:125 y:208
DRAGGED x:128 y:210
DRAGGED x:130 y:209
RELEASED x:130 y:209
```

Because we touched the screen, we first trigger a `mousePressed()` event. By moving the finger slightly while touching the surface, we trigger `mouseDragged()` until we stop moving. Finally, we get a `mouseReleased()`event because we've updated our position since we pressed or touched the screen.

Now that we can now work with the mouse callback methods, we're ready to take a look at the color support that Processing provides, which is one of its strengths. Knowing how to control color values is a fundamental skill that we'll frequently return to as we work with graphics and images throughout the book. We'll come back to the Android touch screen and its multitouch features later in this chapter.

Using Colors

Any geometric primitive we draw on the screen uses a particular `fill()` and `stroke()` color. If we don't say otherwise, Processing will default to a black stroke and a white fill color. We can use the `fill()` and `stroke()` methods to change default values, and we can also use grayscale, RGB, HSB, or hexadecimal color in the Android apps we create.

The `background()` method uses color in the same way, with the exception that it cannot set a value for opacity, formally known as the alpha value.

By default, Processing draws all graphic elements in the RGB (red, green, blue) color mode. An additional alpha value can be used as a fourth parameter to control the opacity of graphic elements drawn on the screen. An alpha value of `0` is fully transparent, and a value of `255` is fully opaque. Values of `0..255`control the level of opacity for an individual pixel.

The `background()` color of a Processing window cannot be transparent. If you provide an alpha parameter for `background()`, the method will just ignore its value. Within `draw()`, the `background()` method is used in most cases to clear the display window at the beginning of each frame. The method can also accept an image as a parameter, drawing a background image if the image has the same size as the Processing window.

Processing provides us with two different color modes that we can switch between using the `colorMode()`method. The color mode can be set to RGB or HSB (hue, saturation, brightness), which we'll explore further in Using HSB Colors. `colorMode()` changes the way Processing interprets color values. Both RGB and HSB can handle alpha values to make objects appear transparent.

We can adjust the value range of the parameters used in `colorMode()` as well. For example, white in the default RGB color mode is defined as `color(255)`. If we change the range to `colorMode(RGB,1.0)`, white is defined as `color(1.0)`.

Here are the parameters `colorMode()` can take. We can specify `mode` as either RGB or HSB and specify`range` in the value range we prefer.

- `colorMode(mode)`
- `colorMode(mode, range)`
- `colorMode(mode, range1, range2, range3)`
- `colorMode(mode, range1, range2, range3, range4)`

Let's now take a look at the three different color methods Processing has to offer. They are good examples of how Processing uses as few methods as possible to get the job done.

Using Grayscale and RGB colors

The `fill()` and `stroke()` methods can take either one, two, three, or four parameters. Since the `background()` method doesn't accept alpha values, it takes either one or three parameters:

```
fill(gray)
stroke(gray)
background(gray)
fill(gray, alpha)
stroke(gray, alpha)
fill(red, green, blue)
stroke(red, green, blue)
background(red, green, blue)
fill(red, green, blue, alpha)
stroke(red, green, blue, alpha)
```

As you can see, your results will differ depending on how many parameters you use. One parameter results in a grayscale value. Two parameters define a grayscale and its opacity (as set by an alpha value). If alpha is set to `0`, the color is fully transparent. An alpha value of `255` results in a fully opaque color. Three parameters correspond by default to red, green, and blue values. Four parameters contain the red, green, and blue values and an alpha value for transparency. Through this approach, Processing reduces the number of core methods by allowing for a different number of parameters and by interpreting them differently depending on the color mode.

To recall the syntax of any particular method, highlight the method you want to look up in the sketch window and choose the Find in Reference option from Help. It's the quickest way to look up the syntax and usage of Processing methods while you are working with your code.

Using Hex Colors

Processing's color method can also handle hexadecimal values, which are often less intuitive to work with but are still fairly common as a way to define color. We'll take a closer look at hexadecimal color values in Read Comma- Separated Web Color Data. Hex color method parameters, such as the `hex` code `#ff8800` for orange, are applied like this:

```
fill(hex)
stroke(hex)
background(hex)
fill()
stroke()
```

Now let's take a look at the `HSB` color mode, which, as we learned earlier, can define a color using hue, brightness, and saturation.

Using HSB Colors

Why should we care about HSB? Because it's a rather excellent color mode for working algorithmically with color, such as when we want to change only the saturation of a UI element. When we switch the default RGB color mode to HSB, the values of the color parameters passed to `fill()` and `stroke()` are not interpreted any more as red, green, blue, and alpha values, but instead as hue, saturation, brightness, and alpha color values. We can achieve seamless transitions between more- and less-saturated color values for UI highlights, for instance, which is very difficult to do properly in RGB. So for the objective of algorithmic color combinations, transitions, and animations that need to be seamless, HSB is great.

When we switch to HSB using the `colorMode(HSB)`, the `fill()`, `stroke()`, and `background()` methods will be interpreted like this:

```
fill(hue, saturation, brightness)
stroke(hue, saturation, brightness)
background(hue, saturation, brightness)
fill(hue, saturation, brightness, alpha)
stroke(hue, saturation, brightness, alpha)
```

When we work algorithmically in HSB, we can access the hue directly using Processing's `hue()` method. It takes a color as a parameter and extracts only the hue value of that color. Similarly, we can get the brightness by using the `brightness()` color method, and we can access the `saturation()` separately as well. The HSB color cylinder is a very useful illustration of this color space to further investigate and better understand the HSB color mode, where all hues are represented within the 360-degree circumference of the color cylinder. Take a quick look at it; we'll come back to it in the next project, Use Mouse Speed to Control Hues.

Now that we've learned about the different ways to assign color values, let's also take a look at the Processing `color` type, which Processing provides for the specific purpose of storing colors.

Using the Color Type

The Processing `color` type can store RGBA or HSBA values in one variable, depending on the `colorMode()` you choose. It's a great type for any app that we build using a color scheme of multiple colors. Using the `color` type, we simply call the `color` variable and apply it to the objects we draw. We can create a color palette in our app without requiring a bunch of individual variables for each value of an RGBA or HSBA color. We would apply the `color` type like this:

```
fill(color)
fill(color, alpha)
```

If `color` included an alpha value of, let's say, `127.5`, a primitive drawn with `fill(color)` would be drawn with `50%` opacity (given a possible max alpha value of `255`). In the unlikely scenario that the same color that already contains an alpha value is used in conjunction with an additional alpha parameter, such as `fill(color, 128)`, the resulting color would be drawn half as transparent as before, or at `25%` opacity.

Processing color methods are overloaded, so they can handle a range of situations—you can use one method for many applications. In other languages, remembering which syntax to use for a particular color effect can be a challenge, but with Processing you need to remember only a small number of methods. When a color value exceeds the default maximum value of `255`, Processing caps it for us. So `fill(300)` has the same result as `fill(255)` does. The same is true for values lower than the default minimum, `0`.

Now that we've learned about the different color modes, methods, and types available to define colors in an Android app, let's refine our previous drawing sketch.

Use Mouse Speed to Control Hues

Now let's explore the HSB mode on the device touch screen display. By adding `colorMode()` to our sketch, we switch the color mode, and by modifying our `fill()` method to work with HSB values, we change our app from grayscale to shades of color. Here's the result:

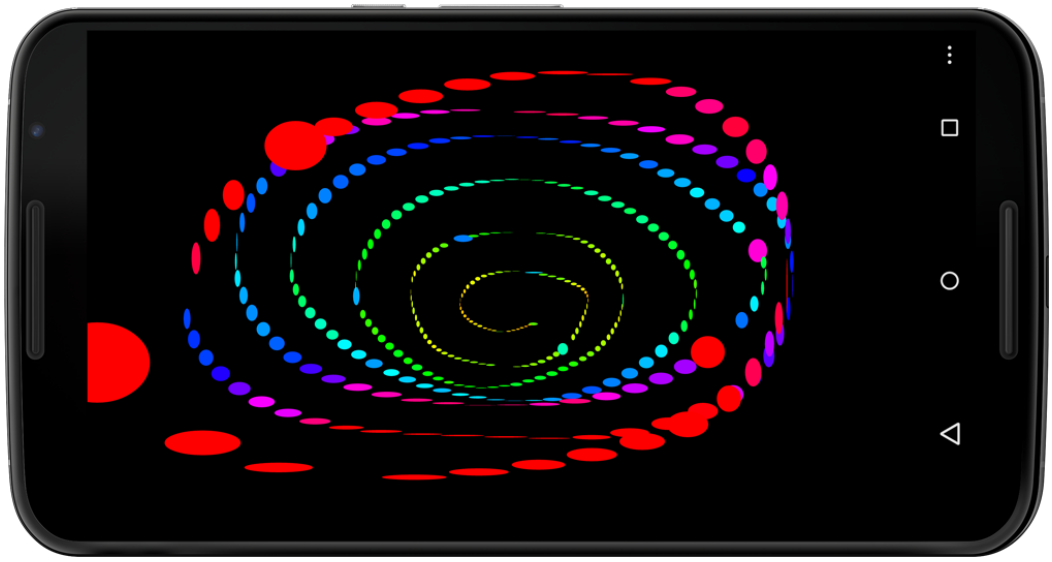


Figure 2.1 — Using mouse speed to control color.

The hue of the ellipses changes depending on how fast you move your finger across the touch screen surface. Slow movements result in greenish, medium in blueish, and fast movements in reddish values.

In this project, we'll keep the screen `orientation()` flexible, which is the default setting for our Processing apps, and we don't have to set anything to make the app change orientation when we hold the device upright or sideways. This means it will change orientation when the built-in device accelerometer sensor decides that the app should adapt to the particular orientation at that moment. When such an orientation change occurs, our `setup()` method will be called again, reinitializing the sketch and executing all the statements we've included in `setup()`. Because we set the screen to black, erasing its contents using the `background(0)`method, a change in the app's orientation will reset the `background()` to black, erasing all the ellipses we've drawn prior to changing the orientation.
We have only two modifications to make using the code from chapter 1. First we switch the color mode to HSB, which also customizes its value range. Then we calculate the speed of our movement by measuring the distance between the previous and the current mouse position using Processing's `dist()` method. The method takes two points as parameters and returns the distance between them. Finally, we apply the distance we've calculated to the hue in the `fill()` method for the ellipses we draw. The default value range for the HSB color modes is `0..255` by default. We'll override the default hue value to use floating point ranges of `0..100` instead, allowing us to use the calculated mouse speed directly to the hue parameter. For the saturation and brightness values, we'll override the default values to use floating point ranges of `0..1.0`.
Let's take a look at the project code.

Display/MouseSpeedHue/MouseSpeedHue.pde

```
void setup()
{
  noStroke();
  background(0);
  colorMode(HSB, 100, 1, 1);                              //  1
}

void draw()
{
  fill(dist(pmouseX, pmouseY, mouseX, mouseY), 1, 1);   //  2
  ellipse(mouseX, mouseY, mouseX-pmouseX, mouseY-pmouseY);
}
```

Here are the modifications we've made.

---

1. Switch the default RGB color mode to HSB using the `colorMode(HSB, 100, 1, 1)` method. Set the second parameter for hue to `100`, allowing for floating point hue values of `0..100`. Set the third parameter to `1`, defining the color saturation values for the range `0..1.0`. Also set the brightness range as `0..1.0` in the fourth parameter.
2. Now set the color `fill()` of the ellipse in our defined HSB mode. Use `dist()` to calculate the distance between the previous mouse position (`pmouseX`, `pmouseY`) and the current one (`mouseX`, `mouseY`), and apply the result to the hue parameter. Use `1` for saturation and brightness. We've defined `1` as the maximum saturation and brightness, so both are set to `100` percent.
   Let's test the app.

Run the App

Rerun the sketch on the device now switched to HSB color, and see how the hue of the ellipses that are drawn changes depending on how fast you move your finger across the screen surface. The changes in the hues occur independently of their saturation and brightness, so all the colors are drawn with maximum saturation and brightness.

If you go back to the HSB color wheel we looked at earlier, you will see how the `360` degrees of the HSB or the wheel correspond to the different colored ellipses you draw on the screen. Slow movement results in yellow color values, and then, with increasing speed, you'll see green, cyan, blue, and magenta as the maximum speed values.
Now that we've mastered the use of `color` in Processing, let's continue our investigation into the multitouch screen panel. We'll go ahead and install a Processing library that will help us work with multitouch gestures and extend the core features Processing provides us. Besides multitouch, the Ketai library makes it easy for us to work with other hardware devices and sensors built into Android phones and tablets. Let's take a look at the Ketai classes and the features it provides. We'll be using these throughout the rest of the book.

Introducing the Ketai Library

The Ketai library for Processing focuses particularly on making it easy to work with the mobile hardware features built into Android phones and tablets. The term "Ketai" is used in Japan to describe its cell phone culture, enabled by mobile handheld devices. The mobile device translates as Keitai Denwa and literally means "something carried in the hand," or handheld. The Ketai library is free software published under the GNU General Public License Version 3 (GPL v3), and it is compatible with Android versions 2.3 Gingerbread, 3.0/3.1 Honeycomb, 4.0 Ice Cream Sandwich, 4.1/4.2/4.3 Jelly Bean, 4.4 KitKat, and 5.0 Lollipop. NFC, Wi-Fi Direct, and updated camera features introduced in 4.0 Ice Cream Sandwich are not available in Gingerbread or Honeycomb. Therefore the Ketai library is available as separate downloads for Gingerbread/Honeycomb and for Ice Cream Sandwich/Jelly Bean. Please refer to Run a Sketch on an Android Device, to find out which version of Android you are running on your device.

Compared to the desktop, the defining feature of a mobile handheld device is that we use it on the go, where we expect cameras, location, and orientation sensors to help us navigate traffic, find relevant locations near by, and snap pictures while we are on the move. We also might be networking with Bluetooth accessories, interacting with products through embedded NFC tags, or paying for merchandise with our mobile devices. The Ketai library helps us develop apps for all of these scenarios.

Libraries are arguably one of the most successful aspects of the open source Processing project. There are more than 130 libraries available for Processing; however, on the Android device we can only use those Java libraries that do not make use of desktop hardware. Libraries extend the easy-to-learn Processing core with classes written for particular contexts, including 3D, animation, compilations, computer vision, data and protocols, geometry, graphic interface, hardware interface, import and export, math, simulation, sound, tools, typography, and video—to name the main categories listed on the Processing website, where the libraries are organized.

Many interface, sound, computer vision, and import/export libraries use code that is specific to the desktop context and are not designed for use on Android devices. Many libraries that could be compatible with Processing for Android are currently updated by library authors to eventually be available for us to use in Android mode. The Library Manager added to Processing 2.0 makes it easy to install libraries from within Processing. We'll use it to install the Ketai library in Install the Ketai Library, below.

There is hardly any computational topic that is not addressed in the Processing libraries. Because all libraries are open source and come with examples and tutorials, Processing is a favorite of students and creative coders alike. Most of the supplemental libraries have been developed for artists and designers for a particular project, so their use is often illustrated with

the actual project that inspired it. Some of those projects can also be found in the online Processing exhibition. This site makes browsing in and "shopping" for free libraries a fun activity and inspiring in its own right. As you download libraries and install them in your Processing sketchbook's `libraries` directory, they remain at this location and available even after you upgrade to a new version of the Processing IDE. The idea of this structure is to separate the Processing developer environment from the sketches that you write and the libraries you collect, keeping them independent of updates to Processing itself.
While there are scores of Processing libraries, only a small number of them work on Android phones and tablets. The Ketai library is designed particularly to provide programmer access to Android sensors, cameras, and networking; it is the only library that has been developed to run solely in Android mode.

I've been working on the Ketai library with Jesus Duran since 2010, with the objective to make it really easy to write apps that can effectively use the mobile hardware features built into Android phones and tablets. Convinced by the idea that phones and tablets evolve rapidly alongside the open source Android OS, the Ketai library makes it possible to consider such devices as a great complement to microcontrollers such as the Arduino—an open hardware sister project to Processing that is built on the same IDE.

Besides their compact form factor, multicore Android phones and tablets are computationally quite powerful, are equipped with a wide range of sensors, and run an operating system that is open source, free, and doesn't require subscriptions—characteristics that are advantageous to innovation, academic use, and DIY culture. What's more, once a mobile phone or tablet is outdated, it remains an inexpensive device, available in abundance and way too functional for a landfill.

The Ketai library values conciseness and legibility in its syntax and makes hardware features available using just a few lines of code. For example, the simple code we use for our accelerometer project (code available here) uses less than thirty lines of code altogether, while the Java sample included in the Android SDKcompletes the task with more than one hundred lines of code. This ratio increases significantly with more complex subjects such as Chapter 7, Peer-to-Peer Networking Using Bluetooth and Wi-Fi Direct, and Chapter 8, Using Near Field Communication (NFC), where Ketai is significantly more concise and easier to understand than the SDK.

Ketai includes a number of classes that make Android hardware sensors and devices available within Processing. The following classes are included in the library, described in more detail in ketai classes, and explained within the relevant chapters:

- KetaiSensor

- KetaiLocation
- KetaiCamera
- KetaiFaceDetector
- KetaiBluetooth
- KetaiWiFiDirect
- KetaiNFC
- KetaiData
- KetaiList
- KetaiKeyboard
- KetaiGesture

Let's go ahead and install the Ketai library now.

Install the Ketai Library

Follow these steps to activate the Processing library. It's a one-time process; you won't need to repeat it.

You can install the Ketai library from within the Processing IDE using the "Add Library..." menu item.

---

1. Choose "Add Library...," which you can find under Sketch ↦ "Import Library..."
2. At the bottom of the window that opens, enter "Ketai."
3. Select the Ketai library that appears in the list and press the Install button on the right.
4. The download starts immediately, and a bar shows the download's progress. When the library is installed, the button on the right changes to Remove.

   Alternatively, you can download and install the library manually from the dedicated website that comes with every Processing library. This process has the advantage that you can read about the library and preview its features alongside a reference and example code for the library.

1. Go to the Ketai library website, http://ketai.org/download, and download the latest `zip` file.
2. Extract the file to the `Documents/Processing/libraries` folder. If the `libraries` subfolder doesn't exist in your sketchbook, create it now and put the Ketai folder inside it.
3. Restart Processing so it can load the newly added library.
4. Check whether the installation was successful by opening Sketch↦ "Import Library..." Under Contributed libraries you should now see the name "Ketai." If it doesn't show up in the list, please refer to Troubleshooting.

   The process for downloading and installing the Ketai library is identical for any other Processing library.

   Let's now move on to our first project—putting the Ketai library to work.

Working with the KetaiGesture Class

`KetaiGesture` gives us access to the most common multitouch gestures used on mobile devices. It provides us with the callback methods that we need to highlight, scale, drag, and rotate objects and UI elements. To select, zoom, focus, and organize the elements we display on the touch screen, we can use a number of gestures that have become user interaction standards on mobile devices. Working off established UI standards, we can build apps that are more intuitive to use and that enable the user to get the job done quickly while on the move. Using the `KetaiGesture` class, we can work with the following callback methods, which report back to us when a certain event has occurred on the touch screen surface, which was triggered by a particular user interaction or multitouch gesture.
Let's take a look at the main methods included in `KetaiGesture`:

```
onTap(float x, float y)
```

- Single Tap—triggered by one short tap on the device screen. Returns the horizontal and vertical position of the single-tap gesture.

```
onLongPress(float x, float y)
```

- Long Press—triggered by tapping and holding a finger at one position on the touch screen for about one second. Returns the horizontal and vertical position of the long press gesture.

```
onFlick(float x, float y, float px, float py, float v)
```

- Flick—triggered by moving a finger in any direction, where the beginning and the end of the gesture occur at two different screen positions while the finger doesn't come to a full stop before lifting it from the screen surface. Returns the horizontal and vertical position where the flick is released, the horizontal and vertical position where the flick started, and the velocity of the flick.

```
onPinch(float x, float y, float d)
```

- Pinch—triggered by a two-finger gesture either away from each other (pinch open) or toward each other (pinch close). The pinch is typically used for zooming in and out of windows or for scaling objects. Returns the horizontal and vertical position of the pinch's centroid and the relative change in distance of the two fingers to each other.

```
onRotate(float x, float y, float angle)
```

- Rotate—triggered by the relative change of the axis rotation defined by two fingers on the touch screen surface. Returns the centroid of the rotation gesture and the relative change of the axis angle.

Let's build an app that puts `KetaiGesture`'s multitouch methods to use.

Detect Multitouch Gestures

For this project, we'll implement the most common user interactions using just one simple geometric primitive—a rectangle—drawn on the screen using Processing's `rect(x, y, width, height)` method. To begin, we'll place a rectangle in a specified size of `100` pixels in the center of the screen. Then we use a series of `KetaiGesture` callback events to trigger changes to the rectangle, including a change of scale, rotation, color, and position, as illustrated in Figure 2.2.

Figure 2.2 — Using multitouch gestures.

The illustration shows a rectangle scaled with a two-finger pinch gesture, turned by a two-finger rotation gesture, placed on a brown background color, and triggered by a flick, as well as a gray fill color caused by a long press. The text "DOUBLE" appears due to a double-tap gesture at the position indicated by the hand silhouette.

We have a number of callback events for the touch surface to try out, so we'll assign each of them with a particular purpose. We'll zoom to fit the rectangle onto the screen using `onDoubleTap()`, randomly change its fill color `onLongPress()` using Processing's `random()` method, scale it `onPinch()`, rotate it `onRotate()`, drag it using `mouseDragged()`, and change the background color `onFlick()`. Besides manipulating color properties and the rectangle, we'll keep track of the multitouch events as they occur by printing a text string to the Processing Console. The code we use to manipulate the properties and the callback methods themselves are not complicated in any way, but we're now dealing with a bit more code than we have before because we're using a series of callback methods in one sketch.

Introducing 2D Transformations

For this project, we'll lock our app into `LANDSCAPE` `orientation()` so we can maintain a clear reference point as we discuss 2D transformations in reference to the coordinate system. To center our rectangle on the screen when we start up, to scale from its center point using the pinch gesture, and to rotate it around its center point using the rotate gesture, we need to work with two-dimensional (2D) transformations.

We'll use the Processing's `rectMode(CENTER)` method to overwrite the default way a rectangle is drawn in Processing, which is from the upper left corner of the rectangle located at position `[x, y]` with a specified `width` and `height`. Instead we draw it from its center point using `rectMode(CENTER)`, which allows us to rotate and scale it around its center point.

A common metaphor to explain 2D transformations is a grid or graph paper. Using this analogy, each grid cell stands for one pixel of our app's display window. The default origin in Processing's coordinate system is always the upper left corner of the window. Each graphic element is drawn relative to this origin onto the screen. To move and rotate our rectangle, we'll use Processing's transformation methods: `translate()` and `rotate()`. We also have a `scale()` method, which we won't use in this sketch.

When we draw graphic objects in Processing on our grid paper, we are used to specifying the rectangle's horizontal and vertical coordinates using x and y values. We can use an alternative method, which is necessary here, where we move our grid (paper) to specified horizontal and vertical coordinates, rotate, and then draw the rotated rectangle at position x and y `[0, 0]`. This way the rectangle doesn't move to our intended position, but our grid paper (coordinate system) did. The advantage is that we can now `rotate()`our `rect()` right on the spot around its center point, something we can't do otherwise.

What's more, we can introduce a whole stack of grid paper if we'd like to by using the Processing methods `pushMatrix()` and `popMatrix()`. When we move, rotate, and scale multiple elements and would like to transform them separately, we need to draw them on

separate pieces of grid paper. The `pushMatrix()`method saves the current position of our coordinate system, and `popMatrix()` restores the coordinate system to the way it was before pushing it.

Like our first project in this chapter, in which we used Processing's `mousePressed()`, `mouseReleased()`, and `mouseDragged()` callback methods to identify touches to the screen, some of the multitouch gestures introduced here fulfill the same purpose. If we'd like to use Processing's mouse methods alongside multitouch methods provided by `KetaiGesture`, we'll need to notify the superclass method `surfaceTouchEvent()` to notify the Processing app that a surface touch event has occurred.

Now let's take a look at our multitouch code.

Display/Gestures/Gestures.pde

```
import ketai.ui.*;                                      //  1
import android.view.MotionEvent;                        //  2

KetaiGesture gesture;                                   //  3
float rectSize = 100;                                   //  4
float rectAngle = 0;
int x, y;
color c = color(255);                                   //  5
color bg = color(78, 93, 75);                           //  6

void setup()
{
  orientation(LANDSCAPE);
  gesture = new KetaiGesture(this);                     //  7

  textSize(32);
  textAlign(CENTER, BOTTOM);
  rectMode(CENTER);
  noStroke();

  x = width/2;                                          //  8
  y = height/2;                                         //  9
}

void draw()
{
  background(bg);
  pushMatrix();                                         //  10
  translate(x, y);                                      //  11
  rotate(rectAngle);
  fill(c);
  rect(0, 0, rectSize, rectSize);
  popMatrix();                                          //  12
}

public boolean surfaceTouchEvent(MotionEvent event) {  //  13
  //call to keep mouseX and mouseY constants updated
  super.surfaceTouchEvent(event);
  //forward events
  return gesture.surfaceTouchEvent(event);
}

void onTap(float x, float y)                            // 14
{
  text("SINGLE", x, y-10);
  println("SINGLE:" + x + "," + y);
```

```
}

void onDoubleTap(float x, float y)                      // 15
{
  text("DOUBLE", x, y-10);
  println("DOUBLE:" + x + "," + y);

  if (rectSize > 100)
    rectSize = 100;
  else
    rectSize = height - 100;
}

void onLongPress(float x, float y)                      // 16
{
  text("LONG", x, y-10);
  println("LONG:" + x + "," + y);

  c = color(random(255), random(255), random(255));
}

void onFlick( float x, float y, float px, float py, float v)  // 17
{
  text("FLICK", x, y-10);
  println("FLICK:" + x + "," + y + "," + v);

  bg = color(random(255), random(255), random(255));
}
void onPinch(float x, float y, float d)                 // 18
{
  rectSize = constrain(rectSize+d, 10, height);
  println("PINCH:" + x + "," + y + "," + d);
}

void onRotate(float x, float y, float angle)            // 19
{
  rectAngle += angle;
  println("ROTATE:" + angle);
}

void mouseDragged()                                     // 20
{
  if (abs(mouseX - x) < rectSize/2 && abs(mouseY - y) < rectSize/2)
  {
    if (abs(mouseX - pmouseX) < rectSize/2)
      x += mouseX - pmouseX;
    if (abs(mouseY - pmouseY) < rectSize/2)
      y += mouseY - pmouseY;
  }
}
```

Let's take a look at the steps we need to take to capture and use multitouch gestures on the Android touch screen.

---

1. Import Ketai's `ui` package to give us access to the `KetaiGesture` class.
2. Import Android's `MotionEvent` package.
3. Define a variable called `gesture` of type `KetaiGesture`.
4. Set a variable we call `rectSize` to `100` pixels to start off.
5. Define the initial color `c` (white), which we'll use as a fill color for the rectangle and text.
6. Define the initial color `bg` (dark green), which we'll use as a background color.
7. Instantiate our `KetaiGesture` object `gesture`.

8. Set the initial value for our variable `x` as the horizontal position of the rectangle.
9. Set the initial value for `y` as the vertical position of the rectangle.
10. Push the current matrix on the matrix stack so that we can draw and rotate the rectangle independent of other UI elements, such as the text.

11. Move to the position `[x, y]` using `translate()`.
12. Pop the current matrix to restore the previous matrix on the stack.

13. Use the Processing method `surfaceTouchEvent()` to notify Processing about mouse/finger-related updates.
14. Use the callback method `onTap()` to display the text string `SINGLE` at the location `(x, y)` returned by `KetaiGesture`.
15. Use the callback method `onDoubleTap()` to display the text string `DOUBLE` at the location returned by `KetaiGesture`, indicating that the user triggered a double-tap event. Use this event to decrease the rectangle size to the original `100` pixels if it's currently enlarged, and increase the rectangle scale to the display height minus `100` pixels if it's currently minimized to its original scale.
16. Use the callback method `onLongPress()` to display the text string "LONG" at the location `(x, y)`returned by `KetaiGesture`. Use this event to randomly select a new color `c` using `random()`, which we'll use as a fill color for the rectangle.
17. Use the callback method `onFlick()` to display the text string `FLICK` at the location `x` and `y` returned by `KetaiGesture`. Also, receive the previous location where the flick has been initiated as `px` and `py`, as well as the velocity `v`.
18. Use the callback method `onPinch()` to calculate the scaled `rectSize` using the pinch distance `d` at the location `x` and `y` returned by `KetaiGesture`.
19. Use the callback method `onRotate()` to calculate the rotation angle `rectAngle` using the `angle`returned by `KetaiGesture`.
20. Use Processing's `mouseDragged()` callback to update the rectangle position (`x` and `y`) by the amount of pixels moved. Determine this amount by subtracting the previous `pmouseX` from the current `mouseX`, and `pmouseY` from `mouseY`. Move the rectangle only if absolute distance between the rectangle and the mouse position is less than half the rectangle's size, or when we touch the rectangle.
    Let's test the app.

Run the App

Run the app on your device. You'll see a square show up in the center of the screen. Drag it to a new location, flick to change the background color, and give it a long tap to change the foreground fill color.

To test the multitouch gestures, put two fingers down on the screen and pinch, and you'll see how the rectangle starts scaling. Now rotate the same two fingers to see the rectangle rotate. If you use more than two fingers, the first two fingers you put down on the screen are in charge.

Finally, double-tap the screen to zoom the square to full screen, and double-tap again to scale it to its initial size of `100` pixels.
This completes our investigation into the multitouch features of the touch screen panel.

Wrapping Up

You've used the touch screen panel as the first hardware device we've worked with. You've learned about mouse speed and all the different color features in Processing and worked with the HSB color mode to manipulate the hue values of the geometric primitive we've drawn. And finally, you are now able to use mouse events and multitouch gestures for your app's user interfaces to control the object you display on the device screen.

You are now well positioned to move on to the next chapter, where we'll focus on the hardware sensors built into Android devices. After all, the "native" user experience of mobile apps relies heavily on hardware devices and sensors, and we are now ready to incorporate them into our apps.

3. Using Motion and Position Sensors

This chapter is about how to interact with sensors on an Android device using the Ketai library. Android devices come packed with sensors that allow us to write mobile apps that react to how we position and move the Android device to make them more engaging, useful, and interactive. Android sensors provide us with information about device motion, position, and environment. We'll focus on motion sensors in this chapter and take a look at some position sensors.

+

Motion sensors allow us to measure how the device is oriented in space and how it accelerates when we move it. The typical accelerometer sensor found on Android devices triggers screen rotations and is used for a wide range of apps to detect shakes, fist bumps, hand gestures, bumpy roads, and other features. Using Ketai, we'll list all the available sensors built into the Android and work with multiple sensors combined in an app that displays values for the magnetic field, light, proximity, and accelerometer sensors.

We'll work with the orientation of an Android device to create an interactive color mixer app. Step by step, we'll start by learning to display raw data from the accelerometer sensor, and then we'll use those values to generate the entire spectrum of color that the Android can generate. Next we'll learn how to store data in an array and how to use the array to display a palette of eight colors that we've created. Finally, we'll use the accelerometer to clear the color palette by shaking the device, and we'll detect that motion in our program. In the process, we'll acquire a sense for the accelerometer sensor, its value range, and its accuracy, and we'll learn to integrate it into the app's user interface. By the end of the chapter, you will know this sensor well enough to transpose this knowledge to other applications.

Introducing the Device Hardware and Software Layers

Writing sensor-based Android apps in Processing involves a series of software layers that build on each other. The list below describes the software stack running on the device hardware, all of which we put to use when we run our sketches—starting with the bottommost hardware layer.

Hardware Besides the central (CPU) and graphics (GPU) processing unit, hardware devices built in the Android include: GSM/3G/4G antennas, digital cameras, an accelerometer sensor, a light sensor, a gyroscope, a geomagnetic field sensor, a capacitive touch screen panel, an audio interface, a speaker, a vibration motor, a battery, a Flash memory interface, and perhaps a hardware keyboard and a temperature sensor.

Linux kernel The bottommost software layer running on the hardware is the Linux kernel, a Unix-like operating system initially developed by Linus Torvalds in 1991. We access all the hardware resources of the device through this layer, which contains drivers for the display, cameras, Bluetooth, Flash memory, Binder (PC), USB, keypad, Wi-Fi, audio, and power.

Android and Java Core Libraries Above the Linux kernel sit the Android native libraries written in C/C++, including Surface Manager, Media Framework, SQLite , OpenGL/ES , FreeType, WebKit, SGL, SSL, and libc. This layer also includes Android Runtime, which contains the core Java libraries and the Dalvik virtual machine. The Dalvik virtual machine creates compact executable files that are optimized for memory and processor speed. The virtual machine allows a high level of control over the actions an app is permitted to take within the operating system. Android applications are typically written in Java using the Java core libraries and compiled to bytecode, which is the format executed by a Java virtual machine. In the Android OS, bytecode is converted into a Dalvik executable (dex) before an app is installed on the Android device.

Processing The language itself is the next layer in our software stack that builds on the Java core libraries. The Android mode in Processing works with the Android libraries in the Android layer. Processing's software architecture allows us to use Java and Android classes directly within the Processing code.

Ketai The library builds on the Processing for Android layer, taking advantage of the Processing, Java, and Android libraries. It's the highest layer in the software stack we'll be working with in projects, besides using a few other libraries that sit on the same level in the hierarchy. Ketai focuses specifically on the hardware features built into Android devices, including the multitouch screen panel, sensors, cameras, and networking devices.

Now that we are aware of the different hardware components and the software layers stacked on top of the hardware layer, let's start with the bottommost hardware layer and take a look at the most common sensors built into our Android device.

Introducing Common Android Sensors

In this chapter, we will work mostly with the accelerometer sensor and use the KetaiSensor class to access it. KetaiSensor is capable of working with all sensors. Some sensors found on the Android device are based on hardware; others are software-based and provided by the Android SDK. For the projects in this chapter, we'll focus on actual electronic hardware sensors built into the Android phone or tablet. Android distinguishes three different sensor-type categories: motion sensors, position sensors, and environment sensors. Most environment sensors have been added to the Android SDK recently (Android 4.0 Ice Cream Sandwich), so they are not typically found in devices yet. Let's take a look at the different sensors Android supports.

Motion Sensors

The following sensors let you monitor the motion of the device:

- Accelerometer (hardware) Determines the orientation of the device as well as its acceleration in three-dimensional space, which we'll use to detect shakes
- Gravity (software-based) Calculates the orientation of the device, returning a three-dimensional vector indicating the direction and magnitude of gravity
- Gyroscope (hardware) Measures the movement of the device, returning the rate of rotation around each device axis—if available, this sensor is often used for games that rely on immediate and precise responses to device movement.
- Linear Acceleration (software-based) Calculates the movement of the device, returning a three-dimensional vector that indicates the acceleration of each device axis, excluding gravity
- Rotation Vector (software-based) Calculates the orientation of the device, returning an angle and an axis—it can simplify calculations for 3D apps, providing a rotation angle combined with a rotation axis around which the device rotated.

Now let's take a look at the sensors that deal with the device's position.

Position Sensors

The following sensors let you to determine the location or position of the device:

- Magnetic Field A three-axis digital compass that senses the bearing of the device relative to magnetic north
- Proximity Senses the distance to an object measured from the sensor that is mounted in close proximity to the device speaker—this is commonly used to determine if the device is held toward, or removed from, the ear.

Now let's take a look at the sensors that measure the device's environment.

Environment Sensors

The following sensors let you monitor environmental properties or measure the device context:

- Light Senses the ambient light level
- Pressure Senses the air pressure (the atmospheric pressure)
- Relative Humidity Senses the humidity of the air (in percent)
- Temperature Senses the ambient air temperature

Since this list will grow and remain a moving target as new generations of devices and APIs are released, the Android Sensor website is the best source for keeping an eye on changes and additions.

Let's start by looking at the `KetaiSensor` class, which we'll use when we work with sensors throughout the book.

Working with the KetaiSensor Class

For the sketches we'll write in this chapter, the following `KetaiSensor` methods are the most relevant:

- `list()` Returns a list of available sensors on the device
- `onAccelerometerEvent()` Returns x-, y-, and z-axis acceleration minus g-force in meters per second squared (m/s^2)
- `onMagneticFieldEvent()` Returns the x, y, and z values for the ambient magnetic field in units of microtesla
- `onLightEvent()` Returns the light level in SI units of lux
- `onProximityEvent()` Returns the distance to an object measured from the device surface in centimeters—depending on the device, a typical output is `0/1` or `0/5`—the sensor is typically located next to the speaker on the device.
- `onGyroscopeEvent()` Returns the x, y, and z rates of rotation around the x-, y-, and z-axes in degrees

Because a multitude of devices on the market exist, it's important that we start by checking the sensors that are built into our Android device. Let's use the `KetaiSensor` class to see what sensors are built into our Android device.

List the Built-In Sensors on an Android Device

Let's find out what sensors are built into our device and available for us to work with.
The `KetaiSensor` class offers a `list()` method that enumerates all Android sensors available in the device and lists them for us in the Processing console.
Open a new sketch window in Android mode and type or copy the following four lines of code; click on the green bar to download the `SensorList.pde` source file if you are reading the ebook.

```
import ketai.sensors.*;
KetaiSensor sensor;
sensor = new KetaiSensor(this);
println(sensor.list());
```

Take a look at the code. First we import the Ketai sensor package, then we create a `sensor` variable of the type `KetaiSensor`, and finally we create a `sensor` object containing all the `KetaiSensor` methods we need. As the last step, we print the sensor `list()` to the console.

Run the App

Run this code on your Android device and take a look at the Processing console. For example, the Google Nexus S reports the following list of sensors:

```
KetaiSensor sensor: KR3DM 3-axis Accelerometer:1
KetaiSensor sensor: AK8973 3-axis Magnetic field sensor:2
KetaiSensor sensor: GP2A Light sensor:5
KetaiSensor sensor: GP2A Proximity sensor:8
KetaiSensor sensor: K3G Gyroscope sensor:4
KetaiSensor sensor: Rotation Vector Sensor:11
KetaiSensor sensor: Gravity Sensor:9
KetaiSensor sensor: Linear Acceleration Sensor:10
KetaiSensor sensor: Orientation Sensor:3
KetaiSensor sensor: Corrected Gyroscope Sensor:4
```

The Asus Transformer Prime tablet reports the following sensors:

```
KetaiSensor sensor: MPL rotation vector:11
KetaiSensor sensor: MPL linear accel:10
KetaiSensor sensor: MPL gravity:9
KetaiSensor sensor: MPL Gyro:4
KetaiSensor sensor: MPL accel:1
KetaiSensor sensor: MPL magnetic field:2
KetaiSensor sensor: MPL Orientation:3
KetaiSensor sensor: Lite-On al3010 Ambient Light Sensor:5
KetaiSensor sensor: Intersilisl29018 Proximity sensor:8
```

The Google Nexus 6 reports a lists 32 software and hardware sensors. The list includes some hardware info, its type, and an ID. Your results, no doubt, will differ; there are a lot of Android makes and models out there today.

The list includes more than hardware sensors. The Android SDK also includes software-based sensors, known as fused sensors. Fused sensors use multiple hardware sensors and an Android software layer to improve the readings from one individual sensor. They make it easier for us

as developers to work with the resulting data. The `Gravity`, `Linear Acceleration`, and `Rotation Vector` sensors are examples of such hybrid sensors, combining gyroscope, accelerometer, and compass data to improve the results. In the list of available sensors, however, no distinction is made between hardware sensors and fused sensors.
This also means that even if you don't update your device hardware, new versions of the Android API might include fused software-based sensor types that might be easier to use or might produce better results. For example, if you browse Android's sensor hardware overview and switch the "Filter by API Level" to `8`(http://developer.android.com/reference/android/hardware/Sensor.html), you will see a list of the sensor types and methods that have been added to the API since the release of API `8`.
As you start adding methods from the Ketai library to the sketch, note that contributed libraries are not highlighted by the Processing IDE because they are not part of the core. This is not a big deal, but it's something you should be aware of.

Here's the code we'll typically use to interact with a device using the classes and methods that the Ketai library provides:

```
import ketai.sensors.*;                                    // 1
KetaiSensor sensor;                                          // 2

void setup()
{
sensor = new KetaiSensor(this);                              // 3
sensor.start();                                              // 4
}

void draw()
{
}

void onAccelerometerEvent(float x, float y, float z) // 5
{
}
```

Let's take a look at the code that is specific to `KetaiSensor`.

1. Import the Ketai sensor library package from `Sketchbook/libraries`.
2. Declare a `sensor` variable of type `KetaiSensor`, and register it for any available Android sensors.
3. Instantiate the `KetaiSensor` class to create a `sensor` object, which makes `KetaiSensor` methods available.
4. Start listening for accelerometer sensor events.
5. Each time the accelerometer changes value, receive a callback for the x, y, and z sensor axes.

Sensor values change at a different rate than the `draw()` method does. By default, `draw()` runs `60` times per second. The sensor can report much faster than that rate, which is why we work with an `onAccelerometerEvent()` callback method. It is called every time we receive a new value from the accelerometer.

Different devices use different accelerometers. Some contain hardware filters that stop reporting values altogether when the device is absolutely still. Others might be more accurate—or noisy—and keep reporting even when the device is seemingly still. Accelerometers are sensitive to the smallest motion. Let's take a look at the raw values such a device will display.

Display Values from the Accelerometer

Using the Ketai library, let's see what the accelerometer has to report. The accelerometer is the most common sensor found in mobile devices and is designed to detect device acceleration and its orientation toward g-force. It returns the x-, y-, and z-axes of the device, measured in meters per second squared. These axes are not swapped when the app's screen orientation changes.

The accelerometer sensor's shortcomings are related to the fact that it cannot distinguish between rotation and movement. For instance, moving the device back and forth on a flat table and rotating it about its axes can produce identical accelerometer values. To differentiate between movement and rotation, we require an additional sensor, the gyroscope, which we'll also use in Chapter 11, Introducing 3D Graphics with OpenGL. When we want to find out how the device is oriented with respect to gravity, however, the accelerometer is the only sensor that can help us.

Let's add some code to output raw accelerometer values onto the screen. We're aiming for the result shown in Figure 3.1, Accelerometer output. We use the `text()` method and some formatting to display accelerometer values. As we move the device, it will also come in handy to lock the screen orientation so we can keep an eye on the quickly changing values. Because we only need to set the screen `orientation(PORTRAIT)` once at startup, the method goes into `setup()`.

Figure 3.1 — Accelerometer output

The picture shows the acceleration of the x-, y-, and z-axes of the device in relation to g-force.

Now let's dive into the code.

Sensors/Accelerometer/Accelerometer.pde

```
import ketai.sensors.*;

KetaiSensor sensor;
float accelerometerX, accelerometerY, accelerometerZ;

void setup()
{
  sensor = new KetaiSensor(this);
  sensor.start();
  orientation(PORTRAIT);
  textAlign(CENTER, CENTER);                          // 1
  textSize(72);                                       // 2
}

void draw()
{
  background(78, 93, 75);
  text("Accelerometer: \n" +                          // 3
    "x: " + nfp(accelerometerX, 2, 3) + "\n" +
    "y: " + nfp(accelerometerY, 2, 3) + "\n" +
    "z: " + nfp(accelerometerZ, 2, 3), width/2, height/2);
}

void onAccelerometerEvent(float x, float y, float z)
{
  accelerometerX = x;
  accelerometerY = y;
  accelerometerZ = z;
}
```

Let's take a closer look at the Processing methods we've used for the first time.

---

1. Align the text to the `CENTER` of the screen using `textAlign()`.
2. Set the text size to `72` using `textSize()`. The default text size is tiny and hard to decipher in motion.
3. Display the data using `text()`. We output a series of strings tied together via the plus sign (+), known as the concatenation operator. This way we can use only one text method to display all the labels and reformatted values we need.
   Acceleration values are measured in m/s^2. If the device is sitting flat and still on the table, the accelerometer reads a magnitude of `+9.81` m/s^2. This number represents the acceleration needed to hold the device up against g-force and the result of the following calculation: acceleration of the device (`0` m/s^2) minus the acceleration due to gravity (`-9.81` m/s^2). If we move and rotate the device, we can observe values in the range of roughly `-10` to `10` m/s^2. Shake the device and the values will surge momentarily to maybe `+-20`m/s^2. Values beyond that become tricky to observe; feel free to try.
   We format the numbers via `nfp()`, a method that helps us to maintain two digits to the left and three digits to the right of the decimal point. This way, values we observe don't jump around as much. The "p" in `nfp()`puts a "+" in front of positive accelerometer values and a "-" in front of negative values, helping us to understand the device orientation better with regard to the accelerometer's nomenclature.

Run the App

In case you didn't already run the sketch in anticipation, now is the time. Remember that the shortcut for Run on Device is ⌘R.

Try placing your device in different positions and observe the acceleration due to gravity reported for each axis. If you lay your Android device flat on a table, for example, the z-axis will report an acceleration of approximately `+9.81` m/s^2. When you hold it vertically in a reading position, notice how the acceleration due to gravity shifts to the y-axis. The screen output is similar to Figure 3.1, Accelerometer output. Tiny movements of the device trigger very observable changes in value, which are reported back to us via `onAccelerometerEvent()`.
Let's now see how a sketch would look using multiple sensors.

Display Values from Multiple Sensors

So far we've worked with the accelerometer, which is a hardware motion sensor built into the Android device. In future chapters we'll want to work with multiple sensors, so let's fire up a few simultaneously and display their values on the Android screen. For this sketch, we'll activate the accelerometer again and add two position sensors and an environment sensor. The magnetic field sensor and the proximity sensors are considered position sensors; the light sensor is an environment sensor.

We could store the three axes returned by the accelerometer and magnetometer sensors in individual floating point variables. A better solution, however, is to work with Processing's `PVector` class.
It can store either a two- or a three-dimensional vector, which is perfect for us, since we can put any two or three values into this package, including sensor values. Instead of three variables for the x-, y-, and z-axes returned by the accelerometer and magnetometer, we can just use one `PVector`, called `accelerometer`. We refer later to an individual value or axis using the `accelerometer.x`, `accelerometer.y`, and `accelerometer.z` components of this `PVector`. The class is equipped with a number of useful methods to simplify the vector math for us, which we'll use later in this chapter to detect a device shake.
For this sketch, let's lock the screen `orientation()` into `LANDSCAPE` mode so we can display enough digits behind the comma for the floating point values returned by the sensors.
To create a sketch using multiple sensors, we follow these steps:

sensors/MultipleSensors.pde

```
import ketai.sensors.*;
```

```
KetaiSensor sensor;
PVector magneticField, accelerometer;
float light, proximity;

void setup()  {
  sensor = new KetaiSensor(this);
  sensor.start();
  sensor.list();
  accelerometer = new PVector();
  magneticField = new PVector();
  orientation(LANDSCAPE);                          // 1
  textAlign(CENTER, CENTER);
  textSize(72);
}
void draw()
{
  background(78, 93, 75);
  text("Accelerometer :" + "\n"
    + "x: " + nfp(accelerometer.x, 1, 2) + "\n"
    + "y: " + nfp(accelerometer.y, 1, 2) + "\n"
    + "z: " + nfp(accelerometer.z, 1, 2) + "\n"
    + "MagneticField :" + "\n"
    + "x: " + nfp(magneticField.x, 1, 2) + "\n"
    + "y: " + nfp(magneticField.y, 1, 2) + "\n"
    + "z: " + nfp(magneticField.z, 1, 2) + "\n"
    + "Light Sensor : " + light + "\n"
    + "Proximity Sensor : " + proximity + "\n"
    , 20, 0, width, height);
}

void onAccelerometerEvent(float x, float y, float z, long time, int accuracy)
{
  accelerometer.set(x, y, z);
}
void onMagneticFieldEvent(float x, float y, float z, long time, int accuracy)  // 2
{
  magneticField.set(x, y, z);
}

void onLightEvent(float v)                         // 3
{
  light = v;
}

void onProximityEvent(float v)                     // 4
{
  proximity = v;
}
public void mousePressed() {                       // 5
  if (sensor.isStarted())
    sensor.stop();
  else
    sensor.start();
  println("KetaiSensor isStarted: " + sensor.isStarted());
}
```

Let's take a closer look at the different event methods.

---

1. Rotate the screen `orientation()`.
2. Measure the strength of the ambient magnetic field in microteslas along the x-, y-, and z-axes.
3. Capture the light intensity, measured in lux ambient illumination.

4. Measure the distance between the device display and an object (ear, hand, and so on). Some proximity sensors support only near (`1`) or far (`0`) measurements.
5. Tap on the touch screen to invoke the `start()` and `stop()` methods for the sensors on the device. This will start and stop all sensors here, as all of them are registered with the same `sensor` object.
   Let's take a look to see if all the sensors return values.

Run the App

Run the sketch on the device, and you should see output similar to this:

Figure 3.2 — Using multiple Android sensors

The image shows the accelerometer, magnetic field, light, and proximity sensor output.

Move and rotate the device to see how sensor values change. The proximity sensor is located on the Android next to the speaker and is typically used to detect whether the device is held against or away from the ear. It returns values in centimeters, and you can use your hand to play with the returned proximity values. Depending on your Android make and model, you get a `0` if you are close to the device and either a `1` or a `5` if you are more than 1 or 5 centimeters away. Current proximity sensors are not accurate enough to use as a measuring tool just yet. Tapping the screen calls the `stop()` method and stops all the sensors. If the app doesn't require sensor updates all the time, stopping sensors is a good way to save some battery power.
Your sensor list might have already shown that your Android has a gyroscope built in. If not, Ketai will report, "Disabling `onGyroscopeSensorEvent()` because of an error" in the console.

To work with the gyro as well, add the following code snippet to the sketch above, add a global `rotation`variable of type `PVector`, and rerun the app on the device:

```
void onGyroscopeEvent(float x, float y, float z) {
rotation.x = x;
rotation.y = y;
rotation.z = z;
}
```

If you have a gyro, you are all set for Chapter 11, Introducing 3D Graphics with OpenGL, where we use it to navigate a 3D environment. No worries though if your device doesn't support it. There are plenty of motion-based apps we'll develop based on the accelerometer, and there are numerous other projects to discover in this book that don't require the gyro.

Let's now move on to the main chapter project and use what we've learned so far to build an app that combines what we know about the accelerometer with the support the Processing language provides for working with colors.

Simulating Sensors in the Emulator

Please keep in mind that features that rely on built-in sensor hardware cannot be emulated on the desktop computer. Because the emulator doesn't have a built-in accelerometer, for example, it can only give you a default value. The emulator does a good job of showing us whether the Processing sketch is compiling happily, but we can't get to an actual user experience. If you'd like to explore further how the emulator can be fed with "simulated" sensor values, you need to download additional software.

Build a Motion-Based Color Mixer and Palette

We're going to build a color mixer that generates hues by mapping the orientation of an Android device relative to its x-, y-, and z-axes to the R, G, and B values of a `color` variable. We've already discussed the Processing `color` type in Using the Color Type. When the sketch is complete, as shown in the image here, you'll be able to create every hue available to your device by rotating it in three-dimensional space.

In Erase a Palette with a Shake, we'll add a feature that lets you erase the stored colors by shaking the device. The color mixer will help us to get a better sense of the Processing `color` type and the value ranges of the accelerometer motion sensor, and it will provide us with a good foundation for working within the device coordinate system.

Mix a Color

Now let's move ahead and connect the accelerometer to change color hues. Since we successfully registered the accelerometer earlier, we can now take the code above to the next level for our color mixer project. The global variables `accelerometerX`, `accelerometerY`, and `accelerometerZ` keep track of raw values already, and it's a small step now to tie color values to device orientation. Earlier we observed magnitude values roughly in the range of `-10` and `10` for each axis. We can now map these raw values to the RGB color spectrum in the default target range of `0..255`. For that, we use the handy `map()` method, which takes one number range (in this case, incoming values of `-10..10`), and maps it onto another (our target of `0..255`):

Here's a description of `map()` parameters. Once we've learned how to use it, we'll find ourselves using it all the time:

```
map(value, low1, high1, low2, high2)
```

- `value` Incoming value to be converted
- `low1` Lower bound of the value's current range
- `high1` Upper bound of the value's current range
- `low2` Lower bound of the value's target range
- `high2` Upper bound of the value's target range

Now let's use `map()` to assign accelerometer values to the three values of an RGB color, and let's use `background()` to display the result, as shown in the image below.

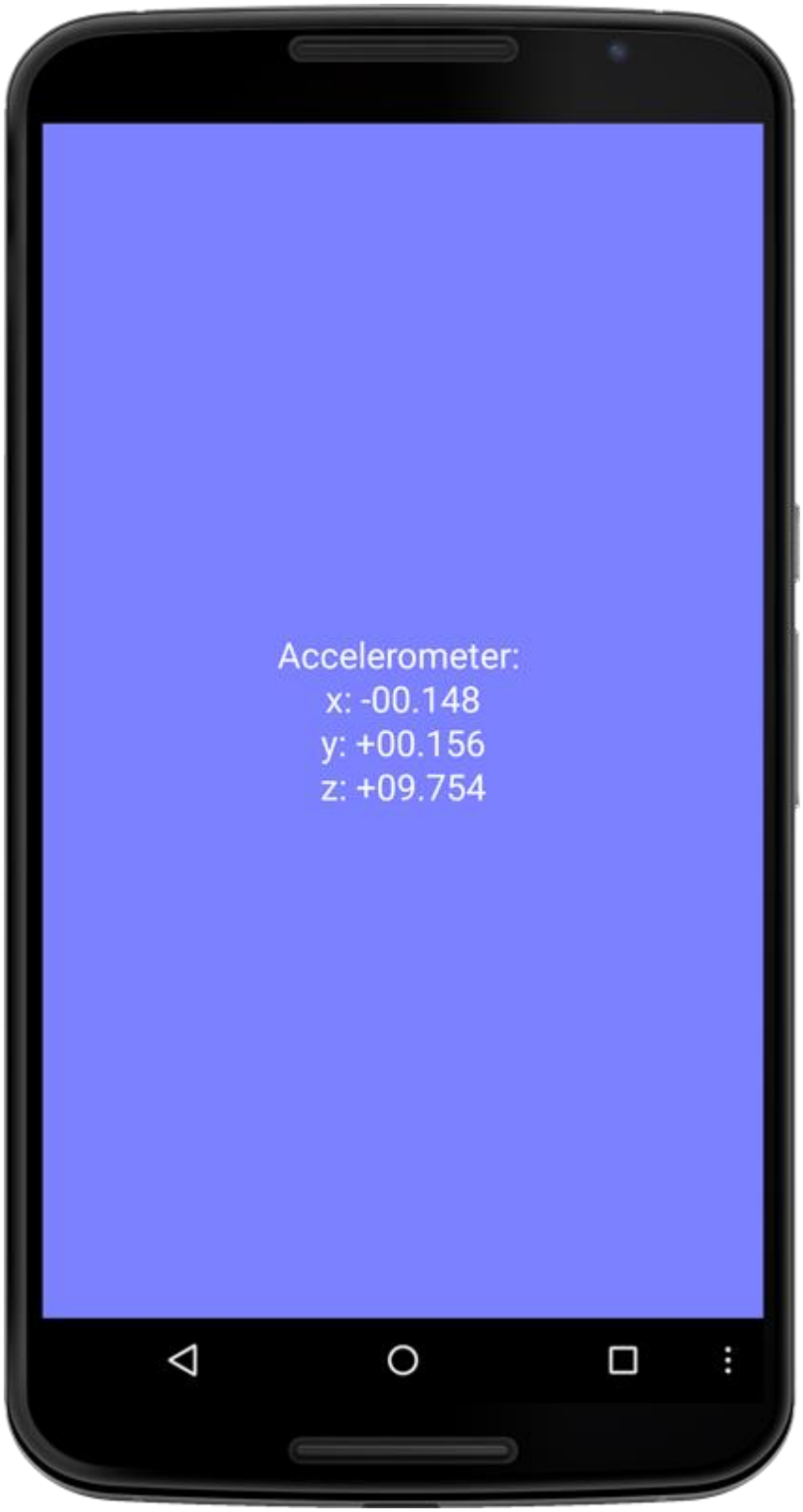


Figure 3.3 — Mapping accelerometer values to RGB color

Accelerometer values for each axis in the range of `-10..10` are mapped to about 50 percent red, 50 percent green, and 100 percent blue values, resulting in a purple background.

We need to add the accelerometer bounds for each axis and `map()` the values to three variables, called `r`, `g`, and `b`. Add the code snippet below to the Accelerometer.pde, at the beginning of `draw()` and adjust the `background()` method:

```
float r = map(accelerometerX, -10, 10, 0, 255);
float g = map(accelerometerY, -10, 10, 0, 255);
float b = map(accelerometerZ, -10, 10, 0, 255);
background(r, g, b);
```

The three color variables (`r`, `g`, and `b`) now translate sensor values in the range of `-10..10` to color values of `0..255`. The sketch then looks something like Figure 3.3.

Sensors/AccelerometerColor/AccelerometerColor.pde

```
import ketai.sensors.*;

KetaiSensor sensor;
float accelerometerX, accelerometerY, accelerometerZ;
float r, g, b;

void setup()
{
  sensor = new KetaiSensor(this);
  sensor.start();
  orientation(PORTRAIT);
  textAlign(CENTER, CENTER);
  textSize(72);
}

void draw() {
  float r = map(accelerometerX, -10, 10, 0, 255);
  float g = map(accelerometerY, -10, 10, 0, 255);
  float b = map(accelerometerZ, -10, 10, 0, 255);
  background(r, g, b);
  text("Accelerometer: \n" +
    "x: " + nfp(accelerometerX, 2, 3) + "\n" +
    "y: " + nfp(accelerometerY, 2, 3) + "\n" +
    "z: " + nfp(accelerometerZ, 2, 3), width/2, height/2);
}

void onAccelerometerEvent(float x, float y, float z) {
  accelerometerX = x;
  accelerometerY = y;
  accelerometerZ = z;
}
```

With this small addition, let's run the sketch on the device.

Run the App

When you run the sketch on the device, notice how the `background()` changes when you tilt or shake it. You are starting to use sketches and ideas from previous sections and reuse them in new contexts. The translation from raw sensor values into a color mixer project is not a big step. To understand how the accelerometer responds to your movement, it is a bit more intuitive to observe color changes displayed on the Android screen rather than fast-changing floating point values.

Now look more closely at the display as you rotate the device. Notice how the red value is linked to rotation around the x-axis, green to the y-axis, and blue to the z-axis. This helps us

figure out how the Android coordinate system is aligned with the actual device. The coordinate system does not reconfigure when the screen orientation switches from `PORTRAIT` to `LANDSCAPE`. This is why we locked the app into `orientation(PORTRAIT)`. We don't have to maintain the one-to-one relationship between the device coordinate system and the Processing coordinate system, but we'd sure have a harder time learning about it. Let's now figure out how to save the colors we generate.

Save a Color

To save any color that we create by rotating our device about its three axes, we need a container that is good for storing color values. Processing provides us with the `color` type, which we looked at briefly in the previous chapter, Using the Color Type.
To implement the color picker, let's rework our accelerometer code, and add a variable named `swatch` to store whatever color we pick when we tap the screen. We can then display the color pick value in an area at the bottom half of the screen, as shown here:

Figure 3.4 — Saving a color swatch

The image shows a color picked from all the possible hues Android can generate, stored in a color swatch.

Let's also display the individual values that correspond to the red, green, and blue variables as text using the `red()`, `green()`, and `blue()` methods to extract color values from the `swatch` color variable.

Sensors/AccelerometerColorPicker/AccelerometerColorPicker.pde

```
import ketai.sensors.*;

KetaiSensor sensor;
float accelerometerX, accelerometerY, accelerometerZ;
color swatch;                                                  // 1
float r, g, b;

void setup()
{
  sensor = new KetaiSensor(this);
  sensor.start();
  orientation(PORTRAIT);
  textAlign(CENTER, CENTER);
  textSize(72);
}

void draw()
{
  // remap sensor values to color range
  r = map(accelerometerX, -10, 10, 0, 255);
  g = map(accelerometerY, -10, 10, 0, 255);
  b = map(accelerometerZ, -10, 10, 0, 255);
  // assign color to background
  background(r, g, b);
  // color picker
  fill(swatch);                                                 // 2
  rect(0, height/2, width, height/2);                           // 3
  fill(0);
  text("Picked Color: \n" +
    "r: " + red(swatch) + "\n" +                                // 4
    "g: " + green(swatch) + "\n" +
    "b: " + blue(swatch), width*0.5, height*0.75);
}

void onAccelerometerEvent(float x, float y, float z)
{
  accelerometerX = x;
  accelerometerY = y;
  accelerometerZ = z;
}

void mousePressed()
{
  // updating color value, tapping top half of the screen
  if (mouseY < height/2)
    swatch = color(r, g, b);                                    // 5
}
```

Let's take a second look at the methods we've added.

---

1. Declare the variable `swatch` to be of type `color`.
2. Apply the swatch color to the `fill()` before drawing the color picker rectangle.
3. Draw the color picker rectangle.
4. Extract the red, green, and blue values individually from the `swatch` color.

5. Update the `swatch` color.
   Let's test the app.

Run the App

Tapping the top half of the screen stores the current `swatch` color, which appears as a strip of color on the bottom half of the screen. The numeric color values displayed as text on the bottom of the screen are taken directly from the `swatch` variable.
The sequence of events can be summarized as follows: We receive the accelerometer values from the hardware sensor and remap them into color values that we then display via the `background()` method in real time. When we tap the screen and pick a color, all three color values are stored in `swatch`. The numeric color value displayed as text is derived directly from the `swatch` color by using Processing's `red()`, `green()`, and `blue()` extraction methods, grabbing each value from `swatch` individually.
Clearly, though, storing one color is not enough. We've organized the screen and code so we can handle multiple colors, so let's take it a step further. We want to store multiple colors in such a way that we can recall them later individually. To implement this effectively, we need a color array.

Build a Palette of Colors

In this section, we'll build a palette of colors using a list of colors, or a color array, as illustrated in Figure 3.5. When you see a color you like, you'll be able to store it as one of eight swatches on the screen. In our example, we are dealing with a color array and we want to store a list of colors in a `palette[]` array. Each data/color entry in the list is identified by an index number that represents the position in the array. The first element is identified by the index number, `[0]`; the second element, `[1]`; and the last element, `palette.length-1`. We need to define the array `length` when we create the array.
ArrayList is an alternative here because it is able to store a varying number of objects. It's great, but it has a steeper learning curve. More info is available at http://processing.org/reference/ArrayList.html.

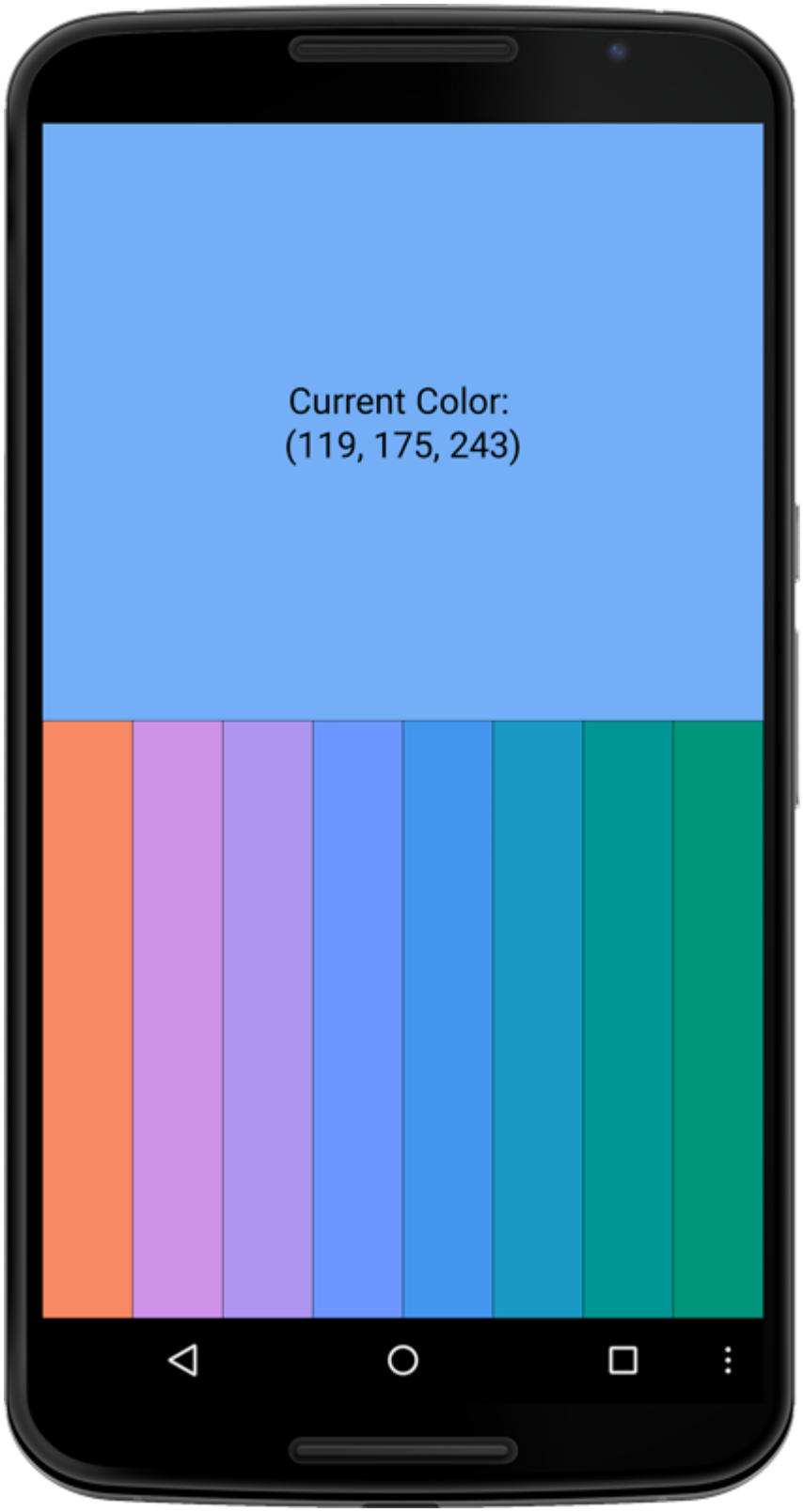


Figure 3.5 — Color mixer app

The image shows the color determined by the device orientation on the top half of the screen and the palette of saved colors at the bottom.

We can create arrays of any data type, for example `int[]`, `String[]`, `float[]`, and `boolean[]`. For a color array that stores up to, let's say, eight colors, we need to change the `swatch` variable from the previous AccelerometerColorPicker.pde, into this:

```
color[] palette = new color[8];
```

As a result, we can then store eight colors sequentially within the `palette` array. This touches on a prime programming principle: build the code to be as adaptable and versatile as possible. In our case, we want the app to work with any number of colors, not just eight. So we need to aim at more (`n`) colors and introduce a `num` variable that can be set to the amount we want to work with (`8`). Sure, the UI might not adapt as neatly if we set `num` to `100`, for example. But the code should be able to handle it without breaking. With adaptability in mind, we also program the GUI independent of the screen's size and resolution. In a diverse and rapidly changing device market, this approach prepares the app for a future filled with Android devices of every conceivable size.

Now that we have an array in which to store our colors, let's talk about its sidekick: the `for` loop. Because arrays are equivalent to multiples, the `for` loop is typically used to parse the array. It's designed to iterate a defined number of times, here `num` times, until it reaches the end of our `palette`. The init, test, and update conditions in a `for` loop are separated by semicolons, here with `i` serving as the counter variable.

When Processing encounters the `for` loop, the counter variable is set to the init condition (`i=0`) and then tested (`i<num`); if the test passes, all statements in the loop are executed. At the end of the loop, the counter variable is updated (`i++`), which here means incremented by one and then tested again, and if the test passes, all statements in the loop are executed. This continues until the test condition is `false`, and Processing continues to interpret the statements following the `for` loop.

Let's now put this into the context of our color mixer sketch.

Sensors/AccelerometerColorPickerArray/AccelerometerColorPickerArray.pde

```
import ketai.sensors.*;
KetaiSensor sensor;
float accelerometerX, accelerometerY, accelerometerZ;
float r, g, b;
int num = 8;                                                    // 1
color[] palette = new color[num];                               // 2
int paletteIndex = 0;                                           // 3

void setup()
{
  sensor = new KetaiSensor(this);
  sensor.start();
  orientation(PORTRAIT);
  textAlign(CENTER, CENTER);
  textSize(72);
}

void draw()
{
  // remap sensor values to color range
  r = map(accelerometerX, -10, 10, 0, 255);
```

```
  g = map(accelerometerY, -10, 10, 0, 255);
  b = map(accelerometerZ, -10, 10, 0, 255);
  // assign color to background
  background(r, g, b);
  fill(0);
  text("Current Color: \n" +
    "(" + round(r) + ", " + round(g) + ", " + round(b) + ")",     // 4
  width*0.5, height*0.25);
  // color picker
  for (int i=0; i<num; i++) {                                     // 5
    fill(palette[i]);                                             // 6
    rect(i*width/num, height/2, width/num, height/2);             // 7
  }
}

void onAccelerometerEvent(float x, float y, float z)
{
  accelerometerX = x;
  accelerometerY = y;
  accelerometerZ = z;
}

void mousePressed()
{
  // updating color value, tapping top half of the screen
  if (mouseY < height/2) {
    palette[paletteIndex] = color(r, g, b);                       // 8
    if (paletteIndex < num-1) {
      paletteIndex++;                                             // 9
    }

    else {
      paletteIndex = 0;                                           // 10
    }
  }
}
```

Let's take a look at the main additions to the sketch.

---

1. Set the quantity of colors to be stored to `8`.
2. Set up the color array to hold the previously defined number of colors.
3. Set an index variable that indicates the current color we are working with in the palette, represented by the index number in the color array.
4. Round the floating point value of the color value to an integer so we can read it better.
5. Iterate through the color array using a `for` loop.
6. Step through all the colors in the list using the `for` loop, and set the fill color for the rectangle to be drawn.
7. Display each color in the array with a rectangle set to the individual fill color in the list. Rectangles are displayed in sequence on the bottom of the screen; their width and position is defined by the number of colors in the list. The rectangle size is determined by the screen width and then divided by the total number of swatches, `num`.
8. Assign the three color values to the palette, cast as color type.
9. Increment the index number, moving on to the next position in the list.

10. Reset the index when the `palette` is full.
    Let's run the sketch now.

Run the App

On the device, we can tap the screen and the color swatches on the bottom of the screen update to the current color we've mixed by moving the device. If we fill all the swatches, it continues again at the beginning of the buffer. This is because we are setting `paletteIndex` back to `0` when the array reaches its end.
Building on the code we've previously developed, we've compiled a number of features into a color mixer prototype. This iterative process is typical when building software. We take small and manageable steps, test/run frequently, and make sure we have always saved a stable version of the code we are working with. This is good practice so that we can always have a fallback version if we get stuck or called away.

Now that we've used the accelerometer to mix and save colors, we've also established that device motion is a UI feature that allows us to interact with the app. We can now continue to build on device motion and add a shake to clear all swatches in the palette. How can we detect a shake?

Erase a Palette with a Shake

Shaking a device can be used as a deliberate gesture for controlling UI elements. On smart phones, it is typically assigned to the Undo command so that a shake can reverse or clear a prior action. Let's take a look at how this gesture can be detected and used by our color mixer.

What is a shake? When we move the device abruptly side to side, forward or backward, up or down, the idea is that our sketch triggers a "shake" event. For the color mixer, we want to use the shake for clearing out all color swatches. The shake needs to be detected no matter how we hold the device and independent of what's up or down. You might already anticipate the issue: we know well that the accelerometer is the ideal sensor for us to detect a shake, but we can't just use its x, y, or z values to trigger the shake when a certain threshold is reached because we can't assume our swaying gesture is aligned any of these axes. So this approach won't work and we need something else.

Any movement in space can be described with a three-dimensional vector that describes both the magnitude and the direction of the movement. You might (or might not) envision such a vector as an arrow in three-dimensional space—a visual representation of the mathematical

construct from the field of trigonometry. A vector is ideal to handle all three axes from our accelerometer.

We are using the `PVector` class again to store three variables in one package, which we are going to use to detect the shake. If we imagine how the vector would react to a shake, it would change erratically in direction and magnitude. In comparison, movement at a constant velocity causes no significant changes to the vector. Hence, if we continuously compare the movement vector of our device to the previous vector, frame by frame, we can detect changes when they reach a certain threshold.
Processing provides a few very useful vector math methods, including `angleInBetween(vector1, vector2)`, to calculate the angle between two given vectors. So if we compare the current accelerometer vector with the vector of the previous frame, we can now determine their difference in angle, summarized into a single numeric value. Because this value describes angular change, we use a threshold to trigger the shake. For now, let's say this threshold angle should be `45` degrees. Alternatively, we could use the `mag()` method to detect a sudden change to the vector's magnitude. We'll work with the change to the vector angle in this example. OK, let's put it together.

Sensors/ColorPickerComplete/ColorPickerComplete.pde

```
import ketai.sensors.*;
KetaiSensor sensor;
PVector accelerometer = new PVector();                  // 1
PVector pAccelerometer = new PVector();                 // 2
float r, g, b;
int num = 8;
color[] palette = new color[num];
int paletteIndex = 0;

void setup()
{
  sensor = new KetaiSensor(this);
  sensor.start();
  orientation(PORTRAIT);
  textAlign(CENTER, CENTER);
  textSize(72);
}

void draw()
{
  // remap sensor values to color range
  r = map(accelerometer.x, -10, 10, 0, 255);            // 3
  g = map(accelerometer.y, -10, 10, 0, 255);
  b = map(accelerometer.z, -10, 10, 0, 255);
  // calculating angle between current and previous accelerometer vector in radians
  float delta = PVector.angleBetween(accelerometer, pAccelerometer);  // 4
  if (degrees(delta) > 45) {                            // 5
    shake();
  }
  // assign color to background
  background(r, g, b);
  fill(0);
  text("Current Color: \n" +
    "(" + round(r) + ", " + round(g) + ", " + round(b) + ")",
  width*0.5, height*0.25);
  // color picker
```

```
  for (int i=0; i<num; i++) {
    fill(palette[i]);
    rect(i*width/num, height/2, width/num, height/2);
  }
  // storing a reference vector
  pAccelerometer.set(accelerometer);                    // 6
}

void onAccelerometerEvent(float x, float y, float z)
{
  accelerometer.x = x;                                  // 7
  accelerometer.y = y;
  accelerometer.z = z;
}

void mousePressed()
{
  // updating color value, tapping top half of the screen
  if (mouseY < height/2) {
    palette[paletteIndex] = color(r, g, b);
    if (paletteIndex < num-1) {
      paletteIndex++;
    }
    else {
      paletteIndex = 0;
    }
  }
}

void shake()
{
  // resetting all swatches to black
  for (int i=0; i<num; i++) {
    palette[i] = color(0);                              // 8
  }
  paletteIndex = 0;
}
```

Here's how we proceed to implement the shake detection using `PVector`.

1. Create a processing vector of type `PVector`.
2. Create a second vector as a reference to compare change.
3. Use the first `.x` component of the `accelerometer` vector. The second component can be accessed via `.y`, and the third component via `.z`.
4. Calculate the `delta` between the current and the previous vector.
5. Check the `delta` in radians against a threshold of `45` degrees.
6. Set the reference vector to the current one as the last step in `draw()`.
7. Assign raw accelerometers to the accelerometer `PVector`.
8. Set all palette colors in the array to the color black.

Let's run the code first and test the shake detection on the device. It helps us better understand some of the shake detection we talked about.

Run the App

If we play with the app, we can mix and pick colors as we did previously, as shown in Figure 3.5. Small wiggles go undetected. As soon as we move the device quickly and a shake is triggered, all color swatches are erased from the palette.

Let's compare some of the small adjustments we made to ColorPickerComplete.pde, to the previous AccelerometerColorPickerArray.pde, and check what we've added. First of all, we eliminated the three floating point variables we had used globally for incoming accelerometer values. Instead, we are using the `PVector`variable `accelerometer` to do the same job. This means we need to update our `map()` method so it uses the vector components `.x`, `.y`, and `.z` of the `accelerometer` vector.
We use the same approach for the `onAccelerometerEvent()` method, where incoming values are now assigned to individual vector components. To assign all three components at once to a vector, we can also use the `set()` method, as illustrated with `pAccelerometer` at the very end of `draw()`.
In terms of additions, we've added the `pAccelerometer` variable so we have something to compare against. We use `angleBetween()` to calculate the angle difference between the current and previous frame and assign it to `delta`. If the difference is larger than `45` degrees, we trigger the `shake()` method, resetting all `palette` colors to black and `paletteIndex` to `0`.
The `degrees()` method used here converts radian values provided by the `angleBetween()` method into degrees. Degrees (ranging `0`..`360`) are far more intuitive to work with than trigonometric measurements in radians, whose range is `0`..`TWO_PI`.
When you take a second look at the app, you can also confirm that `shake()` is triggered consistently independent of device rotation. The shake detection feature completes our color mixer project.

Wrapping Up

You've completed a series of sensor-based apps in this chapter, and you've worked with motion, position, and environment sensors using the `KetaiSensor` class of the Ketai library. You've learned the difference between software and hardware-based sensors and determined which sensors your device supports. You've used multiple sensors in one app, and you could go on to imagine other uses for motion-based features for your apps, such as speedometers for cars, shake detectors for putting your phone in silent mode, "breathometers" for biofeedback and analysis of breathing patterns...and the list goes on. You've also mastered working with color and learned how to mix and map it. You've learned how to work with Processing vectors to store multiple sensor values and to detect shakes.
Now that you know how the accelerometer can be used to determine the movement of an Android device, you are now ready to explore a more complex set of devices, such as GPS.

Our next topic explores how to determine the device's geographic location using Android's geolocation features, which are typically used for navigation and location-based services.

4. Using Geolocation and Compass

Location-based services have changed the way we navigate, share, and shop. Since the FCC ruling in 1996 requiring all US mobile operators to be able to locate emergency callers, location has become embedded in the images we take, the articles we blog, the commercials we watch, and the places we check into. These services rely on location information using latitude and longitude—and sometimes altitude—to describe a north-south and east-west position on the Earth's surface.

When we search for local information, get directions to public transportation, or find the nearest bar or bargain, the Android enables us to zero in on the information that is relevant to us at a particular geographic location. Because the device is aware of its own geolocation, we can navigate, detect where we are heading, and know how we are holding the device in relation to magnetic north. A built-in Global Positioning System (GPS) receiver, accelerometer, and digital compass allow the Android to have a full picture about its location and orientation, which plays an important role for navigation apps and location-based services.

Android apps make use of the Android's Location Manager to calculate a location estimate for the device. Its purpose is to negotiate the best location source for us and to keep the location up-to-date while we are on the move. Updates typically occur when the device detects that we've changed location or when a more accurate location becomes available. An Android device uses two different location providers to estimate its current geographic coordinates: `gps` on the one hand and `network` on the other, the latter based either on the calculated distance to multiple cell towers or on the known location of the Wi-Fi network provider to which we are connected. A special `passive` provider receives location updates from other apps or services that request a location.
Compared with the Global System for Mobile Communications (GSM) and Wi-Fi network localization, GPS is the most well-known and accurate method for determining the location of a device. With thirty-one GPS satellites orbiting about 20,000 kilometers above any spot on the Earth's surface twice a day (every 11 hours, 58 minutes), it's just fantastic how the fingertip-sized GPS receivers built into our smart phones are able to determine the device's latitude, longitude, and altitude at a theoretical accuracy of about three meters.

In this chapter, we'll build a series of navigation apps. We'll start by working with the Android's current geolocation. We'll continue by measuring how far we are located from a

predefined destination. Finally, we'll build an app that helps us navigate toward another mobile device.

Let's first take a look at how the Android device estimates its location.

Introducing the Location Manager

Given its ubiquitous use, working with geolocation data should be simple. In the end, it's just the latitude, longitude, and maybe altitude we are looking to incorporate into our apps. Because there are various location techniques, however, we are interacting with a fairly complicated system and continuously negotiating the best and most accurate method to localize the device. The Location Manager that does that work for us is a software class that obtains periodic updates of the device's geographic location from three sensors available on an Android phone or tablet, including a built-in GPS receiver, a cellular radio, and a Wi-Fi radio. Both the `KetaiLocation` and Android `Location` classes draw their data from the Location Manager, which in turn gets its information from the onboard devices.
Another localization method uses cellular tower signals to determine the location of a device by measuring the distances to multiple towers within reach. This triangulation method is less precise because it depends on weather conditions and relies on a fairly high density of cell towers.

The third method doesn't require GPS or cell towers at all but the presence of a Wi-Fi network. This technique uses the known locations of nearby Wi-Fi access points to figure out the approximate location of the mobile device. Wi-Fi access points themselves lack GPS receivers and therefore lack knowledge of their own geographic locations, but such information can be associated with their physical MAC (media access control) addresses by third parties.

The most notorious case was Google's now abandoned effort to associate GPS coordinates with the MAC address of every wireless access point it encountered as it photographed the streets of US cities and towns with GPS-enabled vehicles for its Google Maps Street View project.

Nowadays, we're the ones who do this work for Google whenever we take an Android device for a stroll. If we have activated Google's location service by selecting Settings ↦ "Location services" on the main menu of our device, then by default we have agreed to "collect anonymous location data" and that "collection may occur even when no apps are running." The MAC addresses of available Wi-Fi networks are sent to Google during this collection process, along with their geographic coordinates. The next user who walks through the same geographic area can then geolocate solely via the Wi-Fi network information, even if GPS is turned off.

It takes a few seconds for the Android to narrow the location estimate and improve its accuracy, so we typically need to start `KetaiLocation` as soon as the app launches. With fewer than ten lines of code, `KetaiLocation` can provide us with our geographic coordinates and notify us of changes in our location via the `onLocationEvent` callback method.
For the location-based apps we'll develop in this chapter, we'll use the following Ketai library and Android classes:

`KetaiLocation class`
A class that simplifies working with Android's Location Manager—it instantiates the Location Manager, registers for location updates, and returns geolocation data.
`Location`
A wrapper for Android's Location Manager that provides us with many useful methods for determining our position, bearing, and speed—if we're only interested in our location, we won't need this class, but we will use some of its features for later projects in this chapter.
Now let's take a look at the `KetaiLocation` methods we'll be using in this chapter.

Introducing GPS

The transmitters built into GPS satellites broadcast with about 50 watts, similar to the light bulb in a desk lamp, and yet the GPS module in the phone is able receive a sequence of numbers sent by all the satellites simultaneously, every microsecond. The atomic clock in each satellite takes care of that. The satellite doesn't know anything about us; it's only transmitting. The receiver in our mobile device makes sense of the transmission by deciphering the sequence of numbers the satellite sends. The GPS receiver then determines from the number sequence (which includes the time it was sent by the satellite) how far each individual radio signal has travelled, using the speed of light as its velocity. If a satellite is close by (about 20,000 kilometers), the signal would take about 67 microseconds to travel. The distance is measured by multiplying the time it has taken the radio signal to reach your phone by the speed of light.

We need to "see" at least four satellites to determine latitude, longitude, and altitude (or three if we assume an incorrect altitude of zero). It's clear that a 50-watt signal from 20,000 kilometers away cannot penetrate buildings. We can only "see" satellites if there are no obstructions. If the signal bounces off a building surface, the estimate is less accurate as a consequence. Because the satellite orbits are arranged so that there are at always six within the line of sight, it's fine if one or two are not "seen" or are inaccurate. Accuracy is higher for military receivers getting a signal every tenth of a microsecond, bringing it theoretically down to 0.3 meters (or about 1 ft). High-end receivers used for survey and measurement can increase accuracy even more—to within about 2 mm.

Working with the KetaiLocation Class

The `KetaiLocation` class is designed to provide us with the longitude, latitude, and altitude of the device, as well as the accuracy of that estimate. Besides the typical `start()` and `stop()` methods, `KetaiLocation`also provides a method to identify the location provider that has been used to calculate the estimate. Let's take a look.

- `onLocationEvent()` Returns the device location, including latitude, longitude, altitude, and location accuracy
- `latitude` Describes the angular distance of a place north or south of the Earth's equator in decimal degrees—positive `lat` values describe points north of the equator; negative values describe points south of the equator (for example, Chicago is located at `41.87338` degrees latitude in the northern hemisphere; Wellington, New Zealand, is located at `-41.29019` degrees latitude in the southern hemisphere).
- `longitude` Describes the angular distance of a place east or west of the meridian at Greenwich, England, in decimal degrees (for example, Chicago, which is west of the Greenwich meridian, is located at -87.648798 degrees longitude; Yanqi in the Xinjiang Province, China, is located at `87.648798` degrees longitude.)
- `altitude` Returns the height of the device in relation to sea level measured in meters
- `accuracy` Returns the accuracy of the location estimate in meters
- `getProvider()` Returns the identity of the location provider being used to estimate the location: `gps` or `network`—it does not distinguish between cellular or Wi-Fi networks.
  Before we can use data from the location provider, we need to take a look at the permissions the sketch needs to access this data.

Setting Sketch Permissions

By default, Android denies permissions to any app that requests access to private data or wants to perform privileged tasks, such as writing files, connecting to the Internet, or placing a phone call. Working with privileged information such as geolocation is no exception.

If we'd like to use the device's location data, we need to ask for permission. Android prompts the user to grant permission if an app requests permission that has not been given to the app before. The Processing IDE (PDE) helps us administer permission requests through the Android Permission Selector, which is available from the menu by selecting Android ↦ Sketch Permissions. There we'll find a list of all the permissions that can be requested by an app on the Android.

As illustrated in Figure 4.1 below, the location permissions need to be set for this app. When we run the sketch on the device and Processing compiles the Android package, it generates a

so-called `AndroidManifest.xml` file that corresponds to our permission settings. We don't need to worry much about the details of `AndroidManifest.xml`; however, as follows, we can see how Processing's Permissions Selector translates our selection into a user-permissions list.

code/Geolocation/Geolocation/AndroidManifest.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1" android:versionName="1.0" package="">
  <uses-sdk android:minSdkVersion="8"/>
  <application android:debuggable="true"
    android:icon="@drawable/icon" android:label="">
    <activity android:name="">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
  <uses-permission
    android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS"/>
</manifest>
```

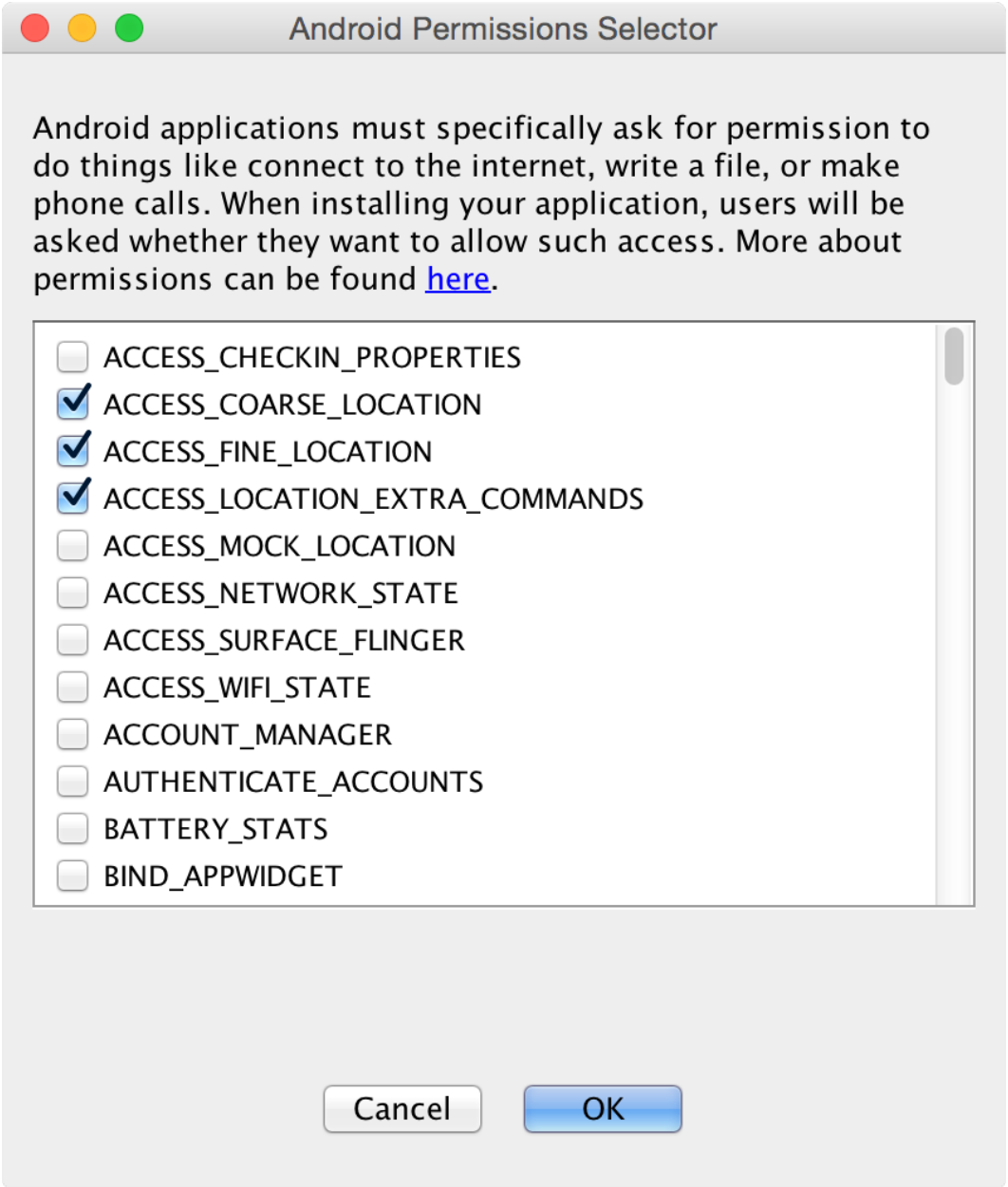


Figure 4.1 — Sketch permissions.

The Android Permissions Selector lists all permissions that can be requested by the Android app. The location permissions required by the first geolocation app are checked.

To make sure our location app is able to work with location data, we need to enable Google's location service on the device under Settings ↦ "Location" and agree to the prompt, shown here:

```
Let Google's location service help apps determine location.
This means sending anonymous location data to Google,
even when no apps are running.
```

Otherwise our app will will display the following warning:

```
Location data is unavailable. Please check your location settings.
```

We've programmed this warning into our sketch, assuming that `getProvider()` returns `none`, which is also the case if Google's location service is switched off.
Let's go ahead and write our first location-based app.

Determine Your Location

As our first step, let's write some code to retrieve and display your device's location, as shown in Figure 4.2.

Figure 4.2 — Displaying location data.

The screen output shows geolocation (latitude, longitude, and altitude), estimation accuracy (in meters), and the current location provider.

This exercise will familiarize us with the kinds of values we'll use to determine our current location on the Earth's surface. Let's display the current latitude, longitude, and altitude on the screen as determined by the Location Manager, as well as display the accuracy of the values and the provider that is used for the calculation. The following example uses `KetaiLocation` to gather this info.

code/Geolocation/Geolocation/Geolocation.pde

```
import ketai.sensors.*;
KetaiLocation location;                                    // 1
double longitude, latitude, altitude;
float accuracy;

void setup() {
  orientation(LANDSCAPE);
  textAlign(CENTER, CENTER);
  textSize(72);
  location = new KetaiLocation(this);                      // 2
```

```
}

void draw() {
  background(78, 93, 75);
  if (location.getProvider() == "none")                          // 3
    text("Location data is unavailable. \n" +
      "Please check your location settings.", width/2, height/2);
  else
    text("Latitude: " + latitude + "\n" +                        // 4
      "Longitude: " + longitude + "\n" +
      "Altitude: " + altitude + "\n" +
      "Accuracy: " + accuracy + "\n" +
      "Provider: " + location.getProvider(), width/2, height/2);
}

void onLocationEvent(double _latitude, double _longitude,
  double _altitude, float _accuracy) {                           // 5
  longitude = _longitude;
  latitude = _latitude;
  altitude = _altitude;
  accuracy = _accuracy;
  println("lat/lon/alt/acc: " + latitude + "/" + longitude + "/"
    + altitude + "/" + accuracy);
}
```

Let's take a look at how the newly introduced class and methods are used in this example.

---

1. Declare the variable `location` to be of type `KetaiLocation`. We'll use this variable to store location updates.
2. Create the `KetaiLocation` object we've called `location`.
3. Check whether we currently have a location provider via the `getProvider()` method.
4. Display location values `latitude`, `longitude`, `altitude`, `accuracy`, and the location provider using `getProvider()`.
5. Whenever a location update occurs, use the `onLocationEvent()` method to retrieve location data and print them to the screen.
   Ketai defaults the Location Manager to provide location updates every ten seconds or whenever the device moves more than one meter. This preset number is geared toward applications that strive for a certain level of accuracy. You can change this update rate by calling the `KetaiLocation` method `setUpdateRate(int millis, int meters)`. The app will try to retrieve a `gps` location first via the Location Manager, and if that fails it will fall back to `network` localization.

Run the App

With the location service turned on, let's run the sketch on our device. Type or copy the code above into your Processing environment and run it on your Android phone or tablet. You should now see your current geographic location. If you are inside a building, chances are that the location estimate is based on the `network` provider, as shown in . In this example, the Location Manager calculated the estimate with an accuracy of `46` meters, which means that the estimate can range from 46 meters, worst case, to "right on" in the best case.

Next, let's disconnect the phone and take it for a little walk. Step outside your building. Watch for a location update and a change in provider.

Great—now head back inside. Take a peek again at your latitude and longitude coordinates, and double-check the location accuracy in Google Maps, as described. How far off are you? If you walk a block, you will be able to observe a change to the third digit after the decimal in either the latitude or longitude, depending on where you are headed. The seemingly small change in this digit represents about 200 feet, which brings us to our next application.

Working with the Location Class

The event method `onLocationEvent()` we worked with earlier returns the latitude, longitude, altitude, and accuracy of the device location—or alternatively, an Android `Location` object. If we look at the `onLocationEvent()` method in more detail, we can use it with the following sets of parameters:

- `onLocationEvent(double latitude, double longitude, double altitude, float accuracy)` Four parameters return the latitude, longitude, altitude, and accuracy of the location estimate.
- `onLocationEvent(Location location)` One parameter returns an Android location object, where Android location methods can be applied directly.

Depending on what location data we need for our location-based app, we can choose our preferred set of parameters from either `latitude`, `longitude`, `altitude`, `accuracy`, or the `Location` type. We can also select a few parameters if we don't require them all.

The `Location` object returned in the second iteration of the `onLocationEvent()` implementation listed here allows us to access any Android `Location` method.

The `Location` class is loaded with useful methods for dealing with the data they contain, and it is a great way to package returned location data for use in an app. Ketai gives us complete access to the `Location`class; let's take a look at some of the `Location` methods we'll be working with.

- `getBearing()` Returns the direction of travel in degrees, measured clockwise in relation to magnetic north
- `getSpeed()` Returns the speed of the device over ground in meters per second (One meter per second is equivalent to `2.236` miles per hour.)
- `distanceTo()` Returns the distance to a given location in meters (The method takes a `Location` object as parameter.)
- `setLatitude()` Sets the latitude of a `Location`.
- `setLongitude()` Sets the longitude of a `Location`.

Now let's work on the next project, where we'll put `Location` methods to work and write an app that determines the distance between two locations.

Determine the Distance Between Two Locations

In this project, we'll calculate the distance between our current device location and another fixed location that we predetermine. We provide the fixed location coordinate through latitude and longitude decimal degree values. To get a better idea about what those values represent, let's first obtain the latitude and longitude values of our current geographic location via Google Maps.

Browse Google Maps on your desktop, and find a location close to a landmark you recognize and know your approximate distance to. Now right-click anywhere close to that landmark on the map. From the menu, choose "Directions to here." You need to be zoomed in all the way so Maps doesn't grab the close-by landmark and display only the landmark's name instead of the latitude and longitude. If you hit a non-landmark spot, Maps will display the `lat` and `lon` values of the location inside the site's destination field. My current location in Brooklyn, for instance, looks like this:

```
40.67937996,-73.9377679
```

Write down your location—we'll use it in the next project. If you use the format shown above, `lat, lon`(latitude comma longitude), Google Maps will understand and take you to this location. This approach is a quick and easy way to double-check a location when you want to test a location app.

Now let's create a sketch to determine the distance between a fixed point and the device, as shown in Figure 4.3.

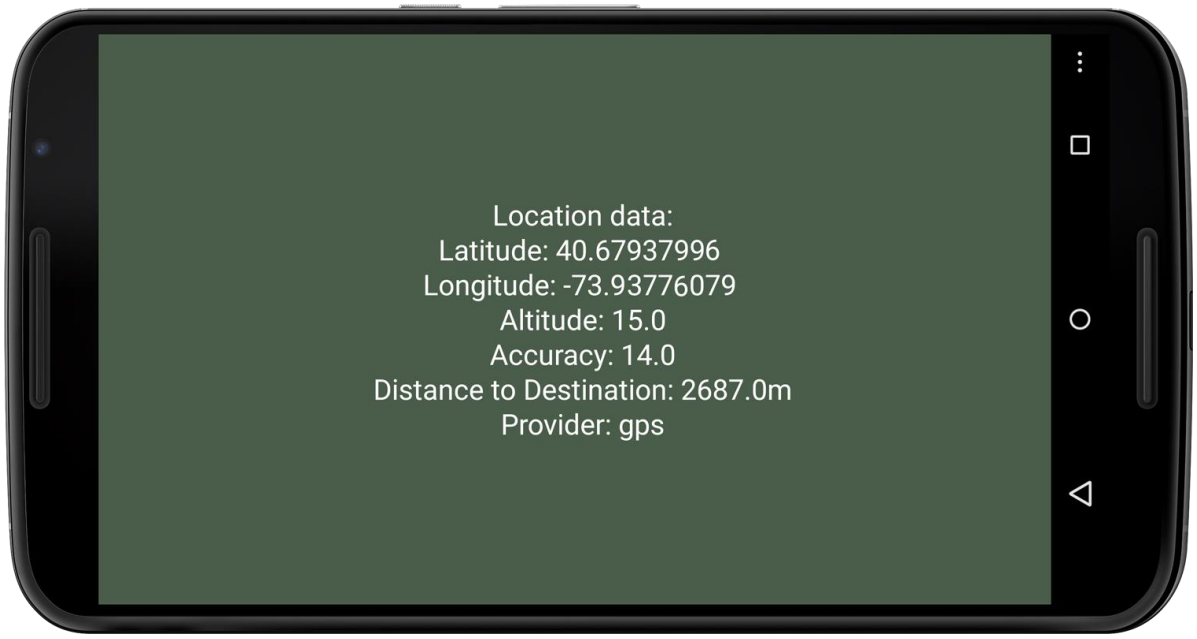


Figure 4.3 — Calculating distance.

The screen output shows the device's current location, the calculated distance to the predefined `bam` destination, and the current location provider.

We'll use both the `KetaiLocation` and Android's `Location` classes. `KetaiLocation` provides us with the current device latitude and longitude, `Location` lets us define a destination location object that we can use to calculate the `distance` between both points. Finally, we'll use the `round()` method to calculate the closest integer and display full meters.

Let's take a look at the code.

code/Geolocation/LocationDistance/LocationDistance.pde

```
import ketai.sensors.*;
double longitude, latitude, altitude, accuracy;
KetaiLocation location;
Location bam;

void setup() {
  location = new KetaiLocation(this);
  // Example location: the Brooklyn Academy of Music
  bam = new Location("bam");                                    // 1
  bam.setLatitude(40.686818);
  bam.setLongitude(-73.977779);
  orientation(LANDSCAPE);
  textAlign(CENTER, CENTER);
  textSize(72);
}

void draw() {
  background(78, 93, 75);
  if (location.getProvider() == "none") {
    text("Location data is unavailable. \n" +
      "Please check your location settings.", 0, 0, width, height);
  } else {
    float distance = round(location.getLocation().distanceTo(bam));  // 2
    text("Location data:\n" +
      "Latitude: " + latitude + "\n" +
      "Longitude: " + longitude + "\n" +
      "Altitude: " + altitude + "\n" +
      "Accuracy: " + accuracy + "\n" +
      "Distance to Destination: "+ distance + "m\n" +
      "Provider: " + location.getProvider(), 20, 0, width, height);
  }
}

void onLocationEvent(Location _location)                        // 3
{
  //print out the location object
  println("onLocation event: " + _location.toString());          // 4
  longitude = _location.getLongitude();
  latitude = _location.getLatitude();
  altitude = _location.getAltitude();
  accuracy = _location.getAccuracy();
}
```

Here's what's new in this sketch compared to our previous project.

---

1. Create an Android `Location` object to store a fixed location against which to compare your current device location. I named mine "bam" (for the Brooklyn Academy of Music). We'll use the `setLatitude()`and `setLongitude()` Android methods to set its values.

2. Use the `distanceTo()` method to compare the device's location via `location.getLocation()` with the fixed `bam` location. The `round()` method calculates the closest integer number to the floating point value returned by `distanceTo()`.
3. Receive a location update using `onLocationEvent()`, which now returns a `Location` object instead of individual values for latitude, longitude, altitude, and accuracy. The different parameter options for `onLocationEvent()` are described next.
4. Use the Android `toString()` method to print a concise, human-readable description of the location object to the console.
   Let's try this sketch.

Run the App

Run the sketch on the device and take a look at the location info, including the distance to your fixed location. In this example, the app calculates the distance to the `bam` `Location` in Brooklyn's Fort Greene neighborhood. So the `distance` will vary significantly depending on the state or country you are currently located in.

Go back to the geolocation you've previously noted via Google Maps. Use this location now to adjust the `bam` location object in `setup()`, and adjust the `setLatitude()` and `setLongitude()` parameters to match your location. Feel free to also adjust the `bam` variable and the `Location` name called `"bam"` to reflect your location—it's not crucial for this sketch though.

Rerun the sketch on the device, and notice how the `distance` has changed. You should be able to confirm the distance to the landmark you've Googled using this app.

Now that you know how to calculate the distance between two points, you're ready to use some additional Android `Location` methods to determine the bearing and speed of an Android phone or tablet when it's in motion. We'll take a look at that topic in the next section.

Determine the Speed and Bearing of a Moving Device

To determine the speed and bearing of a device, three other useful Android `Location` methods can be applied in ways that are similar to what we did with `distanceTo()`. Let's create a new sketch and focus for a moment on travel speed and bearing.

We've mastered latitude, longitude, and altitude and calculated the distance between two points. The next step is to determine where we are heading and how fast we are going.

Because these parameters are only fun to test while we are on the move, let's create a simple

new sketch that focuses on speed and bearing. Then we'll bring it all together in the next section, Find Your Way to a Destination.

Let's take a look.

code/Geolocation/LocationSpeed/LocationSpeed.pde

```
import ketai.sensors.*;
KetaiLocation location;
float speed, bearing;

void setup() {
  orientation(LANDSCAPE);
  textAlign(CENTER, CENTER);
  textSize(72);
  location = new KetaiLocation(this);
}

void draw() {
  background(78, 93, 75);
  text("Travel speed: "+ speed + "\n"
    + "Bearing: "+ bearing, 0, 0, width, height);
}

void onLocationEvent(Location _location) {
  println("onLocation event: " + _location.toString());
  speed = _location.getSpeed();                              // 1
  bearing = _location.getBearing();                          // 2
}
```

Here are the two new Android `Location` methods we are using for this sketch.

1. Get the current travel speed using the Android `Location` method `getSpeed()`, which returns the speed of the device over ground in meters per second.
2. Get the current device bearing using the Android `Location` method `getBearing()`, which returns the direction of travel in degrees.

Let's run the sketch and get ready to go outside.

Run the App

Run the sketch on the device and take the Android for a little trip—again, the app can only give us reasonable feedback when we're on the move. The `onLocationEvent()` method returns a `Location` object containing speed and bearing info, which we extract using the `getSpeed()` method and the `getBearing()` method. The numeric feedback we receive on speed and bearing is useful for the navigation apps we write. If we want to calculate bearing toward a fixed destination instead of magnetic north, however, we should use the `bearingTo()` method instead of `getBearing()`.

We'll look at `bearingTo()` in the next section, where we'll build on a destination finder app.

Find Your Way to a Destination

If we are heading toward a destination and want to use our Android device like a compass to guide us there, we need to calculate the angle toward the destination relative to our location. And to make it at all useful, we also need to consider the direction the device is "looking" relative to geographic north. When used together, these two numbers can then successfully point us to where we want to go. We'll build on the LocationDistance.pde, and add a simple triangle to our user interface that points toward our destination no matter which way the device itself is facing.

The core idea here is that we'll calculate the `bearing` and then use it to rotate a graphic object, a triangle, which will serve as our compass needle. The rotation of our graphic object and text will be performed by moving the triangle to the center of the screen using `translate()`. Then we'll `rotate()` the compass needle by the angle resulting from the difference of the device orientation toward north and the calculated `bearing`toward the destination. We'll calculate the `bearing` using the `bearingTo()` method, which returns values ranging `-180..180` measured from true north—the shortest path between our device location and the destination.
Then we'll draw the triangle and the text showing the distance to the destination in meters and miles. We convert the `distance` from the default measurement unit returned by the Android, meters, into miles by multiplying `distance` by `0.000621371192`. Because `bearing` is measured in degrees and so is the compass azimuth, we'll need to convert it into `radians()` first before performing the rotation. Degree values range `0..360` degrees and radians range `0..TW0_PI.` All trigonometric methods in Processing require parameters to be specified in radians.
We'll use the `PVector` class we've already used earlier so we can keep the code concise and don't use more variables than we need. For numeric feedback, we use the `mousePressed()` method to display the location values and the bearing we'll calculate.
Let's build.

Geolocation/DestinationCompass/DestinationCompass.pde

```
import ketai.sensors.*;
import android.location.Location;

KetaiLocation location;
KetaiSensor sensor;
Location destination;
PVector locationVector = new PVector();
float compass;                                                  // 1

void setup() {
  destination = new Location("bam");
  destination.setLatitude(40.686818);
  destination.setLongitude(-73.977779);
  location = new KetaiLocation(this);
  sensor = new KetaiSensor(this);
  sensor.start();
  orientation(PORTRAIT);
  textAlign(CENTER, CENTER);
```

```
  textSize(72);
  smooth();
}

void draw() {
  background(78, 93, 75);
  float bearing = location.getLocation().bearingTo(destination);  // 2
  float distance = location.getLocation().distanceTo(destination);
  if (mousePressed) {
    if (location.getProvider() == "none")
      text("Location data is unavailable. \n" +
        "Please check your location settings.", 0, 0, width, height);
    else
      text("Location:\n" +
        "Latitude: " + locationVector.x + "\n" +
        "Longitude: " + locationVector.y + "\n" +
        "Compass: "+ round(compass) + " deg.\n" +
        "Destination:\n" +
        "Bearing: " + bearing + "\n" +
        "Distance: "+ distance + " m\n" +
        "Provider: " + location.getProvider(), 20, 0, width, height);
  }
  else {
    translate(width/2, height/2);                                  // 3
    rotate(radians(bearing) - radians(compass));                   // 4
    stroke(255);
    triangle(-width/4, 0, width/4, 0, 0, -width/2);                // 5
    text((int)distance + " meters", 0, 60);
    text(nf(distance*0.000621, 0, 2) + " miles", 0, 140);          // 6
  }
}

void onLocationEvent(Location _location) {
  println("onLocation event: " + _location.toString());
  locationVector.x = (float)_location.getLatitude();               // 7
  locationVector.y = (float)_location.getLongitude();
}
void onOrientationEvent(float x, float y, float z, long time, int accuracy) { // 7
  compass = x;
  // Azimuth angle between magnetic north and device y-axis, around z-axis.
  // Range: 0 to 359 degrees
  // 0=North, 90=East, 180=South, 270=West
}
```

Let's take a look at the code additions.

---

1. Introduce the `compass` variable to store the rotation around the z-axis.
2. Apply the `bearingTo()` method to determine the direction of the destination pointer.
3. Move the triangle to the center of the screen using `translate()`. Translate horizontally by half of the `width` and vertically by half of the `height`.
4. Rotate the triangle toward the destination. The angle is calculated by subtracting the device `bearing`toward the destination from the device orientation toward north stored in `compass`. Both angles are calculated in degrees and need to be converted into `radians()` for the trigonometric `rotate()` method. `rotate()` adds a rotation matrix to the stack, which makes all objects drawn after the method call appear rotated in relation to the default screen orientation.

5. Draw the destination pointer using `triangle()`. Draw the triangle pointing up using three points, starting with the left base, followed by the right base, and finally by the top point, which provides direction.
6. Convert the distance to the destination from meters to miles.

7. Use the `PVector` variable `locationVector` to store the device latitude and longitude.
8. Receive bearing values from the `onOrientationEvent()` method, returning azimuth (z-axis), pitch (x-axis), and roll (y-axis).
   We are now using two methods, `onLocationEvent()` and `onOrientationEvent()`, that operate in concert with each other. One tracks the location of the device in latitude, longitude, and altitude values, and the other determines where the device is pointing.

Run the App

Let's run the app on the device and find out whether we are being pointed in the right direction. Make sure to set the correct permissions again, as we've discussed in Setting Sketch Permissions. For this test, it's quite helpful that we've looked up the destination earlier so we can better gage how well the app is doing.

If you tap the screen, you can observe raw device location values, the `compass` variable we've calculated, and the calculated `bearing` angle of the device. The `distance` toward the destination and the location provider are also displayed on the screen, as shown in Figure 4.4.

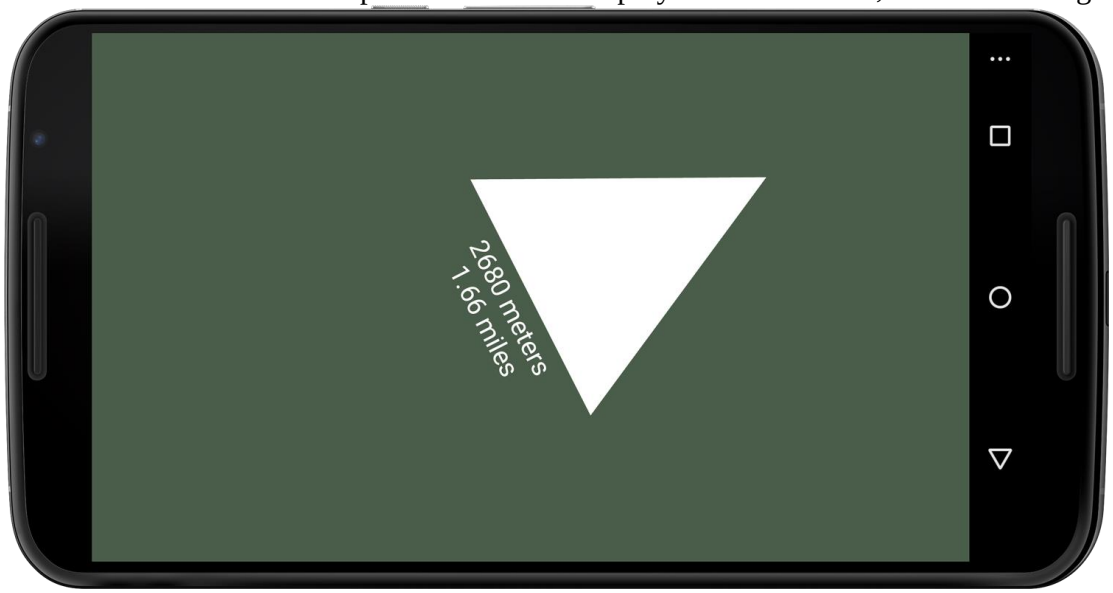


Figure 4.4 — Compass app.

The triangle points to the second device, whose distance is displayed at its base in meters and miles.

We've now used several pieces of location info in concert and created an app that guides us home (or to work, or wherever `destination` is pointing to). Before you rely on your app to find your way, please make sure `destination` is pointing to the right place.
Now that we've seen how to find our way to a fixed destination, the next task in line is to create an app that targets a moving destination. For our next project let's navigate toward another mobile device and address some of the challenges when it comes to sharing locations.

Find a Significant Other (Device)

At first sight, it seems there is not much of a difference between the compass app we've just made and one that guides us toward another mobile device. If we think about it, though, using a hard-coded latitude and longitude as we did in our previous sketch is quite different from retrieving another device's location data in real time. We'll explore networking techniques in detail in Chapter 6, Networking Devices with Wi-Fi. The difficulty is that two mobile devices separated by some distance will not share a common IP subnet that we can use to exchange our location data. So for this task, we need a shared place where each device can write its own latitude and longitude and where each can read the other device's location in return.

For this project, we'll use a web server to facilitate sharing, and we'll equip it with a simple PHP script that takes the location info from each device and writes it to a text file. If one device knows the (made-up) name of the other, it can look it up on that server and we'll have a significant-other location to navigate to. You can certainly download the script , and host it on your own web server as well.

Let's get started. This sketch works with the PHP script on the dedicated web server for this book project. If you point the `serverURL` variable to another destination, you'll store your locations there.

code/Geolocation/DeviceLocator/DeviceLocator.pde

```
import ketai.sensors.*;
double longitude, latitude, altitude, accuracy;
KetaiLocation location;                                          // 1
Location otherDevice;                                            // 2
KetaiSensor sensor;
String serverMessage = "";
String myName = "myNexusID";                                     // 3
String deviceTracked = "myNexusID";                              // 4
String serverURL = "http://ketaiLibrary.org/rapidAndroidDevelopment/location.php"; // 5
float compass;

void setup() {
  otherDevice = new Location("yourNexus");
  sensor = new KetaiSensor(this);
  sensor.start();
  location = new KetaiLocation(this);
  orientation(PORTRAIT);
  textAlign(CENTER, CENTER);
```

```
  textSize(72);
}

void draw() {
  background(78, 93, 75);
  float bearing = location.getLocation().bearingTo(otherDevice);
  float distance = location.getLocation().distanceTo(otherDevice);
  if (mousePressed) {
    if (location.getProvider() == "none")
      text("Location data is unavailable. \n" +
        "Please check your location settings.", 0, 0, width, height);
    else
      text("Location data:\n" +
        "Latitude: " + latitude + "\n" +
        "Longitude: " + longitude + "\n" +
        "Altitude: " + altitude + "\n" +
        "Accuracy: " + accuracy + "\n" +
        "Distance to Other Device: "+ nf(distance, 0, 2) + " m\n" +
        "Provider: " + location.getProvider()+ "\n" +
        "Last Server Message: " + serverMessage, 20, 0, width, height );
  }
  else {
    translate(width/2, height/2);
    rotate(radians(bearing) - radians(compass));
    stroke(255);
    triangle(-width/4, 0, width/4, 0, 0, -width/2);
    text((int)distance + " m", 0, 60);
    text(nf(distance*0.000621, 0, 2) + " miles", 0, 140);
  }
}

void onLocationEvent(Location _location)                          // 6
{
  // Print out the location object
  println("onLocation event: " + _location.toString());
  longitude = _location.getLongitude();
  latitude = _location.getLatitude();
  altitude = _location.getAltitude();
  accuracy = _location.getAccuracy();
  updateMyLocation();
}

void updateMyLocation()
{
  if (myName != "")
  {
    String url = serverURL+"?update="+myName+
      "&location="+latitude+","+longitude+","+altitude;           // 7
    String result[] = loadStrings(url);                           // 8
    if (result.length > 0)
      serverMessage = result[0];
  }
}

void mousePressed()
{
  if (deviceTracked != "")
  {
    String url = serverURL + "?get="+deviceTracked;               // 9
    String result[] = loadStrings(url);
    for (int i=0; i < result.length; i++)
      println(result[i]);
    serverMessage = result[0];
    // Let's update our target device location
    String[] parameters = split(result[0], ",");
    if (parameters.length == 3)                                   // 10
    {
      otherDevice = new Location(deviceTracked);
      otherDevice.setLatitude(Double.parseDouble(parameters[0]));
      otherDevice.setLongitude(Double.parseDouble(parameters[1]));
```

```
      otherDevice.setAltitude(Double.parseDouble(parameters[2]));
    }
  }
  updateMyLocation();                                          // 11
}

void onOrientationEvent(float x, float y, float z, long time, int accuracy)
{
  compass = x;
  // Angle between magnetic north and device y-axis, around z-axis.
  // Range: 0 to 359 degrees
  // 0=North, 90=East, 180=South, 270=West
}
```

There are a few new statements to look at.

---

1. Create a `KetaiLocation` type variable to be updated when our device detects a location update.
2. Create an Android `Location` object to store latitude and longitude data from the target device. The Location object also contains a number of useful methods for calculating bearing and distance.
3. Provide a (unique) phrase or identifier to store the location info.
4. Point to the identifier of the other device.
5. Set the PHP script URL responsible for writing location files.
6. Use Location object to retrieve location updates as opposed to individual variables.
7. Assemble the string that calls the PHP script with attached device name and location data.
8. Trigger the PHP script to write a string containing latitude, longitude, and altitude.
9. Read the other device's location file via the PHP script.
10. Check if we get a valid location containing latitude, longitude, and altitude, as well as parsing numbers contained in the string.
11. Write our location to the server via the PHP script.

    For this device locater app, we maintain a `location` variable that stores our location. We also keep the `otherDevice` `Location`, which this time is responsible for keeping track of a moving target. If we explore the code snippets that we've added to the destination compass app, the `serverURL` variable stands out. It's the path to the web server as a shared place for both devices; the server hosts the PHP script that writes and reads the device locations, which is discussed in the next section. We also introduced two string variables that identify each device. Those are necessary and need to be known to both devices—a shared "phrase" or ID that allows us to look up the other device's location. For instance, our location is identified

via `myName`, the other device refers to the location via `otherDevice`, and vice versa. This is how the exchange is enabled.
Every time we receive a location update from `onLocationEvent()`, `updateMyLocation()` is called to send the device name, latitude, longitude, and altitude to the server. When we tap the screen, we check if there is location info for the remote device called `deviceTracked`. We connect to the same PHP script that takes care of writing the file, this time with a `get` request instead of an `update` request. When the server returns a message, we check if we have a complete data package containing all three parameters: latitude, longitude, and altitude. If that's the case, we parse the info and assign it to the `otherDevice` location object.
This is how the Processing sketch triggers location updates to flows from and to the server to exchange location info between two known devices. If you feel comfortable writing your location to the book's project server defined in `serverURL`, you can give it a shot now and run the sketch on two Android devices (otherwise, please jump to Writing to a Text File on a Web Server). For each, you will have to swap the identifier stored in `myName` and `deviceTracked` for obvious reasons. Now let's test the app.

Run the App

Tap the screen on each device to trigger a location update on the server and observe. You should get a distance between both devices somewhere between `0` and `15` meters. Because our GPS satellites move constantly and the location provider estimates the device location on a constant basis, location, distance, and compass direction will change even when both devices are static. The closer the devices get to each other, the more erratic the compass changes. To test the compass needle, keep your devices are at least 30 feet apart from each other. You can then take the test to the next level by moving with both devices at increasing distances, which is significantly easier with another set of hands.
You can certainly host the PHP script that is responsible for writing the location data to the web server on your own server. Instructions on how the script (and how PHP) works are located in Writing to a Text File on a Web Server.

Wrapping Up

In this chapter, you've created a series of apps where you've learned how to work with location data provided by Android's Location Manager. You've learned that Android devices use the GPS and network methods to determine their geographic location. Given a choice, they will choose the most accurate method available. You can access this information using either the Ketai Library's `KetaiLocation` or the Android's `Location`class.

You are now able to determine the distance between two geolocations and calculate the bearing toward a fixed location. You've also learned how to write a way-finding app that points to another mobile device on the move. You are able to tackle a wide range of apps that build on geolocation. To complete our investigation into Android sensors, we'll look at another very sophisticated device and common sensor next—the Android camera.

5. Using Android Cameras

Now that we've learned to work with several of the most important Android sensors, let's take a look at another device that's found on every Android phone and tablet—the digital camera. Your device includes at least one and sometimes two cameras: the back-facing camera, which you commonly use to take high-resolution pictures and capture video, and the front-facing camera, designed for video calls and chat at a lower resolution. The digital camera can also be used as a sophisticated light-sensitive sensor to build a variety of interactive applications that go beyond pictures and video clips. We'll explore each of these uses and more in this chapter.

We'll start with the back-facing camera and learn how to display what it "sees" as an image on the Android screen. Next we'll learn how to switch between the front- and back-facing cameras found on most devices and add a feature that allows us to save their images to the Android's external storage which is a default public location on the device that can be read by other apps. Depending on the device settings, this can be located on an SD card, in internal storage, or on media mounted over the network. To make it easier to use these features, we'll add a few UI buttons to initiate each task.

Once we have stored an image from a camera, we may want to make further use of it. Additional APIs allow us to stack stored images to create a composite image that consists of a foreground and a background. We'll put this functionality to work by building a photo booth app, where we will create a fake backdrop and superimpose a snapshot on it.

But there's more. The Processing language also provides us with APIs that we can use to analyze the content of the images that we capture at the pixel level. We'll use that capability to build a game that can detect the color of a moving object—red or blue—and display the pattern of its motion on the device screen. To make the activity into a game, two players will compete to fill the screen by waving colored objects above it. The first to fill more than 50 percent of the screen wins. In building the game, we'll get to know the Processing `PImage` class, which allows us to manipulate images and work directly with pixel values.

Finally, we'll end the chapter with a brief look at Android's built-in face recognizer. This lesser-known camera feature is made possible by computer vision algorithms and the increased processing power that's finding its way into Android devices. Android provides a face-finder API that uses pixel-level image analysis to make inferences about what's going in the device's field of view. We'll demonstrate its use with a brief example.

Before we get started on our first project, let's first take a look at some of the camera features and classes we'll be using throughout the chapter to build our camera apps.

Introducing the Android Camera and APIs

Android phones and tablets are typically equipped with two cameras. Camera hardware varies across phones and tablets, but typically the back-facing camera is used to capture images and HD video at a resolution of 5 mega-pixels. The lower-resolution, front-facing camera is designed for video calls. The Google Nexus 6 phone, for example, features a 13-megapixel back-facing camera (4160 x 3120 pixels) with a built-in LED flash and a 2-megapixel front-facing camera.

Mobile cameras don't rely on hardware alone. The Android SDK provides a variety of features through its `Camera` class that make the camera more than just a camera. We can use code to work with camera metering, focus, exposure, white balance, zoom, image capture, and even face detection. Geolocation data can also be added to image metadata so that images can be organized by the location where they were taken. The Google Camera app that ships with Android devices allows users to manipulate those features in its UI. But we're going to learn how apps can use them as well.
To implement the camera features in this chapter, we'll work mainly with a single Ketai library class and a highly versatile Processing type:

- `KetaiCamera` This Ketai library class provides simplified access to the cameras on a device by making Android's `Camera` class available to Processing. When we work with `KetaiCamera`, we define the width, height, and frame rate for the camera preview we'd like to work with. It provides the necessary methods to define basic camera settings (such as resolution) and camera controls. It also provides access to the camera flash and Android's built-in face recognizer.
- `PImage` This is a Processing datatype for storing images (`gif`, `jpg`, `png`, `tif`, and `tga`). It provides a number of methods that help us load, save, and filter images, including access to the image `pixels[]` array that contains information on pixel color values. The methods we are using in this chapter are described further in Working with the PImage Class.

Now let's take a closer look at the `KetaiCamera` methods we'll be using.

Working with the KetaiCamera Class

Besides providing the typical `start` and `stop` methods that we use to control the sensors on a device, we'll use the following more specialized `KetaiCamera` methods for the projects in this chapter:

- `onCameraPreviewEvent()` Returns a preview image from the camera when a new frame is available—the image can then be read into the `KetaiCamera` object using the `read()` method.
- `addToMediaLibrary()` Makes a picture publicly available in the default preferred media storage on the device—the method requires a picture filename or path to the picture. After using the method, pictures are also available as an album in the Gallery app.
- `manualSettings()` and `autoSettings()` Toggles between manual and automatic camera settings—`manualSettings()` locks the current camera exposure, white balance, and focus. `autoSettings()` lets the device adjust exposure, white balance, and focus automatically.
- `enableFlash()` and `disableFlash()` Switches the built-in rear-facing camera flash on and off—this can only be used if the rear camera is on.
- `savePhoto()` Saves a picture in the current camera preview size to the preferred media storage.
- `setPhotoSize()` Sets the picture's size to be saved in a different, for example, higher, resolution.
- `setSaveDirectory()` Defines where to save the pictures to—by default, pictures are saved to the public media storage on the device. The path can also be set to another destination, including private folders. Requires testing whether the directory path is valid.
- `KetaiSimpleFace()` A Ketai wrapper for the `Face` class in Android's `FaceDetector` package, which returns the midpoint location and distance between the eyes recognized by the device cameras.
- `KetaiSimpleFace[]` A `PVector` list containing the position data of detected faces within a camera image—the center point between the left and right eyes and the distance between the eyes are stored in this array.

  With this brief summary of `KetaiCamera`methods for this chapter, let's get started with our first camera app.

Display a Back-Facing Camera Full-Screen Preview

For this initial camera app shown below, we'll display the view seen by the back-facing Android camera.

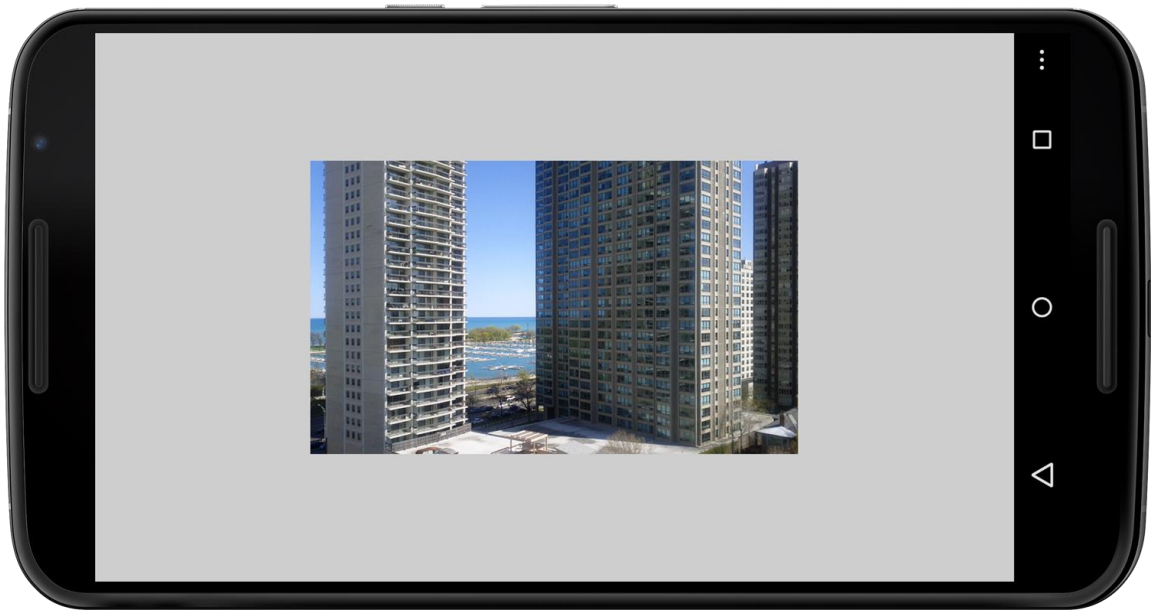


Figure 5.1 — Camera preview app.

The illustration shows a camera preview image at a resolution of 1280 x 768 pixels, displayed on the touch screen of a Google Nexus 6, whose resolution is 2560 x 1440 pixels.

We'll use the `KetaiCamera` class to connect to and start the camera. The `KetaiCamera` class streamlines this process significantly for us. For example, creating a simple camera preview app using `KetaiCamera` takes about ten lines of code, compared with about three hundred documented on the Android developer site.`KetaiCamera` helps us set up and control the camera, and it also decodes the YUV color format provided by the Android camera into the RGB format used in Processing.

`KetaiCamera` works similarly to other Ketai classes that we've explored in Chapter 3, Using Motion and Position Sensors. First we create a `KetaiCamera` object and `start()` the camera. Then we update the screen as soon as we receive a new image from the camera via `onCameraPreviewEvent()`. And finally, we use Processing's own `image` method to display the camera preview.

The code for a basic camera sketch looks like this:

code/Camera/CameraGettingStarted/CameraGettingStarted.pde

```
import ketai.camera.*;
KetaiCamera cam;

void setup() {
  orientation(LANDSCAPE);
  cam = new KetaiCamera(this, 1280, 768, 30);                 // 1
  imageMode(CENTER);                                          // 2
}

void draw() {
  if (cam.isStarted())
    image(cam, width/2, height/2);                            // 3
}
```

```
void onCameraPreviewEvent() {                                   // 4
  cam.read();                                                   // 5
}

void mousePressed() {
  if (cam.isStarted())
  {
    cam.stop();                                                 // 6
  }
  else
    cam.start();
}
```

Let's take a closer look at the steps you take and the methods you use to set up a camera sketch.

---

1. Create an instance of the `KetaiCamera` class to generate a new camera object with a preview width and height of 1280 x 768 pixels and an update rate of 30 frames per second.
2. Call `imageMode()` to tell Android to center its camera images on its screen. All images are now drawn from their center point instead of from the default upper left corner.
3. Display the camera preview using the `image()` method. It requires an image source as well as the `x`and `y` coordinates of the image to display. Optionally, the image can be rescaled using an additional parameter for the image `width` and `height`, which is what we are doing here.
4. Use the `onCameraPreviewEvent()` callback method for notification that a new preview image is available. This is the best time to read the new image.
5. Read the camera preview using the `read()` camera method.
6. Toggle the camera preview on and off when you tap the screen.

Let's try the sketch on the Android phone or tablet.

Run the App

Before we run the sketch, we need to give the app permission to use the camera. Here's how: On the Processing menu bar, select Android ↦ Sketch Permissions. In the Android Permissions Selector that appears, select Camera. As we've done already in Chapter 4, Using Geolocation and Compass earlier in Setting Sketch Permissions, the Android must allow the app to use the camera through a certificate, or it must prompt the user to approve the request to use the camera. If the app has permission to use the camera, the device will remember and not prompt the user anymore. For this app, we only need to check the permission for `CAMERA`. Now run the sketch on the device. The rear-facing camera preview starts up as illustrated in Figure 5.1, in a resolution of 1280px width and 768px height, known as WXGA. Android cameras are set to auto mode, so they adjust focus and exposure automatically. Depending on your phone's native screen resolution, the preview image might cover the screen only partially. You can certainly scale and stretch the preview image, which also changes the image aspect

ratio and distorts the image. For instance, if you set the width and height parameters in the `image()` method to `screenWidth` and `screenHeight` as in the following code, the camera preview will always stretch full screen independent of the screen's size and resolution.

```
image(cam, width/2, height/2, width, height);
```

Go ahead and try the fullscreen mode on your device. For a preview image in a camera app, it doesn't seem like a good idea to stretch the image, though. When we write apps that scale seamlessly across devices, we typically lock and maintain aspect ratios for images and UIs.

As we can see in CameraGettingStarted.pde, the steps we take to get the camera started are like the steps we took working with other sensors (Chapter 3, Using Motion and Position Sensors). First we instantiate a `KetaiCamera` object using a defined `width`, `height`, and `frameRate`. Then we start the camera. And finally, we read new images from the camera using `onCameraPreviewEvent()` and display them. The frame rate in this sketch is set to 30 frames per second, which is the typical playback speed for digital video, giving the appearance of seamless movement. Depending on your device and image conversion performance, the image preview might not be able to keep up with the designated thirty previews per second. In that case, the sketch will try to approach the set frame rate as best it can.
With less than ten lines of code added to the typical processing sketch methods, we've completed our first camera app. The `onPause()` and `exit()` methods are responsible for releasing the camera properly when we pause or exit the app. The methods make sure that other apps can use the cameras and that we don't keep them locked down for our app alone. You can only have one active connection to the cameras at a time.

Toggle Between the Front- and Back-Facing Cameras

Most mobile Android devices come with both the front-facing and back-facing cameras. We need a UI button that toggles between the front and back camera. Let's also activate the flash that's built into most back-facing cameras and add an additional pair of button controls to start and stop the camera. The final app then looks like this:

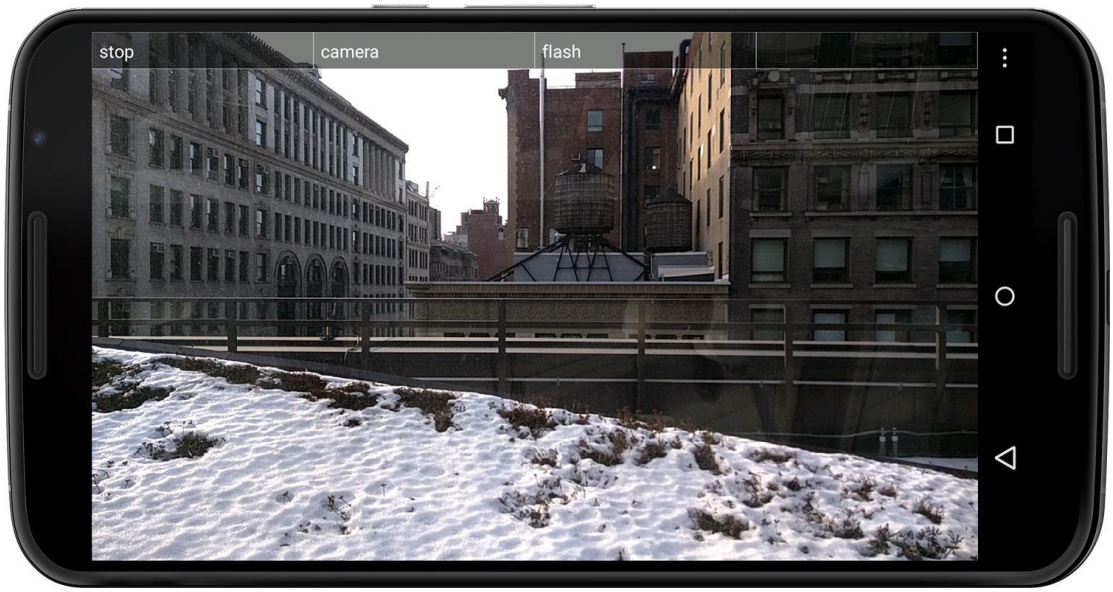


Figure 5.2 — Camera preview app with UI.

The UI added to the Preview app allows users to start and stop the cameras, toggle between the front- and back-facing cameras, and activate the built-in flash.

Android lists all built-in device cameras and allows us to pick the one we'd like to work with. For instance, the Nexus 6 uses the camera index ID `0` for the back-facing camera and `1` for the front-facing camera. Future Android devices might add more cameras to the device, potentially for 3D applications, so having an enumerated list enables Android OS to incorporate them.

Let's build on the previous sketch CameraGettingStarted.pde, adding some camera controls that will remain pretty much the same throughout the chapter. Because this sketch is longer than the previous one, we'll separate it into two tabs: a main tab containing the essential `setup()` and `draw()` methods, which we'll name `CameraFrontBack` (identical to the sketch folder), and a second tab, which we'll call `CameraControls`and will contain the methods we need to `read()` the camera preview, the methods to `start()` and `stop()`the camera, and the UI buttons we'll use to control the camera via the touch screen.

Separating the code this way helps us reduce complexity within the main tab and focus on relevant code for the projects we are working on. We'll store each tab in its own Processing source file, or `pde` file, inside the sketch folder. You can always check what's inside your sketch folder using the menu Sketch ↦ Show Sketch Folder, or the shortcut `K`.

Let's first take a look at the main tab:

code/Camera/CameraFrontBack/CameraFrontBack.pde

```
import ketai.camera.*;

KetaiCamera cam;

void setup() {
```

```
  orientation(LANDSCAPE);
  cam = new KetaiCamera(this, 1280, 768, 30);
  println(cam.list());                                          // 1
  // 0: back camera; 1: front camera
  cam.setCameraID(0);                                           // 2
  imageMode(CENTER);
  stroke(255);
  textSize(48);                                                 // 3
}

void draw() {
  image(cam, width/2, height/2, width, height);
  drawUI();                                                     // 4
}
```

In the main `CameraFrontBack` tab, we've added new features.

1. Print all available device cameras to the Processing console using the `list()` method included in `KetaiCamera`.
2. Set the back-facing camera ID to `0` via `setCameraID()`.
3. Increase the `textSize()` for the UI buttons to `24` pixels.
4. Call the custom `drawUI()` method, taking care of drawing UI buttons.
5. The `draw()` method contains only a call to the `image()` method, used for displaying the camera preview, and a call to the custom `drawUI()` method we defined for our UI elements.
   Now let's explore the second sketch tab called `CameraControls`, where we'll keep all the code that controls the camera.

code/Camera/CameraFrontBack/CameraControls.pde

```
void drawUI() {                                                 // 1
  fill(0, 128);
  rect(0, 0, width/4, 100);
  rect(width/4, 0, width/4, 100);
  rect(2*(width/4), 0, width/4, 100);
  rect(3*(width/4), 0, width/4, 100);
  fill(255);
  if (cam.isStarted())                                          // 2
    text("stop", 20, 70);
  else
    text("start", 20, 70);
  text("camera", (width/4)+20, 70);
  text("flash", 2*(width/4)+20, 70);
}
void mousePressed() {                                           // 3
  if (mouseY <= 100) {                                          // 4
    if (mouseX > 0 && mouseX < width/4) {                       // 5
      if (cam.isStarted())
      {
        cam.stop();
      }
      else
      {
        if (!cam.start())
          println("Failed to start camera.");
      }
    }
    else if (mouseX > width/4 && mouseX < 2*(width/4))          // 6
    {
      int cameraID = 0;
      if (cam.getCameraID() == 0)
        cameraID = 1;
```

```
      else
        cameraID = 0;
      cam.stop();
      cam.setCameraID(cameraID);
      cam.start();
    }
    else if (mouseX >2*(width/4) && mouseX < 3*(width/4))   // 7
    {
      if (cam.isFlashEnabled())                             // 8
        cam.disableFlash();
      else
        cam.enableFlash();
    }
  }
}
void onCameraPreviewEvent() {
  cam.read();
}
void exit() {
  cam.stop();
}
```

In this `CameraControls` tab, we use the following UI elements and camera methods to complete these steps.

1. Display the UI on the screen using a custom `void` function called `drawUI()`. Void functions execute but don't return a value. The UI in this example consists of buttons that use half-transparent rectangles for their backgrounds and text labels for their names.
2. Check if the camera is running using the boolean method `isStarted()`. If the method returns `TRUE`, we display "stop"; otherwise show "start."
3. Capture touch screen input for camera controls using `mousePressed()`.
4. Check if the user is interacting with the UI at the top of the screen using the `mouseY` constant. If we receive user input within the top `40` pixels of the screen, we continue checking the horizontal position via `mouseX`.
5. Check if the user presses the leftmost button to start and stop the camera. Each button occupies one-fourth of the screen width, so we check if the horizontal tap position is within the range `(0..width)/4`. We take the same approach for the other buttons.
6. Check if the user taps the second button, which is responsible for toggling between the rear and the front cameras. We acquire the current camera ID using `getCameraID()` and toggle using `setCameraID()`.
7. Check if the user taps the third button, which is responsible for toggling the camera flash on and off.
8. Check the camera's flash status using the `isFlashEnabled()` method and toggle the flash by calling `enableFlash()` or `disableFlash()`, depending on the returned boolean value.
   Let's go ahead and test the app now.

Run the App

Load or enter the two tabs of the sketch, run it on your device, and take a look at the Processing console. You should see a list of all the built-in cameras on your device with their respective IDs, as shown below.

```
[camera id [0] facing:backfacing, camera id [1] facing:frontfacing]
```

When the app launches, the rear-facing camera becomes the default camera, but it remains paused until we start it up. Press the Start button now. The camera preview should appear on the screen at the defined resolution of 1280 x 768 pixels. Toggle the camera from the front to the back using the Camera button. Start and stop the flash. The camera flash belongs to the back-facing camera and works only when the rear camera is active.

Now that we know how to preview and control the camera, it's time to put it to work—let's snap some pictures. In our next project, we'll learn how to store images on the device.

Snap and Save Pictures

To snap pictures and save them to the external storage of our device, we'll first need to add a `savePhoto()`method to the previous sketch CameraFrontBack.pde. The method takes care of capturing the image and writing it to the device's external storage in a folder that bears the app's name. When the photo is written to this public directory on the SD card, we receive a callback from `onSavePhotoEvent()` notifying us that the writing process is complete. This callback method is also useful if we'd like to notify the device's media library to make the photos available to other applications, which we accomplish with a call to the `addToMediaLibrary()` method. Once we've added photos to the media library, we can browse them in the Gallery—Android's preinstalled app for organizing pictures and video clips shown in Figure 5.3. The larger the captured photo size, the longer it takes to transfer the image buffer and store it on the disk.

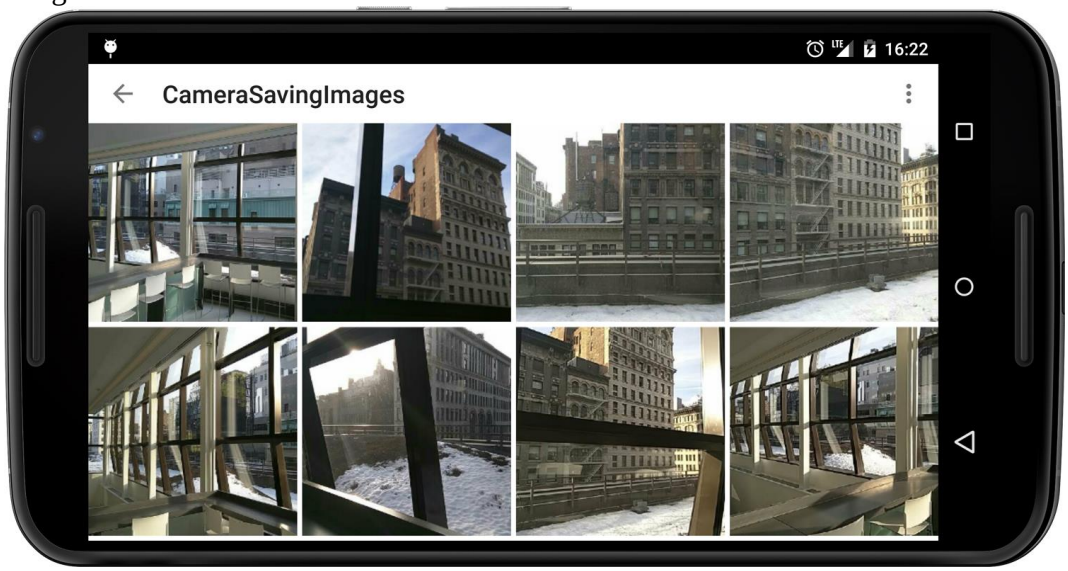


Figure 5.3 — Android gallery.

When we take pictures with our camera app and add them to the public external storage, they are available in an album within Android's Gallery.

To refine the camera app UI, let's also add a Save button that allows us to save the image by tapping the touch screen. Some status info on the current camera settings also seems useful.

For the Save feature, we need to modify the `draw()` method in the main `CameraSavingImages` tab and make some adjustments to `CameraControls`. The following code snippets show only the modifications to the previous code in CameraFrontBack.pde and CameraControls.pde. You can also download the complete `pde`source files from the book's website, and if you're reading the ebook, just click the green rectangle before the code listings.
Let's take a look.

code/Camera/CameraSavingImages/CameraSavingImages.pde

```
import ketai.camera.*;

KetaiCamera cam;

void setup() {
  orientation(LANDSCAPE);
  cam = new KetaiCamera(this, 1280, 768, 30);
  println(cam.list());
  // 0: back camera; 1: front camera
  cam.setCameraID(0);
  imageMode(CENTER);
  stroke(255);
  textSize(48);
}

void draw() {
  background(128);
  if (!cam.isStarted())                                          // 1
  {
    pushStyle();                                                 // 2
    textAlign(CENTER, CENTER);
    String info = "CameraInfo:\n";
    info += "current camera: "+ cam.getCameraID()+"\n";          // 3
    info += "image dimensions: "+ cam.width +                    // 4
      "x"+cam.height+"\n";                                       // 5
    info += "photo dimensions: "+ cam.getPhotoWidth() +          // 6
      "x"+cam.getPhotoHeight()+"\n";                             // 7
    info += "flash state: "+ cam.isFlashEnabled()+"\n";          // 8
    text(info, width/2, height/2);
    popStyle();                                                  // 9
  }
  else
  {
    image(cam, width/2, height/2, width, height);
  }
  drawUI();
}
```

Now let's take a look at the new code we've added to `draw()` and what it does.

1. Check the status through the boolean method `isStarted()`. Returns `TRUE` if the camera is on and `FALSE` if it's off.
2. Save the current style settings using `pushStyle()` to preserve the `stroke()`, `textSize()`, and default `textAlign(LEFT, TOP)` for the UI elements, and add a new `textAlign(CENTER, CENTER)` style using `pushStyle()`. Requires `popStyle()` to restore previous style settings.
3. Get the index number of the currently chosen camera using `getCameraID()`.
4. Get the preview image width (in pixels) of the current camera using `getImageWidth()`.
5. Get the preview image height (in pixels) of the current camera using `getImageHeight()`.
6. Get the image width (pixels) of a photo taken by the current camera using `getPhotoWidth()`. The photo size is separate from the camera preview size.
7. Get the image height (pixels) of a photo taken by the current camera using `getPhotoHeight()`.
8. Inquire about the status of the flash using the boolean method `isFlashEnabled()`. (The flash belongs to the rear camera and can only be used if the back-facing camera is on.)
9. Restore the previous style settings using `popStyle()`.

   Changes to `draw()` mostly concern the text output that gives us some feedback on the camera settings. Next let's examine the modifications to the camera controls.

code/Camera/CameraSavingImages/CameraControls.pde

```
void drawUI() {
  fill(0, 128);
  rect(0, 0, width/4, 100);
  rect(width/4, 0, width/4, 100);
  rect(2*(width/4), 0, width/4, 100);
  rect(3*(width/4), 0, width/4-1, 100);

  fill(255);
  if (cam.isStarted())
    text("stop", 20, 70);
  else
    text("start", 20, 70);

  text("camera", (width/4)+20, 70);
  text("flash", 2*(width/4)+20, 70);
  text("save", 3*(width/4)+20, 70);                          // 1
}

void mousePressed() {
  if (mouseY <= 40) {
    if (mouseX > 0 && mouseX < width/4)
    {
      if (cam.isStarted())
      {
        cam.stop();
      }
      else
      {
        if (!cam.start())
          println("Failed to start camera.");
      }
    }
    else if (mouseX > width/4 && mouseX < 2*(width/4))
    {
      int cameraID = 0;
      if (cam.getCameraID() == 0)
```

```
        cameraID = 1;
      else
        cameraID = 0;
      cam.stop();
      cam.setCameraID(cameraID);
      cam.start();
    }
    else if (mouseX >2*(width/4) && mouseX < 3*(width/4))
    {
      if (cam.isFlashEnabled())
        cam.disableFlash();
      else
        cam.enableFlash();
    }
    else if (mouseX > 3*(width/4) && mouseX < width)              // 2
    {
      if (cam.isStarted()) cam.savePhoto();                       // 3
    }
  }
}

void onSavePhotoEvent(String filename)                          // 4
{
  cam.addToMediaLibrary(filename);                              // 5
}

void onCameraPreviewEvent()
{
  cam.read();
}
```

Take a look at how the code adds the following features.

---

1. Add a UI button `text()` label for saving images.
2. Add a condition to check if the user taps the added Save button.
3. Save the photo to the device's external storage using `savePhoto()`. The method can also take a parameter for a custom file name.
4. Receive notification from the `onSavePhotoEvent()` callback method when a picture is saved to external storage.
5. Add the picture to the device's public preferred media directory on the external storage using `addToMediaLibrary()`.
   With the addition of the `savePhoto()` and `addToMediaLibrary()`, the app is now ready to store pictures in external storage, which makes the images public and available for other apps, such as the Android Gallery app. Once again, let's make sure we've set the permissions we need to write to external storage (see also Setting Sketch Permissions). In the Android Permissions Selector, check the boxes next to Write_External_Storage in addition to Camera. This time, we need both to run this sketch successfully.

Run the App

Run the modified sketch on an Android device and tap Save to save the picture.

Now let's take a look at the Gallery and see if the photos we took show up there properly. Press the Home button on the device and launch the Gallery app, which comes preinstalled with the Android OS. The images you took will appear in the `CameraSavingImages` album that bears the same name as the app. Making the images available publicly allows us to share them with other apps. The use of `addToMediaLibrary()` is certainly optional. If we use only the `savePhoto()` method, the images are still saved to the publicly available external storage, but they won't be visible to other apps using the external storage.
We've now learned how to save images to the external storage of an Android device. In the next project, we'll create a photo booth app that allows us to blend and superimpose the images we capture. To accomplish this task, we'll blend multiple image sources into one. Let's take a look.

Superimpose and Combine Images

In this project, we'll superimpose a snapshot on a background image, as we might do with a friend in a photo booth at a carnival. Using the Android's front-facing camera, we'll create an app that works like a photo booth, with the small twist that we use scenery loaded from a still resource image as the image's background instead of the physical backdrop we might find in an actual photo booth. We want to be able to use the app anywhere, independent of our current surroundings or lighting level. This is why we need to separate the foreground image from its background. Using color pixel calculations, we can erase a background image and superimpose a snapshot onto a scene loaded from an image in a resource file, as shown in Figure 5.4.

The photo booth app combines images from two sources: the preview image acquired by the front-facing camera and an image loaded from a file that will be included with the app.

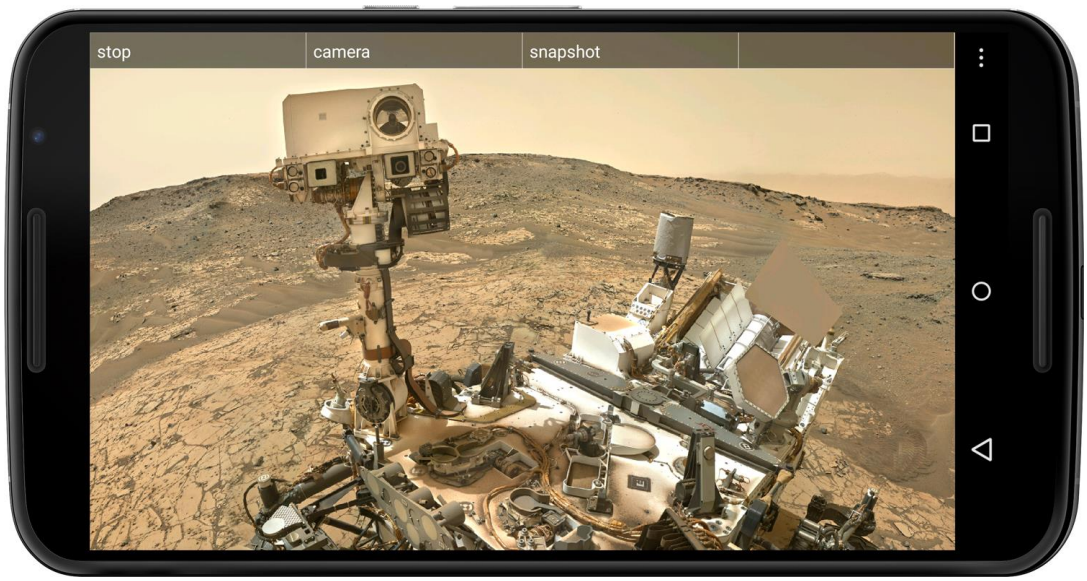


Figure 5.4 — Photo booth app.

The image shows the photo booth app using the rover background image we've chosen.

First, take a snapshot with the device sitting still on the table. When you take the snapshot, be sure to stay out of the camera's field of view. We'll use this snapshot as a reference image, which we'll subtract from the camera's preview image. If we've held the camera steady, this subtraction will leave behind an empty, black image by eliminating all the pixels that have not changed. For example, if the live camera and the snapshot images are identical, any `pixel[n]` that we choose at random will have the identical value in both images. Let's say, for the sake of argument, that the color of a particular pixel is `color(135, 23, 245)`. If we subtract the color value of the pixel in one image from the corresponding pixel in the other—`color(135, 23, 245)`minus `color(135, 23, 245)`—the result is `color(0, 0, 0)`.
When this subtraction of color values is performed for all of the pixels in an `i`mage pair, the resulting effect is that when someone enters the frame of the camera again, the image of the subject will appear to be &lquot;floating&rquot; in front of the background image of our choosing: the landscape of Mars or a view of Lake Michigan from the grounds of the World's Fair. The result: a portable photo booth that we can use to transport ourselves into any scene we'd like.

Let's start by looking in more detail at some of the `PImage` features we'll use.

Working with the PImage Class

`PImage` is a datatype for storing images that supports `.tif`, `.tga`, `.gif`, `.png`, and `.jpg` image formats. Listed below are some of the `PImage` methods that we'll be using for this project:

- `loadImage()` Loads the pixel data for the image into its `pixels[]` array
- `loadPixels()` Loads the pixel data for the image into its `pixels[]` array—this function must always be called before reading from or writing to `pixels[]`.
- `updatePixels()` Updates the image with the data in the `pixels[]` array—the method is used in conjunction with `loadPixels`.
- `pixels[]` Array containing the color of every pixel in the image
- `get()` Reads the color of any pixel or grabs a rectangle of pixels
- `set()` Writes a color to any pixel or writes an image into another
- `copy()` Copies the entire image
- `resize()` Resizes an image to a new width and height—to resize proportionally, use `0` as the value for the width or height parameter.
- `save()` Saves the image to a TIFF, TARGA, GIF, PNG, or JPEG file

Now let's write some code.

For this project, we'll create a new sketch, again with two tabs, and copy the code into each tab individually. We'll call the main tab `CameraPhotoBooth` and the second tab `CameraControls`, which we'll reuse from the previous sketch CameraSavingImages/CameraControls.pde.
Let's first take a look at the main tab.

code/Camera/CameraPhotoBooth/CameraPhotoBooth.pde

```
import ketai.camera.*;

KetaiCamera cam;
PImage bg, snapshot, mux;

void setup() {
  orientation(LANDSCAPE);
  cam = new KetaiCamera(this, 1280, 768, 30);
  cam.setCameraID(1);                                          // 1
  imageMode(CENTER);
  stroke(255);
  textSize(48);
  snapshot = createImage(1280, 768, RGB);
  bg = loadImage("rover.jpg");                                 // 2
  bg.resize(1280, 768);
  bg.loadPixels();
  mux = new PImage(1280, 768);
}

void draw() {
  background(0);
  if (cam.isStarted()) {
    cam.loadPixels();                                          // 3
    snapshot.loadPixels();                                     // 4
    mux.loadPixels();                                          // 5
    for (int i= 0; i < cam.pixels.length; i++)
    {
      color currColor = cam.pixels[i];                         // 6
      float currR = abs(red(cam.pixels[i]) - red(snapshot.pixels[i]) );  // 7
      float currG = abs(green(cam.pixels[i]) - green(snapshot.pixels[i]));
      float currB = abs(blue(cam.pixels[i]) - blue(snapshot.pixels[i]));
      float total = currR+currG+currB;                         // 8
      if (total < 128)
        mux.pixels[i] = bg.pixels[i];                          // 9
      else
        mux.pixels[i] = cam.pixels[i];                         // 10
    }
    mux.updatePixels();                                        // 11
    image(mux, width/2, height/2, width, height);              // 12
  }
  drawUI();
}
```

Here are the steps we need to take in the main tab.

---

1. Set the camera ID to the front-facing camera using `setCameraID()`, which has the index number `1`.
2. Load the `rover.jpg` resource image from the data folder using `loadImage()`, which will serve as a replacement for the background.
3. Load the camera pixel array using `loadPixels()`.
4. Load the snapshot picture pixel array using `loadPixels()`.
5. Load the `mux` pixel array using `loadPixels()` to store the composite photo booth image.

6. Parse the `pixels` array and get the current screen pixel color at array position `i`.
7. Calculate the `red()` difference between the individual camera and snapshot pixel values. Convert the result into an absolute, always positive number using `abs()`. Make the same calculation for the green and blue pixel values.
8. Add the differences for the red, green, and blue values to calculate the `total` difference in color, which will be used as a threshold for the composite image. Values can range from `0` (no change) to `255`(maximum change) for `total`. Use `128` (50 percent change) as the threshold to choose between the live camera or the background image.
9. Set the composite `mux` image to the background image `bg` pixel for small changes in the camera image.
10. Set `mux` to the camera pixel if the camera preview changed a lot.
11. Update the composite `mux` pixels used to display the calculated result using `updatePixels()`.
12. Display the composite image `mux` on the screen using the `image()` method, which now contains the combined pixel data from the live camera and the background image.
    In this app, we've changed the `draw()` method from our previous camera app CameraSavingImages.pde. We focus on combining images in `draw`, where we use a background image—a snapshot taken from the camera preview—and the current camera preview taken in the same location. We calculate the difference between this current camera preview and the snapshot to determine which pixels changed. Then we display the stored background image in all the pixels that did not change and display the live camera pixels where the preview changed. When a person enters the scene after taking the snapshot, those changed pixels function as a mask for the background image. This is why it's also important that the camera doesn't move during the process.

Adding Media Assets to a Sketch

The `setup` method contains a reference to a "canned" image called `rover.jpg`. The image is stored in the sketch's `data` folder. We load the image into the `PImage` variable `bg` at the beginning, when the app starts up. Here we use `PImage` only to store the image. We'll discuss this datatype further in the next project, Working with the PImage Class, where we rely on some useful `PImage` methods to work with pixel values.

The sole purpose of the sketch's `data` folder is to host all necessary media assets and resource files for our sketch, such as images, movie clips, sounds, or data files. If a resource file is outside the sketch's `data`, we must provide an absolute path within the file system to the file. If the file is online, we need to provide a URL. There are three ways to add a media asset to a sketch:

- Drag and drop the file you want to add onto the sketch window from your file system (for example, from the desktop) onto the Processing sketch window you want to add the file to.

Processing will create the `data` folder for you in that sketch and place the resource file inside it.

- Choose Sketch ↦ Add File... from the Processing menu, and browse to the asset.
- Browse to the sketch folder (choose Sketch ↦ Show Sketch Folder).

Now let's check what's changed in `CameraControls`.

code/Camera/CameraPhotoBooth/CameraControls.pde

```
void drawUI()
{
  fill(0, 128);
  rect(0, 0, width/4, 100);
  rect(width/4, 0, width/4, 100);
  rect(2*(width/4), 0, width/4, 100);
  rect(3*(width/4), 0, width/4-1, 100);

  fill(255);
  if (cam.isStarted())
    text("stop", 20, 70);
  else
    text("start", 20, 70);

  text("camera", (width/4)+20, 70);
  text("snapshot", 2*(width/4)+20, 70);
}

void mousePressed()
{
  if (mouseY <= 100) {

    //start/stop the camera
    if (mouseX > 0 && mouseX < width/4)
    {
      if (cam.isStarted())
      {
        cam.stop();
      }
      else
      {
        if (!cam.start())
          println("Failed to start camera.");
        bg.resize(cam.width, cam.height);
      }
    }
    //switch cameras
    else if (mouseX > width/4 && mouseX < 2*(width/4))
    {
      int cameraID = 0;
      if (cam.getCameraID() == 0)
        cameraID = 1;
      else
        cameraID = 0;
      cam.stop();
      cam.setCameraID(cameraID);
      cam.start();
    }
    //take snapshot
    else if (mouseX > 2*(width/4) && mouseX < 3*(width/4))
    {
      cam.manualSettings();                                    // 1
      snapshot.copy(cam, 0, 0, cam.width, cam.height,
        0, 0, snapshot.width, snapshot.height);                // 2
      mux.resize(cam.width, cam.height);
```

```
    }
  }
}

void onCameraPreviewEvent()
{
  cam.read();
}

void exit()
{
  cam.stop();
}
```

In the Camera Controls tab, we reuse the UI button for the flash from the previous code CameraSavingImages/CameraControls.pde and label it "Snapshot." Because the flash belongs to the back-facing camera and it's much easier for us to use the front camera here, we don't need the flash any more for this project. The Snapshot button is now responsible for copying the pixels from `cam` to `snapshot`, as shown below.

1. Set the camera to manual mode using the `manualSettings()` method, locking the current camera exposure, white balance, and focus.
2. Use the `copy()` method to take the snapshot. Use the snapshot image to subtract from the camera preview, erasing the background, and extracting the image foreground of the camera.

Run the App

Now lean the Android upright against something solid so it can remain static, and run the app. When it starts up, press the Snapshot button, capturing a `snapshot` image from the camera preview. Make sure you are out of the camera field of view; if not, you can always retake the snapshot. Now, reenter the scene and see yourself superimposed on the landscape of Mars. Adjust the threshold value of `128` to be higher or lower to best match your lighting situation. You can use any resource image stored in `CameraPhotoBooth/data`, so go ahead and swap it with another image resource of your choice.
This project showed us how to take control of two different image sources and combine them in creative ways. The project can easily be expanded to create a chroma-key TV studio, in which we could superimpose live video of a TV show host onto a studio green screen. But we'll leave that as an exercise for the reader.

Now that we've gained some experience in manipulating images, let's use our ability to process information about pixels to create a two-person drawing game.

Detect and Trace the Motion of Colored Objects

In the drawing game that we'll build in this section, two players will compete to see who can fill the screen of an Android device with the color of a red or blue object first. Without touching the device screen, each player scribbles in the air above it with a blue or red object in an attempt to fill as much space as possible with the object's color. When more than 50 percent of the screen is filled, the player that filled in the most pixels wins. We'll use the front-facing camera as the interactive interface for this game. It's job is to detect the presence of the colors blue or red within its field of vision and capture them each time it records a frame. The game code will increase the score of each player who succeeds in leaving a mark on the screen.

The camera remains static during the game. As Figure 5.5 illustrates, only the primary colors red and blue leave traces and count toward the score. If the red player succeeds in covering more pixel real estate than the blue, red wins. If blue dominates the screen, blue wins. If you are using an Android tablet you can step a little bit further away from the device than is the case for a phone, where the players are more likely to get in each other's way, making the game more competitive and intimate.

Figure 5.5 – Magic marker drawing game.

Red- and blue-colored objects leave color marks, gradually covering the camera preview. The color that dominates wins the game.

The magic marker drawing game uses color tracking as its main feature. As we implement this game, we put Processing's image class, called `PImage`, to use. The main purpose of this datatype is to store images, but it also contains a number of very useful methods that help us manipulate digital images. In the context of this game, we'll use `PImage` methods again to retrieve pixel color values and to set pixel values based on some conditions we implement in our sketch.

Manipulating Pixel Color Values

To create this magic marker drawing game, we need to extract individual pixel colors and decide whether a pixel matches the particular colors (blue and red) we are looking for. A color value is only considered blue if it is within a range of "blueish" colors we consider blue enough to pass the test, and the same is true for red. Once we detect a dominant color between the two, we need to call a winner.

For an RGB color to be considered blue, the `blue()` value of the pixel color needs to be relatively high, while at the same time the `red()` and `green()` values must be relatively low. Only then does the color appear blue. We are using the Processing color methods `red()`, `green()`, and `blue()` to extract R, G, and Bvalues from each camera pixel. Then we determine whether we have a blue pixel, for instance, using a condition that checks if `blue()` is high (let's say `200`) and at the same time `red()` and `green()` are low (let's say `30`) on a scale of `0..255`. To make these relative thresholds adjustable, let's introduce variables called `high` and `low` for this purpose.
Let's take a look. The sketch again contains `CameraControls`, which we don't discuss here because we already know the method to `start` and `stop` the camera.

code/Camera/CameraMagicMarker/CameraMagicMarker.pde

```
import ketai.camera.*;

KetaiCamera cam;
PImage container;
int low = 30;
int high = 100;
int camWidth = 1280;
int camHeight = 768;
int redScore, blueScore = 0;
int win = 0;

void setup() {
  orientation(LANDSCAPE);
  imageMode(CENTER);
  cam = new KetaiCamera(this, camWidth, camHeight, 30);
  // 0: back camera; 1: front camera
  cam.setCameraID(1);
  container = createImage(camWidth, camHeight, RGB);          // 1
}

void draw() {
  if (win == 0) background(0);
  if (cam.isStarted()) {
    cam.loadPixels();
    float propWidth = height/camHeight*camWidth;              // 2
    if (win == 0) image(cam, width/2, height/2, propWidth, height);  // 3
    for (int y = 0; y < cam.height; y++) {
      for (int x = 0; x < cam.width; x++) {
        color pixelColor = cam.get(x, y);                     // 4
        if (red(pixelColor) > high &&
          green(pixelColor) < low && blue(pixelColor) < low) {// 5
          if (brightness(container.get(x, y)) == 0) {         // 6
            container.set(x, y, pixelColor);
```

```
            redScore++;
          }
        }
        if (blue(pixelColor) > high &&
          red(pixelColor) < low && green(pixelColor) < low) { // 7
          if (brightness(container.get(x, y)) == 0) {
            container.set(x, y, pixelColor);
            blueScore++;
          }
        }
      }
    }
    image(container, width/2, height/2, propWidth, height);   // 8
    fill(255, 0, 0);
    rect(0, height, 20, map(redScore, 0, camWidth*camHeight, 0, -height));
    fill(0, 0, 255);
    rect(width-20, height, 20, map(blueScore, 0, camWidth*camHeight, 0, -height));
    if (redScore+blueScore >= camWidth*camHeight * 0.50) {    // 9
      win++;
      if (redScore > blueScore) {                             // 10
        fill(255, 0, 0, win);
      }
      else {
        fill(0, 0, 255, win);
      }
      rect(0, 0, width, height);
    }
    if (win >= 50) {
      container.loadPixels();                                 // 11
      for (int i = 0; i < container.pixels.length; i++) {
        container.pixels[i] = color(0, 0, 0, 0);              // 12
        redScore = blueScore = win = 0;
      }
    }
  }
}

void mousePressed()
{
   if(cam.isStarted())
      cam.stop();
   else
     cam.start();
}
```

There are a couple of new methods for us to look at.

---

1. Create an empty `PImage` called `container` using the `createImage()` method to hold red and blue color pixels that have been detected in the camera preview image. The empty RGB image container matches the size of the camera preview image.
2. Calculate the fullscreen camera preview image width `propWidth` proportional to the camera preview aspect ratio. We get the ratio by dividing the screen `height` by the camera preview height `camHeight`and multiplying that with the `camWidth`.
3. Draw the camera preview image in fullscreen size using `image()` if no player has won the game yet (`win` equals `0`). Match the image height with the screen height and scale the image width proportionately.
4. Get the color value at the image pixel location `x` and `y` using the `PImage` method `get()`. Store the value in the `color` variable `pixelColor`.

5. Check for reddish pixel values within the camera preview using the `red()`, `green()`, and `blue()PImage` methods to extract individual color values from the `color` datatype. Consider only pixel values with a red content greater than the `high` threshold and `low` green and blue values. Use the globals `high` and `low` for the upper and lower limits of this condition.
6. Check if the pixel is already taken by a color using `brightness`. If the `container` is empty and not set yet, it has a brightness value of `0`.
7. Check for blueish pixel value in the camera image. It requires a color with a high blue content, while the red and green values are `low`.
8. Draw the `container` using the `image()` method. This `PImage` contains all the red and blue pixels we grabbed from the camera's preview image.
9. Check for the winner when at least 50 percent of the image is covered, comparing the combined `redScore` and `blueScore` values against `0.50` of all camera preview pixels.
10. Fade to the winning color by changing the `fill` opacity of a colored rectangle covering the screen. To achieve a continuous fade, use the `win` variable for the alpha parameter so that the following rectangle is drawn with decreasing opacity (`0`: fully opaque, `255` fully transparent).
11. Load the pixel data from the container `PImage` into the pixels[] array. The function must be called before writing to (or reading from) `pixels[]`.
12. Empty all `pixels[]` in the container image pixel array. Set all pixels to the `color(0, 0, 0, 0)`, which is a fully transparent black color. The Processing rule is that you must call `loadPixels()` before you read from or write to `pixels[]`, even if some renderers seem not to require this call.
    Now let's test the game using some blueish and reddish objects and test how well the camera picks up their colors. Any kind of object will do as long as its color is a vibrant red or blue—the more intense its hue and brightness the better.

Run the App

Grab a friend and a few blueish and reddish objects, and get ready to scribble madly mid-air and fight for pixel real estate on the Android device. Run the sketch on the device. When the game starts up, the camera preview will appear centered on the screen, stretched to fullscreen size. Reddish and blueish colors are instantly picked up and drawn on top of the preview image. This immediate feedback lets us play with different objects and quickly get an idea about which objects have the greatest color impact as we try to cover the screen.

Try it. The status bar on either side of the screen grows as colors are picked up, showing us how much pixel real estate each player owns. Individual scores are compared with the total number of available pixels. If 50 percent of all pixels are grabbed by the red player, for instance, the red progress bar covers half of the screen height. Once more than 50 percent of

all available pixels are taken, the sketch calls a winner and fades to the winning color. It resets the game to start over.

This game has taken us deep into the world of pixels using all the prior color knowledge we've acquired in Build a Motion Based Color Mixer and Palette. The `PImage` datatype is a convenient way to work with images, which are in principle "just" lists of colors containing red, green, blue, and alpha (transparency) values that we can use for our own purposes, such as our magic marker drawing game.
If your device is up to the challenge, feel free to double the camera resolution via `camWidth` and `camHeight`for better image quality, but consequently you'll have to lower the frame rate. We've discussed that pixel-level calculations are computationally expensive and hence require a speedy Android device to run smoothly. In Chapter 11, Introducing 3D Graphics with OpenGL, we will learn a few tricks that help us put the graphics processing unit (GPU) to use, keeping the central processing unit (CPU) free for other tasks.
Since you've successfully interpreted images on a pixel level, let's take it a step further now and explore how pixel-level image algorithms are used for advanced image processing and computer vision purposes, specifically for Android's face detection API.

Detect Faces

One of the most amazing hidden features of the camera software is the ability to detect faces. We've seen that access to the pixel values enables us to interpret images and make inferences about their content. Such computer vision algorithms have many applications in robotics, automation, and security. The Android face detection API is designed to trigger an event when one or more faces are detected.

Facial recognition is an Android feature that uses complex computer vision algorithms to detect typical facial features, which are recognized by their shape and the position of a person's eyes within the camera's field of view. The Android device uses so-called Haar cascades for face recognition.

The Camera app, for instance, uses this feature to set the focus of the camera on the eyes of a person when taking a photo. Face Unlock added to Ice Cream Sandwich uses face recognition to unlock your device. When you first activate Face Unlock (Security Settings ↦ Face Unlock), you provide an image of your face and a PIN. The device remembers the shape and other characteristics of your face and uses those metrics to compare it to a live camera image when you unlock the screen. Depending on the amount of available light, this feature works uncannily well.

`Face` detection is part of Android's `Camera` class, exposed by `KetaiCamera` so we can use it within the camera apps we develop using the Ketai library. The information we receive about facial features includes the location of the `leftEye()`, the `rightEye()`, the `mouth()`, an individual `id` for each detected face, and a `score` of the confidence level for the detection of the face, with a range of `1..100`. The ability to detect facial features might come as a surprise when we use and expose it. However, modern digital cameras use similar algorithms to auto-set the focus and auto-correct red-eye effects.

The face finder sketch we are writing is based on Android's `Face` detection API. For the sketch, we use the camera's preview image and send it to the face detector. It returns an array of faces to us that contains the metrics of individual facial features that we can use to draw a rectangle where a face is detected. We test the app on the device, point the Android camera to a web page that displays the results of a Google Image search on the term "faces." This way we can see how well the detection works when it has to respond to a variety of faces of different scales and quality. Let's take a look.

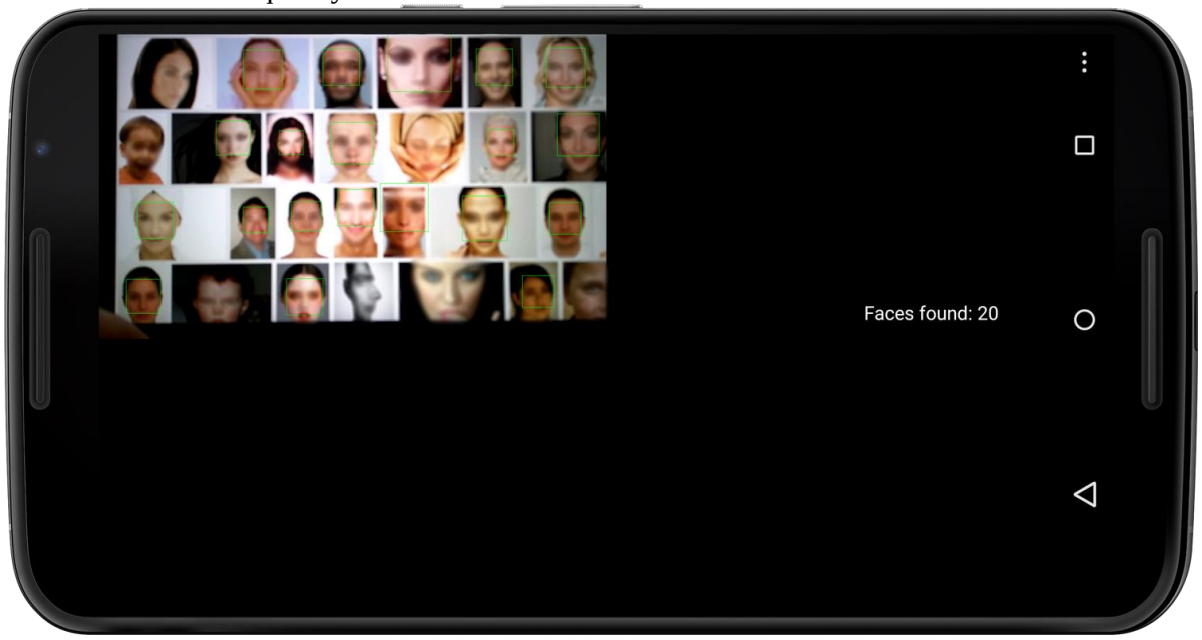


Figure 5.7 — Face Finder app.

The image illustrates Android's Face Detector API, which here displays fourteen faces found by an image search engine. The API does not recognize faces shown in side profiles or cropped portraits.

code/Camera/CameraFaceFinder/CameraFaceFinder.pde

```
import ketai.camera.*;
import ketai.cv.facedetector.*;

KetaiCamera cam;
KetaiSimpleFace[] faces;                                    // 1
boolean findFaces = false;

void setup() {
  orientation(LANDSCAPE);
```

```
  cam = new KetaiCamera(this, 1280, 768, 30);
  rectMode(CENTER);                                          // 2
  stroke(0, 255, 0);
  noFill();                                                  // 3
  textSize(48);
}

void draw() {
  background(0);
  if (cam != null) {
    image(cam, 0, 0);
    if (findFaces)                                           // 4
    {
      faces = KetaiFaceDetector.findFaces(cam, 20);          // 5
      for (int i=0; i < faces.length; i++)                   // 6
      {
        rect(faces[i].location.x, faces[i].location.y,       // 7
        faces[i].distance*2, faces[i].distance*2);           // 8
      }
      text("Faces found: " + faces.length, width*.8, height/2);  // 9
    }
  }
}

void mousePressed () {
  if(!cam.isStarted())
    cam.start();
  if (findFaces)
    findFaces = false;
  else
    findFaces = true;
}
void onCameraPreviewEvent() {
  cam.read();
}
```

Let's take a look at the face finder methods used by the sketch.

---

1. Create an array to store the list of faces found. It contains the x and y location of each face and the distance between the eyes.
2. Center the rectangles that mark found faces around their center points.
3. Turn off the fill color for the green rectangle markers so we can see though them.
4. Check the boolean that lets us turn the face finder on and off.
5. Call the `findFaces()` method in the `FaceFinder` class with the two parameters for the image input (`cam`) and the maximum number of faces to be found (`20`).
6. Parse the results returned from the `faces` array. The array length varies by the number of faces that are found, so we check how often to iterate through the array by testing `faces.length` with the for loop.
7. Draw a rectangle based on the returned face `location` `PVector`. Use `.x()` and `.y()` to access the horizontal and vertical positions of the face location.
8. Use twice the `distance` between the eyes to draw an approximate rectangle marking the detected face.

9. Display the total number of faces found; a maximum of `20` can be found based on our `findFaces()`settings.
   Let's give it a try.

Run the App

Run the app and set the device aside. Now go to your PC and do a Google image search on the word "face." Pick up the Android and aim the camera at the PC display. Google displays a grid of images showing a wide range of faces at different exposures and angles. Now tap the screen to start face detection. You immediately experience a performance hit caused by the face detection algorithm. We've instructed `findFaces` to extract up to twenty faces from the camera's preview.
Once the camera has a clear and steady shot of the faces on the PC display, you can see on the Android screen where green rectangles are overlaid onto the detected areas, as illustrated in . Overall it does a pretty good job. When portraits are cropped or only show faces in profile, the algorithm doesn't consider it a face. To confirm this rule, do a Google search on the term "face profile" and see what happens. Finally, see what "cartoon face" will produce. Using these different search strings helps us to understand what the algorithm requires to interpret a certain pixel area as a face.

Let's move on to the detection of moving human subjects. Use `setCameraID(1)` just before `cam.start();` in `setup` to switch to the front-facing camera. Run the app again, and test the face detection algorithm on your own face. You should observe that the face detection feature begins to work as soon as you face the camera. You need to keep enough distance so your face doesn't appear cropped in the camera preview. If you turn your head to present a profile to the camera, your face won't be detected anymore because the camera can't "see" both of your eyes.
We haven't looked deeply into what the Face API does exactly to extract faces from a list of pixel values, and in this case, we don't need to. Android provides us with a list of faces, the midpoint between the eyes, and their distance. Edge detection and decision trees are the concern of the API. Clearly, this feature, which ships with all current Android devices, can be used for different purposes.

Unlike social media sites that employ face detection algorithms to match a person or identity with an image, the Android is not concerned about that. If we start up face detection in our app, the Android OS will trigger a face event when it "sees" a face, whether or not it knows the identity of that person. For some of your apps, it can be relevant to know whether a person is looking at the screen or not.

Now that you are aware of this feature, it's up to you to use it or look at your device from now on with the level of scrutiny this feature calls for. The face detection project is a good example of why we need to ask for permission to use the `CAMERA` (Setting Sketch Permissions). If we do, the user is prompted to grant or deny this request. Once granted, the app will retain the permission certificate to use the camera, and we won't be prompted any more. In , we'll use the face detection feature to rotate a 3D object based on how we look at the screen. It is one example of where the face detection API serves as an interactive user interface within a 3D environment.

Wrapping Up

In this chapter, you've learned how to work with the cameras on Android devices. You've also learned how to display the images the cameras acquire, how to save them, and how to work with them when they're stored in memory. You're now able to manipulate images down to the pixel level, use the camera to detect colored objects in motion, and display their paths on the screen. You've also learned how to activate and use Android's face recognition feature.

This completes our investigation of a diverse range of sensors found on Android devices. You know how to interact with their touch screens and how to determine their orientation and bearing as well as their motion and geographic location. You can also take pictures with the Android and start to make sense of what the device is seeing. You're ready now to move on to the second part of this book, where we'll learn how to network the Android with PCs and other mobile devices and work with large amounts of data.

6. Networking Devices with Wi-Fi

Social media, location-based services, and multiplayer games are just a few examples of mobile applications that rely on frequent updates delivered over the Internet via cellular, cable, or satellite networks. In this chapter we'll focus on wireless local area networks. The ability to exchange data between Android devices and PCs within a local area network allows us to write mobile apps that can connect multiple users without a mobile carrier data plan.

By the end of this chapter, you'll be able to send data between computers and Androids within a Wi-Fi network. You will be able to write real-time interactive apps running on multiple devices that take advantage of the high bandwidth offered by a Wi-Fi network. This can be useful, for example, to inventory stock in a retail store, monitor a patient, view data from a home security or automation system, or participate in a multiplayer game.

There are four ways to connect devices without sending data over the Internet: Wi-Fi, Wi-Fi Direct, Bluetooth, and near field communication (`NFC`), listed in decreasing order in terms of connection distance, power consumption, and speed. We will cover peer-to-peer networking, to which Wi-Fi Direct and Bluetooth belong, in the next chapter, called Chapter 7, Peer-to-Peer Networking Using Bluetooth and Wi-Fi Direct, and cover NFC in the following one, called Chapter 8, Using Near Field Communication (NFC).
We'll start this chapter by creating an app that exchanges data between an Android device and a desktop PC using the Open Sound Control (`OSC`) networking format. Then we'll build a collaborative drawing app, where we use the Wi-Fi network to share a drawing canvas between two Android devices. As the final project for this chapter, we'll create a game for two players using the accelerometer on each device to control the tilt of a shared playing surface with two balls in play. Local Wi-Fi networks offer us the bandwidth and response time we need for multiplayer games while freeing us from worry about data plans and transmission quotas.

Working with Wi-Fi on Android Devices

Wi-Fi is so ubiquitous in cities that you can find a Wi-Fi network virtually anywhere you go. It's true that most Wi-Fi networks that you encounter while you're on the move are protected and won't allow devices to connect. But most domestic and workplace destinations that we visit regularly—including many coffee shops, libraries, and airports—do offer the opportunity

to subscribe or connect for free. Once connected, the phones, tablets, and laptop devices that you carry will remember a particular Wi-Fi network, making it easy to connect again when you return.

Most Wi-Fi networks are set up to connect to the Internet. But you can also use a wireless network access point to set up a local Wi-Fi network for the sole purpose of connecting multiple Wi-Fi-enabled devices between one another. Many Android devices will even let you create a Wi-Fi hotspot using the device itself (Settings ↦ Wireless & networks ↦ More... ↦ Tethering & portable hotspots).

When a Wi-Fi-enabled device connects to a Wi-Fi access point, it is assigned an IP address. An IP address is a unique identifier (see also Working with Networking Classes) that is used to identify the device within a network. It functions as a numeric label that other devices can use to access it. Likewise, to connect our Android to other devices within the network, we need to know their IP addresses as well.

When two devices wish to communicate, they must also share a common port number in addition to knowing each other's IP addresses. A port is an agreed-upon number that establishes communication in conjunction with the IP address. Certain ports are reserved for services such as FTP (Port 21) or HTTP (Port 80) and should not be used. Port numbers greater than 1023 usually work just fine.

If we're on the move and a known Wi-Fi network is not available to us, the Android device requests an IP address from the cell phone carrier's 3G or 4G network. Although it is possible to network two devices over the carrier network, we cannot sustain a peer-to-peer network as the device connects from cell tower to cell tower. Connecting two (or more) devices inside a Wi-Fi network is significantly different from connecting them outside the network, which is described in further detail in Peer-to-Peer Networking. In this chapter, we'll stay focused on Wi-Fi communications.

Peer-to-Peer Networking (P2P)

When it comes to peer-to-peer networking, the difference between an IP address provided by a local area network and an IP address from a 4G or 3G cellular network is that we can send data to a local area network address directly. If we go outside the local area network, our IP address undergoes network address translation, or `NAT`. There is no way to connect directly to a device without knowing exactly how to translate that new number in reverse. "Getting through" the NAT router is often referred to as "traversing" the NAT, or NAT-busting. Applications such as Skype (voice-over-IP), Hamachi, or LogMeIn (remote desktop) are very good at traversing. IP addresses are managed centrally, and the techniques that companies use

to traverse the NAT are proprietary. It is clear, though, that NAT-busting is a messy and complicated process, one that exploits NAT router and firewall loopholes. Why the trouble? Because of the great benefit that we can have very efficient peer-to-peer connections that provide high update rates at no cost while "off-the-grid".
If we are on the move, we will lose the IP address provided by a cellular network and will get a new one as we hop from tower to tower. A cellular provider might also have services in place that try to maintain a particular address using an address translation that takes place on the carrier's side. Being handed over from cell tower to cell tower or from Wi-Fi network to Wi-Fi network, however, is the nature of being on the move.

In short, for true P2P, we need a public IP address. In the prevalent IPv4 addressing system, there are virtually too few IP addresses available for the number of devices on the planet. An enormous global transition to the new IPv6 addressing system is currently underway, rendering NAT practically obsolete. You can keep an eye on the IPv6 deployment as it unfolds. For now we need to sit tight; all our toasters, clothes, and children have a dedicated IP address.

Let's first take a look at the networking classes we'll be using in this chapter.

Working with Networking Classes

For the networking projects in this chapter, we'll be working with the following classes provided by Processing, Ketai, and oscP5 libraries.

- netP5 Processing's core library for reading and writing data across networks—it allows the creation of clients and servers over a specified port. A server connects to a list of clients for reading and writing data. A client is able to read and write data to that server.
- KetaiNet A library containing Android-device-specific methods to complement `netP5`, including a method to look up the Android device IP address, as it frequently changes when we are on the move
- oscP5 An Open Sound Control library for Processing developed by Andreas Schlegel—`OSC` is a networking protocol for communication among computers, sound synthesizers, and other multimedia devices in many application areas.
  When two or more devices communicate in a distributed application through a network, they typically interact through a server-client computing model. The server provides resources, and a client requests them. There can be more than one client connected to the central server, and multiple clients can communicate with each other through that server. The server is responsible for establishing a connection between itself and one or more clients.

To transfer data between devices, we need a protocol to package, send, and receive different data types in an efficient manner. We'll use the Open Sound Control protocol for that. Let's take a look at what `OSC` has to offer.

Using the Open Sound Control Networking Format

Open Sound Control (`OSC`) is a very flexible and popular data protocol. We can wrap pretty much any data format into an OSC message and send it over the network. We can also package several messages at a time in an OSC bundle before sending it on its way, giving us a lot of control over how much we send at once. So instead of taking our data, chopping it into individual pieces (bytes), and precariously assembling it on the other side of a network, we can send different data types within one OSC message. OSC allows us to change how many pieces of data we put into a message as well, and we don't have to change the way we package it. This is particularly useful when we build networking apps, where we start with a simple message and add data to the payload as we get more complex.

For the projects in this chapter, we'll use the `netP5` library to read and write data over the network. It's already available because it's part of Processing's core library.

The `KetaiNet` class is also available because it's part of the Ketai library, which we've already installed.

Let's go download the `oscP5` library now. You can find it at http://www.sojamo.de/libraries/oscP5/ and also among other Processing libraries in the Data/Protocols section of the Processing libraries website.

Let's follow the same process we already used to install the Ketai library, which is the same process we use for installing any Processing library:

---

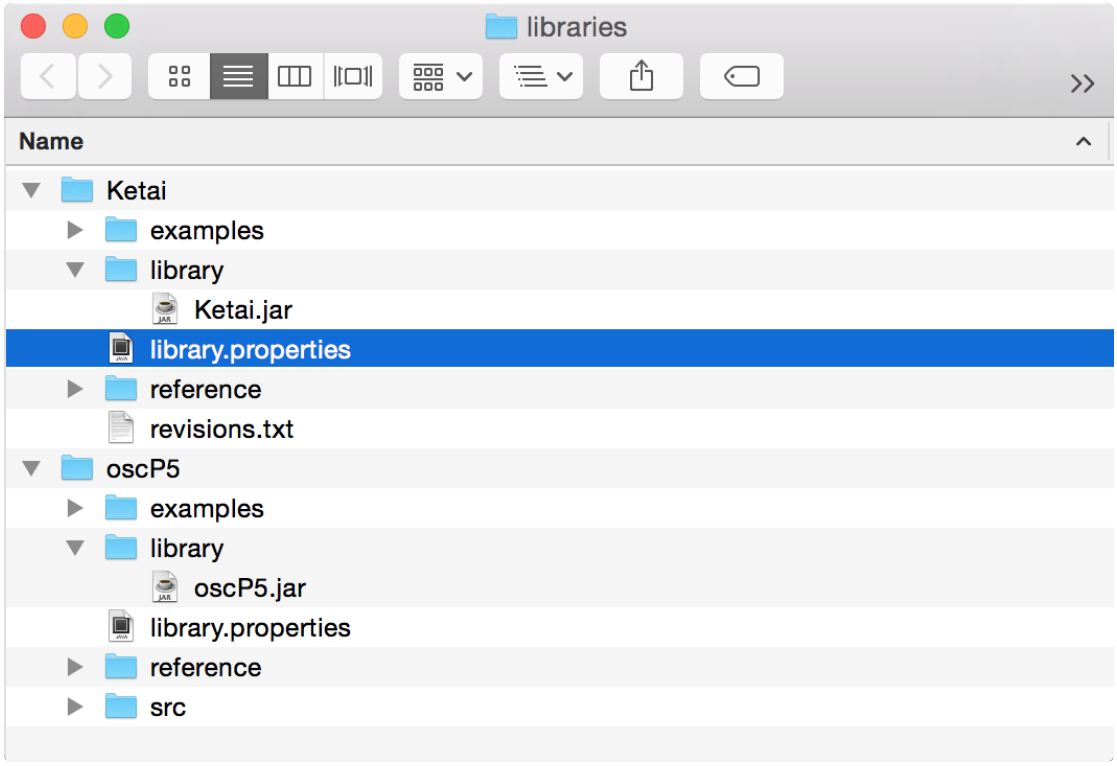
1. Choose "Add Library...," which you can find under Sketch ↦ "Import Library..."
2. In the search field of the window that opens, enter `oscP5`.
3. Select the "oscP5" library that appears in the list, and press the Install button on the right.
4. The download starts immediately, and a bar shows the download's progress. When the library is installed, the button on the right changes to Remove.

Alternatively, you can use your web browser to download the library and install it manually within Processing's library folder:

---

1. Download and unzip the oscP5 library folder.

2. Check what's inside the folder. Locate the `library` subfolder that contains the actual Processing library `.jar`. The OSC library and most other Processing libraries also include `reference`, `examples`, and `src` subfolders.
3. Move the complete `oscP5` directory, including `library`, `reference`, `examples`, and `src` into the Processing sketchbook, located at `Documents/Processing/libraries`.
   The sketchbook `libraries` folder now looks something like the figure below. Now let's put OSC to work and network the Android with the PC.



Network an Android with a Desktop PC

For our first project, we're going to network a desktop PC and an Android device and then use the Wi-Fi network to exchange data between them. Wireless local area networks provide us with a high-bandwidth connection, which allows us to write applications that let us interact with peers within the network in real time. We can send fairly large data payloads without noticeable delays, making it a good choice for a diverse range of multiuser applications.

We'll need to import the networking classes described in Working with Networking Classes, so we can exchange data over a common port, as illustrated in Figure 6.1. We'll use oscP5, which builds on and requires Processing's core netP5 library for the exchange of data. We also use the `KetaiNet` class to look up the Android's IP address, and we'll use the familiar `KetaiSensor` class to receive accelerometer data.

Figure 6.1 - Connecting an Android to a PC.

The screen output shows remote mouse data from the PC on the Android screen (left) and accelerometer data from the Android in the PC display window (right). The local Android IP address and the remote PC address are shown at the bottom of the Android screen.

Before we get started, let's make sure that both devices are connected to the same Wi-Fi network. Go ahead and check the Android (Settings ↦ Wireless & networks) to activate Wi-Fi. If your device is not already connected to a Wi-Fi network, choose the same network that the PC is connected to. Once connected, write down the IP address that has been assigned to the Android device. On your desktop machine, check your network settings so your desktop computer is connected to the same network as the Android. You can use an Ethernet connection for the PC as well, as long as you are connected to the same network as the Android.

We'll build this application in two steps: first we'll write a sketch for the Android device and then for the PC.

Program the Android Device

Before you can connect your Android to the PC, you first need to figure out the IP address of the desktop computer on the local network. Make sure your PC is on the same network as the Android via Wi-Fi.

- On a Mac, you'll find your IP address under System Preferences ↦ Network.
- On a PC, try Control Panel ↦ Network and Internet.
- On Linux you can go to Administration ↦ Network Tools.

My IP address looks like this:

```
192.168.1.2
```

Your address most likely looks different. Write yours down, as it is not very intuitive, and this needs to be correct to connect successfully.

We'll first code the Android sketch using the oscP5 `NetAddress` class to specify the destination of the OSC message. We'll create a `NetAddress` object called `remoteLocation` and consisting of the IP address of the remote device—in this case our PC—and the port number (12000) that both devices will use to communicate. For this first sketch, the OSC message we send will consist of three floating point numbers, the values of the x-, y-, and z-axes of the accelerometer that we'll `add()` to the message before it's sent. In turn, we'll receive three integer values from the desktop PC, consisting of the x and y positions of the mouse cursor, followed by a `0` or a `1`, depending on whether the mouse button is pressed (`1`) or not (`0`). Now let's take a look at the code for the sketch:

code/Networking/WiFiDataExchangeAndroid/WiFiDataExchangeAndroid.pde

```
import netP5.*;                                          // 1
import oscP5.*;
import ketai.net.*;
import ketai.sensors.*;

OscP5 oscP5;
KetaiSensor sensor;

NetAddress remoteLocation;
float myAccelerometerX, myAccelerometerY, myAccelerometerZ;
int x, y, p;
String myIPAddress;
String remoteAddress = "192.168.1.2";                    // 2 Customize!

void setup() {
  sensor = new KetaiSensor(this);
  orientation(PORTRAIT);
  textAlign(CENTER, CENTER);
  textSize(72);
  initNetworkConnection();
  sensor.start();
}
```

```
void draw() {
  background(78, 93, 75);

  text("Remote Mouse Info: \n" +                          // 3
  "mouseX: " + x + "\n" +
    "mouseY: " + y + "\n" +
    "mousePressed: " + p + "\n\n" +
    "Local Accelerometer Data: \n" +
    "x: " + nfp(myAccelerometerX, 1, 3) + "\n" +
    "y: " + nfp(myAccelerometerY, 1, 3) + "\n" +
    "z: " + nfp(myAccelerometerZ, 1, 3) + "\n\n" +
    "Local IP Address: \n" + myIPAddress + "\n\n" +
    "Remote IP Address: \n" + remoteAddress, width/2, height/2);

  OscMessage myMessage = new OscMessage("accelerometerData"); // 4
  myMessage.add(myAccelerometerX);                        // 5
  myMessage.add(myAccelerometerY);
  myMessage.add(myAccelerometerZ);
  oscP5.send(myMessage, remoteLocation);                  // 6
}

void oscEvent(OscMessage theOscMessage) {
  if (theOscMessage.checkTypetag("iii"))                  // 7
  {
    x =  theOscMessage.get(0).intValue();                 // 8
    y =  theOscMessage.get(1).intValue();
    p =  theOscMessage.get(2).intValue();
  }
}

void onAccelerometerEvent(float x, float y, float z)
{
  myAccelerometerX = x;
  myAccelerometerY = y;
  myAccelerometerZ = z;
}

void initNetworkConnection()
{
  oscP5 = new OscP5(this, 12000);                         // 9
  remoteLocation = new NetAddress(remoteAddress, 12000);  // 10
  myIPAddress = KetaiNet.getIP();                         // 11
}
```

Here are the steps outlining what the sketch does.

---

1. Import the Processing networking library `netP5` to read and write data over the network. Import the `oscP5` library to send data using the OSC protocol. Import the Ketai networking class to look up the device's current IP address and the `KetaiSensor` class to work with the accelerometer sensor.
2. Set the remote IP address variable (`remoteAddress`) of the desktop to exchange data with.
3. Print all info about remote mouse position, state, and local accelerometer data. Android accelerometer data `myAccelerometerX`, `myAccelerometerY`, and `myAccelerometerZ` are presented with one digit to the left and two digits to the right of the decimal point and a plus or minus number prefix using the `nfp()`method. At the bottom of the screen we display our local Android IP address followed by the remote desktop IP.

4. Create a new outgoing OSC message (`myMessage`) with an assigned label (`accelerometerData`) that contains our local accelerometer info. OSC labels can also be used on the receiving side to distinguish between multiple incoming messages.
5. Add the `x`, `y`, and `z` accelerometer axes to the outgoing OSC message.
6. Send the OSC message `myMessage` to `remoteLocation`.
7. Check the incoming OSC message for the `iii` value pattern, which specifies a packet of three integer values.
8. Once a complete OSC data package containing three integers is detected, we set `x`, `y`, and `p` to the incoming values.
9. Instantiate an OSC object from the `oscP5` library and start an OSC connection on port `12000`.
10. Set the destination IP and port number to the `remoteAddress` at port number `12000`; the port number must be identical to successfully exchange data.
11. Look up the Android IP address assigned by the Wi-Fi network using `getIP()`.
    The `oscP5` library relies on some methods from the core network library in Processing called `netP5`, which is why we `import` both at the beginning of the code. To work with the accelerometer, we use the `KetaiSensor` class again, which is why we `import` the `ketai.sensors` package. To look up the Android's assigned Wi-Fi IP address we use the `getIP()` method contained in the `ketai.net` package. Make sure to customize `remoteAddress` to match your desktop IP address.
    Now we are ready on the Android side to start talking.

Open Sound Control

Developed by Matt Wright and Adrian Freed at the Center for New Music and Audio Technologies in 1997, the OSC protocol has been used for a variety of applications, including sensor-based electronic music instruments, mapping data to sound, multiuser controls, and web interfaces, to name a few. OSC messages consist of numeric and symbolic arguments, 32-bit integers and floats, time tags, strings, and blobs. Messages can be bundled so they can act simultaneously when received. Pattern matching allows OSC to specify multiple targets for a single message as well. This allows us to broadcast values to a number of devices. Optional time tags (64 bit) allow highly accurate synchronization of timed events. Many data exchange applications don't require the optional time tags, as they utilize only the OSC data structure, triggering events upon delivery.

Although less convenient and more fundamental in nature, other widespread communication protocols include TCP (Transmission Control protocol), UDP (User Datagram Protocol), and asynchronous serial communication. They use different ports to exchange data, and they

“shake hands” slightly differently. Handshaking is the process of negotiating communication parameters on both sides before the actual communication begins.

Run the App

Before we run the sketch, let’s check the `INTERNET` permissions in the Android Permissions Selector that we’ll need in order to send data through the network. We’ve already worked with different types of permissions for geolocation and cameras, and we follow the same procedure (Setting Sketch Permissions) in the Permissions Selector, choosing `INTERNET` from the Android ↦ Sketch Permissions dialog.
We’ll take the following steps to network the Android and the PC. First, we’ll run the Android sketch we’ve just created on the Android device. Then we’ll check the Processing console to see if OSC is up and running. Since the Android device is connected to the PC via USB, it gives us some feedback with regard to the OSC status when the app starts up. Finally, when the app is running on the device and the OSC server is running, we move on to run the PC sketch and start the connection.

Now run the sketch on the Android device. It should start up fine, even if we don’t have a partner to talk to yet. When the app is starting up, the device reports to the console that OSC is initialized and that the OSC server is running on port `12000`.
Here’s the output you can expect to see in the PDE (Processing IDE) console.

```
PROCESS @ UdpClient.openSocket udp socket initialized.
PROCESS @ UdpServer.start() new Unicast DatagramSocket
created @ port 12000 INFO @ OscP5 is running. you (127.0.0.1)
are listening @ port 12000 PROCESS @ UdpServer.run()
UdpServer is running @ 12000
```

With the app launched on the Android, let’s shift our focus to the desktop to complete the OSC server-client network.

Program the PC

The sketch for the PC is nearly identical to the one for the Android sketch. It’s a bit more concise because there’s no accelerometer data to capture on the desktop, and we don’t have to look up the device IP address because we’ve already written it down. The desktop sketch receives accelerometer values from the Android and sends its `mouseX`, `mouseY`, and `mousePressed` values in return. Let’s take a look:

code/Networking/WiFiDataExchangePC/WiFiDataExchangePC.pde

```
import oscP5.*;
import netP5.*;

OscP5 oscP5;
```

```
NetAddress remoteLocation;
float accelerometerX, accelerometerY, accelerometerZ;

void setup() {
  size(480, 480);
  oscP5 = new OscP5(this, 12000);
  remoteLocation = new NetAddress("192.168.1.3", 12000);  // 1 Customize!
  textAlign(CENTER, CENTER);
  textSize(24);
}

void draw() {
  background(78, 93, 75);
  text("Remote Accelerometer Info: " + "\n" +
    "x: "+ nfp(accelerometerX, 1, 3) + "\n" +
    "y: "+ nfp(accelerometerY, 1, 3) + "\n" +
    "z: "+ nfp(accelerometerZ, 1, 3) + "\n\n" +
    "Local Info: \n" +
    "mousePressed: " + mousePressed, width/2, height/2);

  OscMessage myMessage = new OscMessage("mouseStatus");
  myMessage.add(mouseX);                                  // 2
  myMessage.add(mouseY);                                  // 3
  myMessage.add(int(mousePressed));                       // 4
  oscP5.send(myMessage, remoteLocation);                  // 5
}

void oscEvent(OscMessage theOscMessage) {
  if (theOscMessage.checkTypetag("fff"))                  // 6
  {
    accelerometerX =  theOscMessage.get(0).floatValue();  // 7
    accelerometerY =  theOscMessage.get(1).floatValue();
    accelerometerZ =  theOscMessage.get(2).floatValue();
  }
}
```

On the desktop, we make the following adjustments.

---

1. Point OSC to the remote Android IP address `remoteLocation`, displayed on the Android as "Local IP Address." Go ahead and customize this address using your Android's IP address now.
2. Add the horizontal mouse position `mouseX` to the OSC message.
3. Add the vertical mouse position `mouseY` to the OSC message.
4. Add the `mousePressed` boolean, cast as an integer number to send either `0` or `1` via OSC.
5. Send the OSC message `myMessage` on its way to the Android via port `12000`.
6. Check the OSC message for packages containing three incoming floating point values patterned "fff."

7. Assign incoming floating point values to `accelerometerX`, `accelerometerY`, and `accelerometerZ`, shown on the desktop screen.
   We are sending three global integers `x`, `y`, and `p` from the desktop to the Android (the horizontal and vertical mouse position and `mousePressed`) and will receive `accelerometerX`, `accelerometerY`, and `accelerometerZ` in return. For the data exchange, we are using port `12000` in both sketches. This port number (`12000`) could change,

but it must be identical on both sides to work properly and shouldn't conflict with the lower numbers for ports already in use.

Run the App

Let's run the sketch on the PC in Java mode. The display window starts up on the desktop, and we can now move the mouse in the window to send OSC messages containing mouse info to the Android. On the Android screen we see the horizontal and vertical position of the mouse update and the mouse button state change. Changing the orientation of the Android device gives us a range of accelerometer values, which we can observe on the desktop screen. Value updates seem to occur instantaneously. There is no perceivable lag time, and while we are certainly only sending a few values, it gives us an idea about the bandwidth Wi-Fi has to offer—a highly interactive setup.

If communication fails, make sure you've adjusted `remoteAddress` in the Android sketch to match the IP address of your desktop PC. It's close to impossible that your Wi-Fi router assigned the same IPs used in the example sketches here. And while you are at it, go ahead and also check that the port number matches on both sides. The IP address must be correct, and port numbers must match to exchange data successfully.
Let's note that when we use OSC networking, it won't complain if there is no other device to talk to. The connection sits and waits until another device enters the conversation on port `12000`. Likewise, OSC doesn't throw a networking error when a device leaves the conversation; it can also reconnect at any time. This is another great feature of the OSC communication protocol, whether we use it on the Android or the desktop—a robust connection process combined with a straightforward method to send messages containing different data types.
In terms of networking across devices, this is a major milestone we can continue to build on. It's a small step for us to change the values we've sent via OSC to take on different new tasks. So for the next project, we'll use Networking/WiFiDataExchangeAndroid/WiFiDataExchangeAndroid.pde to create a drawing canvas that the Android and the PC can share.

Share Real-Time Data

For our next project, we're going to create a program for the Android and the PC that allows users of the two devices to draw on a shared surface, or virtual whiteboard, as shown in Figure 6.2, below. We'll refine the previous sketches Networking/WiFiDataExchangeAndroid/WiFiDataExchangeAndroid.pde and Networ

king/WiFiDataExchangePC/WiFiDataExchangePC.pde that we've written to connect the Android and the desktop PC. The Wi-Fi network has the necessary bandwidth and update rates that we need to draw collaboratively. Whatever one of the users draws will appear instantaneously on the other's device, and vice versa.

Let's start by programming the Android; then we'll program the PC.

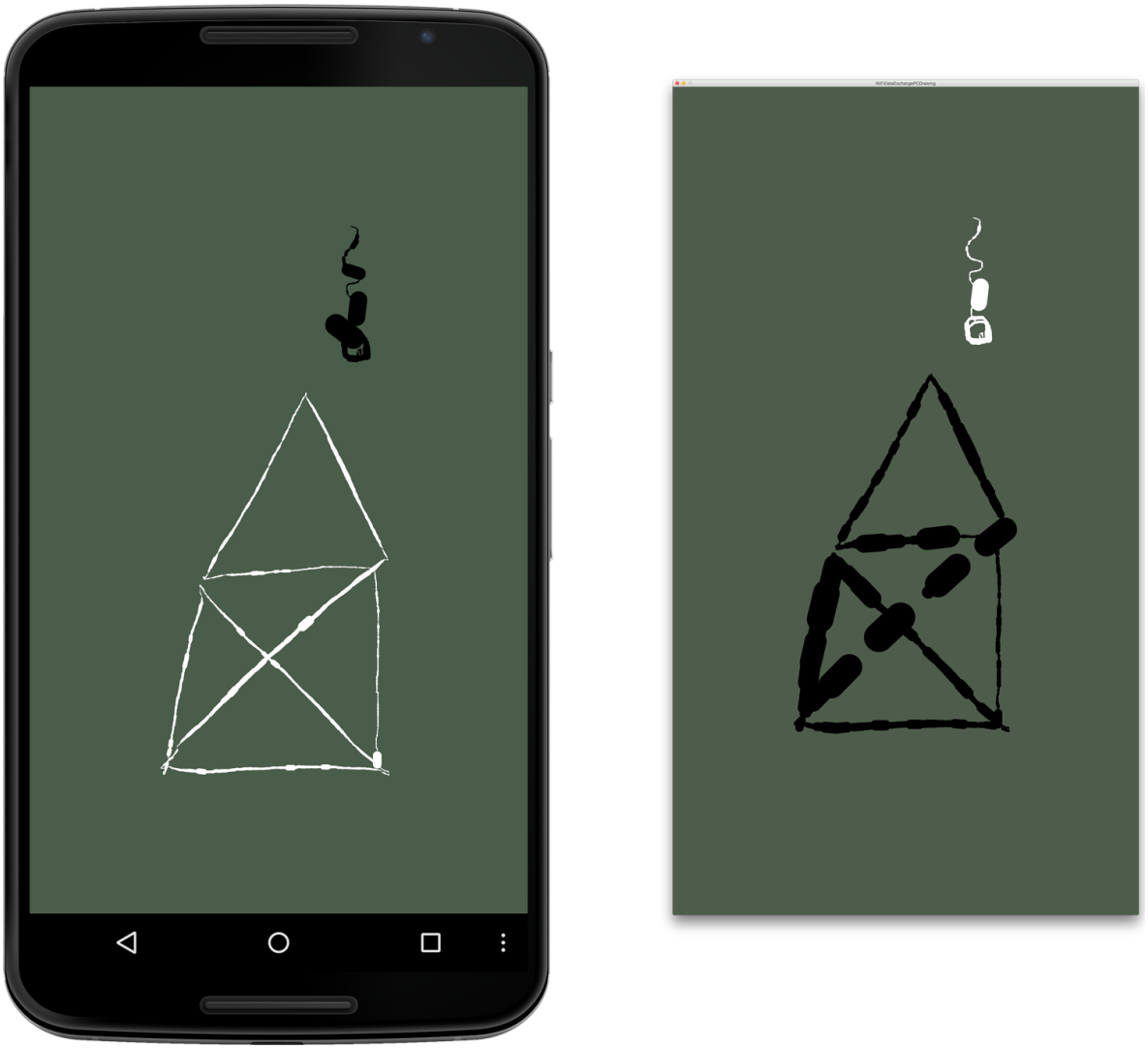


Figure 6.2 - Networked drawing app.

The image illustrates the app running on the Android (left) and the sketch running on the desktop PC (right).

Program the Android

Compared to the previous sketch, where we sent accelerometer data from the Android to the desktop and mouse info from the desktop to the Android, we'll focus now on the `mouseX` and `mouseY` integer values we'll need to draw, sending only those two constants

back and forth using OSC. The sketches for the Android and the PC are identical, with the exception of the single line of code that specifies the remote IP address. Since we now know the IP addresses of both the Android and the PC, we can complete this project using only the oscP5 and netP5 libraries.

Let's take a look:

code/Networking/WiFiDataExchangeAndroidDrawing/WiFiDataExchangeAndroidDrawing.pde

```
import oscP5.*;
import netP5.*;

OscP5 oscP5;
NetAddress remoteLocation;
int x, y, px, py;

void setup() {
  orientation(PORTRAIT);
  oscP5 = new OscP5(this, 12001);
  remoteLocation = new NetAddress("192.168.1.2", 12001);  // 1
  background(78, 93, 75);
}

void draw() {
  stroke(0);
  float remoteSpeed = dist(px, py, x, y);                 // 2
  if (remoteSpeed < 100) strokeWeight(remoteSpeed);       // 3
  line(px, py, x, y);                                     // 4
  px = x;                                                 // 5
  py = y;                                                 // 6
  if (mousePressed) {
    stroke(255);
    float speed = dist(pmouseX, pmouseY, mouseX, mouseY); // 7
    strokeWeight(speed);                                  // 8
    if (speed < 100) line(pmouseX, pmouseY, mouseX, mouseY); // 9
    OscMessage myMessage = new OscMessage("AndroidData");
    myMessage.add(mouseX);
    myMessage.add(mouseY);
    oscP5.send(myMessage, remoteLocation);
  }
}

void oscEvent(OscMessage theOscMessage) {
  if (theOscMessage.checkTypetag("ii"))
  {
    x =  theOscMessage.get(0).intValue();
    y =  theOscMessage.get(1).intValue();
  }
}
```

Here are the steps we take to change the `draw()` and `oscEvent()` methods from the previous sketch Networking/WiFiDataExchangeAndroid/WiFiDataExchangeAndroid.pde:

1. Create a new OSC connection on port `12001` to avoid conflicts with the previous sketch Networking/WiFiDataExchangePC/WiFiDataExchangePC.pde, which runs an OSC connection on port `12000`.
2. Calculate the speed of the remote mouse, which we'll use for the stroke weight.

3. Define a stroke weight for the line drawing sent from the desktop via OSC. It is determined by the mouse speed and calculated from the difference between the previous to the current mouse position using the `dist()` method.
4. Draw a line from the previous mouse position stored in `px` and `py` to the current mouse position's `x`and `y`, received via OSC.
5. Assign the current horizontal mouse position `x` to the previous position `px` once we are done drawing.
6. Assign the current vertical mouse position `y` to the previous position `py` once we are done drawing.
7. Calculate the speed of the fingertip moving across the screen using the distance `dist()` from the previous position (`pmouseX`, `pmouseY`) to the current position (`mouseX`, `mouseY`).
8. Set the stroke weight for the line drawing on the Android to the fingertip's `speed`.
9. Draw a line from the previous to the current finger position.

We moved a port number higher compared to the previous sketch so that the number would not conflict with the already established connection there. An OSC conflict with an already established connection on a specific port would be reported to the Processing console like this:

```
ERROR @ UdpServer.start() IOException, couldnt create new DatagramSocket @ port 12000
java.net.BindException: Address already in use
```

If we stopped the app that occupies the port, we can reuse that port number. To stop a sketch, hold down the home button on the Android device and swipe the app horizontally to close it, or choose Settings ↦ Manage Apps ↦ Wi-FiDataExchangeAndroid ↦ Force Stop.

Because we've moved to port number `12001` for this sketch, we won't run into a conflict. Let's go ahead and test the app now.

Run the Android App

Load the sketch for the Android and run it on the device. When the app launches, the console reports that we are running an OSC server on port `12001`:

```
 UdpServer.run() UdpServer is running @ 12001
```

When you move your finger across the touch screen surface, you draw white lines in different thicknesses, depending on how fast you are going.

The Android sketch is now complete, and we can move on to the PC sketch so that we have two devices that can doodle collaboratively.

Program the PC

Now let’s work on the PC sketch. As mentioned earlier, the Android and desktop sketch involved in this virtual canvas project are identical; we only need to make sure the IP address matches the remote Android device. Let’s take a look at the following code snippet, where only the IP address differs from the Android sketch.

code/Networking/WiFiDataExchangePCDrawing/WiFiDataExchangePCDrawing.pde

```
import oscP5.*;
import netP5.*;

OscP5 oscP5;
NetAddress remoteLocation;
int x, y, px, py;

void setup() {
  size(1440, 2560);
  oscP5 = new OscP5(this, 12001);
  remoteLocation = new NetAddress("192.168.1.3", 12001);  // 1
  background(78, 93, 75);
}

void draw() {
  stroke(0);
  float remoteSpeed = dist(px, py, x, y);
  strokeWeight(remoteSpeed);
  if (remoteSpeed < 100) line(px, py, x, y);
  px = x;
  py = y;
  if (mousePressed) {
    stroke(255);
    float speed = dist(pmouseX, pmouseY, mouseX, mouseY);
    strokeWeight(speed);
    if (speed < 100) line(pmouseX, pmouseY, mouseX, mouseY);
    OscMessage myMessage = new OscMessage("PCData");
    myMessage.add(mouseX);
    myMessage.add(mouseY);
    oscP5.send(myMessage, remoteLocation);
  }
}

void oscEvent(OscMessage theOscMessage) {
  if (theOscMessage.checkTypetag("ii"))
  {
    x =  theOscMessage.get(0).intValue();
    y =  theOscMessage.get(1).intValue();
  }
}
```

To enable a graphical output on the PC, we add the following single line of code.

---

1. Adjust the IP address as a parameter in the `NetAddress` object `remoteLocation` to match the Android IP.

Now we are ready to doodle and communicate both ways.

Run the PC App

Let’s go ahead and run the sketch on the desktop PC. The Android sketch is already running. If you draw with your mouse in the display window on the desktop, you will cause white lines to appear on the screen whose weight increases the faster you draw.

Now let’s go back to the Android and draw on its touch screen surface while keeping an eye on the desktop window. Notice that the lines you draw “locally” appear in white, while those that appear “remotely” are in black. Keep doodling using either the desktop or the Android, and you should see the same image you draw on one device appear in reverse colors on the other, as shown in Figure 6.2 (above). Black and white marks perpetually override each other as we keep doodling.

You’ve exchanged data between the Android and a desktop PC using Wi-Fi; now is a good time to test OSC communication between two Android devices using Wi-Fi. For this, you’ll need to go find a second Android device.

Run the Sketch on a Pair of Androids

Now that you have located a second Android device—let’s call it Android 2—you can go ahead and confirm that you are able to send OSC messages between a pair of Android devices as well. Let’s make sure again that you are on the correct network with the second device, and choose Settings ↦ “Wireless & networks.” Write down the IP address of the second device. With the IP addresses of both devices ready, open the sketch Networking/WiFiDataExchangeAndroidDrawing/WiFiDataExchangeAndroidDrawing.pde, that we’ve already loaded onto the first device.

Adjust the IP address for `remoteLocation` to match Android 2, and run the sketch on Android 1. The app starts up and Android 1 is ready. Repeat the steps on the other device, adjusting the IP address to match Android 1 and running the sketch on Android 2. If you are still running the other app on port `12001`, adjust the port number to use another port.
You’ve mastered networking Android devices using Wi-Fi. Now let’s explore a slightly more advanced networking app, where we work with an extended set of OSC messages to develop a simple marble-balancing game for two devices.

Network a Pair of Androids for a Multiplayer Game

For this project, we are going to build on the previous sketch, in which we connected two Android devices using Wi-Fi and OSC. Let’s build a simple multiplayer game where each player uses an Android device to tilt a marble toward a selected target, as illustrated in Figure 6.3. Instead of a scenario in which one player controls the outcome on a single device, two

devices will share a virtual game board whose orientation reflects the actions of the two players. One player influences the tilt of the other's game board, and vice versa.

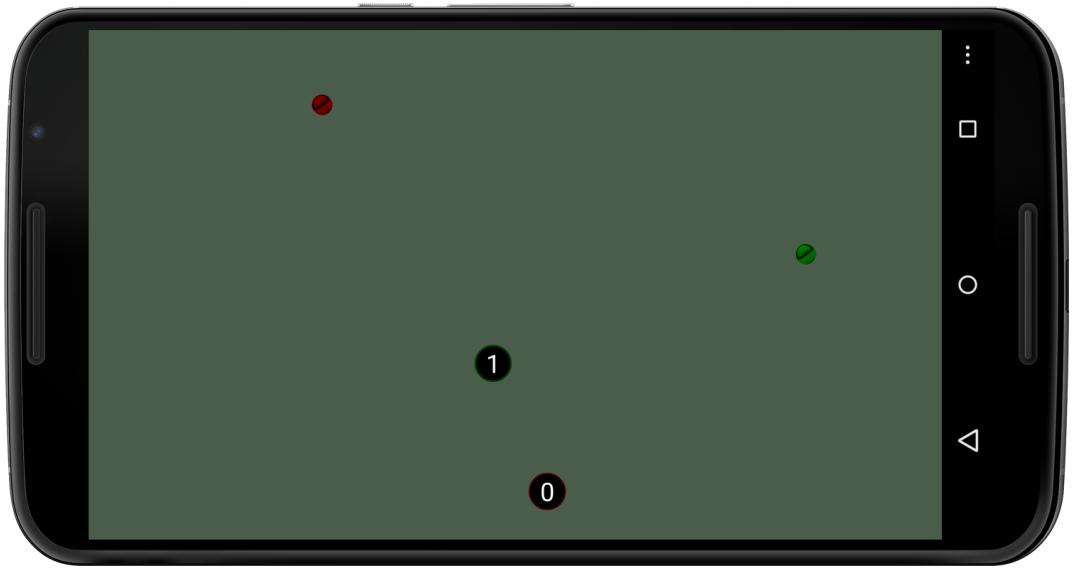


Figure 6.3 - Multiplayer balancing game

Played with two Android devices, two players compete in navigating individual marbles to the targets on a shared board, where the orientation of both devices influence each other.

To make the marbles for this game look three-dimensional, we'll load an image that provides us with the image texture we need for the desired effect. We'll use Processing's `PImage` class again to store the image.We can load `gif`, `jpg`, `tga`, and `png` images using the `loadImage()`. We are going to use a PNG-formatted image because it supports a transparent background. With a transparent background, the image will appear to float on the surface of the game board without the background color showing up as a rectangle as the marble moves across it. `PImage` also offers us a `tint()` method, which we can use to create two differently colored marbles from one image.

The sketch we are going to build is identical on both devices with the exception of the value assigned to the variable `remoteAddress`, which points to the other Android device. This time we'll send a few more values via OSC compared with the earlier Networking/WiFiDataExchangeAndroidDrawing/WiFiDataExchangeAndroidDrawing.pde—seven instead of three, including the position of the marble, its speed, the position of each target, and the score for each player. OSC allows us to mix the data types we send within one message, so we'll send four floating point values followed by three integers and determine a valid OSC message using the `checkTypetag(ffffiii)`method.

To assign random positions to the marbles and targets when the app starts up, we'll use Processing's `random()` method. It generates random floating point numbers every time the method is called. We can use it with one parameter (floating point or integer number),

causing `random()` to generate values ranging from zero to that number. We can also use it with two parameters, making `random()` return values ranging from the first to the second number parameter.
Let's take a look:

code/Networking/MultiplayerBalance/MultiplayerBalance.pde

```
import oscP5.*;
import netP5.*;
import ketai.sensors.*;

OscP5 oscP5;
KetaiSensor sensor;
NetAddress remoteLocation;
float x, y, remoteX, remoteY;
float myAccelerometerX, myAccelerometerY, rAccelerometerX, rAccelerometerY;
int targetX, targetY, remoteTargetX, remoteTargetY;
int score, remoteScore;
float speedX, speedY = .01;
PImage marble;
String remoteAddress = "192.168.1.3";  //Customize!
void setup() {
  sensor = new KetaiSensor(this);
  orientation(LANDSCAPE);
  textAlign(CENTER, CENTER);
  textSize(72);
  initNetworkConnection();
  sensor.start();
  strokeWeight(5);
  imageMode(CENTER);
  marble = loadImage("marble.png");                       // 1
  init();
}

void draw() {
  background(78, 93, 75);
  // Targets
  fill (0);
  stroke(0, 60, 0);
  ellipse(targetX, targetY, 100, 100);
  stroke (60, 0, 0);
  ellipse(remoteTargetX, remoteTargetY, 100, 100);
  noStroke();
  fill(255);
  text(score, targetX, targetY);
  text(remoteScore, remoteTargetX, remoteTargetY);
  // Remote Marble
  tint(120, 0, 0);                                        // 2
  image(marble, remoteX, remoteY);                        // 3
  // Local Marble
  speedX += (myAccelerometerX + rAccelerometerX) * 0.1;   // 4
  speedY += (myAccelerometerY + rAccelerometerY) * 0.1;

  if (x <= 25+speedX || x > width-25+speedX) {
    speedX *= -0.8;                                       // 5
  }
  if (y <= 25-speedY || y > height-25-speedY) {
    speedY *= -0.8;
  }
  x -= speedX;                                            // 6
  y += speedY;
  tint(0, 120, 0);
  image(marble, x, y);
  // Collision
  if (dist(x, y, targetX, targetY) < 25) {
```

```
    score++;
    background(60, 0, 0);
    init();
  }

  OscMessage myMessage = new OscMessage("remoteData");    // 7
  myMessage.add(x);
  myMessage.add(y);
  myMessage.add(myAccelerometerX);
  myMessage.add(myAccelerometerY);
  myMessage.add(targetX);
  myMessage.add(targetY);
  myMessage.add(score);
  oscP5.send(myMessage, remoteLocation);
}

void oscEvent(OscMessage theOscMessage) {
  if (theOscMessage.checkTypetag("ffffiii"))              // 8
  {
    remoteX = theOscMessage.get(0).floatValue();
    remoteY = theOscMessage.get(1).floatValue();
    rAccelerometerX = theOscMessage.get(2).floatValue();
    rAccelerometerY = theOscMessage.get(3).floatValue();
    remoteTargetX = theOscMessage.get(4).intValue();
    remoteTargetY = theOscMessage.get(5).intValue();
    remoteScore = theOscMessage.get(6).intValue();
  }
}

void onAccelerometerEvent(float _x, float _y, float _z)
{
  myAccelerometerX = -_y;
  myAccelerometerY = _x;
}

void initNetworkConnection()
{
  oscP5 = new OscP5(this, 12000);
  remoteLocation = new NetAddress(remoteAddress, 12000);
}
void init() {                                             // 9
  x = int(random(25, width-25));
  y = int(random(25, height-25));
  targetX = int(random(25, width-25));
  targetY = int(random(25, height-25));
}
```

Here's what we are working with for the balancing game.

---

1. Load a marble image from the sketch `data` folder using `loadImage()`.
2. Tint the remote marble red to distinguish the two players. The `tint()` method is applied to the `marble.png` image, drawn next.
3. Draw the tinted marble image at the remote position (`remoteX`, `remoteY`).
4. Calculate the horizontal marble speed `speedX` by adding the horizontal accelerometer values of both devices, called `myAccelerometerX` and `rAccelerometerX`. Reduce the speed by multiplying the accelerometer sum by a factor of `0.1`. Do the same for the vertical direction in the following line.
5. Bounce the marble off the screen edge whenever its distance from the edge is less then `25` pixels, which happens to equal half the diameter of the marble image. Consider

the `speedX`, and dampen the speed at every bounce, reducing it to 80% (`0.8`) of the previous speed. Do the same for the y-axis next.
6. Update the horizontal position `x` of the local marble.
7. Add all the local marble positions (`x` and `y`), the local accelerometer values (`myAccelerometerX` and `myAccelerometerY`), the local target (`targetX` and `targetY`), and the local `score` to the OSC message (`myMessage`), and send it to the other device.
8. Look for a package of four floating point numbers followed by three integers, and parse the message to assign values for the remote position (`remoteX` and `remoteY`), the remote accelerometer values (`rAccelerometerX` and `rAccelerometerY`), the remote target position (`remoteTargetX` and `remoteTargetY`), and the remote score (`remoteScore`).
9. Initialize the local marble position and target position to appear randomly on the screen.

Now let's test the game.

Run the App

To run the game, let's first check the IP addresses on both of the Android devices we'll be using so that we can adjust the `remoteAddress` variable in the sketch for each Android device. Connect the first Android device (Android 1) to the desktop computer via USB and load the `MultiPlayerBalance` sketch into Processing. Locate the `remoteAddress` variable in the code; we'll adjust this in a moment.
Now look up the IP address of the second Android device (Android 2), which is currently not connected via USB cable. Navigate to Settings ↦ "Wireless & networks" on the device, and tap the Wi-Fi network the device is connected to. The IP address assigned to the device is shown on the bottom of the list. Write it down; it's the IP address for Android 2.

Now go back to the Processing sketch and adjust the IP address for `remoteAddress` to match the IP address (for Android 2) that you've just looked up.
Run the sketch on Android 1, which is already connected via USB cable. When the sketch is launched on the device, disconnect Android 1 and connect Android 2. We'll repeat those steps for the other device.

With Android 2 connected via USB cable, locate `remoteAddress` in the code so we can adjust the IP address again. Look up the IP address of Android 1 now, which is currently not connected via USB cable. Write it down.
Go back to the Processing code and adjust the IP address for `remoteAddress` to match the IP address (of Android 1) that you've just looked up.
Run the sketch on Android 2, which is already connected via USB cable. When the sketch launches on the device, we are ready to play.

Grab a friend and hand over one device. Now hold the device as level as possible to balance the green (local) marble toward its target, shown with a green stroke. Your friend is pursuing the same objective, which makes the balancing act more difficult as it probably interferes with the path of your marble toward the target. Once you've hit the target, or vice versa, the score increases and the game restarts. You can see score updates immediately when the target is reached. Enjoy your game!

This two-player game concludes our explorations into wireless networking using Wi-Fi networks.

Wrapping Up

You've mastered the exchange of data across devices within a Wi-Fi network. You've learned how to package diverse data types into OSC messages, send them across wireless local area networks, and unpack them on the other side. You've got devices talking to each other, which you can now easily expand into other application contexts, including making the Android a remote control for other devices on the network.

But if we don't have access to a Wi-Fi network and we'd like to exchange data with other Android devices directly, peer-to-peer, what can we do? All fourth-generation Android devices are equipped with Bluetooth, so in the next chapter we'll take a look at this short-distance wireless technology, and we'll also explore an emerging peer-to-peer networking standard known as Wi-Fi Direct. Then, in the following chapter, we'll explore near field communication, the emerging short-range wireless standard for contactless payment and smart cards. When you're done, you'll be an Android networking pro.

7. Peer-to-Peer Networking Using Bluetooth and Wi-Fi Direct

In this chapter, we'll give peer-to-peer (`P2P`) networks the attention they deserve. We've mastered the exchange of data between Android devices using Wi-Fi. Now it's time to end our dependence on wireless infrastructure. Popular services such as Skype and BitTorrent are only two examples that use peer-to-peer technology. However, the concept of P2P communication doesn't stop with telephony or file sharing and has little do with copyright.

P2P networking has several advantages. First of all, it's free. We don't require a carrier network or access to `Wi-Fi` infrastructure, and we won't be restricted by data quotas. P2P still works if wireless infrastructure is unavailable or overwhelmed due to high demand, for instance. It uses less power due to its short range and can help protect privacy because information remains decentralized. And finally, information flows directly from one device to the other—we can control the information flow and choose whether data is saved or retained. P2P communication between two devices doesn't preclude us from also reaching out to web or cloud servers. For example, if we are connected P2P while we are on the move, we can update an online database as soon as a carrier network becomes available. Both networking paradigms can coexist and complement each other. P2P has the advantage that it can reliably provide us with instantaneous feedback across devices due to the very small lag time, and it provides the transmission rates that are crucial for some multiuser or multiplayer apps.

The most common way to implement P2P exchanges of data between Android devices is to use Bluetooth, which is available on all Android devices shipping today. We'll start by building a remote cursor app, where we visualize the cursor position of two devices connected via Bluetooth. Then we'll build a peer-to-peer survey app that lets us create a poll for multiple users. Connected via Bluetooth, users pick answers from a common multiple choice questionnaire displayed on their device screens. Individual answers are gathered and visualized in real time, giving each user immediate feedback on the collective response and the distribution of answers as they accumulate.

Then we'll move on to Wi-Fi Direct, an emerging peer-to-peer networking standard where each device can serve as the Wi-Fi access point for the network. We'll revisit the remote cursor app and modify it to use Wi-Fi Direct so that we can directly compare its performance to Bluetooth. Wi-Fi Direct is designed to provide a higher bandwidth and better network range than Bluetooth. To get started, let's first take a look at the main classes we'll use in this chapter.

Introducing Short-Range Networking and UI Classes

For the apps we'll develop in this chapter, we'll use the following networking and UI classes from the Ketai library:

- `KetaiBluetooth` A Ketai class for working with Bluetooth on Android devices—the class contains the necessary methods for Bluetooth discovery, pairing, and communication using the popular Bluetooth standard
- `KetaiWiFiDirect` A Ketai class to simplify working with Wi-Fi Direct on Android devices—the class contains the necessary methods for Wi-Fi Direct peer discovery and data exchange. In a Wi-Fi Direct network, every Wi-Fi Direct-enabled device can serve as the access point for the other devices in the Wi-Fi network.
- `KetaiOSCMessage` A Ketai class that is identical to the oscP5 library's `OscMessage` class we worked with in Using the Open Sound Control Networking Format, with the difference being that it allows us to create `KetaiOSCMessage` using a byte array—it makes some private methods in `OscMessage` public so we can use it for Bluetooth communication.
- `KetaiList` A Ketai UI class that makes it easier to work with the native Android `ListView` widget—this class contains methods to populate, display, refresh, and retrieve strings from a selected list item. A `KetaiList` can be created using a `String` array or a `String` `ArrayList`.
- `KetaiKeyboard` A class included in the Ketai `UI` package that allows us to toggle the Android software keyboard on and off without importing additional Android UI classes

We'll start with Bluetooth because it's the most ubiquitous peer-to-peer technology. Let's take a closer look at the Bluetooth methods that Ketai provides.

Working with the KetaiBluetooth Class

Besides the usual `start()` and `stop()` methods, `KetaiBluetooth` provides the following methods for discovering, pairing, and connecting Bluetooth devices:

- `onBluetoothDataEvent()` Returns data sent via Bluetooth, including the device name where it originated as a `String` (useful when more than one Bluetooth device is connected) and the Bluetooth data as `byte[]` array
- `makeDiscoverable()` Makes a Bluetooth device discoverable for 300 seconds
- `discoverDevices()` Scans for discoverable Bluetooth devices
- `getDiscoveredDeviceNames()` Returns a list of all Bluetooth devices found within range of the Bluetooth radio
- `connectToDeviceByName()` Connect to a device using its assigned Bluetooth name
- `broadcast()` Writes data to all connected Bluetooth devices
- `getPairedDeviceNames()` Provides a list of devices that have been successfully paired with the Android device—paired devices can reconnect automatically if they are discoverable and within range, and they do not need to repeat the pairing process.
  Now that we've seen the classes and methods that we'll use to build apps in this chapter, let's now take a closer look at Bluetooth, the most ubiquitous peer-to-peer standard.

Introducing Bluetooth

Every Android device ships with a Bluetooth radio. It is a popular communication standard used for consumer electronics, health and wellness devices, PC peripherals, sports and fitness equipment, and smart home appliances, and it is typically found in wireless game controllers, headsets, keyboards, printers, and heart rate sensors, to name a few. Because it is so ubiquitous, it will remain an important protocol even as new networking standards emerge.

The standard is managed by the Bluetooth Special Interest Group, which includes more than sixteen thousand member companies. "Bluetooth" refers to a tenth-century Danish king who united dissonant tribes into a single kingdom. The implication is that Bluetooth has done the same for the device and accessory market, uniting communication protocols into one universal standard.

Bluetooth uses short-wavelength radio frequencies of 2400-2483.5 MHz and allows us to transfer data within a range of about 30 feet. It requires relatively little power and rarely experiences interference from other devices. But before we can transfer data, we must first pair the devices involved. Once we've done that successfully, the Android stores a list of known devices, which we can use to reconnect them without pairing them again. If we already know the unique 48-bit address of the Bluetooth device to which we'd like to connect, we can skip the pairing process entirely.

If the Bluetooth radio is powered on, any Bluetooth-enabled device can send an inquiry to initiate a connection. If a device is discoverable, it sends information about itself to other Bluetooth devices within reach, including its own device name, allowing the devices to pair

and exchange data securely. If we send data between Android devices, pairing and acceptance of the connection by the device owner is required for security reasons.

Let's now connect two Android devices via Bluetooth.

Working with the Android Activity Life Cycle

When we launch a Processing sketch as an app on the Android, we create an Android activity ready for us to interact via the touch screen interface. We typically don't need to deal with the activity life cycle on the Android because Processing takes care of it for us. To activate Bluetooth (and later `NFC`), we need to initialize the Bluetooth object we'll be working with at the very beginning of the activity life cycle.
When a new app or activity starts up, Android adds it to the stack of activities already running and places it on top of the stack to run in the foreground. The previous activity goes in the background and will come to the foreground again when the current activity exits.

We can summarize the four states an activity can take like this:

- The activity is active and running in the foreground on top of the stack.
- The activity lost focus because another non-fullscreen activity runs on top of the activity.
- The activity stopped because another activity is covering it completely.
- The paused or stopped activity is killed to make memory available for the active activity.

When an activity goes though this life cycle, Android provides the following callback methods for us to use. When the activity starts up, Android calls the following:

`onCreate()`	Called when the activity is starting
`onStart()`	Called after `onCreate()` when the activity starts up—if the activity is already running, `onRestart()` is called before `onStart()`.
`onResume()`	Called after `onStart()` when the activity becomes visible

After `onResume()`, the activity is running in the foreground and active. If we launch another activity, Android calls these methods:

`onPause()`	Called when another activity comes in the foreground

`onStop()`	Called after `onPause()` when the activity is no longer visible
`onDestroy()`	Called after `onStop()` when the activity is finishing or destroyed by the system

Enabling Bluetooth

To work with Bluetooth for the Bluetooth apps we'll create in this chapter, we will need to launch a new activity to initialize our Bluetooth right at the beginning when the activity starts up using `onCreate()`. Once Bluetooth is active, this activity returns to us the Bluetooth object we need via `onActivityResult()`, which is called when the app starts up immediately before `onResume()` in the activity life cycle. We'll look at the code to enable Bluetooth in more detail in this P2P/BluetoothCursors/EnableBluetooth.pde.
For the projects in this book, we'll need to deal with the life cycle only for Bluetooth and `NFC`. We'll work more with the activity life cycle in Enable NFC and Bluetooth in the Activity Life Cycle, where we initiate `NFC`and Bluetooth. For all other apps, we can let Processing handle the life cycle. Future versions of Processing might allow libraries to work with lifecycle callback methods, so we don't need to include such code inside the sketch.

Connect Two Android Devices via Bluetooth

In the following sketch, we'll work with three tabs. The main tab, `BluetoothCursors`, contains our usual `setup()` and `draw()` methods and global variables. The second tab, `EnableBluetooth`, contains code that is necessary to enable Bluetooth on startup, registering our Bluetooth class when the so-called Android activity is created (this step might not be necessary in future versions of Processing). Processing allows us to not dive too deep into the Android application life cycle, and we'll try to keep it that way. The third tab, called `UI`, contains all the code we'll use for GUI elements like menus, an Android list to select Bluetooth devices, and the software keyboard to enter user input. When the sketch is complete, we'll get a screen similar to the one shown in Figure 7.0.

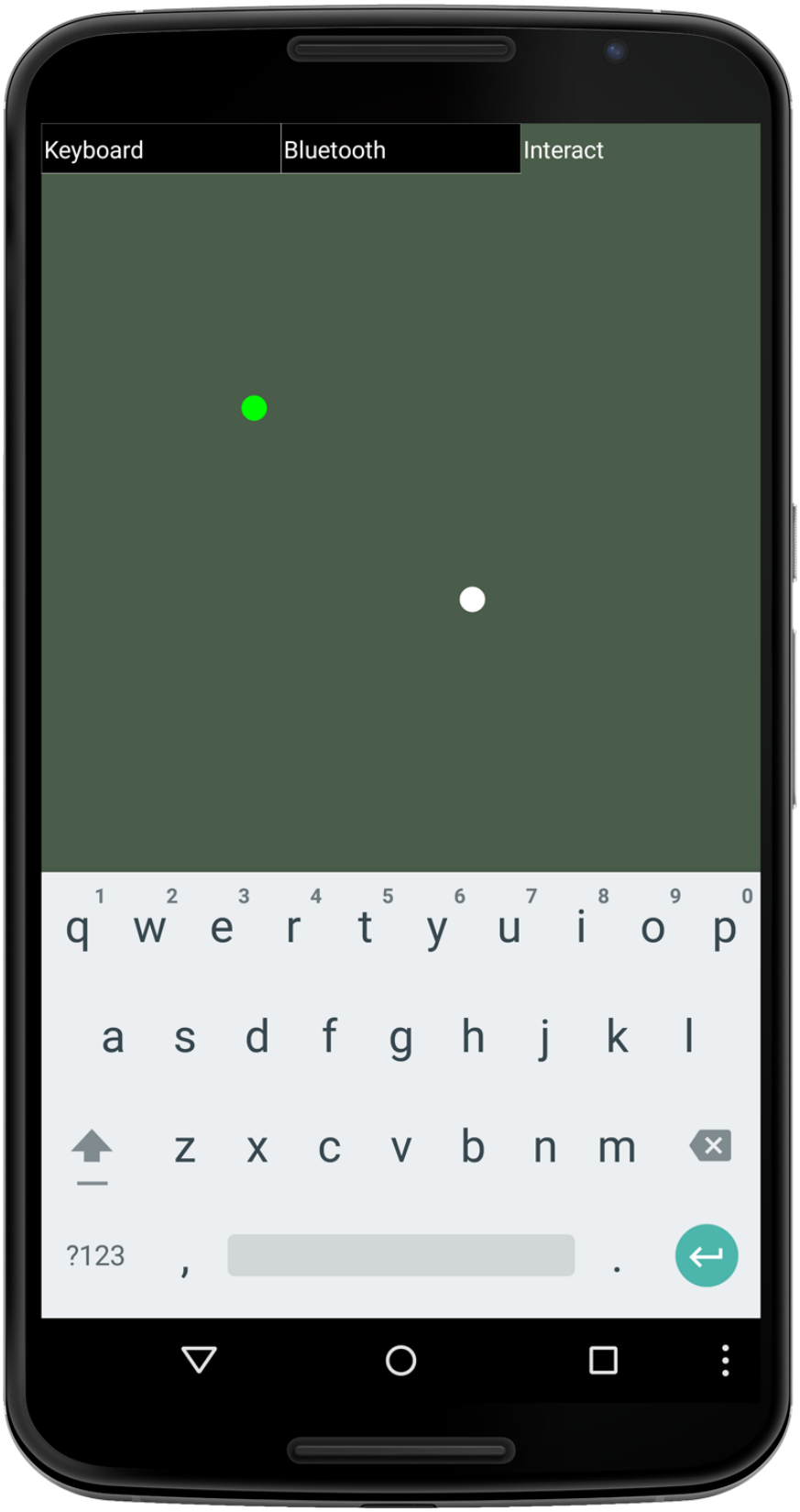


Figure 7.0 - Bluetooth Cursors app.

The illustration shows the local (white) and the remote (red) mouse pointer positions, marked as ellipses on the screen. The software keyboard is made visible using the keyboard tab shown at the top. Also shown are the Bluetooth and Interact tabs, which we use to interact with the cursors.

The code needs to facilitate the Bluetooth pairing process as follows: We start by making both Androids discoverable for Bluetooth by listing discovered devices. Then we choose the device to connect to from the list of discovered devices (you might be surprised to see what shows up). Finally, we pair and connect the devices to transfer the data via OSC, which we've already used in Using the Open Sound Control Networking Format. We'll need to confirm the Bluetooth connection in a popup window because we will be connecting the devices for the first time.

We'll use the Android software keyboard to discover other Bluetooth devices, make the device itself discoverable, connect to another device, list already paired devices, and show the current Bluetooth status. To work with the keyboard, we'll use the `KetaiKeyboard` class. And to show and pick discoverable Bluetooth devices to connect to, we'll use the `KetaiList` class, making it easy for us to work with a native Android list without importing additional packages.

Working with a KetaiList

We can create a `KetaiList` object using either a `String` array or a `String` `ArrayList`. Because an `ArrayList` stores a variable number of objects and we have a variable number of discoverable Bluetooth devices to connect to, it's the better choice for us here. We can easily add or remove an item dynamically in an `ArrayList`, and because we work with Bluetooth device names in our sketch, we'll create an `ArrayList`of type `String`.
Andreas Schlegel has updated his excellent ControlP5 UI library (as of 09/2012) to also work with the Android mode in Processing 2, making it a great tool to develop all custom aspects of UI elements, such as controllers, lists, sliders, buttons, and input forms. Although the UI elements do not use Android's native UI classes, ControlP5 elements can be fully customized to match the look and feel of your app while still maintaining consistency with the Android's UI style guide.

Let's get started with the main tab of our `BluetoothCursors` sketch.

code/P2P/BluetoothCursors/BluetoothCursors.pde

```
// Required Bluetooth methods on startup
import android.os.Bundle;                                   // 1
import android.content.Intent;                              // 2

import ketai.net.bluetooth.*;
import ketai.ui.*;
import ketai.net.*;
import oscP5.*;

KetaiBluetooth bt;                                          // 3

KetaiList connectionList;                                   // 4
String info = "";                                           // 5
PVector remoteCursor = new PVector();
boolean isConfiguring = true;
```

```
String UIText;

void setup()
{
  orientation(PORTRAIT);
  background(78, 93, 75);
  stroke(255);
  textSize(48);

  bt.start();                                              // 6

  UIText =  "[b] - make this device discoverable\n" +      // 7
    "[d] - discover devices\n" +
    "[c] - pick device to connect to\n" +
    "[p] - list paired devices\n" +
    "[i] - show Bluetooth info";
}

void draw()
{
  if (isConfiguring)
  {

    ArrayList<String> devices;                             // 8
    background(78, 93, 75);

    if (key == 'i')
      info = getBluetoothInformation();                    // 9
    else
    {
      if (key == 'p')
      {
        info = "Paired Devices:\n";
        devices = bt.getPairedDeviceNames();               // 10
      }
      else
      {
        info = "Discovered Devices:\n";
        devices = bt.getDiscoveredDeviceNames();           // 11
      }

      for (int i=0; i < devices.size(); i++)
      {
        info += "["+i+"] "+devices.get(i).toString() + "\n";  // 12
      }
    }
    text(UIText + "\n\n" + info, 5, 200);
  }
  else
  {
    background(78, 93, 75);
    pushStyle();
    fill(255);
    ellipse(mouseX, mouseY, 50, 50);
    fill(0, 255, 0);
    stroke(0, 255, 0);
    ellipse(remoteCursor.x, remoteCursor.y, 50, 50);       // 13
    popStyle();
  }

  drawUI();
}

void mouseDragged()
{
  if (isConfiguring)
    return;

  OscMessage m = new OscMessage("/remoteMouse/");          // 14
  m.add(mouseX);
```

```
  m.add(mouseY);

  bt.broadcast(m.getBytes());                                  // 15
  // use writeToDevice(String _devName, byte[] data) to target a specific device
  ellipse(mouseX, mouseY, 20, 20);
}

void onBluetoothDataEvent(String who, byte[] data)         // 16
{
  if (isConfiguring)
    return;

  KetaiOSCMessage m = new KetaiOSCMessage(data);             // 17
  if (m.isValid())
  {
    if (m.checkAddrPattern("/remoteMouse/"))
    {
      if (m.checkTypetag("ii"))                              // 18
      {
        remoteCursor.x = m.get(0).intValue();
        remoteCursor.y = m.get(1).intValue();
      }
    }
  }
}

String getBluetoothInformation()                              // 19
{
  String btInfo = "Server Running: ";
  btInfo += bt.isStarted() + "\n";
  btInfo += "Discovering: " + bt.isDiscovering() + "\n";
  btInfo += "Device Discoverable: "+bt.isDiscoverable() + "\n";
  btInfo += "\nConnected Devices: \n";

  ArrayList<String> devices = bt.getConnectedDeviceNames();  // 20
  for (String device: devices)
  {
    btInfo+= device+"\n";
  }

  return btInfo;
}
```

Here are the steps we need to take.

---

1. Import the `os.Bundle` Android package containing the `onCreate()` method we need to initialize Bluetooth.
2. Import the `content.Intent` Android package containing the `onActivityResult()`, which is called when the app starts up.
3. Create a `KetaiBluetooth` type variable, `bt`.
4. Create a `KetaiList` variable that we'll use to select the device to connect to.
5. Create a `String` variable, `info`, to store changing status messages that we want to output to the Android screen.
6. Start the `bt` Bluetooth object.
7. Provide instructions for connecting to the other device using the keyboard.
8. Create an `ArrayList` of type `String` to store the Bluetooth device, `devices`.

9. Retrieve a Bluetooth status update and assign it to our `info` variable for screen output when we press `i` on the keyboard.
10. Get a list of paired devices and assign it to the `devices` `ArrayList` to update the `KetaiList` when we press the `p` on the keyboard.
11. Get a list of discovered devices and assign it to the `devices` `ArrayList` to update the `KetaiList` when we press `d` on the keyboard.
12. Append each Bluetooth device entry in `devices` to our `info` text output, converting each individual `ArrayList` item into human-readable text using the `toString()` method.
13. Use the `x` and `y` components of the `remoteCursor` `PVector`, which stores the remote cursor location, and draw an ellipse at the exact same x and y location.
14. Create a new `OSC` message `m` to add our `mouseX` and `mouseY` cursor position to.
15. Broadcast the `OSC` message containing the cursor position to all connected devices using the `OSCbroadcast()` method. Alternatively, we can use `writeToDevice(String _devName, byte[] data)` to send the message to only one specific device.
16. Receive the `byte[]` array when the new Bluetooth `data` is sent from the remote device.
17. Receive the `data` as an `OSC` message.
18. Check if the `OSC` message contains two integer values for the mouse x and y position. We've also checked if the `OSC` message is valid and if the message we've sent contains the label "remoteCursor."
19. Return a `String` containing Bluetooth status info, including if Bluetooth `isStarted()` and `isDiscoverable()`, as well as the individual names of connected Bluetooth devices.
20. Get a list of all connected Bluetooth devices using `getConnectedDeviceNames()`.

We've completed the crucial components of our sketch in `setup()` and `draw()`. To enable Bluetooth when the app starts up, we'll need to work with the activity life cycle as described in Working with the Android Activity Life Cycle. We'll put the code to enable Bluetooth into the tab named `EnableBluetooth`. Let's take a look.

code/P2P/BluetoothCursors/EnableBluetooth.pde

```
void onCreate(Bundle savedInstanceState) {                // 1
  super.onCreate(savedInstanceState);
  bt = new KetaiBluetooth(this);                          // 2
}

void onActivityResult(int requestCode, int resultCode, Intent data) {
  bt.onActivityResult(requestCode, resultCode, data);     // 3
}
```

These are the steps we need to take to enable Bluetooth.

1. Use the Android `onCreate()` method to initialize our Bluetooth object. The method is called when the activity is starting.
2. Initialize the Bluetooth object `bt` when the activity is created.
3. Return the `bt` object to the sketch using `onActivityResult()`, called right before `onResume()` in the activity life cycle.
   We looked at the required `onCreate()` and `onActivityResult()` methods to initialize Bluetooth at the beginning of the activity.

Programming the UI

Now let's return to the part of the code that is responsible for all the UI elements, which we'll put in the `UI`tab of the sketch. It takes care of GUI elements and keyboard menu items. Because the Bluetooth pairing process requires us to select a device from a whole list of discovered devices (you'll probably be surprised by how many Bluetooth devices are broadcasting around you), we'll use a `KetaiList` to simplify the selection process. We'll also need the keyboard to make menu selections during the pairing process, and we'll use the `KetaiKeyboard` class to toggle the keyboard on and off. For the `KetaiList`, we'll use the `onKetaiListSelection()` method to capture when the user picks an item from the list. And to show and hide the `KetaiKeyboard`, we'll work with the `toggle()` method.
Here are the steps to programming the UI.

code/P2P/BluetoothCursors/UI.pde

```
// UI methods

void mousePressed()

{
  if (mouseY <= 100 && mouseX > 0 && mouseX < width/3)
    KetaiKeyboard.toggle(this);                          // 1
  else if
    (mouseY <= 100 && mouseX > width/3 && mouseX < 2*(width/3)) //config button
    isConfiguring=true;                                  // 2
  else if
    (mouseY <= 100 && mouseX >  2*(width/3) && mouseX < width) // draw button
  {
    if (isConfiguring)
    {
      background(78, 93, 75);
      isConfiguring=false;
    }
  }
}

void keyPressed() {
  if (key =='c')
  {
    //If we have not discovered any devices, try prior paired devices
    if (bt.getDiscoveredDeviceNames().size() > 0)
      connectionList = new KetaiList(this, bt.getDiscoveredDeviceNames());  // 3
    else if (bt.getPairedDeviceNames().size() > 0)
```

```
      connectionList = new KetaiList(this, bt.getPairedDeviceNames());  // 4
  }

  else if (key == 'd')
    bt.discoverDevices();                                  // 5
  else if (key == 'b')
    bt.makeDiscoverable();                                 // 6
}

void drawUI()
{
  pushStyle();                                             // 7
  fill(0);
  stroke(255);
  rect(0, 0, width/3, 100);

  if (isConfiguring)
  {
    noStroke();
    fill(78, 93, 75);
  }
  else
    fill(0);

  rect(width/3, 0, width/3, 100);
  if (!isConfiguring)
  {
    noStroke();
    fill(78, 93, 75);
  }
  else
  {
    fill(0);
    stroke(255);
  }
  rect((width/3)*2, 0, width/3, 100);
  fill(255);

  text("Keyboard", 5, 70);                                 // 8
  text("Bluetooth", width/3+5, 70);                        // 9
  text("Interact", width/3*2+5, 70);                       // 10
  popStyle();
}

void onKetaiListSelection(KetaiList connectionList)       // 11
{
  String selection = connectionList.getSelection();       // 12
  bt.connectToDeviceByName(selection);                    // 13
  connectionList = null;                                  // 14
}
```

Let’s take a look at the steps.

---

1. Toggle the Android’s software keyboard using `toggle()`. Make the `KetaiKeyboard` visible if it’s hidden; hide it if it’s visible.
2. Set the boolean variable `isConfiguring` to `true` so we can switch to the Bluetooth configuration screen and for the pairing process.
3. Assign the list of paired Bluetooth devices to the `KetaiList` `connectionList`, given there are paired devices.

4. Assign the list of discovered Bluetooth devices to the `KetaiList` `connectionList` using the String array returned by `bt.getDiscoveredDeviceNames()`, given there are discovered devices.
5. Discover Bluetooth devices using `discoverDevices()` when we press `d` on the software keyboard.
6. Make the device discoverable using the `makeDiscoverable()` method.
7. Save the current style settings using `pushStyle()` to preserve the stroke, text size, and alignment for the UI elements. Use the method in combination with `popStyle()` to restore the previous style settings.
8. Draw the Keyboard UI tab.
9. Draw the Bluetooth UI tab.
10. Draw the Interact UI tab.
11. Call the `onKetaiListSelection()` event method when a user selects an item from the `KetaiList`.
12. Get the `String` that the user picked from the `KetaiList`.
13. Connect to the selected Bluetooth device using the `connectToDeviceByName()` method.
14. Remove all items from the list by setting the `connectionList` object to `null`.

Now let's test the app.

Run the App

Let's set the correct Android permissions before we run the sketch. Open the Android Permission Selector as we've done previously (see Setting Sketch Permissions), and check the following permissions:

- `BLUETOOTH`
- `BLUETOOTH ADMIN`
- `INTERNET`

Now connect your first device to your workstation with a USB cable and run the sketch. When the app is compiled, all three tabs in the sketch, `BluetoothCursors`, `EnableBluetooth`, and `UI`, will be compiled into one app. You will see our Bluetooth tab active and the menu options displayed on the screen. Before we interact, let's install the app on our second Android device. Disconnect your first device from the USB port, connect your second Android, and install the identical `BluetoothCursors` sketch on the device. The sketch launches, and we are ready to pair the two devices.

On your first device (currently disconnected from USB), show the software keyboard by pressing the Keyboard tab. Then press `b` on the keyboard. If Bluetooth is turned off for your device (Settings ↦ Bluetooth), you will be prompted to allow Bluetooth, as shown below. Otherwise the device will become discoverable for 300 seconds.

```
Android 1
[b] - make this device discoverable
[d] - discover devices
[c] - pick device to connect to
[p] - list paired devices
[i] - show Bluetooth info

%%%b
Alert:
An app on your phone wants to
make your phone discoverable
by other Bluetooth devices for
300 seconds.
%%%ALLOW
```

Now switch to your second device (currently connected to USB), and follow the process of discovering devices. Pick the name of the first Android device:

```
Android 2
%%%d
Discovered Devices
[0] Nexus 6
%%%c
%%%Nexus 6
Android 1
Bluetooth pairing Request
%%%pair
Android 2
%%%p
Paired Devices
[0] Nexus 6
%%%i
Sever Running: true
Device Discoverable: true

Connected Devices:
Nexus S (78:47:1D:A6:20:48)
```

When your screen output looks like what's shown above, the Bluetooth server is running on your second device and you have your first device show up in the list of connected devices. You are now ready to interact.

Press the Interact screen tab on both devices. You'll see a white dot for the local cursor and a red one for the remote one. As you move your finger over the screen surface of one Android device, observe the other device and see how the red dot moves magically to that position.

Congratulations! You've established a data connection between two Android devices using Bluetooth. Now it all depends on your Bluetooth antenna, which should reach a distance of about thirty feet. If you have a friend nearby to test this, try it out. Otherwise, it will not be possible to observe how the connection goes out of range.

The process of discovering and pairing Bluetooth devices can seem cumbersome. However, Bluetooth can't just accept an incoming connection without confirmation for good security

reasons. Once paired, we can reconnect automatically by picking the device address again from the list of paired devices. This is a sketch refinement you can try. If you'd like to "unpair" previously paired devices on your Android, tap the device name under Settings ↦ Bluetooth ↦ Paired Devices, and choose Unpair.

We will implement this remote cursor's app using Wi-Fi Direct later in this chapter P2P/WiFiDirectCursors/WiFiDirectCursors.pde, and you can then compare how the two standards perform in terms of update rate and wireless range.

Since you've mastered peer-to-peer networking using Bluetooth, let's build on our Bluetooth skills and create a survey app for multiple Bluetooth users.

Create a Survey App Using Bluetooth

We'll now move on to the chapter project, which is a survey app for multiple users using Bluetooth networking. Such an app is useful for teaching, decision-making, and learning assessments. We'll build on our Bluetooth skills and put them into practice. We'll learn how to send different data types via `OSC` over Bluetooth, share numeric data across devices in real time, and learn how to work with custom Processing classes.
For this survey app, we'll broadcast a number of questions that you can imagine as text slides shared by multiple users. All survey participants respond to the questions through each person's individual device by picking one out of three answers from a multiple choice list. We'll tally the responses in real time and send an update to all peer devices as illustrated in Figure 7.1 - Bluetooth survey app, giving each user instantaneous feedback on the survey as it unfolds.

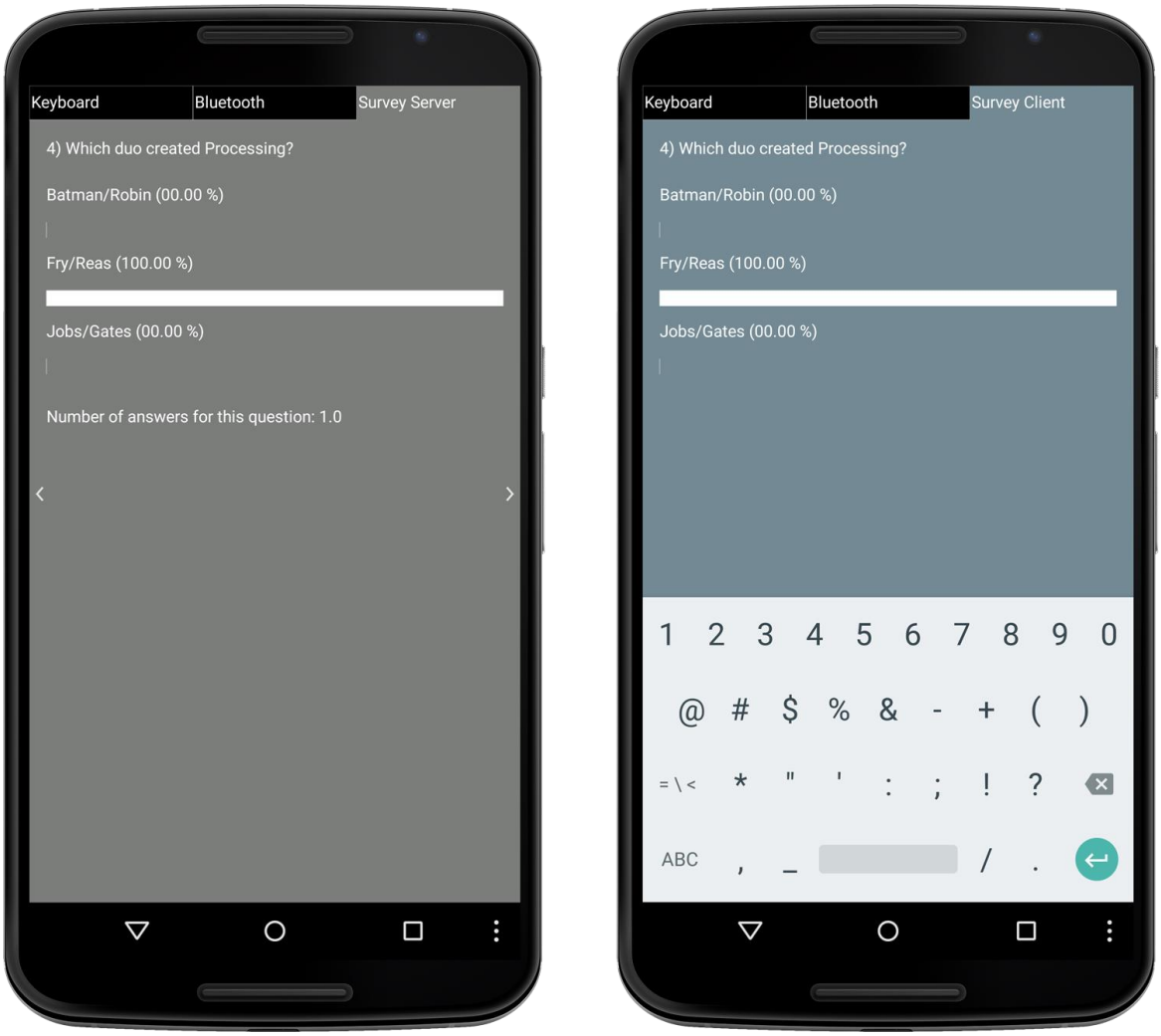


Figure 7.1 - Bluetooth survey app.

The illustration shows the Bluetooth server running on one device (left) and a client running on the other (right). The server determines which question is displayed on the client screens by pressing the arrow pointing to the right (next question) and left (previous question).

Both server and client receive real-time feedback on how the question was answered. There are many examples of survey and polling applications that are available online. They typically use proprietary online databases or a dedicated hardware infrastructure. Virtually no app exists using peer-to-peer connectivity on mobile devices—let's change that. Let's go ahead and write a survey app using some of the code we've already created for the remote cursor app in Connect Two Android Devices via Bluetooth.

For this survey app, we'll work with four tabs for this sketch: the main tab, which we'll name `BluetoothSurvey`, the `EnableBluetooth` tab, which is identical to the tab with the same name P2P/BluetoothCursors/EnableBluetooth.pde, a `Question` tab for a custom Processing class we'll write to take care of our Q & A, and a slightly modified version of the `UI` tab, which we developed P2P/BluetoothCursors/UI.pde.

Our approach is as follows. We'll write a sketch that we'll load onto every device participating in the survey. This way we can distribute the app without making adjustments for each individual device. The app needs to figure out whether it serves as the Bluetooth server for the other devices or connects as a client. As participants, we then send different messages to the other devices using `OSC` (Using the Open Sound Control Networking Format), for example, to request the current question, to submit an answer, or to get an update on the statistics. We'll give each `OSC` message a dedicated label, which we can then use to determine what to do when an `OSC` event occurs. When we receive an `OSC` message, we check its label using `checkAddrPattern()`, and depending on what pattern we detect, we can respond accordingly.

Program the BluetoothSurvey Main Tab

Let's take a look at our main tab, which contains the following methods: `setup()`, `draw()`, `onBluetoothDataEvent()`, `getBluetoothInformation()`, `loadQuestions()`, `requestQuestions()`, and `findQuestion()`.

code/P2P/BluetoothSurvey/BluetoothSurvey.pde

```
// Required Bluetooth methods on startup
import android.os.Bundle;
import android.content.Intent;

import ketai.net.bluetooth.*;
import ketai.ui.*;
import ketai.net.*;
import oscP5.*;

KetaiBluetooth bt;

KetaiList connectionList;
String info = "";
boolean isConfiguring = true;
String UIText;
color clientColor = color(112, 138, 144);
color serverColor = color(127);

boolean isServer = true;

ArrayList<Question> questions = new ArrayList<Question>();// 1
ArrayList<String> devicesDiscovered = new ArrayList();
Question currentQuestion;                                  // 2
int currentStatID = 0;
Button previous, next;

void setup()
{
  orientation(PORTRAIT);
  background(78, 93, 75);
  stroke(255);
  textSize(48);
  rectMode(CORNER);
  previous = new Button("previous.png", 30, height/2);
  next = new Button("next.png", width-30, height/2);
```

```
  bt.start();
  if (isServer)
    loadQuestions();

  UIText =  "[b] - make this device discoverable\n" +
    "[d] - discover devices\n" +
    "[c] - connect to device from list\n" +
    "[p] - list paired devices\n" +
    "[i] - show Bluetooth info";

  KetaiKeyboard.show(this);
}

void draw()
{
  if (isConfiguring)
  {
    ArrayList<String> devices;

    if (isServer)
      background(serverColor);  //green for server
    else
      background(clientColor); //grey for clients

    if (key == 'i')
      info = getBluetoothInformation();
    else
    {
      if (key == 'p')
      {
        info = "Paired Devices:\n";
        devices = bt.getPairedDeviceNames();
      }
      else
      {
        info = "Discovered Devices:\n";
        devices = bt.getDiscoveredDeviceNames();
      }

      for (int i=0; i < devices.size(); i++)
      {
        info += "["+i+"] "+devices.get(i).toString() + "\n";
      }
    }
    text(UIText + "\n\n" + info, 5, 200);
  }
  else
  {
    if (questions.size() < 1)
      requestQuestions();

    if (questions.size() > 0 && currentQuestion == null)
      currentQuestion = questions.get(0);                 // 3

    if (isServer)
      background(serverColor);
    else
      background(clientColor);
    pushStyle();
    fill(255);
    stroke(255);
    ellipse(mouseX, mouseY, 20, 20);
    if (currentQuestion != null)
      currentQuestion.display(50, 200);
    //      text(currentQuestion.toString(), 75, 75);     // 4
    popStyle();
  }
  drawUI();
  broadcastStats();
}
```

```
void onBluetoothDataEvent(String who, byte[] data)
{
  //   but allows construction by byte array
  KetaiOSCMessage m = new KetaiOSCMessage(data);
  if (m.isValid())
  {
    print(" addrpattern: "+m.addrPattern());

    //handle request for questions
    if (m.checkAddrPattern("/poll-request/"))             // 5
    {
      if (isServer)
      {
        int lastID =  m.get(0).intValue();

        for (int j = 0; j < questions.size(); j++)
        {
          Question q = questions.get(j);

          if (q.id <= lastID)
            continue;

          OscMessage msg = new OscMessage("/poll-question/");  // 6
          msg.add(q.id);
          msg.add(q.question);
          msg.add(q.answer1);
          msg.add(q.answer2);
          msg.add(q.answer3);
          bt.broadcast(msg.getBytes());
        }
      }
    }

    else if (m.checkAddrPattern("/poll-question/"))       // 7
    {
      if (isServer)
        return;

      //id, question, choice a, choice b, choice c
      if (m.checkTypetag("issss"))                        // 8
      {
        int _id = m.get(0).intValue();
        println("processing question id: " + _id);

        //we already have this question...skip
        if (findQuestion(_id) != null)
          return;

        Question q = new Question(                        // 9
        m.get(0).intValue(),
        m.get(1).stringValue(),
        m.get(2).stringValue(),
        m.get(3).stringValue(),
        m.get(4).stringValue());
        questions.add(q);
      }
    }
    else if (m.checkAddrPattern("/poll-answer/"))         // 10
    {
      if (!isServer)
        return;
      //question id + answer
      if (m.checkTypetag("ii"))                            // 11
      {
        Question _q = findQuestion(m.get(0).intValue());
        if (_q != null)
        {
          println("got answer from " + who + " for question " +
            m.get(0).intValue() + ", answer: " + m.get(1).intValue());
```

```
          _q.processAnswerStat(m.get(1).intValue());       // 12
          OscMessage msg = new OscMessage("/poll-update/");
          println("sending poll update for question " + _q.id + "-" +
            _q.total1 + "," + _q.total2 + "," + _q.total3);
          msg.add(_q.id);
          msg.add(_q.total1);
          msg.add(_q.total2);
          msg.add(_q.total3);
          bt.broadcast(msg.getBytes());
        }
      }
    }
    //update answer stats
    else if (m.checkAddrPattern("/poll-update/") && !isServer) // 13
    {
      //question id + 3 totals
      if (m.checkTypetag("iiii"))                          // 14
      {
        int _id = m.get(0).intValue();
        Question _q = findQuestion(_id);
        if (_q != null)
        {
          println("got poll update for question " +
            _id + " vals " + m.get(1).intValue() + ", " +
            m.get(2).intValue() + "," + m.get(3).intValue());

          _q.updateStats(m.get(1).intValue(),             // 15
          m.get(2).intValue(),
          m.get(3).intValue());
        }
      }
    }
    else if (m.checkAddrPattern("/poll-current-question/") && !isServer)
    {
      int targetQuestionId =  m.get(0).intValue();
      Question q = findQuestion(targetQuestionId);
      if (q != null)
        currentQuestion = q;
    }
  }
}

String getBluetoothInformation()
{
  String btInfo = "BT Server Running: ";
  btInfo += bt.isStarted() + "\n";
  btInfo += "Device Discoverable: "+bt.isDiscoverable() + "\n";
  btInfo += "Is Poll Server: " + isServer + "\n";
  btInfo += "Question(s) Loaded: " + questions.size();
  btInfo += "\nConnected Devices: \n";

  ArrayList<String> devices = bt.getConnectedDeviceNames();
  for (String device: devices)
  {
    btInfo+= device+"\n";
  }
  return btInfo;
}

void loadQuestions()
{
  String[] lines;

  lines = loadStrings("questions.txt");                   // 16

  if (lines != null)
  {
    for (int i = 0; i < lines.length; i++)
    {
      Question q = new Question(lines[i]);
```

```
      if (q.question.length() > 0)
      {
        q.id = i+1;
        questions.add(q);                                    // 17
      }
    }
  }
}

void requestQuestions()
{
  //throttle request
  if (frameCount%30 == 0)
  {
    int lastID = 0;

    if (questions.size() > 0)
    {
      Question _q = questions.get(questions.size()-1);
      lastID = _q.id;
    }

    OscMessage m = new OscMessage("/poll-request/");      // 18
    m.add(lastID);
    bt.broadcast(m.getBytes());
  }
}

Question findQuestion(int _id)                             // 19
{
  for (int i=0; i < questions.size(); i++)
  {
    Question q = questions.get(i);
    if (q.id == _id)
      return q;
  }
  return null;
}

void broadcastStats()
{
  if (!isServer)
    return;

  if (frameCount%60 == 0)
  {
    if ( currentStatID > 0 && currentStatID <= questions.size())
    {
      Question _q = findQuestion(currentStatID);
      if (_q != null)
      {
        println("sending poll update for question " + _q.id + "-" +
          _q.total1 + "," + _q.total2 + "," + _q.total3);
        OscMessage msg = new OscMessage("/poll-update/");  // 20
        msg.add(_q.id);
        msg.add(_q.total1);
        msg.add(_q.total2);
        msg.add(_q.total3);
        bt.broadcast(msg.getBytes());
        currentStatID++;
      }
    }
    else {
      if (questions.size() > 0)
        currentStatID = questions.get(0).id;
    }
    sendCurrentQuestionID();
  }
}
void sendCurrentQuestionID() {
```

```
  if (currentQuestion == null)
    return;
  OscMessage msg = new OscMessage("/poll-current-question/");
  msg.add(currentQuestion.id);
  bt.broadcast(msg.getBytes());
}
```

These are the main steps we need to take to implement the survey app.

---

1. Create an `ArrayList` called `questions`, which we'll use to store objects of the custom `Question` class we'll write to store questions and corresponding answers.
2. Store the current question in a variable, `currentQuestion`, which is presented on all devices simultaneously for the survey.
3. Set the `currentQuestion` to the first question in the `questions` `ArrayList`.
4. Present the `currentQuestion` on the screen as formatted output `String`, using our custom `toString()`method in the `Question` class.
5. Check if the `OSC` message we receive via `onBluetoothDataEvent()` contains the label "poll-request."
6. Create a new `OscMessage` `msg` with the label "poll-question" for each object in our `questionsArrayList`, add the question and answers stored in the `Question` object to the `OSC` message, and `broadcast()` the message to all connected users.
7. Check if we received an `OSC` message labeled "poll-question."
8. Check if the `OSC` message labeled "poll-question" contains an `integer` followed by four `String`values, if we are connected as a client.
9. Create a new `Question` object based on the received `integer` and `String` data.
10. Check if the received `OSC` message is labeled "poll-answer."
11. Check if the "poll-answer" message contains two `integer` values if we are the server app.
12. Find the corresponding question to the answer we've received and process the answer, adding to the tally of responses.

13. Check if the `OSC` message is labeled "poll-update."
14. Check if the `OSC` message labeled "poll-update" contains four `integer` values.
15. Update the poll statistic for the question we've received via OSC by updating the total number of responses for each answer.

16. Load the questions from the `questions.txt` flat file located in the `data` folder of the sketch using `loadStrings()` if we operate as the server. `loadStrings()` loads each line of the text file as an item in the `String` array `lines`. If we are connected as a client, we receive the questions peer-to-peer from the server.
17. `add()` the `Question` object to the `questions` `ArrayList`.
18. Request question if we don't have any in our `questions` `ArrayList`. Retry every thirtieth frame, or once a second, at a default `frameRate` of `30` fps.

19. Find the corresponding question to the received answer ID in the `questions` `ArrayList` method.
20. When the custom method `broadcastStats()` is called and the device is serving as the Bluetooth server, send an update with all totals to all connected client devices using an `OSC` message labeled “poll-update.”

Program the Question Tab

Let’s now work on our questions and answers. We’ll build a custom Processing class called `Question` that can take care of multiple questions and the corresponding answers for us. To make the number of questions we can work with flexible, we’ve already set up an `ArrayList` for the `questions` variable. Each question comes with three possible answers, which belong to a specific question; that’s why it’s best we work with a custom class.
A class is a composite of data stored in variables and methods. If we use a custom class, we can keep the Q & A data together in one object that consists of the actual question, three answers, three totals, and the answer given by the individual participant. We can also write a couple of methods that help us keep track of the totals for each answer and return the statistic back to us for a text output.

Now let’s take a look at the Question class, which gets its own tab in the sketch.

code/P2P/BluetoothSurvey/Question.pde

```
// Question Class

class Question                                          // 1
  int id=0;
  String question="";
  String answer1, answer2, answer3="";
  int total1, total2, total3 = 0;
  int myAnswer, correctAnswer;

  Question(String _row)                                 // 2
  {
    String[] parts = split(_row, '\t');                 // 3
    if (parts.length == 4)
    {
      question = parts[0];
      answer1 = parts[1];
      answer2 = parts[2];
      answer3 = parts[3];
    }
  }

  Question(int _id, String q, String a1, String a2, String a3)  // 4
  {
    id = _id;
    question = q;
    answer1 = a1;
    answer2 = a2;
    answer3 = a3;
  }
```

```
void updateStats(int s1, int s2, int s3)                // 5
{
  total1 = s1;
  total2 = s2;
  total3 = s3;
}
void processAnswerStat(int _answer)                     // 6
{
  if (_answer == 1)
    total1++;
  else if (_answer == 2)
    total2++;
  else if (_answer == 3)
    total3++;
}

void setAnswer(int _answer)
{
  myAnswer = _answer;
  processAnswerStat(_answer);
}

float getAnswerStat(int _answer)                        // 7
{
  if (_answer == 1)
    return total1;
  else if (_answer == 2)
    return total2;
  else if (_answer == 3)
    return total3;
  return 0;
}

void saveResults()                                      // 8
{
  String line = question + "\t" +
    answer1 + "\t" + total1 + "\t";
  line += answer2 + "\t" + total2 + "\t" +
    answer3 + "\t" + total3;
}

boolean isAnswered()
{
  if (myAnswer == 0)
    return false;
  return true;
}

void display(int x, int y)                              // 9
{
  pushStyle();
  pushMatrix();
  translate(x, y);
  if (myAnswer == 0  && !isServer)
  {
    text(id+") " + question + "\n\n" +
      "[1] " + answer1 + "\n" +
      "[2] " + answer2 + "\n" +
      "[3] " + answer3, 0, 0);
  }
  else
  {

    float total = total1+total2+total3;

    //avoid div by 0
    if (total == 0)
      total = 1;
```

```
      float lineheight = textAscent()+textDescent();
      lineheight = 50;
      text( id+") " + question, 0, 0);

      textAlign(LEFT, TOP);
      translate(0, lineheight*2);

      text(answer1 + " (" + nf((total1/total)*100, 2, 2) + " %) ", 0, 0);
      translate(0, lineheight*2);
      rect(0, 0, map((total1/total)*200, 0, 200, 0, width-100), lineheight-5);
      translate(0, lineheight*2);

      text(answer2 + " (" +nf( (total2/total)*100, 2, 2 ) + " %) ", 0, 0);
      translate(0, lineheight*2);
      rect(0, 0, map((total2/total)*200, 0, 200, 0, width-100), lineheight-5);
      translate(0, lineheight*2);

      text(answer3 + " (" +nf( (total3/total)*100, 2, 2) + " %) ", 0, 0);
      translate(0, lineheight*2);
      rect(0, 0, map((total3/total)*200, 0, 200, 0, width-100), lineheight-5);
      translate(0, lineheight*3);

      if (isServer)
        text("Number of answers for this question: " + total, 0, 0);
    }
    popMatrix();
    popStyle();
  }
}
```

Let's take a closer look at the `Question` class variables and methods.

1. Create a custom Processing `class` called `Question`. The class definition does not take parameters.
2. Create the constructor for the `Question` class, taking one `String` parameter `_row`.
3. Split the `String`, which contains one row of `questions.txt`. Use the tab character, `\t`, as a delimiter to `split()` the string.
4. Add a second constructor for `Question`, taking five parameters instead of a `String` like the first constructor does. Custom classes can be overloaded with multiple constructors to accommodate multiple types of parameters, here an `integer` for the ID, followed by a `String` for the question, followed by three `String` values for the answers.
5. Update the totals for each answer when we receive an answer from a participant.
6. Process the statistic for each answer (how many times each answer has been picked compared to the total number of answers).
7. Get the statistic for each answer.
8. Save each answer and its statistic.
9. Return the `String` output presented on the screen. If no answer has been given yet on the device, show the question and the multiple choice answers; otherwise, show the statistics as well.

   Now it's time modify the `UI` tab.

Program the UI Tab

We need to make few adjustments to the `UI` tab to modify it for our survey app based on the previous code P2P/BluetoothCursors/UI.pde . Most of it is redundant and otherwise called out.

code/P2P/BluetoothSurvey/UI.pde

```
// UI methods
void mousePressed() {
  if (mouseY <= 100 && mouseX > 0 && mouseX < width/3)
    KetaiKeyboard.toggle(this);
  else if (mouseY <= 100 && mouseX > width/3 && mouseX < 2*(width/3))
    isConfiguring=true;
  else if (mouseY <= 100 && mouseX >  2*(width/3) && mouseX < width &&
    bt.getConnectedDeviceNames().size() > 0)
  {
    if (isConfiguring) {
      background(127);
      isConfiguring=false;
    }
  }

  if (bt.getConnectedDeviceNames().size() > 0)  {
    if (currentQuestion == null)
      return;
    if (previous.isPressed() && isServer) //previous question
    {
      if (findQuestion(currentQuestion.id-1) != null)
        currentQuestion = findQuestion(currentQuestion.id-1); // 1
      sendCurrentQuestionID();
    }
    else if (next.isPressed()  && isServer) //next question
    {
      if (findQuestion(currentQuestion.id+1) != null)
        currentQuestion = findQuestion(currentQuestion.id+1); // 2
      else
        requestQuestions();
      sendCurrentQuestionID();
    }
  }
}

void keyPressed() {
  if (!isConfiguring && !isServer) {
    if (currentQuestion != null && !currentQuestion.isAnswered())
      if (key == '1') {
        currentQuestion.setAnswer(1);
        OscMessage m = new OscMessage("/poll-answer/");
        m.add(currentQuestion.id);
        m.add(1);                                          // 3
        bt.broadcast(m.getBytes());
      }
      else if (key == '2') {
        currentQuestion.setAnswer(2);
        OscMessage m = new OscMessage("/poll-answer/");
        m.add(currentQuestion.id);
        m.add(2);                                           // 4
        bt.broadcast(m.getBytes());
      }
      else if (key == '3') {
        currentQuestion.setAnswer(3);
        OscMessage m = new OscMessage("/poll-answer/");
        m.add(currentQuestion.id);
          m.add(3);                                         // 5
        bt.broadcast(m.getBytes());
```

```
      }
  }
  else if (key =='c') {
    if (bt.getDiscoveredDeviceNames().size() > 0)
      connectionList = new KetaiList(this, bt.getDiscoveredDeviceNames());
    else if (bt.getPairedDeviceNames().size() > 0)
      connectionList = new KetaiList(this, bt.getPairedDeviceNames());
  }
  else if (key == 'd')
    bt.discoverDevices();
  else if (key == 'm')
    bt.makeDiscoverable();
}

void drawUI() {
  pushStyle();
  fill(0);
  stroke(255);
  rect(0, 0, width/3, 100);

  if (isConfiguring)
  {
    noStroke();
    fill(127);
  }
  else if (bt.getConnectedDeviceNames().size() > 0 && isServer)
  {
    previous.draw();
    next.draw();
  }
  rect(width/3, 0, width/3, 100);

  if (!isConfiguring)
  {
    noStroke();
    if (isServer)
      fill(serverColor);
    else
      fill(clientColor);
  }
  else
  {
    fill(0);
    stroke(255);
  }
  rect((width/3)*2, 0, width/3, 100);

  fill(255);
  text("Keyboard", 5, 65);
  text("Bluetooth", width/3+5, 65);

  if (bt.getConnectedDeviceNames().size() > 0)
    if (isServer)
      text("Survey Server", width/3*2+5, 65);
    else
      text("Survey Client", width/3*2+5, 65);

  popStyle();
}

void onKetaiListSelection(KetaiList connectionList)
{
  String selection = connectionList.getSelection();
  bt.connectToDeviceByName(selection);
  connectionList = null;
  isServer = false;
}

class Button
{
```

```
  PImage icon;
  PVector location;
  int Size = 100;

  public Button(String imagefile, int x, int y)
  {
    icon = loadImage(imagefile);
    location = new PVector(x, y);
  }

  void draw()
  {
    pushStyle();
    imageMode(CENTER);
    if (icon != null)
      image(icon, location.x, location.y);
    else
      ellipse(location.x, location.y, Size, Size);
    popStyle();
  }

  boolean isPressed()
  {
    return (dist(location.x, location.y, mouseX, mouseY) < Size/2);
  }
}
```

Here are the changes we've made to the `UI` tab.

1. Jump to the previous question by decrementing the ID for `currentQuestion`.
2. Jump to the next question by incrementing the ID for `currentQuestion`.
3. Send an `OSC` message called "poll-answer" if we press `1` on the keyboard. The message contains the current question ID followed by the answer `1`.
4. Send an `OSC` message called "poll-answer" if we press `2` that contains the current question ID followed by the answer `2`.
5. Send an `OSC` message called "poll-answer" if we press `3` that contains the current question ID followed by the answer `3`.

Now it's time to test the app.

Run the App

Run the app on the Android device you've currently connected via USB. When the app is compiled, disconnect that device and run it on the other device. If you have a third (or fourth) device available, load the sketch onto as many Android devices as you'd like to test with.

Now follow the steps we took earlier in Run the App, to connect two devices via Bluetooth for the remote cursor app.

Finally, press the Survey tab and answer your first question. Press the left and right arrows to move through the questions. Respond to the questions using the software keyboard and notice how the statistics change as you punch in `1`, `2`, and `3`.

You've completed a survey app for multiple Bluetooth devices, where we've been diving deep into the peer-to-peer networking process. Although very ubiquitous, Bluetooth has its limitations in terms of transmission bandwidth and range.

Less ubiquitous than Bluetooth but more powerful in terms of bandwidth and range, is the emerging Wi-Fi Direct P2P networking standard. Let's take a look at the final networking standard we'll discuss in this chapter, which is also fairly easy to work with.

Working with Wi-Fi Direct

Wi-Fi Direct was introduced in Android 4.0 (API level 14) to enable Android devices to connect directly to each other via Wi-Fi without a fixed Wi-Fi access point. Each device in a Wi-Fi Direct network can serve as an access point for any of the other devices in the network. Like Bluetooth, Wi-Fi Direct allows us to discover Wi-Fi Direct devices and connect to them if confirmed by the user. Compared to Bluetooth, Wi-Fi Direct offers a faster connection across greater distances and is therefore the preferred networking protocol for multiplayer games or multiuser applications, when every connected device supports Wi-Fi Direct.

In many ways, Wi-Fi Direct is very familiar to us when it comes to allowing devices to connect to each other. We've also worked with Wi-Fi already and sent `OSC` messages over the wireless local area network in Chapter 6, Networking Devices with Wi-Fi. When we use Wi-Fi Direct, we combine the P2P aspects and pairing process of Bluetooth with the ease of use of Wi-Fi.
Wi-Fi Direct is currently supported on a few of the newest Android devices, so the following section may describe operations that are not possible on your device just yet. But because it's a powerful standard, we'll discuss it now and compare it to Bluetooth's wireless peer-to-peer performance using our earlier remote cursor sketch P2P/BluetoothCursors/BluetoothCursors.pde.

Let's take a look at the `KetaiWi-FiDirect` class first, which makes working with Android's Wi-Fi Direct features an easy task. For this sketch, we'll work with the following `KetaiWi-FiDirect` methods:

`connectToDevice()`	Connects to a specific Wi-Fi Direct-enabled device
`getConnectionInfo()`	Gets the status of the Wi-Fi Direct connection
`getIPAddress()`	Gets the IP address of a specified `Wi-Fi` Direct device
`getPeerNameList()`	Returns the list of connected `Wi-Fi` Direct devices

Now, let's go ahead and implement the remote cursor app using Wi-Fi Direct.

Use Wi-Fi Direct to Control Remote Cursors

We've already implemented the remote cursors app earlier in this chapter in Connect Two Android Devices via Bluetooth. In order to compare Wi-Fi Direct to Bluetooth, we'll implement the same remote cursor app, replacing its Bluetooth peer-to-peer networking functionality with Wi-Fi Direct. Large portions of the code are identical to the Bluetooth version of the sketch shown in P2P/BluetoothCursors/BluetoothCursors.pde and P2P/BluetoothCursors/UI.pde, so we will focus only on the code that we'll change from Bluetooth to Wi-Fi Direct.

Using Wi-Fi Direct, we'll be able to use the `OSC` protocol again to send data to remote devices, as we've already done in Network an Android with a Desktop PC.

Modify the Main Tab

code/P2P/WiFiDirectCursors/WiFiDirectCursors.pde

```
import ketai.net.wifidirect.*;                          // 1
import ketai.net.*;
import ketai.ui.*;
import oscP5.*;
import netP5.*;

KetaiWiFiDirect direct;                                 // 2
KetaiList connectionList;
String info = "";
PVector remoteCursor = new PVector();
boolean isConfiguring = true;
String UIText;

ArrayList<String> devices = new ArrayList();             // 3
OscP5 oscP5;                                             // 4
String clientIP = "";                                    // 5

void setup()
{
  orientation(PORTRAIT);
  background(78, 93, 75);
  stroke(255);
  textSize(48);

  direct = new KetaiWiFiDirect(this);                    // 6

  UIText =  "[d] - discover devices\n" +
    "[c] - pick device to connect to\n" +
    "[p] - list connected devices\n" +
    "[i] - show WiFi Direct info\n" +
    "[o] - start OSC Server\n";                          // 7
}

void draw()
{
  background(78, 93, 75);
```

```
  if (isConfiguring) {
    info="";
    if (key == 'i')
      info = getNetInformation();                          // 8
    else if (key == 'd') {
      info = "Discovered Devices:\n";
      devices = direct.getPeerNameList();                  // 9
      for (int i=0; i < devices.size(); i++)
      {
        info += "["+i+"] "+devices.get(i).toString() + "\t\t"+devices.size()+"\n";
      }
    }
    else if (key == 'p')  {
      info += "Peers: \n";
    }
    text(UIText + "\n\n" + info, 5, 200);
  }
  else {
    pushStyle();
    noStroke();
    fill(255);
    ellipse(mouseX, mouseY, 50, 50);
    fill(255, 0, 0);
    ellipse(remoteCursor.x, remoteCursor.y, 50, 50);
    popStyle();
  }
  drawUI();
}

void mouseDragged() {
  if (isConfiguring)
    return;

  OscMessage m = new OscMessage("/remoteCursor/");
  m.add(pmouseX);
  m.add(pmouseY);

  if (oscP5 != null) {
    NetAddress myRemoteLocation = null;

    if (clientIP != "")
      myRemoteLocation = new NetAddress(clientIP, 12000);
    else if (direct.getIPAddress() != KetaiNet.getIP())
      myRemoteLocation = new NetAddress(direct.getIPAddress(), 12000);

    if (myRemoteLocation != null)
      oscP5.send(m, myRemoteLocation);
  }
}

void oscEvent(OscMessage m) {
  if (direct.getIPAddress() != m.netAddress().address())  // 10
    clientIP = m.netAddress().address();                  // 11
  if (m.checkAddrPattern("/remoteCursor/")) {
    if (m.checkTypetag("ii")) {
      remoteCursor.x = m.get(0).intValue();
      remoteCursor.y = m.get(1).intValue();
    }
  }
}

String getNetInformation()                                // 12
{
  String Info = "Server Running: ";
  Info += "\n my IP: " + KetaiNet.getIP();
  Info += "\n initiator's IP:  " + direct.getIPAddress();
  return Info;
}
```

Let’s take a look at the steps we took to change our remote cursor app to use Wi-Fi Direct.

1. Import the Ketai library's Wi-Fi Direct package.
2. Create a variable `direct` of the type `KetaiWiFiDirect`.
3. Create an `ArrayList` for discovered Wi-Fi Direct devices.
4. Create a `OscP5`-type variable, `oscP5`, as we've used already in Chapter 6, Networking Devices with Wi-Fi.
5. Create a `String` variable to hold the client IP address.
6. Create the `WiFiDirect` object, `direct`.
7. Include a keyboard menu item `o` to start the `OSC` server
8. Get the Wi-Fi Direct network information.
9. Get the list of connected Wi-Fi Direct peer devices.
10. Check if the Wi-Fi Direct server IP address is different from our device IP, which means we are connecting as a client.
11. Set the `clientIP` variable to the device IP address, since we've determined we are connecting as a client to the Wi-Fi Direct network.
12. Get the Wi-Fi Direct information, including our IP address and the server's IP address.

For the UI, we won't change very much compared to the previous `UI` P2P/BluetoothCursors/UI.pde. Let's take a look.

Modify the UI Tab

Now it's time to see what's needed to modify the `UI` tab to support Wi-Fi direct. We'll need to adjust the discover key (`d`) to call the Wi-Fi Direct `discover()` method and the info key (`i`) to get the Wi-Fi Direct connection info using `getConnectionInfo()`. Also, we need to introduce an `OSC` key (`o`) to the menu, allowing us to start `OSC` networking now over Wi-Fi Direct.

code/P2P/WiFiDirectCursors/UI.pde

```
// UI methods
void mousePressed() {
  //keyboard button -- toggle virtual keyboard
  if (mouseY <= 100 && mouseX > 0 && mouseX < width/3)
    KetaiKeyboard.toggle(this);
  else if (mouseY <= 100 && mouseX > width/3 && mouseX < 2*(width/3)) //config
  {
    isConfiguring=true;
  }
  else if (mouseY <= 100 && mouseX >  2*(width/3) && mouseX < width) { // draw
    if (isConfiguring) {
      isConfiguring=false;
    }
  }
```

```
}
void keyPressed() {
  if (key == 'c') {
    if (devices.size() > 0)
      connectionList = new KetaiList(this, devices);
  }

  else if (key == 'd') {
    direct.discover();                                    // 1
    println("device list contains "  + devices.size() + " elements");
  }
  else if (key == 'i')
    direct.getConnectionInfo();                           // 2
  else if (key == 'o') {
    if (direct.getIPAddress().length() > 0)
      oscP5 = new OscP5(this, 12000);                     // 3
  }
}

void drawUI()
{
  pushStyle();
  fill(0);
  stroke(255);
  rect(0, 0, width/3, 100);

  if (isConfiguring)
  {
    noStroke();
    fill(78, 93, 75);
  }
  else
    fill(0);

  rect(width/3, 0, width/3, 100);

  if (!isConfiguring)
  {
    noStroke();
    fill(78, 93, 75);
  }
  else
  {
    fill(0);
    stroke(255);
  }
  rect((width/3)*2, 0, width/3, 100);

  fill(255);
  text("Keyboard", 5, 65);
  text("WiFi Direct", width/3+5, 65);
  text("Interact", width/3*2+5, 65);

  popStyle();
}

void onKetaiListSelection(KetaiList klist)
{
  String selection = klist.getSelection();
  direct.connect(selection);                              // 4
  connectionList = null;
}
```

Now let's see what we adjusted for the `UI` tab of the Wi-Fi Direct remote cursor sketch.

1. Discover Wi-Fi Direct devices if we press `d` on the keyboard.
2. Get the Wi-Fi Direct information.

3. Initialize the `OSC` connection on port `12000`.
   Everything looks quite familiar from the Bluetooth version of this sketch. The difference is that we are connecting and sending `OSC` messages like we've done in Chapter 6, Networking Devices with Wi-Fi.
   Now let's test the app.

Run the App

Run the app on your Wi-Fi Direct-enabled Android device. Disconnect the USB cable, and run the app on your second Wi-Fi Direct device. Press `d` on the keyboard to discover Wi-Fi Direct devices. Then press `c`to show the list of discovered devices and pick your second device. You need to allow the Wi-Fi Direct connection request. Once you do, press the Interact tab on both devices and move your finger across the screen. Notice how quickly both cursors respond; there seems to be no noticeable delay, and the motion is continuous at a high frame rate. Compare the performance of the Wi-Fi Direct remote cursor app to the Bluetooth remote cursor app we've installed earlier. You can observe that Wi-Fi Direct performs better. Now grab a friend and put the connection range to the test for both flavors of the remote cursor sketch and see which one has the better range.

This concludes our explorations in the world of peer-to-peer networking. Now you are independent of networking infrastructure and ready to program your multiuser and multiplayer apps for users within close proximity.

Wrapping Up

We've learned that we don't need Wi-Fi/3G/4G infrastructure to interact with other Android users. We can write a range of apps that use peer-to-peer principles. We've learned about Bluetooth discovery, pairing, connecting, and exchanging data. We've learned that Wi-Fi Direct uses a similar discovery process as Bluetooth but it provides more bandwidth and greater connection distances.

With a range of networking projects under our belt, it's time now to move on to another emerging standard, near field communication, or NFC, which allows us not only to interact with other NFC-enabled Android devices but also with NFC tags embedded in stickers, posters, objects, or point-of-sale payment systems.

8. Using Near Field Communication (NFC)

Now that we've learned how to create peer-to-peer networks using Bluetooth and Wi-Fi Direct, it's time for us to dive into a more user-friendly method for connecting Android devices and sharing data. Near field communication (`NFC`) is an emerging short-range wireless technology designed for zero-click transactions and small data payloads. In a zero-click transaction between Android devices, you simply touch them back-to-back—that's it. For instance, we can invite a friend to join a multiplayer game or exchange contact information simply by touching our devices.

+

Using `NFC`, we can also exchange images, apps, and other data between devices without first pairing them—a feature that Google calls Android Beam. Beam is Android's trademark for `NFC` when the protocol is used for device-to-device communication. It was introduced with the release of Ice Cream Sandwich and is now a standard feature with all new Android devices. Google began promoting Beam with the release of Jelly Bean. When two unlocked Android devices facing back-to-back are brought near each other, Beam pauses the app that is currently running in the foreground and waits for us to confirm the `NFC` connection by tapping the screen. We'll use Android Beam's `NFC` features for the peer-to-peer networking app we'll write in this chapter.

With a maximum range of about four centimeters—slightly less than the length of a AA battery—NFC's reach is limited when compared to that of Bluetooth or Wi-Fi Direct, providing a first level of security. One shortcoming of the technology is that it's fairly slow and not designed for large data payloads. We can use it, however, to initiate a higher-bandwidth connection, for instance via Bluetooth, which we'll do later in this chapter.

Because `NFC` is quick and user-friendly, it promises to revolutionize the point-of-sale (`POS`) industry, which has been monetized by services such as Google Wallet and other merchandise services promoting `NFC`. For example, we can pay for a purchase at a MasterCard PayPass or Visa payWave terminal or read an `NFC` tag embedded in a product. Most Android smart phones shipped in the US today, and some tablets like the Nexus 7, come with `NFC` built in.

In addition to simplifying device interaction, `NFC` also promises to bridge the worlds of bits and atoms seamlessly, enabling us to interact with physical objects that have `NFC` chips embedded in them, also known as tags. Tags are `RFID` chips that contain `NFC`-formatted data. They can be mass produced for less than a dollar each and can store a small amount of data,

often in the form of a URL that points to a website with more information. Most tags are read-only, some can be written to, and some can even be programmed.
In this chapter, we'll first use `NFC` to initiate a P2P connection between two Android devices by simply touching them back-to-back. Then we'll use that connection to send an increasingly detailed camera preview from one device to another, introducing us also to recursion as a programming concept. At the end of this chapter we'll be writing apps to read and write `NFC` tags. If you don't have an `NFC` tag at hand, you can get them at your local (online) store.
Let's take a closer look at the `NFC` standard first.

Introducing NFC

Near field communication can be used to exchange data between any pair of `NFC` equipped devices. When we develop peer-to-peer apps for mobile devices, zero-click interaction via `NFC` is the key to effortless networking via Bluetooth or Wi-Fi Direct. Support for `NFC` was first added to Android in 2011 with the release of Ice Cream Sandwich, and although the technology is not yet ubiquitous, most Android phones and some tablets sold in the US today ship with it.
With `NFC`, we don't need to make a device discoverable, compare passcodes, or explicitly pair it with another because we establish the `NFC` network simply by touching one device to another. Only the deliberate back-to-back touch of two devices will trigger an `NFC` event. Simply walking by a payment reader or getting close to another device on a crowded bus is not sufficient. Front-to-front contact between devices won't do the trick either. Only when both devices are within four centimeters of each other will a connection result.
`NFC` is high frequency (`HF`) `RFID` technology operating at 13.56 MHz. It's a subset of `RFID`, but near field communication overcomes many of the security and privacy concerns that businesses and individuals have expressed about the use of `RFID`, which was reflected in the `RFID` mandates implemented in 2005 by Walmart and the United States Department of Defense for their global supply chains. `RFID` does not have a range limitation, but `NFC` does.
`NFC` tags are passive devices and don't require batteries. They get their power through induction from a powered `NFC` device such as an Android phone or tablet. Induction occurs when a conductor passes through a magnetic field, generating electric current. If an `NFC` tag (conductor) comes close to the magnetic coil embedded in a powered `NFC` device (magnetic field), an electric current is induced in the tag, reflecting radio waves back to the `NFC` device (data). Because the tags lack power and only reflect the radio waves emitted by an `NFC` device, they are also referred to as passive tags.
`NFC` tags come in different shapes and sizes and are roughly between the size of a quarter and a business card, as shown here:

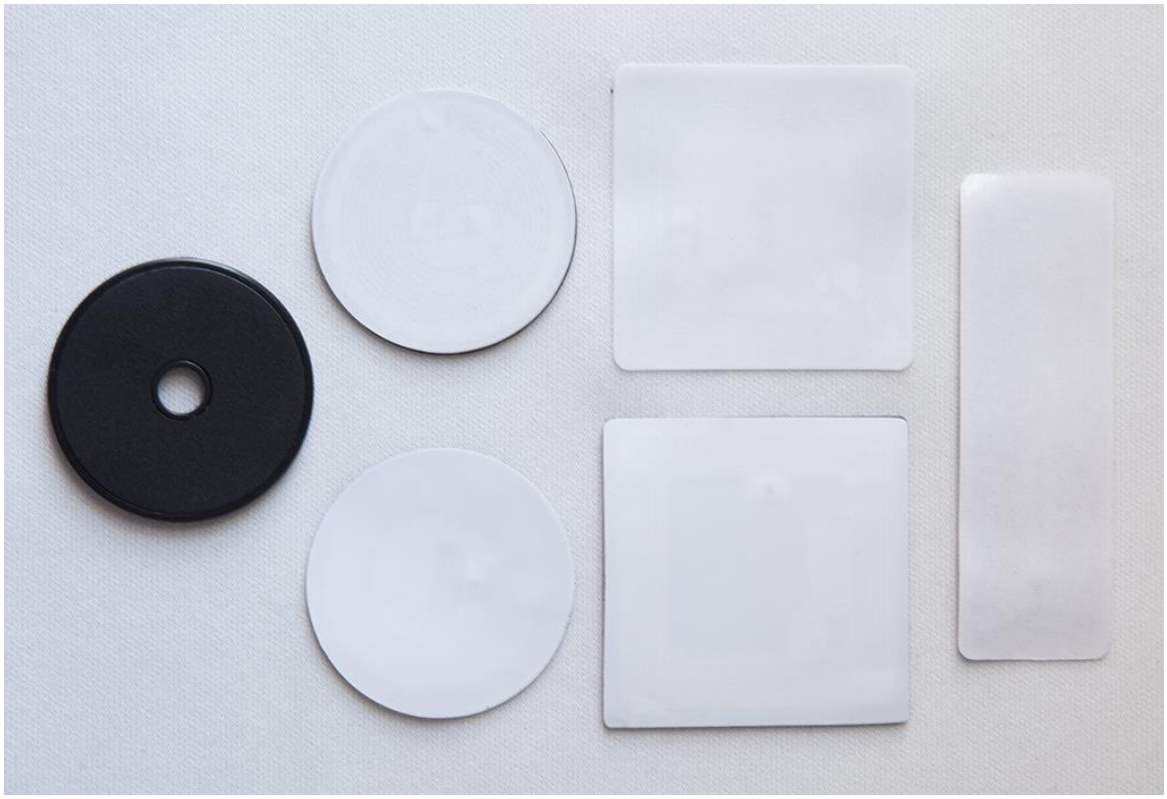


Figure 8.0 - NFC tags.

Different types of tags with different form factors: (from left to right) a heat resistant token, round and square stickers, and a machine washable tag `NFC` tags carry small amounts of data, typically between 40 bytes or characters and 1 kilobyte. Because of this limited amount of usable memory, tags are commonly used to store a URL that points to more information online.

Android devices respond to `HF` `RFID` tags because they are using the same radio frequency that `NFC`uses. For example, when you touch your biometric passport, the `RFID` tag inside the book from your library, the tag from your last marathon attached to your shoe, or a tagged high-end product or piece of clothing, your Android signals with a beep that it has encountered a tag. This doesn't mean that we can successfully decipher such an `RFID` tag. The information stored on your ID card, passport, PayPass, or payWave card is encrypted. What's more, `RFID` tags embedded in products typically store inventory IDs that don't comply with the `NFC` specification, which means we can't decipher the tag.

However, it is possible for our `NFC`-enabled Android phone or tablet to emulate an `HF` `RFID` tag, which is the reason we can use the Android to pay at a PayPass or payWave terminal. Because the PayPass and payWave tag content is proprietary and encrypted, we also can't simply write a point-of-sale app of our own—like Google Wallet and bank-issued apps, we need to comply with the proprietary encryption to get a response. An increasing number of transit authorities have started to use `RFID` and `NFC` as well and are collecting fares using this wireless technology.

The first level of security and privacy protection for near field communication is proximity. Due to `NFC`'s limited range, we don't need to make our device discoverable or browse for nearby Bluetooth devices. When we use `NFC` to exchange data directly between two Android

devices, the second level of security is that the apps running on both devices must be running in the foreground—otherwise a dialog pops up to confirm. Also, when we use Beam, we need to tap the screen once while the devices are near each other to confirm the exchange. The data we exchange via `NFC` is not encrypted by default on the Android. If your goal is to write an app that requires security, like your own point-of-sale app, you need to deal with encryption separately.

We'll use the `KetaiNFC` class for the apps in this chapter, making it simpler for us to work with Android's `NFC` package. Let's take a look at the `KetaiNFC` class next.

Working with the KetaiNFC Class and NDEF Messages

`KetaiNFC` allows us to easily access the NFC methods of Android's `nfc` package from within a Processing sketch. It allows us to receive `NFC` events and read and write tags in the NFC Data Exchange Format (NDEF), the official specification defined by the `NFC` Forum. Android implements this data format with the `Ndef` class, which provides methods for us to read and write `NdefMessage` data on `NFC` tags. An `NdefMessage` can contain binary data (`NdefMessage(byte[])`) or `NdefRecord` objects (`NdefMessage(NdefRecord[]`).

An `NdefRecord` object contains either MIME-type media, a `URI`, or a custom application payload. `NdefMessage` is the container for one or more `NdefRecord` objects.

We'll use the `KetaiNFC` class to write `NDEF` data. Depending on the data type provided to `write()`, it will create the corresponding `NdefMessage` for us accordingly. For the projects in this chapter, we'll use the following `NFC` methods:

- `write()` A Ketai library method to write a text `String`, `URI`, or `byte[]` array—depending on the datatype provided to `write()` as a parameter, it sends an `NFC` message formatted in that datatype.
- `onNFCEvent()` An event method returning `NDEF` data of different types, such as a text `String` or `byte`array—depending on the datatype returned, `onNFCEvent()` can be used to respond differently depending on the `NDEF` message returned.

Let's get started by creating a peer-to-peer connection using `NFC`.

Share a Camera Preview Using NFC and Bluetooth

The idea of this project is to allow two or more individuals to quickly join a peer-to-peer network using `NFC`and Bluetooth. `NFC` is a good choice because it reduces the number of steps users must complete to create a network. We'll take advantage of the user-friendly `NFC` method

to pair devices and rely on the `NFC`-initiated higher-bandwidth Bluetooth network to handle the heavy lifting.

Our sketch will use a recursive program to send an increasingly accurate live camera image from one Android device to another. Once we've paired the two devices via `NFC`, we'll begin with a camera preview that consists of only one large "pixel," which we'll draw as a rectangle in our program. Each time we tap the screen on the remote device, we will increase the resolution of the transmitted camera preview by splitting each pixel of the current image into four elements, as illustrated below. In the next level, each of those pixels is split again into four, and so on—exponentially increasing the preview resolution until the image becomes recognizable. The color is taken from the corresponding pixel of the camera preview pixel located exactly in the area's center.

Figure 8.1 - Broadcast pixels using NFC and Bluetooth.

Touching NFC devices back-to-back initiates the Bluetooth connection, starting a two-directional pixel broadcast. The camera preview is then sent from one device to the other and displayed there. The top image shows the sampled camera image after four taps, the bottom image after two.

When we run the app on the networked Androids, we will get a sense of how much data we can send via Bluetooth and at what frame rate. We'll revisit concepts from previous chapters where we worked with a live camera preview, Chapter 5, Using Android Cameras, and sent Bluetooth messages, Chapter 7, Peer-to-Peer Networking Using Bluetooth and Wi-Fi Direct, now using `NFC` to initiate the network.

Generate a Low-Resolution Preview

Let's go ahead and work on the main tab of our sketch, where we'll put our camera code, and write a function to generate images of the camera preview at higher and higher resolutions. The program works by repeatedly calling itself, a technique known to programmers as recursion. This technique allows us to iterate through the image until we reach a number of divisions that we'll set beforehand with a variable we'll name `divisions`. Setting a limit is important since the recursion would otherwise continue forever, eventually "freezing" the app.
Let's name the recursive function `interlace()`. Each time it runs when we tap the screen, it will split each pixel in the current image into four new pixels.
The `interlace()` method we'll create works with the `divisions` parameter to control how many recursions will be executed. We'll start with a `divisions` value of `1`, for one division. Each time we tap the screen, `divisions` will increase to `2`, `3`, and so on, which will also increase the `level` parameter in our `interlace()` method. There we are using `level` to check that it has a value greater than `1` before recursively calling the `interlace()` method again to split each pixel into four.
In the main tab we also import the Ketai camera package, which is familiar to us from Chapter 5, Using Android Cameras. We'll create a `KetaiCamera` object that we'll name `cam`.
The `cam` object will read the image each time we receive a new frame from the camera.
For this sketch, we'll use the following tabs to organize our code:

- `NFCBTTransmit` Contains the main sketch, including our `setup()` and `draw()` methods, along with the `interlace()` method for recursively splitting the camera preview image. It also contains a `mousePressed()` method to increase the global variable `divisions`, used as a parameter for `interlace()`, and a `keyPressed` method that allows us to toggle the local camera preview on and off.
- `ActivityLifecycle` Contains all the methods we need to start `NFC` and Bluetooth correctly within the activity life cycle. We require a call to `onCreate()` for launching Bluetooth, `onNewIntent()` to enable `NFC`, and `onResume()` to start both `NFC` and Bluetooth.

- `Bluetooth` A tab for the two Bluetooth methods, `send()` and `onBluetoothDataEvent()`, to send Bluetooth messages and receive others in return.
- `NFC` The tab that contains the `setupNFC()` method to create the `NFC` object we are working with and the `onNFCEvent()` method that launches the Bluetooth connection when we received the other device's Bluetooth `ID` via `NFC`.

  We'll create each of those tabs step by step and present the code for each component separately in the following sections.

  Let's first take a look at our main tab.

code/NFC/NFCBTTransmit/NFCBTTransmit.pde

```
// Required methods to enable NFC and Bluetooth
import android.content.Intent;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;

import ketai.net.*;
import oscP5.*;
import netP5.*;

import ketai.camera.*;
import ketai.net.bluetooth.*;
import ketai.net.nfc.*;

KetaiCamera cam;

int divisions = 1;                                      // 1
String tag = "";

void setup()
{
  orientation(LANDSCAPE);
  noStroke();
  frameRate(10);
  background(127);
  rectMode(CENTER);                                     // 2
  textAlign(CENTER, CENTER);
  textSize(48);
  bt.start();
  cam = new KetaiCamera(this, 720, 480, 10);
  ketaiNFC.beam("bt:"+bt.getAddress());
  text("Beam and tap screen to start camera", width/2, height/2);
}

void draw()
{
  if (cam.isStarted() && bt.isStarted())
  {
    interlace(cam.width/2, cam.height/2, cam.width/2, cam.height/2, divisions); // 3
  }
  else
    ketaiNFC.beam("bt:"+bt.getAddress());
}

void interlace(int x, int y, int w, int h, int level)     // 4
{
  if (level == 1)
  {
    color pixel = cam.get(x, y);                        // 5
    send((int)red(pixel), (int)green(pixel), (int)blue(pixel), x, y, w*2, h*2);  // 6
```

```
  }

  if (level > 1) {
    level--;                                                  // 7
    interlace(x - w/2, y - h/2, w/2, h/2, level);             // 8
    interlace(x - w/2, y + h/2, w/2, h/2, level);
    interlace(x + w/2, y - h/2, w/2, h/2, level);
    interlace(x + w/2, y + h/2, w/2, h/2, level);
  }
}

void onCameraPreviewEvent()
{
  cam.read();
}

void mouseReleased()
{
  if (!cam.isStarted())
    cam.start();

  divisions++;                                                // 9
}
```

Here are the steps we need to recursively process the live camera image.

---

1. Set the initial number of divisions to `1`, showing one fullscreen rectangle.
2. Center the rectangle around the horizontal and vertical location where it is drawn, using `rectMode()`.
3. Call the recursive function with starting values for each parameter, starting in the center of the camera preview.

4. Use the following parameters for `interlace()`: horizontal position `x`, vertical position `y`, rectangle width `w`, rectangle height `h`, and the number of `divisions`.
5. Get the pixel color at the defined `x` and `y` location in the camera preview image from the pixel located in the exact center of each rectangular area we use for the low-resolution preview.
6. Send the pixel data using our user-defined function `send()`.
7. Decrease the `limit` variable by 1 before recalling the recursive function. Decrease this variable and call the function only if the limit is greater than 1 to provide a limit.
8. Call `interlace()` recursively from within itself using a new location and half the width and height of the previous call as parameters.
9. Increment the number of divisions when tapping the screen.

Now that we are done with our coding for the camera and the recursive program to create a higher-and-higher resolution image preview, let’s create the code we need to activate `NFC` and Bluetooth in the activity life cycle.

Enable NFC and Bluetooth in the Activity Life Cycle

To use NFC and Bluetooth, we need to take similar steps in the activity life cycle as we've done for our Bluetooth peer-to-peer app. In Working with the Android Activity Life Cycle, we looked at the callback methods called during an activity life cycle. For this project, we need tell Android that we'd like to activate both NFCand Bluetooth. Let's put the lifecycle code for the activity into an ActivityLifecycle tab.

At the very beginning of the life cycle, onCreate(), we'll launch KetaiBluetooth by initiating our KetaiBluetooth object, and we'll tell Android that we intend to use NFC. We do so using an intent, which is a data structure to tell Android that an operation needs to be performed. For example, an intent can launch another activity or send a result to a component that declared interest in it. Functioning like a kind of glue between activities, an intent binds events between the code in different applications. We need an Intent to launch NFC.

When NFC becomes available because our activity is running in the foreground on top of the activity stack, we get notified via onNewIntent(), because we asked for such notification with our intent in onCreate(). This is where we tell Android that we use the result of the returned intent with our ketaiNFC object, launching NFC in our sketch. An activity is always paused before receiving a new intent, and onResume() is always called right after this method.

When Bluetooth is available as the result of the Bluetooth activity we launched onCreate() while instantiating KetaiBluetooth, the connection is handed to us via onActivityResult(), which we then assign to our bt object.

Finally, onResume(), we start our Bluetooth object bt and instantiate our NFC object ketaiNFC. Let's take a look at the actual code for ActivityLifecycle.

code/NFC/NFCBTTransmit/ActivityLifecycle.pde

```
void onCreate(Bundle savedInstanceState) {                // 1
  super.onCreate(savedInstanceState);
  bt = new KetaiBluetooth(this);
  ketaiNFC = new KetaiNFC(this);
  ketaiNFC.beam("bt:"+bt.getAddress());
}

void onNewIntent(Intent intent)                           // 2
{
 if (ketaiNFC != null)
    ketaiNFC.handleIntent(intent);
}

void onActivityResult(int requestCode, int resultCode, Intent data) // 3
{
 bt.onActivityResult(requestCode, resultCode, data);
}

void exit() {                                              // 4
  cam.stop();
}

void onDestroy()                                           // 5
{
  super.onDestroy();
  bt.stop();
}
```

We need these steps to initiate NFC and Bluetooth correctly within the activity life cycle.

1. Instantiate the Bluetooth object bt to start a Bluetooth activity. Register the NFC intent when our activity is running by itself in the foreground using FLAG_ACTIVITY_SINGLE_TOP.
2. Receive the NFC intent that we declared in onCreate(), and tell Android that ketaiNFC handles it.
3. Receive the Bluetooth connection if it started properly when we initiated it in onCreate().
4. Release the camera when another activity starts so it can use it.
5. Stop Bluetooth and the camera when the activity stops.

All of this happens right at the beginning when our sketch starts up. The callback methods we are using require some getting used to. Because NFC and Bluetooth launch in separate treads or activities from our sketch—and not sequentially within our sketch—we need the callback methods to get notified when the Bluetooth activity and the NFC intent have finished with their individual tasks.
And because we depend on the successful delivery of the NFC payload for our Bluetooth connection, we need to use those callback methods and integrate them into the activity life cycle of our sketch. Processing and Ketai streamline many aspects of this programming process; when it comes to peer-to-peer networking between Android devices, we still need to deal with those essentials individually.
Now let's move on to the NFC tab, where we put the NFC classes and methods.

Add the NFC Code

We don't need much code to import NFC and make the KetaiNFC class available to the sketch. When we receive an NFC event using onNFCEvent(), we take the Bluetooth address that has been transferred as a text String and use it to connect to that device using connectDevice(). Let's take a look at the code.

code/NFC/NFCBTTransmit/NFC.pde

```
KetaiNFC ketaiNFC;

void onNFCEvent(String s)                                  // 1
{
  tag = s;
  println("Connecting via BT to " +s.replace("bt:", ""));
  bt.connectDevice(s.replace("bt:", ""));                  // 2
  divisions = 1;
}
```

Here are the NFC steps we take.

1. Receive the String from the NFC event using the onNFCEvent() callback method.

2. Connect to the Bluetooth address we've received, removing the prefix "bt:" first.

Finally, let's take a look at the `Bluetooth` tab.

Add the Bluetooth Code

In the `Bluetooth` tab, we import the necessary Ketai Bluetooth and `OSC` package to send the Bluetooth messages. Let's use a custom function called `send()` to assemble the `OSC` message, sending out the color, location, and dimension of our pixel.
If we receive such a pixel from the networked Android via `onBluetoothDataEvent()`, we unpack the data contained in the `OSC` message and draw our pixel rectangle using a custom function, `receive()`.
Let's take a look at the code.

code/NFC/NFCBTTransmit/Bluetooth.pde

```
PendingIntent mPendingIntent;
KetaiBluetooth bt;
OscP5 oscP5;

void send(int r, int g, int b, int x, int y, int w, int h)
{

  OscMessage m = new OscMessage("/remotePixel/");         // 1
  m.add(r);
  m.add(g);
  m.add(b);
  m.add(x);
  m.add(y);
  m.add(w);
  m.add(h);

  bt.broadcast(m.getBytes());                             // 2
}

void receive(int r, int g, int b, int x, int y, int w, int h)  // 3
{
  println("r:"+r+" | g:"+g+" | b:"+b+" | x:"+x+" | y:"+y+" | w:"+w+" | h:"+h);
  fill(r, g, b);
  float s = displayWidth/cam.width; // scale full display
  rect(x*s, y*s, w*s, h*s);
  // background(r, g, b); // try this if running slow
}

void onBluetoothDataEvent(String who, byte[] data)
{
  KetaiOSCMessage m = new KetaiOSCMessage(data);
  if (m.isValid())
  {
    if (m.checkAddrPattern("/remotePixel/"))
    {
      if (m.checkTypetag("iiiiiii"))                      // 4
      {
        receive(m.get(0).intValue(), m.get(1).intValue(),
        m.get(2).intValue(), m.get(3).intValue(),
        m.get(4).intValue(), m.get(5).intValue(), m.get(6).intValue());
      }
```

```
    }
  }
}
```

Here are the steps we take to send and receive `OSC` messages over Bluetooth.

1. Add individual values to the `OscMessage` `m`.
2. Send the `byte` data contained in the `OSC` message `m` via Bluetooth using `broadcast()`.
3. Receive individual values sent via `OSC`, and draw a rectangle in the size and color determined by the received values.
4. Check if all seven integers in the `OSC` message are complete before using the values as parameters for the `receive()` method.

   Now with our recursive program, camera, `NFC`, and Bluetooth code completed, it's time to test the app.

Run the App

Before we run the app, we need to set two permissions. Open the Permission Selector from the Sketch menu and select `CAMERA` and `INTERNET`.

Now browse to the sketch folder and open `AndroidManifest.xml` in your text editor, where you'll see that those permissions have been set. Add `NFC` permissions so the file looks something like this:

code/NFC/NFCBTTransmit/AndroidManifest.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1"
android:versionName="1.0" package="">
  <uses-sdk android:minSdkVersion="15" android:targetSdkVersion="23"/>
  <application android:debuggable="true" android:icon="@drawable/icon" android:label="">
    <activity android:name="">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.nfc.action.TECH_DISCOVERED"/>
      </intent-filter>
      <intent-filter>
        <action android:name="android.nfc.action.TAG_DISCOVERED"/>
      </intent-filter>
    </activity>
  </application>
  <uses-permission android:name="android.permission.BLUETOOTH"/>
  <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
  <uses-permission android:name="android.permission.CAMERA"/>
  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.NFC"/>
</manifest>
```

Run the app on the device that is already connected to the PC via USB. When it launches, disconnect and run the app on your second Android device. Now it's time for the moment of truth—touch both devices back-to-back and confirm the P2P connection.

You should see a colored rectangle on each display, taken from the camera preview of the other device. If you move your camera slightly, you'll recognize that its color is based on a live feed. Tap each screen to increase the resolution and observe what happens on the other device, then tap again. Each new division requires more performance from the devices as the number of pixels we send and display increases exponentially.

Keep tapping and you will observe how the app slows as the size of the data payload increases.

Now that we've learned how to send a Bluetooth ID via `NFC` Beam technology to another device, let's move on to reading and writing `NFC` tags.

Read a URL from an NFC Tag

Moving on to the world of `NFC` tags, our sketches will get significantly shorter. Tags come in different shapes and sizes, as illustrated in Figure 8.0, and they mostly store a few dozen characters, which is why most tags contain a `URL` pointing to a website. For this first app, we'll create a sketch that can read `NFC` tags.

Because we are dealing mostly with `URL`s, let's also include some Processing code that lets us open the link in the device browser. We'll check if it is a valid URL before we launch the browser. When we launch our sketch on the device, the app will wait for an `NFC` event to occur, which will be triggered when we touch the tag with the device. We'll also want to display the tag's content on the device, as shown below.

Figure 8.2 - Read an NFC tag.

When you approach the NFC tag with the Android device, it outputs the collected text/URL on the display. Tapping the screen will take you to the URL saved on the tag.

Enable NFC

To get started, let's enable `NFC` using the now familiar activity lifecycle methods we used in the previous sketch. All the lifecycle code we need to enable `NFC` goes into our `EnableNFC` tab. This tab is identical to Enable NFC and Bluetooth in the Activity Life Cycle, with the exception that we don't have to start up Bluetooth as well. Let's take a look at the code.

code/NFC/NFCRead/EnableNFC.pde

```
PendingIntent mPendingIntent;

void onCreate(Bundle savedInstanceState) {
  ketaiNFC = new KetaiNFC(this);
  super.onCreate(savedInstanceState);
  mPendingIntent = PendingIntent.getActivity(this, 0, new Intent(this,
  getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);
}

void onNewIntent(Intent intent) {
  if (ketaiNFC != null)
    ketaiNFC.handleIntent(intent);
}
```

Now let's move on to our main sketch, `NFCRead`.

Add the Main Tab

Now that we've set up everything so `NFC` can start up properly, let's take a look at the main tab, where we read the tag. When we receive an `NFC` event that includes a text `String`, we then use Processing methods to clean the `String`, check if it's a valid `URL`, and link to the browser. We use `trim()` to remove whitespace characters from the beginning and the end of the `String`. Then we can use the resulting `String` directly with Processing's `link()` method, which opens the browser and shows the website stored on the tag. To check if it's a valid `URL`, we use the `indexOf()` method, which tests if a substring is embedded in a string. If it is, it returns the index position of the substring, and if not, it returns `-1`.
Here is the code.

code/NFC/NFCRead/NFCRead.pde

```
// Required NFC methods on startup
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;

import ketai.net.nfc.*;

String tagText = "";
KetaiNFC ketaiNFC;

void setup() {
  ketaiNFC = new KetaiNFC(this);
  orientation(LANDSCAPE);
  textSize(144);
  textAlign(CENTER, CENTER);
}

void draw() {
  background(78, 93, 75);
  text("Tag:\n"+ tagText, width/2, height/2);             // 1
}

void onNFCEvent(String txt) {                             // 2
  tagText = trim(txt);                                    // 3
}

void mousePressed() {
  if (tagText.indexOf("http://") == 0)                    // 4
    link(tagText);                                        // 5
}
```

1. Specify how to display the content of the tag stored in `tagText` on the device display.
2. Receive a `String` from the tag when the device touches it and an `NFC` event occurs.
3. Assign a clean version of the `String` to `tagText` after removing whitespace characters from the beginning and end using `trim()`.
4. Receive a `String` from the tag when the device touches it and an `NFC` event occurs.
5. Jump to the link stored on the tag using `link()`. Follow the link in the device browser when tapping the screen, given there is text stored on the tag.

Before we run the sketch, we'll again need to make sure that we have the appropriate `NFC` permissions.

Set NFC Permissions

Because `NFC` permissions are not listed in Processing's Android Permissions Selector, where we usually make our permission selections (Setting Sketch Permissions), we need to modify `AndroidManifest.xml`directly to enable `NFC`. Processing typically takes care of creating this file for us when we run the sketch, based on the selection(s) we've made in the Permission Selector, and it re-creates the file every time we change our permission settings. Also, when we make no permission selections at all, Processing creates a basic manifest file inside our sketch folder.

Since we are already editing the Android manifest file manually, let's jump ahead and also add an intent filterthat launches our app when a tag is discovered. This way, `NFCRead` will start when the app is not yet running and resume when it is running in the background.

Let's take a look at the sketch folder and see if `AndroidManifest.xml` already exists inside it. Open the sketch folder by choosing Sketch ↦ Show Sketch Folder. You should see two Processing source files in the folder for this sketch, one named `EnableNFC` and the other `NFCRead`.

Now to create a manifest, return to Processing and choose Android ↦ Sketch Permissions from the menu. Although we won't find an `NFC` check box in there, it will still create an `AndroidManifest.xml` template for us that we can modify.

To modify the manifest, navigate to the sketch folder and open `AndroidManifest.xml`. Make your changes to the manifest so it looks like the following `xml` code.

code/NFC/NFCRead/AndroidManifest.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1"
android:versionName="1.0" package="">
  <uses-sdk android:minSdkVersion="15" android:targetSdkVersion="23"/>
  <uses-permission android:name="android.permission.NFC"/>  <!--<callout
id="co.nfc.permission"/>--><application android:debuggable="true"
android:icon="@drawable/icon" android:label="">
    <activity android:name="">
      <intent-filter>
        <action android:name="android.nfc.action.TECH_DISCOVERED"/> <!--<callout
id="co.tech.discovered"/>--></intent-filter>
      <intent-filter>
        <action android:name="android.nfc.action.NDEF_DISCOVERED"/> <!--<callout
id="co.intent.ndef"/>--></intent-filter>
      <intent-filter>
        <action android:name="android.nfc.action.TAG_DISCOVERED"/>  <!--<callout
id="co.tag.discovered"/>--><category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

In the manifest `xml`, we take the following steps.

1. Set `NFC` permission.

2. Have the app look for a tag.
3. Make Android look for an `NDEF` tag.
4. If an `NDEF` tag is discovered and the app is not already running in the foreground, make the activity the default.

Now that the appropriate `NFC` permissions are in place, let's run the app.

Run the App

Run the app on the device. When it starts up, our `tagText` `String` is empty. You can tap the screen but nothing happens, because we don't yet have a link to jump to.

Now approach your tag with the back of your device. A few centimeters before you touch the tag, you will hear a beep, which signals that an `NFC` event has occurred. The `URL` stored on the tag should now appear on your display (Read a URL from an NFC Tag). If you have another one, try the other tag and see if it has a different `URL`.

Now that you've successfully read `NFC` tags, it's time to learn how to write them as well, either to add your own URL or to change the `NDEF` message on there.

Write a URL to an NFC Tag

The `NFC` device built into the Android can also write to `NFC` tags. Most tags you get in a starter kit can be repeatedly rewritten. So if you'd like to produce a small series of `NFC`-enabled business cards, provide a quick way to share information at a fair booth, or create your own scavenger hunt with `NFC` stickers, you can write tags with your Android.

Let's build on our previous sketch and add a feature to do that. The app must still be able to read a tag to confirm that our content has been successfully written. To write tags, let's use the software keyboard to type the text we want to send as a `String` to the tag, as illustrated in the figure below. If we mistype, we need the backspace key to delete the last character in our `String`. Once we've completed the `String`, let's use the `ENTER` key to confirm and `write()` the tag. The transmission to write the actual tag is then completed when we touch the tag. Finally, when we come in contact with the tag again, we read its content.

Figure 8.3 - Read and write NFC tags.

Use the keyboard to input a URL and press Enter. The string will then be written to the tag on contact.

Let's introduce a variable called `tagStatus` to provide us with some onscreen feedback during this process. The sketch itself is structured identically to our previous example NFC/NFCRead/NFCRead.pde. We'll keep the `EnableNFC` tab and the permissions we set for `AndroidManifest.xml`.
Let's take a look at the main tab.

code/NFC/NFCWrite/NFCWrite.pde

```
// Required NFC methods on startup

import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import ketai.net.nfc.*;
import ketai.ui.*;                                          // 1

KetaiNFC ketaiNFC;

String tagText = "";
String tagStatus = "Tap screen to start";                   // 2

void setup() {
  if (ketaiNFC == null)
    ketaiNFC = new KetaiNFC(this);
  // orientation(LANDSCAPE);
  textAlign(CENTER, CENTER);
  textSize(144);
}

void draw() {
  background(78, 93, 75);
  text(tagStatus + " \n" + tagText, 0, 50, width, height-100);  // 3
}
```

```
void onNFCEvent(String txt) {
  tagText = trim(txt);
  tagStatus = "Tag:";
}

void onNFCWrite(boolean result, String message) {
  if (result)
    tagStatus = "Writing Complete.";
}

void mousePressed() {
  KetaiKeyboard.toggle(this);                              // 4
  tagText = "";
  tagStatus = "Type tag text:";
  textAlign(CENTER, TOP);                                  // 5
}

void keyPressed() {
  if (key != CODED)                                        // 6
  {
    tagStatus = "Write URL, then press ENTER to transmit";
    tagText += key;                                        // 7
  }
  if (key == ENTER)                                        // 8
  {
    ketaiNFC.write(tagText);                               // 9
    tagStatus = "Touch tag to transmit:";
    KetaiKeyboard.toggle(this);
    textAlign(CENTER, CENTER);
  } else if (keyCode == 67)                                // 10
  {
    tagText = tagText.substring(0, tagText.length()-1);    // 11
  }
}
```

Let's take a look at the steps we need to take to write a tag.

---

1. Import the Ketai user interface package to show and hide Android's software keyboard.

2. Declare a variable `tagStatus` to give us feedback on the text input and writing status.
3. Show the tag status and the current text input to write to the tag.

4. Toggle the Android software keyboard to type the text `String`.
5. Adjust the display layout to `TOP`-align vertically so we can accommodate the software keyboard.
6. Check if the key press is `CODED`.
7. Add the last character typed on the software keyboard to the `tagText String`.
8. Check if the Enter `key` is pressed on the keyboard.
9. Prepare writing the content of the `tagText` variable to the tag on contact using `write()`.
10. Check if the coded backspace button `67` is pressed on the software keyboard to remove the last character in `tagText`.
11. Remove the last character in the `tagText String` using the `String` method `substring()` and `length()`. Determine the current `length()` of the `String` and return a `substring()` from the first character `0` to the next-to-last character.

Run the App

Run the app on the device and follow the instruction on the display. Start by tapping the screen, which will cause the software keyboard to appear. Type your text. If you mistype, you can correct it using the backspace button. When you are done, finish up by tapping the Enter key. Now the device is ready to write to the tag. Touch the tag with the back of the device, and if the sketch is operating properly, you should hear a beep, which indicates the device has found and written to the tag. Touch the tag again to read what's now on it. That's it!

Now that we've completed both reading and writing tags, we know how easy this kind of interaction is to do and we've got an idea of how useful it can be.

Wrapping Up

With all these different networking techniques under your belt, you've earned the networking badge of honor. Now you'll be able to choose the right networking method for your mobile projects. You already know how to share data with a remote device and you've mastered peer-to-peer networking for those that are nearby. You've seen how near field communication can be used to initiate a peer-to-peer connection between two devices and to read `NFC` tags. Each of these highly relevant mobile concepts complement each other and can be combined—making for a whole lot of apps you can build.

Now that we've seen how to share all kinds of data between networked devices, it's time to take a closer look at the databases and formats we can use both locally and remotely.

9. Working with Data

Sooner or later, we'll need to be able to store and read data. To keep track of user choices and settings, we need to write data into a file or database stored on the Android device. We can't always rely on a carrier or network connection to read and write data from the Web or the Cloud, so we require a repository on the Android device; that way we can stop the app or reboot the phone without losing data and provide continuity between user sessions. Users expect mobile devices to seamlessly integrate into their daily routines and provide them with information that is relevant to their geographic and time-specific context. Entire books have been dedicated to each section in this chapter. As we create each chapter's projects, we'll try our best to remain focused on the Android specifics when we are working with data and to explore only those formats and techniques you're most likely to use.

Processing has received a major upgrade to its data features, which were compiled into a comprehensive `Table` class. The `Table` class allows us to read, parse, manipulate, and write tabular data in different datatypes. With Processing 2.0, Ben Fry, one of its principal authors, has now integrated the methods and techniques from his seminal Visualizing Data (Fry, '08) into the Processing core, making it easier for us to visually explore data.
Using the `Table` class, we'll be visualizing tab- and comma-separated data in no time. We'll learn how to work with private and public data storage on the Android, keeping data accessible only for our app, or alternatively sharing it with other apps via Android's external storage. We'll read data from the internal and external storage and write data into tab-separated value files stored on the Android.
To demonstrate how sensors, stored data, and Processing techniques for displaying data can be combined, we'll create an app that acquires real-time earthquake data and displays the result. The app will read, store, and show all reported earthquakes worldwide during the last hour using the data access techniques you'll learn in this chapter and data collected by the US Environmental Protection Agency (`EPA`). For the project, we'll make use of the `Location` code introduced in Chapter 4, Using Geolocation and Compass. In a second step, we'll refine the earthquake app to indicate when new earthquakes are reported using timed device vibrations.
In the following chapter of this two-part introduction to data, we'll work with SQLite, the popular relational database management system for local clients like the Android and used by many browsers and operating systems. It implements the popular Structured Query Language (`SQL`) syntax for database queries, which we can use to access data that we've stored locally on our device. We'll first get SQLite running with a simple sketch and learn how to use `SQL` queries to retrieve data from one or more tables, and then we'll use it to capture, store, browse, and visualize sensor values. These are the tools we'll need to write some more ambitious data-driven projects.

Let’s first take a closer look at data, data storage, and databases.

Introducing Databases

To autocomplete the words that users type, guess a user’s intent, or allow users to pick up where they have left off requires that our apps work with files and databases; there is simply no alternative. For example, when we pull up a bookmark, check a score, or restore prior user settings, we read data that we’ve written earlier into a local file or database. Data is essential for mobile apps because both time and location are ubiquitous dimensions when we use our Android devices, and it requires additional considerations when we develop, compared to desktop software. We expect our mobile apps to also be functional when cellular or Wi-Fi networks are unavailable. In such a scenario, apps typically rely on the data that has been saved in prior sessions, which typically get updated when a network becomes available.

Whether they are stored in tabular files or as object-oriented records, databases generally share one characteristic—structured data in the form of text and numbers, separated into distinct categories, or fields. The more specific the fields, the better the results and sorts the database can deliver. As we know, for example, dumping all of our receipts into one shoebox is not the kind of database structure an accountant or financial advisor would recommend: labeled files and folders are far more efficient and searchable.

Recommending which type of organization, or data architecture, is the best for a particular task goes beyond the scope of this book. It’s the subject of numerous anthologies (such as Visualizing Data (Fry, '08)), which do a great job of breaking down appropriate table relations, data types, and queries. We’re going to limit the scope of our exploration to tab- and comma-separated values (`TSV` and `CSV`) because they are easy to use and very common, and we’ll balance it with the more powerful SQLite data management system, providing us with more complex queries and the most widely deployed data management system out there.
The most common structural metaphor for representing a database is a table (or a couple of them). Known to us from spreadsheets, a table uses columns and rows as a data structure. Columns, also known as fields, provide the different categories for a table; rows contain entries in the form of numbers and text that always adhere to the structure provided by the columns.

+

Processing provides us with a `Table` class that lets us read, parse, manipulate, and write tabular data, which we’ll be using throughout the chapter. It’s a very useful class that is built into Processing’s core and provides us with methods akin to what we’d expect from a

database. However, it is used as an object stored in memory only until we explicitly write the contents to a file.

Working with the Table Class and the File System

Throughout this chapter, we'll work with Processing's `Table` class, and particularly with the following methods:

`Table`	A comprehensive Processing class to load, parse, and write data in different file formats—it provides similar methods that we'd find in a database.
`getRowCount()`	A `Table` method, which returns the number of rows or entries inside a table
`getInt()`, `getLong()`, `getFloat()`, `getDouble()`, `getString()`	A series of `Table` methods to retrieve the different value types from a specified row and column provided to the methods as two parameters separated by comma
`addRow()`	A `Table` method to add a new row to the table
`writeTSV()`	A `Table` method to write a `tsv` file to a specified location in the file system, provided to the method as a parameter
`Environment`	An Android class that provides access to environment variables such as directories
`File`	A Java method to create a file path to a specified location in the file system
`URL`	A Java class for a Uniform Resource Locator, a pointer to a resource on the Web
`BufferedReader`	A Java class that reads text from a character-input stream and buffers them so we can read individual characters as complete text—we use it in this chapter to make sure we've received all the comma-separated values stored in our online data source.
`InputStreamReader`	A Java class reading a bytes stream and decoding the data into text characters

`sketchPath()`	A Processing method returning the file path to the sketch in the file system—if we run our sketch on the Android device, it returns the path to the app's location within Android's file system.
`KetaiVibrate()`	A Ketai class giving access to the built-in device vibration motor
`vibrate()`	A `KetaiVibrate` method to control the built-in device vibration motor—this can be used without parameters for a simple vibration signal, a duration parameter in milliseconds, or an array of numbers that trigger a pattern of vibrations, `vibrate(long[] pattern, int repeat)`.

Since we are writing to the device's file system in this chapter, let's take a look at the options we have.

Working with Android Storage

The Android device is equipped with the following storage types, which are available to our apps for saving data. We can keep our data private or expose it to other applications, as we've done deliberately in Snap and Save Pictures, to share images we took.

- Internal Storage This is used to store private data on the device memory. On the Android, files we save are saved by default to internal storage and are private, which means other applications cannot access the data. Because they are stored with our application, files saved to internal storage are removed when we uninstall an app. We'll use the internal storage in Save User Data in a TSV File.

- External Storage All Android devices support a shared external storage that we can use to store files. The external storage can be a removable SD card or a non-removable storage medium, depending on your Android device's make and model. If we save a file to external storage, other applications can read the file and users can modify or remove it in the USB mass storage mode when we connect the device to the desktop. In 4.1 Jelly Bean or earlier versions of Android, we need to be careful when writing essential data there for the app to run, and we can't use it for any data we need to keep private. Since 4.2 Jelly Bean, Android provides a `READ_EXTERNAL_STORAGE` permission for protected read access to external storage. In the Settings menu on the Android device, a new developer option called `Protect USB storage`allows us to activate a read-access restriction and test this new permission option for protected read access to external storage.
- SQLite Databases SQLite support on the Android provides us with a simple database management tool for our local storage, both internal and external, which we'll explore in the next chapter, Chapter 10, Using SQLite Databases.

- Network Connections We've explored the possibility of working with a web server for stored data already in Find a Significant Other (Device), which is an option for files that do not have to be readily available for our app to run. This is not a good option for user preferences, as we can't rely on a carrier or Wi-Fi network to reach the server.
- Cloud Services Cloud services are becoming increasingly popular and are another option for extending the storage capabilities of Android devices. Google's Cloud platform, for instance, provides SDKs to integrate the Cloud into your apps alongside a Google Cloud Messaging service to send a notification to your Android device if new data has been uploaded to the Cloud and is available for download. Also, Google Drive provides an SDK to integrate Google's document and data storage service into our apps.

We'll focus on Android's internal storage in this chapter. Let's get started and use the `Table` class now to read data from a tab-separated file.

Read a Tab-Separated Grocery List

Let's get started by working with a familiar list, a grocery list for our favorite pasta recipe, which we'll display on the device screen. Let's color-code the ingredients based on where we have to go to get them. We'll work with tab-separated values stored in a text file called `groceries.tsv`, which is located in the `data` folder of our sketch. The file contains eleven items saved into individual rows, each row containing the amount, unit, item, and source for each ingredient on our list, separated by a tab character. The first row contains the labels for each column, which we'll keep for our reference but not display on the Android screen, as shown in Figure 9.0.

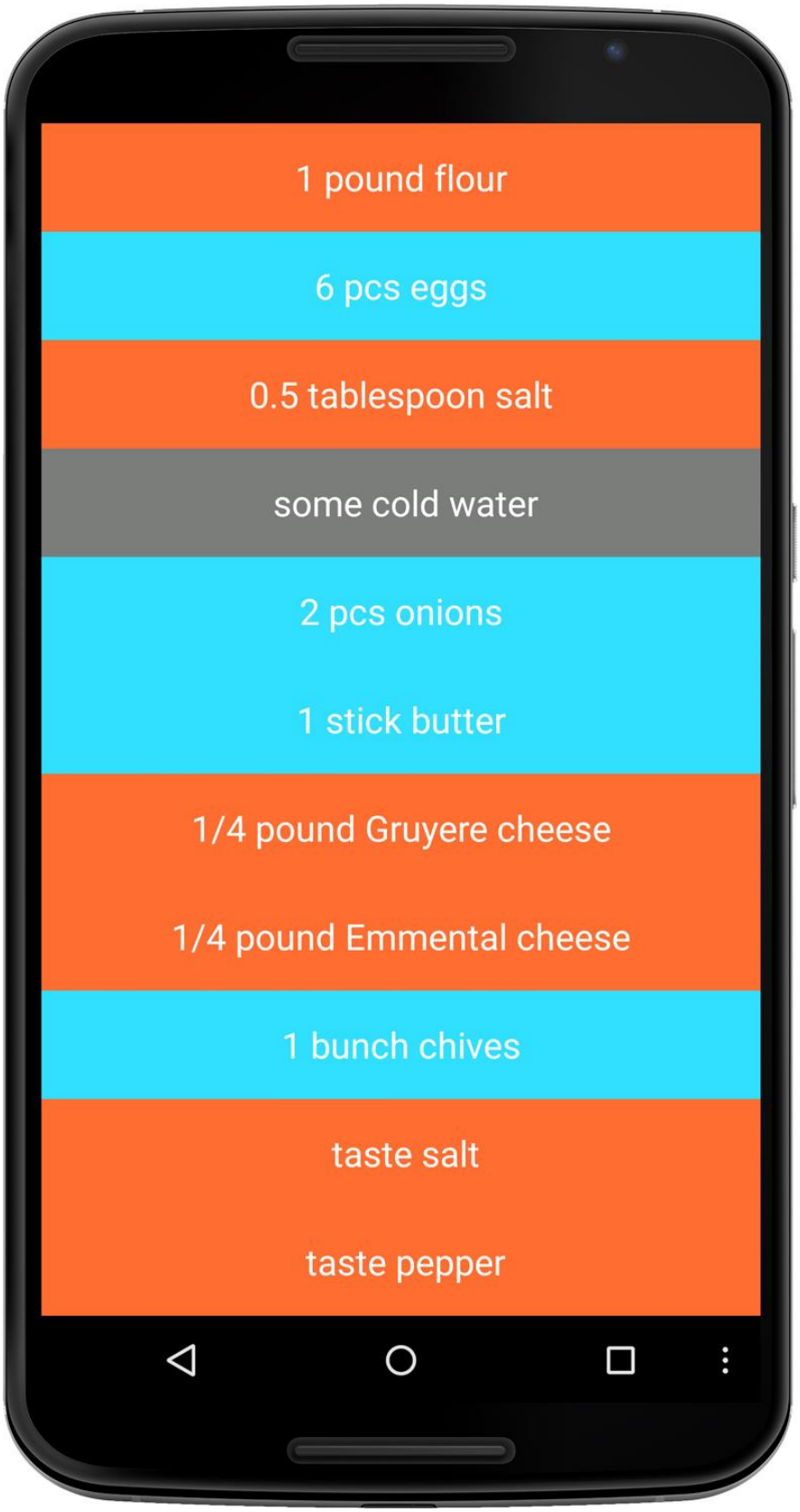


Figure 9.0 - Reading groceries items from a tab-separated data file.

The eleven items we need for this favorite pasta recipe are listed, color-coded by source (cyan for “market” and orange for “store”).

To implement this sketch, we'll use Processing's `Table` class for loading and parsing the file's contents row by row. When we initialize our `Table` object in `setup()`, we provide a file path to our data as a parameter, and `Table` object will take care of loading the data contained in the file for us. The grocery items and amounts contained in the `groceries.tsv` file each use one row for one entry, separated by a new line ("\n") character. A tab separates the amount from the volume unit, item description, and the source where we'd like to get it.
Let's look at the text file containing our grocery items, which are saved into individual rows and separated by tabs.

code/Data/DataReadGroceries/data/groceries.tsv

```
amount    unit    item    source
1    pound    flour    store
6    pcs    eggs    market
0.5    tablespoon    salt    store
    some    cold water
2    pcs    onions    market
1    stick    butter    market
1/4    pound    Gruyere cheese    store
1/4    pound    Emmental cheese    store
1    bunch    chives    market
    taste    salt    store
    taste    pepper    store
```

The file extension `tsv` indicates here that the groceries data is tab-separated. If the file is named `groceries.txt` instead, we can use the option parameter in `loadTable()` to indicate the tab-separated data structure, like this:

```
loadTable("groceries.txt", "tsv");
```

We can also add `header` in the options parameter to indicate that the file includes a header row.

```
loadTable("groceries.txt", "header, tsv");
```

Now let's take a look at our code.

code/Data/DataReadGroceries/DataReadGroceries.pde

```
Table groceries;

void setup()
{
  groceries = loadTable("groceries.tsv");                 // 1
  textSize(72);
  rectMode(CENTER);                                       // 2
  textAlign(CENTER, CENTER);                              // 3
  noStroke();
  noLoop();                                               // 4
  background(0);
  int count = groceries.getRowCount();                    // 5

  for (int row = 1; row < count; row++) {
    float rowHeight = height/(count-1);                   // 6
    String amount = groceries.getString(row, 0);          // 7
    String unit = groceries.getString(row, 1);            // 8
    String item = groceries.getString(row, 2);            // 9

    if (groceries.getString(row, 3).equals("store")) {    // 10
      fill(color(255, 110, 50));                          // 11
```

```
    }
    else if (groceries.getString(row, 3).equals("market")) {
      fill(color(50, 220, 255));
    }
    else {
      fill(127);
    }

    rect(width/2, rowHeight/2, width, rowHeight);         // 12
    fill(255);
    text(amount + " " + unit + " " + item, width/2, rowHeight/2);  // 13
    translate(0, rowHeight);                              // 14
  }
}
```

Here are the steps we take to read our text file and display the grocery list.

---

1. Load `groceries.tsv` by providing the file name as a parameter to the `Table` object `groceries`.
2. Set the rectangle drawing mode to `CENTER` so that the x and y location of the rectangle specifies the center of the rectangle instead of the default upper left corner.
3. Center the text labels for our rows horizontally and vertically within our text area.
4. Do not loop the sketch (by default it loops 60 times per second), because we are not updating the screen and do not interact with the touch screen interface. This optional statement saves us some `CPU` cycles and battery power; the sketch produces no different visual output if we don't use this statement.
5. Count the number of rows contained in `groceries` and store this number in `count`—used also to position the rectangles.
6. Calculate the `rowHeight` of each color rectangle by dividing the display height by the number of rows in the file.
7. Get the text string from column `0` that contains the `amount` using `getString()`.
8. Get the text string from column `1` that contains the measurement `unit`.
9. Get the `item` description from column `2`, `unit`.
10. Check if the location stored in column `3` matches "store."
11. Check if the location stored in column `3` matches "market."
12. Draw a rectangle with the fill color `c` horizontally centered on the screen and vertically moved down by half a `rowHeight`.
13. Output the text label for each named color centered within the row rectangle.
14. Move downward by one `rowHeight` using `translate()`.

Let's now move on to reading comma-separated values from a text file.

Read Comma-Separated Web Color Data

In the next sketch, we'll work with hexadecimal values of web colors and juxtapose them with their official name from the HTML web specification. Our data source contains comma (",")

separated values (`CSV`), which we read from the file stored in the `data` directory of our sketch. The `CSV` file contains sixteen rows, each containing two values separated by a comma. The first value contains a `String` that is one of the named colors in the W3C's `HTML color` specification. The second value contains a text `String` that represents the hexadecimal value (or "hex value," for short) of that named color. When we juxtapose a text description with its color in a list of individually labeled swatches, our sketch will display a screen like that shown in Figure 9.1. To distribute each swatch vertically, we use the `translate()` method we've implemented already in Find Your Way to a Destination.

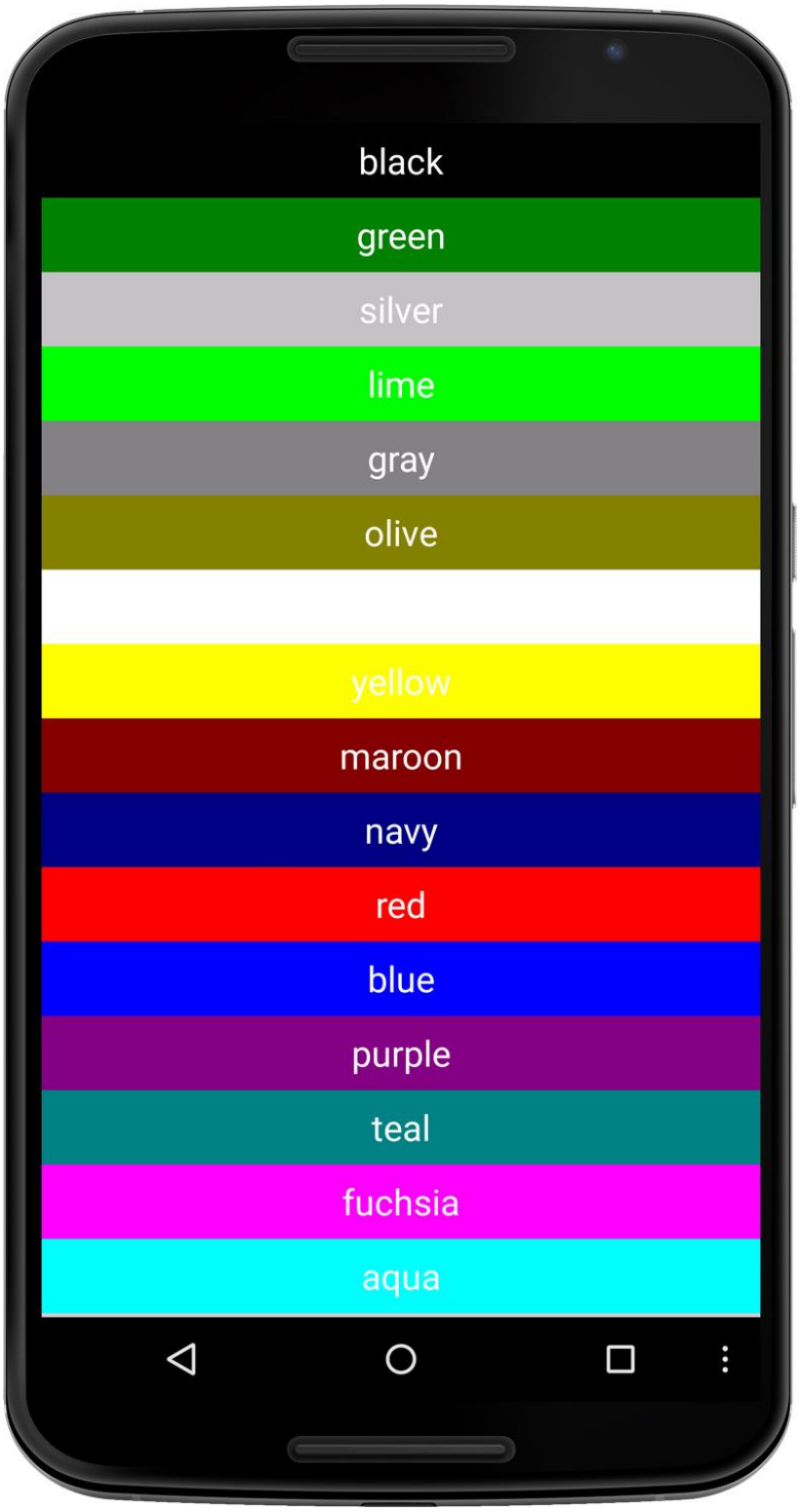


Figure 9.1 - Reading comma-separated color values.

Sixteen named colors from the HTML specification are stored in a csv file and juxtaposed with their hexadecimal color value.

Hexadecimal is a numbering system with a base of `16`. Each value is represented by symbols ranging `0..9` and `A..F` (`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`)—sixteen

symbols that each represent one hexadecimal value. Two hex values combined can represent decimal numbers up to `256` (`16` times `16`)—the same number we use to define colors in other Processing color modes such as `RGB` and `HSB`(see Using Colors Chapter 2).

In most programming languages, hexadecimal color values are typically identified by a hash tag (`#`) or the prefix `0x`. The hex values stored in column 1 of our file contains a `#` prefix. We need to convert the text `String` representing the hex color in the first column into an actual hex value we can use as a parameter for our `fill()` method. For that, we use two Processing methods, `substring()` and `unhex()`, to bring the hex value into the correct format, and then we convert the `String` representation of a hex number into its equivalent integer value before applying it to `fill()`.

The `substring()` method allows us to remove the `#` prefix, and `unhex()` allows us to convert `String` into hex. The `unhex()` method expects a hexadecimal color specified by eight hex values, the first two (first and second) defining the alpha value or transparency; the next two (third and fourth), red; then the next two (fifth and sixth), green; and finally the last two values (seventh and eighth), the blue value of that color. When we read the color from the file, we'll prepend `"FF"` so we get fully opaque and saturated colors.

Let's first take a peek at the color data we copy into `colors.csv`.

code/Data/DataRead/data/colors.csv

```
black,#000000
green,#008000
silver,#C0C0C0
lime,#00FF00
gray,#808080
olive,#808000
white,#FFFFFF
yellow,#FFFF00
maroon,#800000
navy,#000080
red,#FF0000
blue,#0000FF
purple,#800080
teal,#008080
fuchsia,#FF00FF
aqua,#00FFFF
```

Open the file in your text editor and copy the file into your sketch's `data` folder. The file contents are fairly easy for us to grasp, as the file only contains two columns and sixteen rows. Our approach would be the same if we faced a `csv` file containing fifty columns and one thousand rows. Note that there are no spaces after the commas separating the values, as this would translate into a whitespace contained in the value following the comma.

Now let's take a look at our Processing sketch.

code/Data/DataRead/DataRead.pde

```
Table colors;

void setup(){
```

```
  colors = loadTable("colors.csv");                        // 1

  textSize(72);
  rectMode(CENTER);
  textAlign(CENTER, CENTER);
  noStroke();
  noLoop();
  int count = colors.getRowCount();

  for (int row = 0; row < count; row++) {
    color c =  unhex("FF"+colors.getString(row, 1).substring(1));  // 2
    float swatchHeight = height/count;
    fill(c);                                               // 3
    rect(width/2, swatchHeight/2, width, swatchHeight);
    fill(255);
    text(colors.getString(row, 0), width/2, swatchHeight/2);
    translate(0, swatchHeight);
  }
}
```

Here are the steps we take to load and parse the data.

---

1. Load the text file containing colors as comma-separated values.
2. Define a hex color from the `String` contained in column `1` after removing the `#` prefix using `substring()` and prepending `"FF"` for a fully opaque alpha channel.
3. Set the `fill()` color for the color rectangle to the hex color we've retrieved from the text file.

Now that you know how it works, let's run the app.

Run the App

When our sketch runs, the `colors.csv` file will be included as a resource and installed with our app on the device. You'll see the sixteen named colors in `HTML` as individual swatches filling up the screen. Because we haven't locked `orientation()` in this sketch, the display will change its orientation depending on how we hold the device. We've implemented the position and alignment of the swatches in a variable manner based on the current display's `width` and `height`, so the sketch will scale to any orientation or display size on our Android phone or tablet.

Now that we know how to read data from a file, let's now move ahead and read and write tab-separated values.

Save User Data in a TSV File

In this project, we'll learn how to save user data. We'll implement a simple drawing sketch that allows us to place a sequence of points on the Android touch screen. When we press a button on the device triggering a `keyPressed()` event, the resulting drawing doodle consisting of individual points is saved to the app folder on the Android device.

Using the menu button on the device as a trigger, we'll write each horizontal and vertical position x and y into a text file using tab-separated values. To keep track of how many points we've saved into our file, we'll output our row count on the display as well. If we pause or close the app and come back later, the points we've saved will be loaded into the sketch again, and we can continue where we left off. If we add to the drawing and press the menu button again, the new points will be appended to our `data.tsv` file and saved alongside our previous points.

We'll revisit the simple drawing concepts from Introducing the Android Touch Screen, and Share Real-Time Data, and use the `mouseX` and `mouseY` location of our fingertip to continuously draw points on the screen, as shown in Figure 9.2. Using Java's `File` class, we'll also learn about Android storage and file paths, because we are creating a `tsv` file inside our app. This file will only be available for our app and not be usable by other locations, keeping the data private.

In terms of working with data, we'll start this time from scratch. We won't be copying an existing data source into the sketch's `data` folder. Instead we'll create data via the touch screen interface and write it into a file located in our sketch folder. This time, we use tab-separated values and save the data into a `data.tsv` file.

There is no significant difference between the two delimiters. Instead of a comma, `TSV` uses a tab (`\t`) to separate values. The most important thing to consider when choosing between the two formats is this: if you use comma-separated values, you cannot have entries that include a comma in a text string, and if you use tab-separated values, you cannot have entries that use tabs without changing the table structure and making it impossible to parse the file correctly. You can modify `CSV` and `TSV` text files in any text editor, and your operating system might already open it up with your default spreadsheet software. I personally have an easier time deciphering tab-separated values because tabs lay out the columns in a more legible way, which is why I prefer `TSV`. The `Table` class can handle, read, and write either format equally well, so from a technical perspective, it really doesn't make much of difference how we store our data.

To implement this sketch, we'll revisit the handy `PVector` class we already used in , to store value pairs in a vector. When we worked with an existing file earlier, we were certain that the `csv` file exists. Now when we run the sketch for the first time, the `data.tsv` file we'll be using to store our data won't exist yet, and we'll need to create one using Processing's `Table` method `writeCSV()`. To check if `data.tsv` exists from a prior session, we'll use the `try` `catch` construct typically used in Java to catch exceptions that would cause our app to crash. We use it in our sketch to check if `data.tsv` already exists within our app. If we are trying to load the file when it does not exist the first time around, we'll receive an exception, which we can use to create the file.

To draw, we'll use Processing's `mouseDragged()` method again, called every time we move our finger by one or more pixels while tapping the screen. This means that we will add new points to our table only when we move to a new position. The point count we display at the top of the screen will give us some feedback whenever we've added a new point to the list. To save the points to the Android's internal storage, press one of the device buttons.
Let's take a look at the sketch.

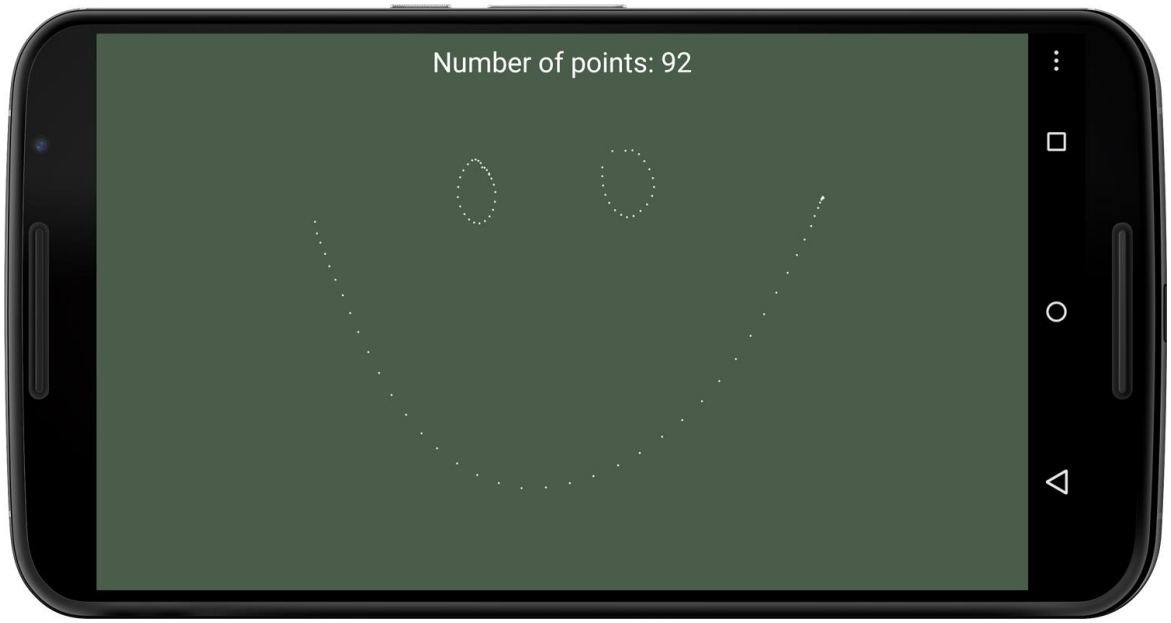


Figure 9.2 - Write data to an Android.

The illustration shows a total of eighty data points drawn on the touch screen, stored in the points PVector array, and saved to the Android storage in a file called data.tsv when we press a button.

code/Data/DataWrite/DataWrite.pde

```
Table tsv;                                                  // 1
ArrayList<PVector> points = new ArrayList<PVector>();      // 2
void setup()
{
  orientation(LANDSCAPE);
  noStroke();
  textSize(72);
  textAlign(CENTER);
  try {                                                      // 3
    tsv = new Table(new File(sketchPath("")+"data.tsv"));   // 4
  }
  catch (Exception e) {                                      // 5
    tsv = new Table();                                       // 6
  }
  for (int row = 1; row < tsv.getRowCount(); row++)
  {
    points.add(new PVector(tsv.getInt(row, 0), tsv.getInt(row, 1), 0));  // 7
  }
}

void draw()
{
  background(78, 93, 75);
```

```
  for (int i = 0; i < points.size(); i++)
  {

    ellipse(points.get(i).x, points.get(i).y, 5, 5);      // 8
  }
  text("Number of points: " + points.size(), width/2, 100);
}

void mouseDragged()
{

  points.add(new PVector(mouseX, mouseY));                // 9
  String[] data = {
    Integer.toString(mouseX), Integer.toString(mouseY)    // 10
  };
  tsv.addRow();                                           // 11
  tsv.setRow(tsv.getRowCount()-1, data);                  // 12
}

void keyPressed()
{
  try {
    tsv.save(new File(sketchPath("")+"data.tsv"), "tsv"); // 13
  }
  catch(IOException iox) {
    println("Failed to write file." + iox.getMessage());
  }
}
```

We take the following steps to create a new `Table`, add points to it, and save those points into a file.

1. Create a new variable called `tsv` of type `Table`.
2. Create a `PVector` `ArrayList` called `points` to store the x and y location of a fingertip.
3. Try reading the `data.tsv` file from the Android sketch folder, if it exists.
4. Create the `tsv` `Table` object using a parameter. For the parameter, use Java's `File` class and Processing's `sketchPath()` for the file path, which the `Table` class will attempt to load—causing an exception that the file doesn't exist.
5. Catch the `java.io.FileNotFoundException`, which you can see in the console if the `data.tsv` file doesn't exist at the defined location.
6. Create a new `tsv` `Table` object without a parameter if it's the first time we run the sketch and the `data.tsv` file doesn't exist.
7. Parse the `tsv` `Table` object row by row and add a new `PVector` to our `points` `ArrayList` for every record in our `data.tsv` file. Do nothing if `getRowCount()` returns `0`.
8. Parse the `points` `ArrayList` and draw a five-pixel-wide and -high `ellipse()` for each item using the `PVector`'s `x` and `y` components for the x and y location.
9. Add a new `PVector` to the `points` `ArrayList` when a new `mouseDragged()` event is triggered.
10. Create a `String` array called `data` containing two `String` values using Java's `toString()` method to convert the `mouseX` and `mouseY` values into a text string.
11. Add a row to the `Table` object using the `addRow()` `Table` method.
12. Set the last row in the `Table` object to the `String` array `data`.

13. Using `writeTSV()`, write our `tsv` `Table` object to the `data.tsv` file inside our app at the `sketchPath()` location on the Android. Trigger the method using `keyPressed()`, which will detect if we press any menu button or key on the keyboard.
    We won't need to give the sketch permission to access the Android's internal storage because we are writing to the memory allocated for our app. Let's run the code.

Run the App

When the app starts up, use your finger and doodle on the screen. It leaves behind a trace of points drawn sixty times per second. Each time, we set the point position using `mouseX` and `mouseY` and add the point to our `PVector` array, making the point count go up.
Press `MENU` to save all point coordinates to internal storage. Now let's test whether we've written our data successfully by closing and restarting our app.
Press the `HOME` key to go back to the home screen. Now press and hold the `HOME` button to open the recent app screen, showing `DataWrite` alongside other apps you've launched recently.
To close the `DataWrite` app or any app that's running, swipe the app icon horizontally left or right, and it will close. Let's reopen the `DataWrite` app now to see if our points are still there by navigating to the apps installed on the device using the Apps button.
Reopen the app. The sketch launches, showing all the previously saved points we've doodled. Great, you've built an app that stored data on the Android device.

Now that you've learned how to write data to the app using a specified location in internal storage, it's time to explore how to share data with other apps using Android's external storage.

Write Data to External Storage

Building on our previous Data/DataWrite/DataWrite.pde, let's now make some modifications so we can write our data to the Android's external storage. This allows us to share files with other applications, as we've done when we worked with the camera and saved pictures to the external storage in Snap and Save Pictures. We can also copy our data to the desktop by mounting the device as USB mass storage.

The process of mounting the device as USB mass storage is inconsistent across devices and is manufacturer-specific. Some devices like the Nexus S offer to "Turn on USB storage" when you connect the device to the desktop via a USB cable. Other devices like the Galaxy S3 now require an app to launch the device as mass storage. Either way, Android devices typically

offer such a feature, and we'll take a look at the `data.tsv` file once we've created it on the external storage.

To work with the file path to the external storage, we need to `import` Android's `android.os.Environment` package, which will give us access to the `Environment` class and its `getExternalStorageDirectory()` method, including the file path method `getAbsolutePath()`. We use both methods to create the path `String` for our sketch, writing to and reading from `data.tsv` on the external storage.

Let's take a look at the code snippet showing `keyPressed()`, where we only modify our file path for writing `data.tsv` to the external storage. The path for reading `data.tsv` is, and must be, identical.

code/Data/DataWriteExternal/DataWriteExternal.pde

```
import android.os.Environment;

Table tsv;
ArrayList<PVector> points = new ArrayList<PVector>();

void setup()
{
  orientation(LANDSCAPE);
  noStroke();
  textSize(72);
  textAlign(CENTER);

  try {
    tsv = new Table(new File(
    Environment.getExternalStorageDirectory().getAbsolutePath() +
      "/data.tsv"));
  }
  catch (Exception e) {
    tsv = new Table();
  }
  for (int row = 1; row < tsv.getRowCount(); row++)
  {
    points.add(new PVector(tsv.getInt(row, 0), tsv.getInt(row, 1), 0));
  }
}

void draw()
{
  background(78, 93, 75);
  for (int i = 0; i < points.size(); i++)
  {
    PVector p = points.get(i);
    ellipse(p.x, p.y, 5, 5);
  }
  text("Number of points: " + points.size(), width/2, 100);
}

void keyPressed()
{
  try {
    tsv.save(new File(
    Environment.getExternalStorageDirectory().getAbsolutePath() + // 1
    "/data.tsv"), "tsv");
  }
  catch (IOException iox) {
    println("Failed to write file." + iox.getMessage());
  }
```

```
}

void mouseDragged()
{
  points.add(new PVector(mouseX, mouseY, 0));
  String[] data = {
    Integer.toString(mouseX), Integer.toString(mouseY)
    };
    tsv.addRow();
  tsv.setRow(tsv.getRowCount()-1, data);
}
```

---

1. Use Android's `Environment` method `getExternalStorageDirectory()` to get the name of the external storage directory on the Android device, and use `getAbsolutePath()` to get the absolute path to that directory. Work with that path as a parameter for Java's `File` object, providing a `File`-type parameter for Processing's `writeTSV()Table` method, used to write the actual `TSV` file.

   Let's test the app now.

Run the App

Before we can write to the external storage, we need to give the appropriate permission to do so in the Permissions Selector. Open the Android Permissions Selector, scroll to Write External Storage and check the permission box.

Now run the sketch on your device. It looks and behaves identically to the previous sketch shown in Figure 9.2 - Write data to an Android. Draw some points on the screen and save it by pressing any of the menu keys. The only difference here is that we save `data.tsv` now into the root of your Android's external storage directory.

Let's browse the external storage and look for our `data.tsv` file. Depending on your Android make and model, try unplugging the USB cable connecting your device and the desktop, and plug it back in. You should be prompted to "Turn on USB" storage. If this is the case, go ahead and confirm (on some devices, try browsing to Settings ↦ "More..." on the Android and look for a USB mass storage option. Alternatively, look for the USB mass storage process recommended by your device manufacturer).

When you turn on USB storage, the device lets you know that some apps will stop; go ahead and OK that. Now move over to your desktop computer and browse to your USB mass storage medium, often called `NO NAME` if you haven't renamed it. Click on your mass storage device, and right there in the root folder, find a file called `data.tsv`.

Check `data.tsv` by opening it in your favorite text editor. You'll find two columns there neatly separated by a tab; in each row, you'll find a pair of integer values. This is perfectly sufficient for our project. More complex data projects typically require a unique identifier for each row, a row in one table to point to a specific record in another. We'll look into this when we are in Chapter 10, Using SQLite Databases, later in this chapter.

Now that we've learned how to use `CSV` and `TSV` data stored on the Android device, let's explore how to load comma-separated values from a source hosted online in the next project.

Visualize Real-Time Earthquake Data

Let's create an app to track earthquakes, putting our newly acquired data skills to work on a nifty data visualization project. The objective of the project is to visualize the location and magnitude of all of the earthquakes that have been reported worldwide during the last hour. We'll use live data hosted on the U.S. Geological Survey Organization's website and visualize it as an animated map, shown in Figure 9.3. The `CSV` data format that we'll work with again is typically available on governmental sites, such as for the `USGS`or the US Census.
When we take a look at the text data source containing comma-separated values, we can see the following data. The file linked here is a sample of the live online source, saved on February 24, 2015, which we'll use as a fallback in case we don't have an Internet connection. The first row contains the field labels.

code/Data/DataEarthquakes/data/all_hour_2015-02-24.txt

```
time,latitude,longitude,depth,mag,magType,nst,gap,dmin,rms,net,id,updated,place,type
2015-02-24T21:43:49.600Z,19.4117,-
155.2845,9.4,1.9,ml,,39.5999968320002,0,0.21,hv,hv60858986,2015-02-24T21:49:39.000Z,"5km WSW
of Volcano, Hawaii",earthquake
2015-02-24T21:43:27.020Z,36.0663333,-
117.6928333,0.54,1.57,ml,30,58,0.07162,0.23,ci,ci37101903,2015-02-24T21:55:35.640Z,"23km E of
Coso Junction, California",earthquake
2015-02-24T21:41:38.560Z,34.0128333,-
117.2171667,16.86,1.88,ml,73,34,0.08989,0.3,ci,ci37101895,2015-02-24T21:52:27.900Z,"6km SE of
Loma Linda, California",earthquake
2015-02-24T21:39:55.690Z,37.3209991,-
122.105835,0.03,1.83,md,9,163,0.03779,0.1,nc,nc72400960,2015-02-24T21:55:04.419Z,"3km S of
Loyola, California",earthquake
2015-02-24T21:22:35.300Z,19.4045,-155.2823,7.6,2,Ml,,21.6,0.00898315,0.23,hv,hv60858971,2015-
02-24T21:28:46.169Z,"5km WSW of Volcano, Hawaii",earthquake
```

The actual data source we'll work with is hosted online:

`http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_hour.csv`

Follow the link in your browser, and you'll see the current live `CSV` file consisting of comma-separated values. We won't need to use all of the fields in order to visualize the location and magnitude of each earthquake, but we will need `latitude`, `longitude`, and `mag`.
To display the geographic location of each earthquake, we'll use an equirectangular projection world map,which stretches the globe into a rectangular format. This allows us to translate the longitude and latitude values for each earthquake into an x and y location that we can display on our device screen. Such a projection maps the longitude meridians to regularly spaced vertical lines and maps latitudes to regularly spaced horizontal lines. The constant intervals between parallel lines lets us overlay each earthquake's geolocation accurately in relation to the world map.

The map includes the complete range of longitude meridians from -180 to 180 degrees, but only a portion of the latitude degree spectrum—from -60 to 85 degrees instead of the usual -90 to 90 degrees. The poles are not included in the map, which are the most distorted portion of an equirectangular projection map. Because they are less populated and less frequently the source of earthquakes, they are also less relevant for our app, and we can use the map's pixel real estate for its more populated land masses.

To use our pixel real estate most effectively, we'll draw the world map full screen, covering the complete `width` and `height` of the Android screen and introducing some additional distortion to our data visualization due to the device's own aspect ratio. Because both the map and the location data scales depend on the display `width` and `height`, our information remains geographically accurate.

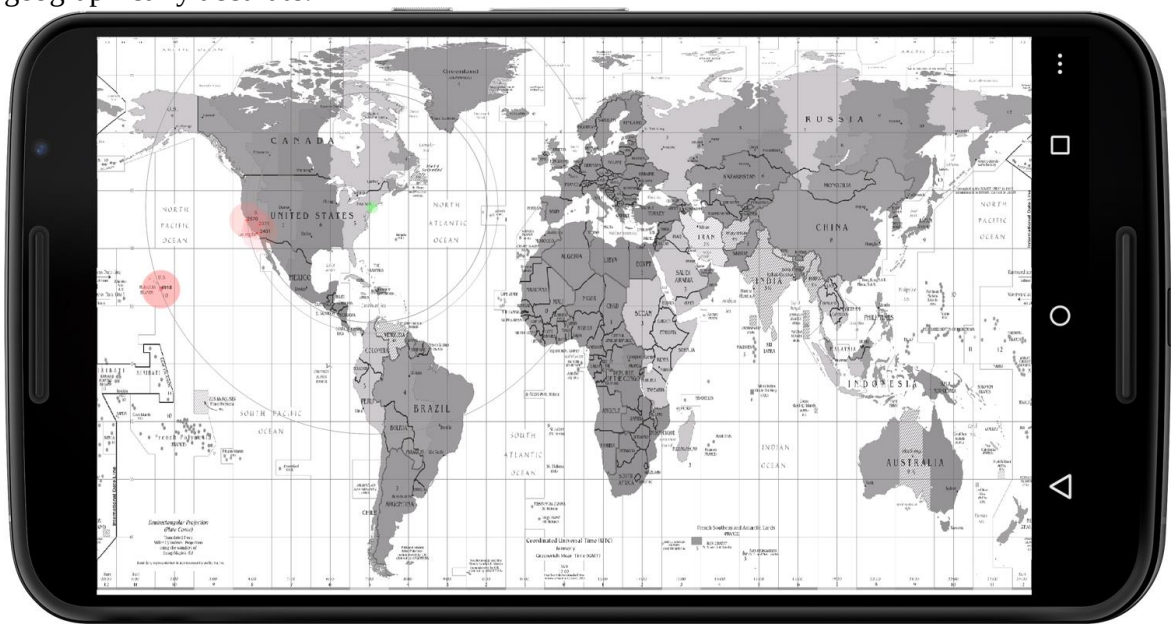


Figure 9.3 - Earthquakes reported worldwide during the last hour.

The device location is indicated by a green circle. Red circles indicate the locations of earthquakes reported within the hour—the size and pulse frequency indicate their magnitude.

Using a data file that is hosted online changes the way we load the file into Processing's `Table` class. Unlike our earlier examples, where we loaded the file from the Android's storage, we won't know ahead of time whether we can successfully connect to the file due to a very slow or an absent Internet connection, for instance. So we'll use the `try` `catch` construct we've seen in Data/DataWrite/DataWrite.pde, again to attempt loading from the online source. If it fails, catch the exception and load a data sample stored in our sketch's `data` folder as a fallback.
Let's take a look at the code.

code/Data/DataEarthquakes/DataEarthquakes.pde

```
import ketai.sensors.*;

Table earthquakes, delta;

int count;
PImage world;
String src =
  "http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_hour.csv";  // 1

void setup()
{
  location = new KetaiLocation(this);
  try {

    earthquakes = loadTable(src, "header, csv");          // 2
  }
  catch
    (Exception x) {
    println("Failed to open online stream reverting to local data");
    earthquakes = loadTable("all_hour_2015-02-24.txt", "header, csv");      // 3
  }
  count = earthquakes.getRowCount();
  println(count+" earthquakes found");

  orientation(LANDSCAPE);
  world = loadImage("worldMap.png");
}

void draw ()
{
  background(127);
  image(world, 0, 0, width, height);                      // 4

  for (int row = 0; row < count; row++)
  {
    float lon = earthquakes.getFloat(row, 2);             // 5
    float lat = earthquakes.getFloat(row, 1);             // 6
    float magnitude = earthquakes.getFloat(row, 4);       // 7
    float x = map(lon, -180, 180, 0, width);              // 8
    float y = map(lat, 85, -60, 0, height);               // 9
    noStroke();
    fill(255, 127, 127);
    ellipse(x, y, 10, 10);
    float dimension = map(magnitude, 0, 10, 0, 500);      // 10
    float freq = map(millis()%(1000/magnitude),
      0, 1000/magnitude, 0, magnitude*50);                // 11
    fill(255, 127, 127, freq);                            // 12
    ellipse(x, y, dimension, dimension);

    Location quake;
    quake = new Location("quake");                        // 13
    quake.setLongitude(lon);
    quake.setLatitude(lat);
    int distance = int(location.getLocation().distanceTo(quake)/1609.34);     // 14
    noFill();
    stroke(150);
    ellipse(myX, myY, dist(x, y, myX, myY)*2, dist(x, y, myX, myY)*2);        // 15
    fill(0);
    text(distance, x, y);                                 // 16
  }

  // Current Device location
  noStroke();
  float s = map(millis() % (100*accuracy*3.28), 0, 100*accuracy*3.28, 0, 127); // 16
  fill(127, 255, 127);
  ellipse(myX, myY, 10, 10);
  fill(127, 255, 127, 127-s);
  ellipse(myX, myY, s, s);
}
```

Here are the steps we need to take to load and visualize the data.

---

1. Define an `src` string containing the `URL` to the data source hosted online.
2. Load the data into the `earthquakes` `Table` object using the `header, csv` to indicate that we have a header row and use a comma-separated data structure.
3. Use the fallback local data source `all_hour_2015-02-24.txt` stored in the `data` folder of our sketch if the connection to the online source fails. The local file is a sample of the online source using the same data structure.
4. Draw the world map fullscreen over the screen's `width` and `height`, with the upper left image corner placed at the origin.
5. Get the longitude of the individual earthquake stored in each table row as a floating point number from the field index number `2`, which is the sixth column in our data source.
6. Get the latitude of the earthquake as a `float` from the field index number `1`.
7. Get the magnitude of the earthquake as a `float` from the field index number `4`.
8. Map the retrieved longitude value to the map's visual degree range (matching the Earth's degree range) of `-180..180`, and assign it to the horizontal position `x` on the screen.
9. Map the retrieved latitude value to the map's visual degree range of `85..-60`, and assign it to the vertical position `y` on the screen.
10. Map the dimension of the red circles visualizing the earthquakes based on their individual `magnitudes`.
11. Calculate a blink frequency for the red circles based on the milliseconds (`millis()`) passed since the app started, modulo 1000 milliseconds, resulting in a frequency of once per second, and then divide it by the earthquake's magnitude to blink faster for greater magnitudes.
12. Use the blink frequency variable `freq` to control the alpha transparency of the red ellipses with the `RGB`value `color(255, 127, 127)`.
13. Create a new Android `Location` object "quake," and set it to the latitude and longitude of the individual earthquake.
14. Calculate the distance of the current device `location` stored in the `KetaiLocation` object, and compare it to the `quake` location stored in the Android object. Divide the resulting number in meters by `1609.34`to convert it to miles.
15. Draw a circle with a gray outline indicating the distance of the current location to the earthquake. Use Processing's distance method `dist()` to calculate the distance between both points on the screen, and draw an ellipse with a width and a height of double that distance.
16. Draw a text label indicating the distance from the current device location to the earthquake at the position of the earthquake on the screen.
17. Draw a slowly animated green circle to indicate the current device's location on the map. The pulse rate is one second for a 100-foot accuracy, or 0.1 seconds for a 10-foot accuracy.

Let's look at the `Location` tab next, which includes all the necessary code to determine the location of our device. It's very similar to our Geolocation/Geolocation/Geolocation.pde.

code/Data/DataEarthquakes/Location.pde

```
// Location Code

KetaiLocation location;                                    // 1

float accuracy;
float myX, myY;                                            // 2

void onLocationEvent(double lat, double lon, double alt, float acc)
{
  myX = map((float)lon, -180, 180, 0, width);              // 3
  myY = map((float)lat, 85, -60, 0, height);               // 4
  accuracy = acc;                                          // 5
  println("Current Longitude: " + lon + " Longitude: " + lat);
}
```

Here's what we need to do to determine our device location.

---

1. Create a `KetaiLocation` variable named `location`.
2. Create two floating point number variables to store the x and y position of the device relative to the world map so we can use it in `draw()`.
3. Map the `lon` value we receive from the Location Manager relative to the screen width.
4. Map the `lat` value we receive from the Location Manager relative to the screen height.
5. Assign the accuracy value we receive from the Location Manager to the global `accuracy` variable, and use it for the blink rate of the green ellipse indicating the current device's location.
   Let's test the app now.

Run the App

We need to make sure we set the correct Android permissions again to run this sketch. Not only do we need to select `INTERNET` from the Android Permission Selector under Sketch Permissions, we also need to check `ACCESS_COARSE_LOCATION` at least, or if we want to know it more accurately we check `ACCESS_FINE_LOCATION`as well. A couple of hundred feet matter less in this application, so the Fine Location is optional.
Run the sketch on your device. When it starts up, the app will try to connect to the data source online. If your device doesn't have a connection to the Internet, it will catch the exception ("Failed to open online stream reverting to local data") and load a sample stored as a fallback inside our data folder.

Your device might not have an updated coarse location available, so it might take a couple of seconds until the green circle moves into the correct location on the world map; the gray

concentric circles are tied to the (green) device location and indicate the distance to each individual earthquake.

Try Another Source

Try another source from the `USGS`'s data feed, where you can find `CSV` files containing other earthquake data using the same file structure we've seen earlier in the Data/DataEarthquakes/data/all_hour_2015-02-24.txt.
Replace the `src` text string with this `URL`:
http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_week.csv

Now rerun the code. Looking at the seven-day-period visualization, you can see how vibrant our planet is, even though we've limited the scope of the application to earthquakes of magnitude 2.5 and higher. In terms of their physical impact, experts say earthquakes of magnitude 3 or lower are almost imperceptible, while earthquakes of magnitude 7 and higher can cause serious damage over large areas.

Because the comma-separated data structure of this seven-day-period data file is identical to the one we used earlier in the last hour, we don't have to do anything else besides replace the source `URL`. The app loads the additional data rows containing the earthquake data and displays it independent of how many earthquakes are reported, as shown in the image below.

Figure 9.4 - Earthquakes reported worldwide during the last seven days.

Only earthquakes with a magnitude of 2.5 and higher are included in this data source.

Let’s now refine the earthquake app using a feature that we’re quite familiar with—device vibration.

Add Vibes to the Earthquake App

How much we know and care about earthquakes and the alert systems that are used to warn people about them probably depends on where you live and what kind of incident history your area has. Clearly the earthquake app we’ve developed so far has educational value and is not designed as a warning system. Wouldn’t it be great, though, if we could keep the app running and receive a notification when a new earthquake incident is reported? Maybe, but we have neither the time nor the attention span to keep looking at our device screen, so let’s use a very familiar feedback device built into our Android, the tiny `DC` motor that makes it vibrate.
We can take this quite literal translation from Earth’s vibrations to device vibrations quite far, as we can control the duration of each device’s vibrations and can even use a vibration pattern. So let’s refine our earthquake app by mapping each individual earthquake magnitude to an individual vibration duration and the number of earthquakes to the number of vibrations. Furthermore, we can continue to check the online data source for reports of new earthquakes and vibrate the device each time a new one appears. We won’t modify the visual elements of the app any further, but we’ll focus on manipulating the audio-tactile response from the vibration motor built into the device to give us the effect we want.

To refine our app in this way, we can work with `KetaiVibrate`, which gives us straightforward access to the device’s vibration motor. We’ll also need an additional Processing `Table` object so we can compare our data to the data received from the live data source and add new quakes to the `earthquakes` `Table` when we determine they have occurred. Let’s take a look at the code, focusing on the `vibrate()` and `update()` methods that provide the functionality we’re looking for. Besides the main tab, we’ll use the `Location` tab we’ve seen already in the previous iteration of the app in the Data/DataEarthquakes/Location.pde.

code/Data/DataEarthquakesShake/DataEarthquakesShake.pde

```
import ketai.sensors.*;
import ketai.ui.*;

Table history;
PImage world;
String src =
  "http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_hour.csv";
KetaiVibrate motor;                                          // 1
int lastCheck;

void setup()
{
  location = new KetaiLocation(this);
  try {
    history = loadTable(src, "header, csv");
```

```
  }
  catch
    (Exception x) {
    println("Failed to open online stream reverting to local data");
    history = loadTable("all_week_2015-02-24.txt", "header, csv");
  }
  orientation(LANDSCAPE);
  world = loadImage("worldMap.png");
  lastCheck = millis();

  motor = new KetaiVibrate(this);                          // 2
}

void draw () {
  background(127);
  image(world, 0, 0, width, height);

  if (history.getRowCount() > 0) {
    for (int row = 0; row < history.getRowCount(); row++) {
      float lon = history.getFloat(row, 2);
      float lat = history.getFloat(row, 1);
      float magnitude = history.getFloat(row, 4);
      float x = map(lon, -180, 180, 0, width);
      float y = map(lat, 85, -60, 0, height);

      noStroke();
      fill(0);
      ellipse(x, y, 10, 10);
      float dimension = map(magnitude, 0, 10, 0, 500);
      float freq = map(millis()%(1000/magnitude),
      0, 1000/magnitude, 0, magnitude*50);
      fill(255, 127, 127, freq);
      ellipse(x, y, dimension, dimension);

      Location quake;
      quake = new Location("quake");
      quake.setLongitude(lon);
      quake.setLatitude(lat);
      int distance = int(location.getLocation().distanceTo(quake)/1609.34);

      noFill();
      stroke(150);
      ellipse(myX, myY, dist(x, y, myX, myY)*2, dist(x, y, myX, myY)*2);
      fill(0);
      text(distance, x, y);
    }
  }
  // Current Device location
  noStroke();
  float s = map(millis() % (100*accuracy*3.28), 0, 100*accuracy*3.28, 0, 127);
  fill(127, 255, 127);
  ellipse(myX, myY, 10, 10);
  fill(127, 255, 127, 127-s);
  ellipse(myX, myY, s, s);

  if (millis() > lastCheck + 10000) {                      // 3
    lastCheck = millis();
    update();
  }
}

void vibrate(long[] pattern)
{
  if (motor.hasVibrator())                                 // 4
    motor.vibrate(pattern, -1);                            // 5
  else
    println("No vibration service available on this device");
}

void update()
```

```
{
  println(history.getRowCount() + " rows in table before update" );

  ArrayList<Float> magnitudes = new ArrayList<Float>();
  Table earthquakes;

  try {
    earthquakes = loadTable(src);
  }
  catch
    (Exception x) {
    println("Failed to open online stream, reverting to local data");
    earthquakes = loadTable("all_week_2015-02-24.txt", "header, csv");
  }

  if (earthquakes.getRowCount() > 1)

    for (int i = 1; i < earthquakes.getRowCount(); i++)   // 6
    {
      if (findInTable(history, 1, earthquakes.getString(i, 1)))
      {
        continue;
      }
      String[] rowString = earthquakes.getStringRow(i);
      history.addRow();                                   // 7
      history.setRow(history.getRowCount()-1, rowString); // 8
      //Magnitude field is number 6
      Float magnitude = new Float(earthquakes.getFloat(i, 6));  // 9
      magnitudes.add(magnitude);
      println("adding earthquake: " + earthquakes.getString(i, 1));
    }

  long[] pattern = new long[2*magnitudes.size()];         // 10

  int j = 0;
  for (int k=0; k < pattern.length;)
  {
    pattern[k++] = 500;
    pattern[k++] = (long)(magnitudes.get(j) * 100);
    j++;
  }

  motor.vibrate(pattern, -1);                             // 11
  println(history.getRowCount() + " rows in table after update" );
}

boolean findInTable(Table t, int col, String needle) {    // 12
  for (int k=0; k < t.getRowCount(); k++) {
    if (needle.compareTo(t.getString(k, col)) == 0)       // 13
      return true;
  }
  return false;
}
```

Let’s take a look at the modifications we need to make to frequent updates and add vibration feedback to the earthquake app.

---

1. Create a `motor` variable of type `KetaiVibrate`.
2. Create the `KetaiVibrate` object `motor`.
3. Check if the ten-second interval has expired.
4. Check if the device has a vibration motor built in and available.

5. Vibrate the device motor using the `KetaiVibrate` method `vibrate()` using the vibration `pattern` as the first parameter and “no-repeat” (`-1`) as the second parameter.
6. Iterate through the `earthquakes` table; determine the number of rows contained in it using `getRowCount()`.
7. Add a new row into the `history`.
8. Set the new row in the `history` table to the new entry found in the `earthquakes` table.
9. Get the magnitude of the new entry we’ve found.

10. Create an array of `long` numbers to be parsed into the `vibrate()` method as a duration pattern.
11. Call the `vibrate` method using the `pattern` we’ve assembled as an array of `long` numbers.
12. Create a `boolean` custom method to iterate through a table and find a specific `String` entry we call `needle`. Use the table name, the column index number we are searching in, and the `needle` as parameters for the method.
13. Iterate through the table `t` contents in column `col`, and compare the entry to the `needle`. Return `true` if we find a matching entry, and `false` if we don’t.
    Let’s test the sketch.

Run the App

Run the sketch on your device. Visually, everything looks familiar. Every ten seconds the app is checking back to the `EPA`’s server for updates. When the app detects a new earthquake, you’ll hear the device vibrate briefly. A 2.5-magnitude earthquake results in a quarter-second vibration. If you receive two or more updates, you’ll hear two or more vibrations.
When earthquakes disappear from the data source hosted by the `EPA`, we still keep them in our `historyTable`. So for as long as the app is running, we’ll accumulate earthquake records, and we show them all collectively on the world map.
This completes our earthquake app as well as our investigation in comma- and tab-separated data structures.

Wrapping Up

This concludes our examination of databases and tables, a highly relevant subject when developing mobile applications. You’ll be able to read and write data to the Android into private and public directories and work with comma- and tab-separated values. This will allow you to save settings and application states so you can get your users started where you left off.

For more complex projects, where our interactions with the data become more complicated and our questions more detailed, we might need to work with a database, which gives us the

crucial features to search, sort, and query the data we are working with. The most commonly used local database management system on mobile devices is SQLite, which we'll learn about in the next chapter.

10. Using SQLite Databases

In this second part of our introduction to data, we'll work with SQLite, the popular relational database management system for local clients such as the Android, used also by many browsers and operating systems to store data. It implements the popular Structured Query Language (`SQL`) syntax for database queries, which we can use to access data stored locally on our Android device.

SQLite is a fairly simple and fast system, is considered very reliable, and has a small footprint that can be embedded in larger programs. It offers less fine-grained control over access to data than other systems like PostgreSQL or MySQL does, but it is simpler to use and administer, which is the main objective of the technology. It works very well as a file format for applications like Computer-Aided Design (`CAD`), financial software, and record keeping. It is often used in cellphones, tablet computers, set-top boxes, and appliances because SQLite does not require administration or maintenance. This simple database management system can be used instead of disk files, like the tab- or comma-delimited text files we've worked with in Chapter 9, Working with Data, replacing them with ad-hoc SQLite disk files.

In this chapter, we'll first get SQLite running with a simple sketch and learn how to use `SQL` queries to retrieve data from a SQLite table. Then we'll create an app that uses SQLite to capture accelerometer data from the sensor built into the Android. We'll use the recorded sensor values to create a time series visualization of the data. Finally, we'll query the data set we've recorded based on a certain device orientation we are looking for, and we'll highlight the sensor value that matches our query criteria.

Let's take a look at the classes and methods that allow us to use SQLitedatabases for more complex data-driven apps.

Working with SQLite Databases

Now that we've seen most of Processing's `Table` features, it's time we take a look at the widely used SQLite database management system for local clients. It is based on the the popular Structured Query Language syntax for database queries and will look very familiar if you've worked with `SQL` before. Ketai gives us access to Android's `SQLiteDatabase` class and provides us with the essential methods we need to create, query, and update content in the database tables.

The Ketai `KetaiSQLite` class is what we need to create full-fledged local SQLite databases on the device. For the projects in this chapter, we'll use it to store a number of points that we'll create by tapping the touch screen interface, and later we'll use it to record accelerometer sensor data using the `KetaiSensor` class we've seen in Chapter 3, Using Motion and Position Sensors. Let's get started by taking a look at the relevant Processing and Ketai methods we'll be working with throughout the chapter.

We'll create two SQLite projects, one to get us up and running with a few random values in a SQLite database. The next project will take advantage of the `KetaiSensor` class to capture accelerometer data directly into a SQLite database, which we'll browse and visualize on the Android display.

For the SQLite app we'll create in this chapter, we'll discuss `SQL` queries only very briefly. If you are unfamiliar with the language or would like to explore `SQL` queries further later on, you can find a more thorough reference online for the statements outlined next.

Let's take a look at the `KetaiSQLite` class and SQLight basics.

Working with the KetaiSQLite Class

To use SQLite on the Android, we'll work with the following `KetaiSQLite` methods.

`KetaiSQLite`	Ketai class for working with SQLite databases—it can be used to create a KetaiSQLite database or to load an existing database.
`execute()`	KetaiSQLite method for executing a SQLite query to a database, which doesn't return data
`query()`	KetaiSQLite method for sending a SQLite query to a database, returning data
`getRecordCount()`	KetaiSQLite method returning all records in a specified table, using the table name as parameter
`getDataCount()`	KetaiSQLite method returning all records in a database across all tables

Now let's take a look at the most important declarative `SQL` we'll use in our database project.

`CREATE`	`SQL` statement for creating a table in a database with specified fields and data types
`INSERT`	`SQL` statement for creating a new row in a table
`SELECT`	`SQL` statement to query a database, returning zero or more rows of data

`WHERE`	expressions
*	`SQL` wildcard that stands for “all” in a query
`INTEGER`, `TEXT`	`SQL` field data types
`PRIMARY KEY`	`SQL` statement to make a field the primary key
`AUTOINCREMENT`	`SQL` statement to make an integer field, which is typically also the primary key to automatically increment by one in order to be unique

Now let’s get started with our first `KetaiSQLite` database.

Implement a Data Table in SQLite

In the next project, we’ll create a simple record-keeping sketch using the SQLite data management system, which can store a list of individual names and the IDs associated with them. Using text strings for the names we store and integer values for the associated IDs allows us to explore two different data types within the database. There is essentially no limit to the number of entries we can add to the SQLite table, besides the usual memory restrictions we have in Android’s internal storage.

The goal of this SQLite project is to familiarize ourselves with the steps we need to follow to create a SQLite database, a new table inside the database, and data entries inside that table. To see whether we are successful, we’ll output the contents of the table to the Android screen as shown in the image below. We’ll use the `KetaiSQLite` class for this project and the remaining projects of this chapter.

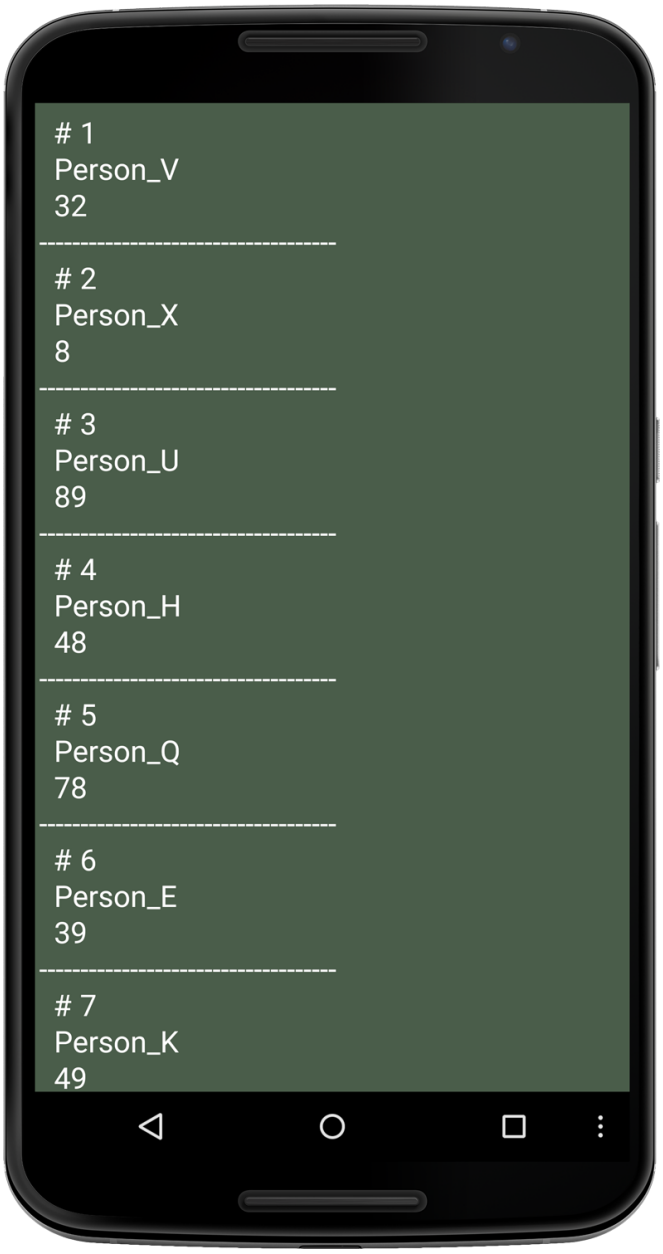


Figure 10.0 - Working with a SQLiteDatabase.

This screen output shows the initial five entries inserted in the data table. Each record contains a unique ID assigned by the database management system shown with a # prefix, a random name, and a random age.

To implement this SQLite sketch we'll first create a `KetaiSQLite` object, then create a new table called `data`using a `CREATE TABLE` statement, after that, `INSERT` data into the table, and finally, `SELECT` all the table contents to output it on the device screen. This is the most concise way to complete the necessary steps when we are working with a database.
To keep it simple, we'll populate the table only with values for two fields we call `name` and `age`. Besides those two fields, we'll also implement a field called `_id`, which is good practice: we should always use this to provide a unique identifier for each record. To keep each ID unique, we'll use `SQL`'s `AUTOINCREMENT` feature, which takes care of

incrementing the numeric integer ID each time we add a new record to the table. This ensures that all table rows have a number assigned to it, each unique throughout the table.

To create five sample entries inside the `data` SQLite table, we'll use a `for()` loop to generate a random `name` text `String` and a random `age` for the respective fields.

Note that we are neither using our familiar `setup()` nor the `draw()` method for this sketch. We don't need them here because we are writing a static sketch that executes our statements to create, populate, and display the database content, and that's it. In all those cases, we need to use `setup()` and `draw()` as we did in all the sketches we've developed so far.

Let's take a look at the code.

code/SQLite/SQLite/SQLite.pde

```
import ketai.data.*;

KetaiSQLite db;                                          // 1
String output = "";
String CREATE_DB_SQL = "CREATE TABLE data (" +
"id INTEGER PRIMARY KEY AUTOINCREMENT, " +
"name TEXT, age INTEGER)";
orientation(PORTRAIT);
textSize(72);
db = new KetaiSQLite( this);                             // 2

if ( db.connect() ) {
  if (!db.tableExists("data"))                           // 3
    db.execute(CREATE_DB_SQL);                           // 4
  for (int i=0; i < 5; i++)                              // 5
    if (!db.execute(
        "INSERT into data (`name`,`age`) " +             // 6
        "VALUES ('Person_" + (char)random(65, 91) +
        "', '" + (int)random(100) + "' )"
        )
       )
       println("error w/sql insert");
  println("data count for data table after insert: " +
    db.getRecordCount("data"));                           // 7

  // read all in table "table_one"
  db.query( "SELECT * FROM data" );                       // 8
  while (db.next ())                                      // 9
  {
    output +="-----------------------------------\n";
    output += "  # " + db.getString("id") + "\n";         // 10
    output += "  " +db.getString("name") + "\n";
    output += "  " +db.getInt("age") + "\n";
  }
}
background(78, 93, 75);
text(output, 10, 10);                                     // 11
```

We need to take the following steps to implement our SQLite database table.

---

1. Define a `KetaiSQLite` variable called `db`.
2. Create the `db` object of type `KetaiSQLite`.

3. Check if the `data` table exists from a prior session; if not, create a new `data` table as defined in the query string `CREATE_DB_SQL` using three fields of type `INTEGER`, `TEXT`, and `INTEGER` for the respective fields `_id`, `name`, and `age`.
4. Execute the `SQL` query to create the `data` table using the `KetaiSQLite` method `execute()`.
5. Loop five times to create five initial entries in the `data` table.
6. `INSERT` values for the `data` table into the `name` and `age` fields using a random character suffix to make the "Person_" string unique (characters 65..90 represent A..Z in the ASCII character table) and using a random integer number ranging 0..99 for `age`.
7. Get the record count for the `data` table using `KetaiSQLite`'s `getRecordCount()` method.
8. Send a query to the database requesting all (`*`) records from the `data` table.
9. Parse all records in the table using a while loop until there are no more to be found.
10. Create an `output` `String` containing all records in the `data`.
11. Show the `output` text to give us feedback on the `data` table contents.

Let's run the sketch now.

Run the App

Run the sketch on your device. You'll see five records similar to Figure 10.0. To retrieve the individual entries of each record, we use the `getString()` and `getInt()` methods, which take the table's field names as parameters. If we use a field name that doesn't exist, the `getString()` and `getInt()` methods will return `0`. You can check this out by adding the following line of code to the `output` string.

```
output += db.getInt("foo") + "\n"; //doesn't exist, so we get '0'
```

If you are interested in working with an existing SQLite database, you can use the `KetaiSQLite` class to load it from the sketch `data` folder. You'd load the `example.sqlite` database as shown in the following code snippet.

```
KetaiSQLite db; KetaiSQLite.load(this, "example.sqlite", "example"); db = new
KetaiSQLite(this, "example");
```

You are now able to work with SQLite databases on the Android, which also lets you explore aspiring data-driven projects. Let's put the new skills in practice by creating a sketch that lets us record accelerometer sensor data to a SQLite database.

Record Sensor Data into a SQLite Database

To see how useful a database can be, let's go one step further and create an app that lets us record sensor data directly into a SQLite database table. We'll then use `SQL` queries to browse the sensor data we've recorded from the accelerometer and visualize the data on the Android

screen as a time series. A time series plots data points recorded at fixed intervals. For our example, we will record a data point every time we receive a new accelerometer value. Alongside the accelerometer sensor values x, y, and z, we'll record time as Unix time (measured in milliseconds since January 1, 1970 UTC) using Android's `System` method, `currentTimeMillis()`. This allows us to identify precisely at what time (and date) the data has been captured. The Unix time stamp will also serve as the unique ID in our `data` table. So for our `data` table, we'll need the following table structure and data types created by the following `SQL` query.

```
CREATE TABLE data ( time INTEGER PRIMARY KEY, x FLOAT, y FLOAT, z FLOAT)
```

Let's look at each part of the `SQL` statement separately.

`CREATE TABLE`	The `SQL` keyword to create a table
`data`	The name we give the created table
`timeINTEGERPRIMARY KEY`	Defines the first `time` field we'll create with the datatype `INTEGER`—the `id` field also functions as the `PRIMARY KEY` for the table. In a database that can use multiple tables that relate to each other, the primary key uniquely identifies each record in the table.
`x`	A field of type `FLOAT` that we use to store the value reported from the accelerometer's *x*-axis
`y`	A field of type `FLOAT` that we use to store the value reported from the accelerometer's *y*-axis
`z`	A field of type `FLOAT` that we use to store the value reported from the accelerometer's *z*-axis

Here's our approach to visualizing the time series data from the accelerometer sensor.

To display our time series on the screen, we'll work with a pair of variables called `plotX` and `plotY`, taking each of our data points and mapping it to the correct horizontal and vertical positions on the screen. We calculate `plotX` by using the record counter `i` to determine the total number of entries. We then use this number to spread the collected data over the full display `width`. We determine the vertical position `plotY`for each point by mapping each x, y, and z sensor value in relation to the display `height`.

Because the device reports a value equal to `1` g when it rests on the table (g-force equals 9.81 m/s^2, as we know from Display Values from the Accelerometer), let's use `2` g as the assumed maximum so we can move and shake the device and still show those higher values on the screen. Values of `0` g are shown centered vertically on the screen; positive values plot in the upper half of the display, and negative values in the bottom half.

Let's take a look at the code.

code/SQLite/DataCapture/DataCapture.pde

```
import ketai.data.*;
import ketai.sensors.*;

KetaiSensor sensor;
KetaiSQLite db;
boolean isCapturing = false;
float G = 9.80665;
String CREATE_DB_SQL =
  "CREATE TABLE data ( time INTEGER PRIMARY KEY, x FLOAT, y FLOAT, z FLOAT);";  // 1

void setup() {
  db = new KetaiSQLite(this);
  sensor = new KetaiSensor(this);
  orientation(LANDSCAPE);
  textAlign(LEFT);
  textSize(72);
  rectMode(CENTER);
  frameRate(15);

  if ( db.connect() ) {                                    // 2
    if (!db.tableExists("data"))                           // 3
      db.execute(CREATE_DB_SQL);                           // 4
  }
  sensor.start();
}

void draw() {
  background(78, 93, 75);
  if (isCapturing)
    text("Recording data...\n(tap screen to stop)" +"\n\n" +
      "Current Data count: " + db.getDataCount(), width/2, height/2);  // 5
  else

    plotData();                                            // 6
}

void keyPressed() {
    if (keyCode == BACK) {                                 // 7
      db.execute( "DELETE FROM data" );
    }
    else if (keyCode == MENU) {                            // 8
      if (isCapturing)
        isCapturing = false;
      else
        isCapturing = true;
    }
}

void onAccelerometerEvent(float x, float y, float z, long time, int accuracy) {
  if (db.connect() && isCapturing) {
    if (!db.execute(
      "INSERT into data (`time`,`x`,`y`,`z`) VALUES ('" +  // 9
      System.currentTimeMillis() + "', '" + x + "', '" + y + "', '" + z + "')"
      )
    )
      println("Failed to record data!" );                  // 10
  }
}
void plotData() {
  if (db.connect()) {
    pushStyle();
    line(0, height/2, width, height/2);
    line(mouseX, 0, mouseX, height);
    noStroke();
    db.query("SELECT * FROM data ORDER BY time DESC");     // 11
    int  i = 0;
```

```
    while (db.next ())                                      // 12
    {
      float x = db.getFloat("x");                           // 13
      float y = db.getFloat("y");
      float z = db.getFloat("z");
      float plotX, plotY = 0;

      fill(255, 0, 0);
      plotX = map(i, 0, db.getDataCount(), 0, width);
      plotY = map(x, -2*G, 2*G, 0, height);
      ellipse(plotX, plotY, 3, 3);                          // 14
      if (abs(mouseX-plotX) < 1)
        text(nfp(x, 2, 2), plotX, plotY);                   // 15

      fill(0, 255, 0);
      plotY = map(y, -2*G, 2*G, 0, height);
      ellipse(plotX, plotY, 3, 3);                          // 16
      if (abs(mouseX-plotX) < 1)
        text(nfp(y, 2, 2), plotX, plotY);

      fill(0, 0, 255);
      plotY = map(z, -2*G, 2*G, 0, height);
      ellipse(plotX, plotY, 3, 3);                          // 17
      if (abs(mouseX-plotX) < 1)
      {
        text(nfp(z, 2, 2), plotX, plotY);
        fill(0);
        text("#" + i, mouseX, height);
      }
      i++;
    }
    popStyle();
  }
}
```

Let’s take a look at the steps we need to take to implement the sketch.

---

1. Define a `String` called `CREATE_DB_SQL` containing the `SQL` query to create a new table called `data`. Use four columns, or fields (called `time`, `x`, `y`, and `z`) to store sensor data. Associate the data type `INTEGER` with the `time` field and make it the `PRIMARY KEY`, and use the datatype `FLOAT` for the sensor axis.
2. Connect to the SQLite database that we’ve associated with the sketch when we created `db`.
3. Create the `data` table if it doesn’t already exist in the SQLite database, using the `CREATE_DB_SQLString` we’ve declared earlier.
4. Send the table query `CREATE_DB_SQL` to the database using `execute()`, a method that doesn’t return values but just executes a query.
5. Give some feedback on the display while we are recording data points.
6. Call the custom function `plotData()`, taking care of the data visualization.
7. If the `DELETE` key is pressed, erase all content in the `data` table using a `DELETE` query, which leaves the table structure intact.
8. Capture a `MENU` key event and use it to start and stop recording data.
9. Use an `INSERT` `SQL` query to add a record into the `data` table every time we receive a sensor value via `onAccelerometerEvent()`. Use the

Android `System` method `currentTimeMillis()` to request the current UTC time in milliseconds.

10. If the insertion of the new record fails, print an error message to the console.
11. Use a `SELECT` `SQL` statement to request all entries from the `data` table, sorting them by the `UTC` field in descending order.
12. Use `while()` to parse through the `data` table for as long as the `next()` method returns `TRUE` and there are more entries to browse.
13. Get the x-axis value from the record using the `getFloat()` method with the field name `x` as the parameter. Do the same for the y- and z-axes.
14. Draw a red `ellipse()` at the location's horizontal position `plotX`.
15. Show the `x` value in a text label aligned with the ellipse if the horizontal position of the ellipse matches the horizontal finger position. Use the same approach for the y- and z-axes later on.
16. Draw the y-axis values using the same approach we took for x but with a green fill color.
17. Draw the z-axis values using the same approach we took for x and y but using a blue fill color.

Now let's run the app.

Run the App

Run the sketch on the device. When you run it for the first time, the SQLite `data` table will be created first. Press the menu key on the device to start recording accelerometer data. While you record sensor data, you'll see the record count increase. Press Menu again to stop the recording process.

You'll see a screen output similar to the image below, showing the x-, y-, and z-axis values scattered as red, green, and blue dots around the vertical center representing `0`. Positive g-force values are shown on the top half of the display, and negative values on the bottom.

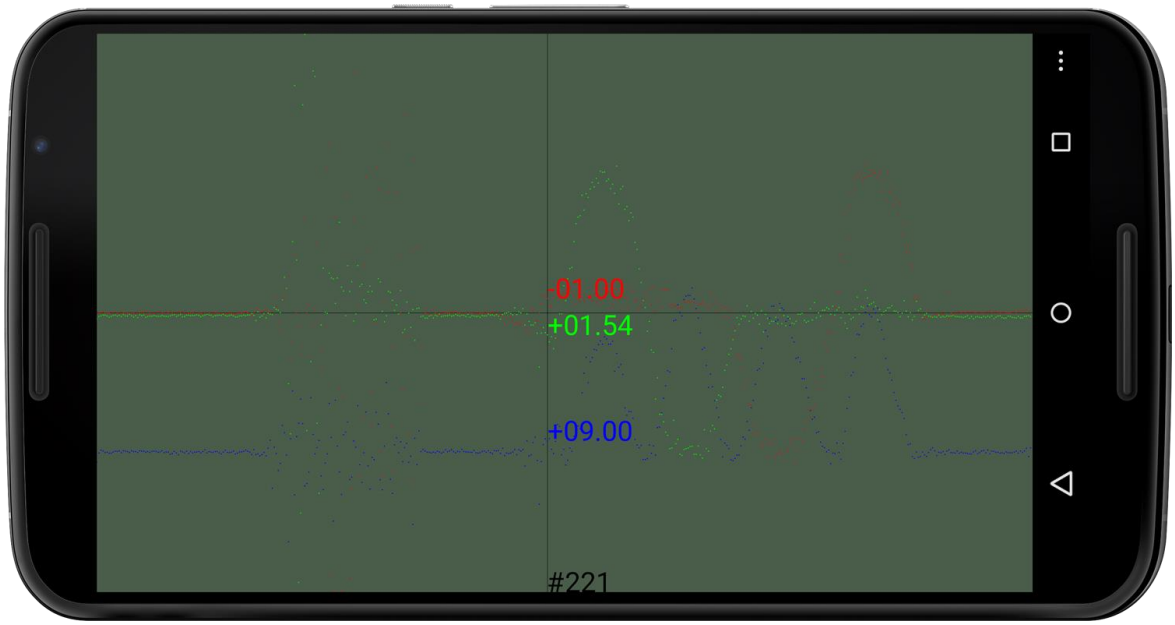


Figure 10.1 - Capturing sensor data in SQLite.

We visualize the recorded x (red), y (green), and z (blue) accelerometer values using SQLite entries. On the left we see increasing device shakes and rest, and on the right, continuous device rotation.

You can continue adding points to the database by continuing to periodically press the Recent button. Once you reach a couple hundred points, you will note that it takes a bit longer to write and read the data. Our sensor reports values so fast that we record a couple hundred values into our SQLite database in just a few seconds. And because our database is stored on the device's SD card, the more values we've captured, the longer it will take to write.

You can reduce the number of entries into our database, for example, by recording only every other sensor value in `onAccelerometerEvent()`. Depending on your application, you can also write to the database at a reduced time interval. If you'd like to erase all data from the database, press the Back button. It triggers a `DELETE` query so we can start from scratch.
The process of creating tables and inserting and selecting data is in principle the same for any SQLite database project. Whether you have four or forty fields, a hundred or a thousand rows, this project can serve as a template for many of your data-driven Android projects.

Besides working with all the data in our database table, there is another important aspect of a database we need to explore—selecting data using a condition and returning only values that match that condition, which brings us to our next section.

Refine SQLite Results Using WHERE Clauses

Since we recorded data into a SQLite database, we can do much more with the data than parsing our `data`row by row. For example, `KetaiSQLite` methods can help us get the minimum

and maximum values of a particular field in our table. This comes in very handy when we're displaying time-series graphs and want to distribute our data points evenly across a display in a way that takes full advantage of the available pixel real estate of each Android device.
Conditional `SQL` queries using `WHERE` clauses allow us to search table records that match a particular value we specify in our clause. It returns all rows for which the `WHERE` clause is `true`. `WHERE` clauses are often used with one of the following operators: `=`, `<>`, `>`, `>=`, `<`, `<=`, or `LIKE`.
For instance, adding `WHERE x > 5` to our `SELECT` statement will only return records that have values greater than `5` in the `x` field of our `data` table. Similarly, we could request from the `data` table in our previous sketch SQLite/SQLite/SQLite.pde, only the `name` of a person older than `age 21`. This way we can quickly implement many of the user-driven interactions we know from searching an online store for books only by a particular author, or shopping for merchandise from a particular brand.
Let's explore `WHERE` clauses based on the code we've just worked on to visualize sensor data stored in our SQLite database table we've called `data`. We'll leave the structure of the sketch intact but add a query that uses a `WHERE` clause to find only those records that match our condition.
As a condition for our `WHERE` clause, let's look for all the records that indicate the device is resting flat on the table (display pointing up). This is only the case if the z-axis shows a value of approximately `1` g, or `+9.81`, while the x- and y-axis values are close to `0`. Let's use a white circle to indicate data points that match our condition, as shown in Figure 10.2, below.

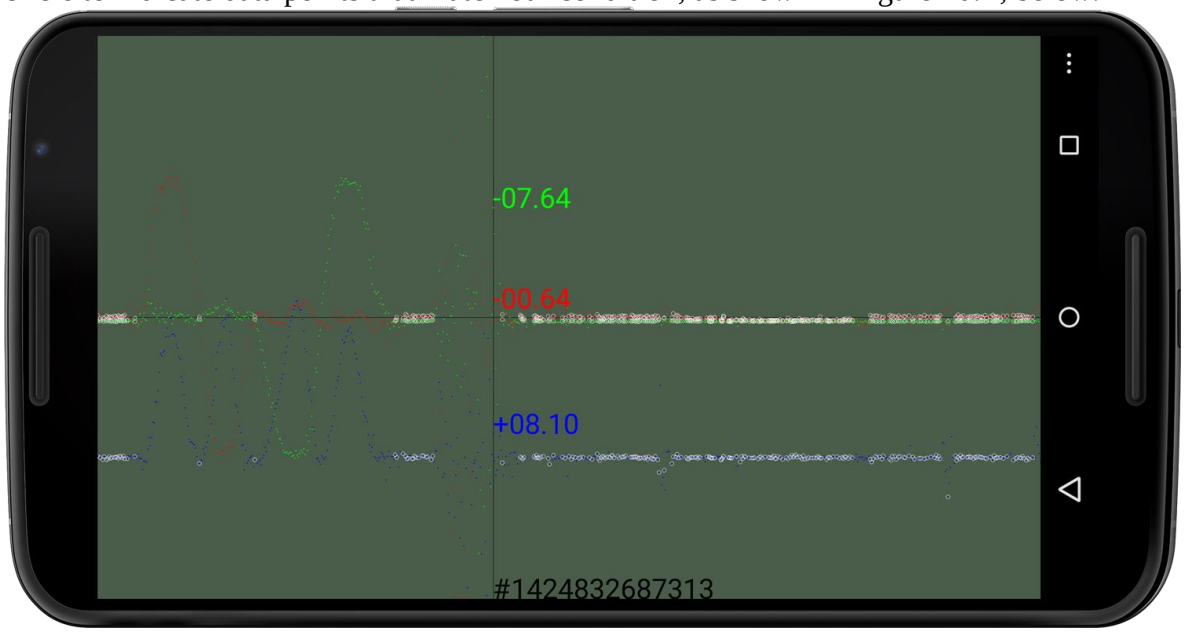


Figure 10.2 - Refining SQL queries using WHERE clauses.

Data points recorded when the Android remains flat and still are highlighted via white circles. The number on the bottom of the screen indicates the Unix time when the record was created.

We add the `WHERE` clause to the previous `SELECT` statement we've used. We are allowing values differing up to `1` m/s^2 from the value we are looking for.

```
SELECT * FROM data WHERE z > 9.5 AND abs(x) < 0.3 AND abs(y) < 0.3
```

Instead of the scheme we used in our previous sketch, let's specify time using the Unix time stored in the `time` field of the `data` table. This is more accurate because we don't receive sensor updates at an exact interval. Instead, we use the exact moment we've received new values in the form of a time stamp, and we use this time stamp to plot the data exactly when it occurred on our horizontal time axis. Essentially we are plotting each recorded data point proportional to the lowest (right) and highest (left) recorded time stamp. To correctly represent a Unix time value, we'll need thirteen digits. For that level of precision, we'll use the Java long datatype, a datatype that can handle long integers. The `map()` method we've used throughout the book is not designed to handle such large integer values, so we simply build our helper method `getLong()` based on Processing's `map()` algorithm.

Let's look at the `plotData()` and `mapLong()` methods we are now working with, building on the SQLite/DataCapture/DataCapture.pde.

code/SQLite/DataCaptureClause/DataCaptureClause.pde

```
import ketai.data.*;
import ketai.sensors.*;

KetaiSensor sensor;
KetaiSQLite db;
boolean isCapturing = false;
float G = 9.80665;

String CREATE_DB_SQL =
  "CREATE TABLE data ( time INTEGER PRIMARY KEY, x FLOAT, y FLOAT, z FLOAT);";

void setup()
{
  db = new KetaiSQLite(this);
  sensor = new KetaiSensor(this);

  orientation(LANDSCAPE);
  textAlign(CENTER, CENTER);
  textSize(72);
  rectMode(CENTER);
  frameRate(15);
  if ( db.connect() )
  {
    if (!db.tableExists("data"))
      db.execute(CREATE_DB_SQL);
  }
  sensor.start();
}

void draw() {
  background(78, 93, 75);
  if (isCapturing)
    text("Recording data...\n(press MENU button to stop)" +"\n\n" +
      "Current Data count: " + db.getDataCount(), width/2, height/2);
  else
    plotData();
}
```

```
void keyPressed() {
  if (keyCode == BACK) {
    db.execute( "DELETE FROM data" );
    key = 0;
  }
  else if (keyCode == MENU) {
    if (isCapturing)
      isCapturing = false;
    else
      isCapturing = true;
  }
}

void onAccelerometerEvent(float x, float y, float z, long time, int accuracy)
{
  if (db.connect() && isCapturing)
  {
    if (!db.execute(
      "INSERT into data (`time`,`x`,`y`,`z`) VALUES ('" +
      System.currentTimeMillis() + "', '" + x + "', '" + y + "', '" + z + "')"
      )
    )
      println("Failed to record data!" );
  }
}

void plotData()
{
  if (db.connect())
  {
    pushStyle();
    textAlign(LEFT);
    line(0, height/2, width, height/2);
    line(mouseX, 0, mouseX, height);
    noStroke();
    db.query("SELECT * FROM data");
    long myMin = Long.parseLong(db.getFieldMin("data", "time")); // 1
    long myMax = Long.parseLong(db.getFieldMax("data", "time")); // 2
    while (db.next ())
    {

      long t = db.getLong("time");                          // 3
      float x = db.getFloat("x");
      float y = db.getFloat("y");
      float z = db.getFloat("z");
      float plotX = 0;
      float plotY = 0;

      fill(255, 0, 0);
      plotX = mapLong(t, myMin, myMax, 0, width);         // 4
      plotY = map(x, -2*G, 2*G, 0, height);
      ellipse(plotX, plotY, 3, 3);
      if (abs(mouseX-plotX) < 1)
        text(nfp(x, 2, 2), plotX, plotY);

      fill(0, 255, 0);
      plotY = map(y, -2*G, 2*G, 0, height);
      ellipse(plotX, plotY, 3, 3);
      if (abs(mouseX-plotX) < 1)
        text(nfp(y, 2, 2), plotX, plotY);

      fill(0, 0, 255);
      plotY = map(z, -2*G, 2*G, 0, height);
      ellipse(plotX, plotY, 3, 3);
      if (abs(mouseX-plotX) < 1)
      {
        text(nfp(z, 2, 2), plotX, plotY);
        fill(0);
        text("#" + t, mouseX, height);                    // 4
      }
```

```
    }
    noFill();
    stroke(255);
    db.query("SELECT * FROM data WHERE z > 9.5 AND abs(x) < 0.3 AND abs(y) < 0.3"); // 5
    while (db.next ())
    {
      long t = db.getLong("time");
      float x = db.getFloat("x");
      float y = db.getFloat("y");
      float z = db.getFloat("z");
      float plotX, plotY = 0;

      plotX = mapLong(t, myMin, myMax, 0, width);
      plotY = map(x, -2*G, 2*G, 0, height);
      ellipse(plotX, plotY, 10, 10);                        // 5

      plotY = map(y, -2*G, 2*G, 0, height);
      ellipse(plotX, plotY, 10, 10);

      plotY = map(z, -2*G, 2*G, 0, height);
      ellipse(plotX, plotY, 10, 10);
    }
    popStyle();
  }
}

// map() helper method for values of type long
float mapLong(long value, long istart, long istop, float ostart, float ostop) { // 6
 return (float)(ostart + (ostop - ostart) * (value - istart) / (istop - istart));
}
```

Now let's see what changes we've made to our previous sketch SQLite/DataCapture/DataCapture.pde.

---

1. Get the minimum value of the `time` field in the `data` table using `KetaiSQLite`'s `getFieldMin()`method. Use the datatype `long` to hold the returned thirteen-digit Unix time value.
2. Get the maximum value of the `time` field in the `data` table using `getFieldMax()`. Use the datatype `long` to hold the time value.
3. Parse the data, including the `time` field, which contains a `long` value that we store in the variable `t`.
4. Calculate the `plotX` position of our data point based on the `time` value stored in `data`.
5. Draw a text label for the data point's Unix time stamp.
6. Use a `WHERE` clause that only returns results where the condition is `true`.
7. Draw a white circle around the data points that match the device's rest state.
8. Define the user-defined method `mapLong()` as working identically to `map()` but handling values of datatype `long`.

   Now let's test our sketch.

Run the App

Run the sketch on the device, and you'll see it start up empty again, as no database table `data` exists in this modified sketch with a new name. When you start recording using your menu key, make sure to lay the device flat and let it rest in that position for a moment before you move it around a bit.

Now stop recording data. You'll see that the data points recorded at rest are highlighted by a white ring. Those data points that don't match our condition remain without a highlight. This means that our `SQL` query did the job, returning only the records that match z-axis values greater than `+9.5` and x- and y-axis values smaller than `0.3` m/s^2.

You can try other `WHERE` clauses and see the same data highlighted differently, depending on the conditions and expressions you've used.

Working with SQLite databases is certainly not limited to sensor data. The tables we create can contain any type of numbers and text strings, making the projects in this chapter an ideal template to explore your other data-driven app ideas.

Wrapping Up

Working with local SQLite databases, you can now also develop your aspiring data-driven applications and take advantage of the powerful Structured Query Language to filter, search, and sort your queries. Being able to work with data will allow you to improve all the projects we've worked on in this book, as well as your future projects, and help make your apps more usable and useful.

Now that we've learned how to work with the Android file system and databases, we are now ready to work with 3D graphics and with linked assets such as objects, materials, and textures.

11. Introducing 3D Graphics with OpenGL

Rich graphics are the staple of sophisticated mobile games and data visualizations, and recent Android phones are well equipped with the necessary graphics hardware to deliver 2D and 3D graphics rendering without degrading performance. When we play 3D games or interact with complex data visualizations, the geometry that composes such a scene must be redrawn a few dozen times per second on the device screen&—ideally sixty times or more&—for animations and interactions to appear smooth and fluent. Besides the geometry, which consists of points, lines, and polygons, we typically also work in a 3D scene with textures, lighting, and virtual cameras to control the appearance of shapes and objects and to change our perspective within the scene.

+

All Android phones and tablets sold today support `OpenGL ES`, a lightweight implementation of the popular Open Graphics Library for embedded systems&—the industry standard for developing interactive 2D and 3D graphics applications across platforms. It's a free application programming interface and graphics pipeline that allows the software applications we create to leverage the graphics hardware built into our desktop computers, game consoles, and mobile devices for better graphics performance.

Processing is a great environment to create sophisticated graphics, and it comes with an `OpenGL` library that can be used in all modes. On the Android, Processing takes advantage of the graphics processing unit, or `GPU`, built into the mobile device. Only hardware acceleration makes it possible to animate thousands of data points, text characters, polygons, image textures, and lighting effects while maintaining a sufficiently high frame rate. Despite the multicore `CPU`s built into the latest Android devices today, we rely on the graphics hardware and `OpenGL` to achieve the smooth animations and special effects we are looking for.

In this chapter we'll create basic 3D shapes, use lights and textures, and manipulate the viewpoint of a virtual camera. We'll create apps that employ 3D cubes and spheres and use different types of lighting. We'll texture a sphere with a NASA image of the Earth at night and superimpose our camera preview as texture onto a 3D box. We'll also render a large amount of text with custom lighting, and we'll position it dynamically on the screen using our familiar touch screen input. We'll continue our investigation into shapes and 3D objects in Chapter 12, Working with Shapes and 3D Object, as the second part of our introduction to 3D graphics.

Let's get started by taking a closer look at `OpenGL` in Processing.

Introducing 3D Graphics and OpenGL

The Open Graphics Library (`OpenGL`) is the industry standard API for creating 3D (and 2D) computer graphics and runs on most platforms. It can be used to create complex 3D scenes from graphic primitives such as points, lines, and polygons. One of `OpenGL`'s most important features is that it provides an interface for communicating with the accelerator hardware, or `GPU`, typically built into computer graphic cards, game consoles, and mobile devices.
The `GPU` is responsible for providing us with the frame rates we need to animate geometry, render complex textures and shades, and calculate lighting effects. If a particular device hardware does not support all of `OpenGL`'s feature sets, the library uses software emulation via the `CPU`instead, allowing `OpenGL` to still run the application on most platforms but at a lower frame rate.
Andres Colubri has now integrated his GLGraphics library into Processing 2.0, providing us with a major upgrade to the P2D and P3D graphics renderer in Processing, with exciting new features for creating cutting-edge graphics that make use of `OpenGL` hardware acceleration. When we use the `OpenGL` renderer in Processing 2.0, we can gain a high level of control over textures, image filters, 3D models, and `GLSL`shaders—`OpenGL`'s shading language for rendering effects on graphics hardware. Now we can also create groups of shapes using Processing's `createShape()` method and retain geometry in the `GPU`'s memory for significantly improved graphics performance. If you'd like to preview your sketch in the emulator using `OpenGL` hardware acceleration, you need to add the `GPU` emulation hardware property when you create your Android Virtual Device (`AVD`).
Let's take a look at the methods we'll use to work with the 3D graphics and `OpenGL` classes and methods we'll use in this chapter.

- `size(width, height, MODE)` Defines the dimension of the display window in pixels, followed by an optional render mode parameter. By default, the renderer is set to `2D`. Use `3D` to draw 3D shapes. You should call `size()` only once in `setup()` as the first line of code.
- P3D and OPENGL Identical render modes defined in the `size()` method&—the `OPENGL` renderer used to differ from `P3D`, but it is now an alias pointing to the `P3D` renderer. `P3D` allows us to use the z-axis as the third dimension in our sketch, oriented perpendicular to the device screen. Larger z values move objects further into the scene, negative z values toward us. The `P3D` renderer uses OpenGL hardware.
- `displayHeight` A Processing constant that returns the current height of the device display in pixels
- `displayWidth` A Processing constant that returns the current width of the device display in pixels

Let's get started with basic 3D geometry and lighting.

Work with 3D Primitives and Lights

For our first 3D app, let's explore simple geometry and lights to get a feel for the 3D coordinate system and learn how to control the position and scale of 3D objects on the device's display. We'll get started with two 3D primitives, the `sphere()` and the `box()`, and also define the scene lights using an ambient and a directional light source, as illustrated in Figure 11.0.

To create the 3D scene, we use the `size()` method, which we've used so far only for the desktop applications we've created. The method defines the window `width` and `height`, but it can also be used to replace the default Processing 2D (`P2D`) renderer. For the Android apps we've created so far, we haven't used `size()`, because if we don't use it, Processing defaults the app to full screen, which is standard for all applications on the Android.

For our 3D scene, we need to switch from the default `P2D` to the `P3D` renderer. Both Processing renderers are very similar, but the `P2D` render has optimized smoothing settings to display 2D figures. If we use the `size()` method with two parameters, our sketch defaults to the `P2D` renderer. If we provide a third one for the render mode, `P3D`, we've created a 3D sketch.

Processing also provides us with two constants to retrieve the display width and height of the device we are running our app on. They're called `displayWidth` and `displayHeight`, and we can also use those constants to set our app to full screen via the `size()` method.

Working with Lights

The different types of virtual light sources available in Processing are defined by their direction, amount of falloff, and specular values. Falloff defines the falloff rate for lights due to increased distance to the object they illuminate, defined in Processing as a constant, linear, or quadratic parameter using the `lightFalloff()`method. A specular light is the highlight that appears on shiny objects, defined by the `lightSpecular()`method, which provides a more realistic 3D impression based on how specular light interacts with material surfaces. The material appearance of objects can be defined via the `specular()`method as well as via the `shininess()` and `emissive()` methods.

- `lights()` Sets the default values for ambient light, directional light, falloff, and specular values, making the 3D objects in the scene appear lit with medium ambient light and look dimensional.
- `ambientLight()` Adds an ambient light to the scene, which is a type typically used in a 3D scene alongside directional light sources&—it makes objects appear evenly lit from all sides. The method uses three parameters for the light color and three parameters for the position of the light.

- `directionalLight()` Adds a directional light to the scene coming from a defined direction—it illuminates an object more where the light hits perpendicular to the surface and less so at smaller angles. The method uses three parameters for the light color and three parameters for its direction.
- [`pointLight()` Adds a point light to the scene; an omnidirectional light source emitting light from one point&—it uses the light color and position within the scene as parameters.
- `spotLight()` Adds a spotlight to the scene, which offers the most control through parameters including the light color, position, direction, angle of the spotlight cone, and the concentration of the spotlight

Create a 3D Scene

We'll use the `lights()` method for this project, which sets default light values for the scene. The default values for ambient light are then set to `ambientLight(128, 128, 128)`, defining a medium bright white ambient light. In addition, we'll use the touch screen interface to control a directional light, where we translate the finger position on the display into a direction for this directional light, allowing us to change the objects' illumination interactively.

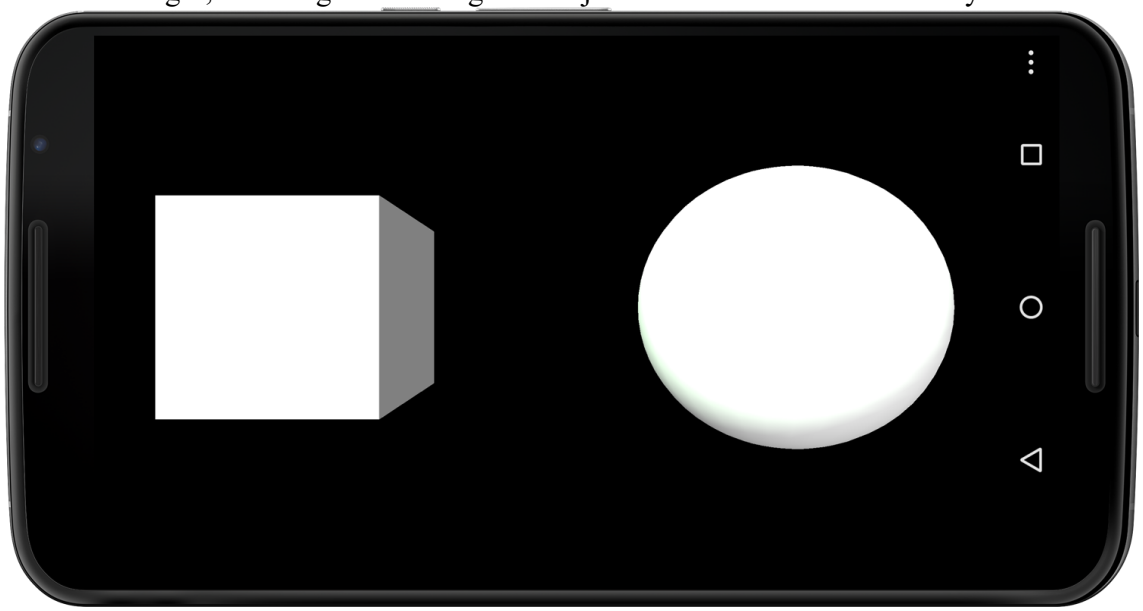


Figure 11.0 - Using 3D primitives and lights.

The illustration shows a cube and a sphere illuminated by two light sources, a default ambient light and a directional light source, that you control with your fingertip.

Let's take a look at our program, which is fairly concise for this basic scene.

code/Mobile3D/PrimitivesLights/PrimitivesLights.pde

```
void setup() {
  size(displayWidth, displayHeight, P3D);                 // 1
```

```
  orientation(LANDSCAPE);
  noStroke();
}
void draw()
{
  background(0);
  float lightX = map(mouseX, 0, width, 1, -1);            // 2
  float lightY = map(mouseY, 0, height, 1, -1);           // 3

  lights();                                               // 4
  directionalLight(200, 255, 200, lightX, lightY, -1);    // 5

  translate(width/4, height/2, 0);
  box(height/3);
  translate(width/2, 0, 0);
  sphere(height/4);
}
```

Let's take a look at the methods we use to place the 3D primitives and control the directional light.

---

1. Turn on the Processing 3D renderer using the `size()` method. Define the current device width as the app `width`using the `displayWidth` constant, and define the app `height` using the `displayHeight` constant.
2. Calculate the horizontal direction of the light source by mapping the `width` of the screen to values with a range of [-1..1].
3. Calculate the horizontal direction of the light source by mapping the `height` of the screen to values with a range of [-1..1].
4. Call the default lights for the scene using default lighting values. Use the `lights()` method within `draw()` to retain the default lights in the scene instead of `setup()`, where only default lighting values are set.
5. Use a directional light source with a greenish hue, and direct it based on the horizontal and vertical position of a fingertip on the touch screen.

Let's test the app.

Run the App

Now run the sketch on the device. This time we don't need to set Android permissions or import any libraries because we are not using hardware that requires authorization and it's all part of Processing's core functionality. When the app starts up, move your finger across the screen and see the directional light change direction based on your input.

A greenish halo around the specular highlights comes from our colored directional light source, and because we also have the default ambient light at work through calling the `lights()` method, the objects in our scene do not appear completely dark on the opposite side of the directional light source.

Now that we've created a 3D scene using basic 3D primitives and lights, we are ready to work with textures that we can superimpose onto 3D geometry.

Apply an Image Texture

For this next project we'll render a 3D night view of the Earth using a NASA image texture showing the light concentrations that emanate from the urban centers of our planet. We'll use the `NASA` `JPEG` image as a texture wrapped around a 3D `sphere()` shown in Figure 11.1 below. This time we'll create the 3D primitive using Processing's `PShape` class so we are able to apply a texture onto that shape. We'll create the sphere with the `createShape()` method and a `SPHERE` parameter, and then we'll use the `shape()` method to display the 3D object on the device screen.

The NASA satellite image of the Earth seen at night is also an equirectangular projection, as we used already in Visualize Real-Time Earthquake Data, for visualizing earthquakes. An image texture with such a projection stretches perfectly around our sphere, recompensating for the distortions toward the poles that we observe in the flattened `JPEG` image.

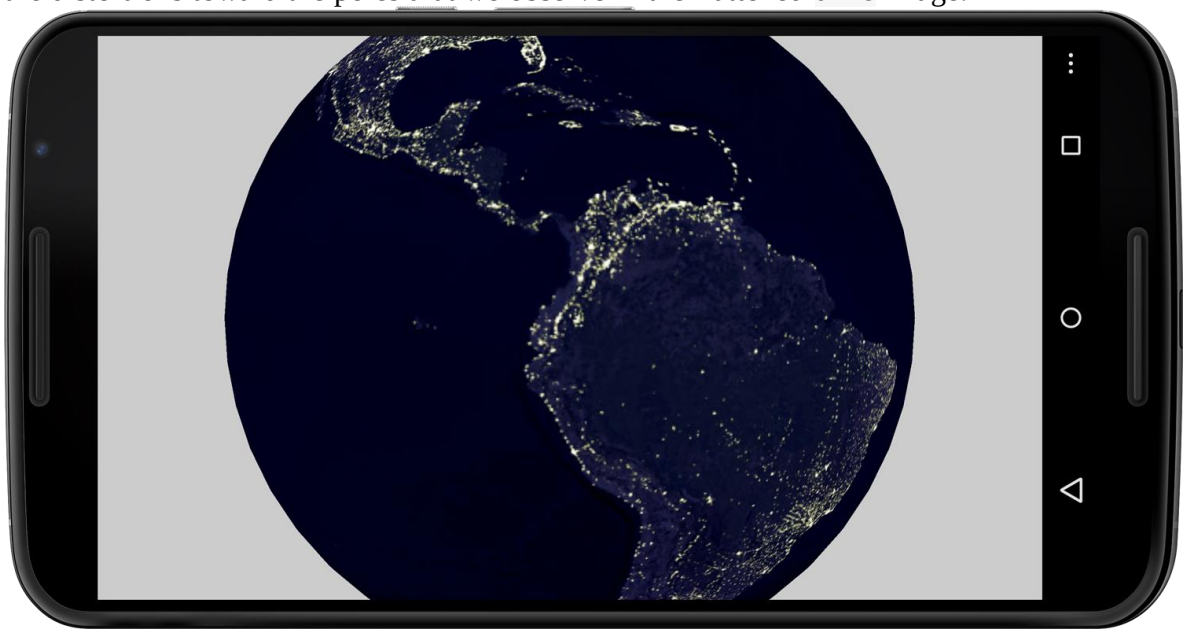


Figure 11.1 - Applying an image texture to a sphere.

The bright spots on the image that covers the sphere show urban centers&—such as New York City and Rio de Janeiro&—in North and South America.

code/Mobile3D/TexturedSphere/TexturedSphere.pde

```
PShape sphereShape;                                     // 1
PImage sphereTexture;                                   // 2

void setup() {
  size(displayWidth,  displayHeight,  P3D);
  orientation(LANDSCAPE);
```

```
  noStroke();
  fill(255);

  sphereTexture = loadImage("earth_lights.jpg");          // 3
  sphereShape = createShape(SPHERE, height/2);            // 4
  sphereShape.setTexture(sphereTexture);                  // 5
}

void draw() {
  translate(width/2, height/2, 0);
  rotateY(TWO_PI * frameCount / 600);                     // 6
  shape(sphereShape);
}
```

Let's take a look at the code.

Here's what we need to do to apply the image texture.

---

1. Create a `PShape` variable called `sphereShape`.
2. Create a `PImage` variable called `sphereTexture`.
3. Load the `JPEG` image texture using `loadImage()`.
4. Create the `SPHERE` `PShape` object we'll use to render the Earth, with a size of one-third the device's screen height.
5. Apply the image texture to the shape using the `PShape` `setTexture()` method.
6. Rotate the sphere slowly on the spot around the y-axis at a rate of one revolution per second. Use the `frameCount`constant to calculate the rotation.

Now let's run the sketch.

Run the App

Run the sketch on the device and you'll see a sphere covered with our NASA image texture on the screen. It covers one-third the screen height and rotates slowly around its vertical axis. We don't control this scene interactively, but we can watch the Earth rotate and we can observe the bright spots where densely populated cities are located.

Now that we've learned how to use static images as textures for 3D objects, we can move on to a moving image texture, which we'll discuss in the next section.

Use the Camera Preview as 3D Texture

Let's map a live camera preview of our front-facing Android camera next. When we use images as textures for 3D objects in Processing, we can take advantage of the fact that moving images are handled essentially like static images, displayed as a `PImage` object but updated every time we receive a new image from the camera. Building on the previous image texture project Apply an Image Texture, we can use the Android camera previews as textures via

the `KetaiCamera` class we've worked with in Display a Back-Facing Camera Full-Screen Preview.

For this project we'll use the `BOX` 3D primitive instead of the `SPHERE`, and we'll map the camera preview on every face of the box, as shown in Figure 11.2 below. We are already familiar with the code to instantiate a `KetaiCamera` object and the methods to start and stop the camera, which we'll reuse from Display a Back-Facing Camera Full-Screen Preview.

Figure 11.2 - Use a camera preview as a 3D texture.

The camera preview image is mapped on every face of the 3D box as an image texture.

Now let's take a look at the parts of the code that deal with the 3D box and camera texture.

code/Mobile3D/CameraTexture/CameraTexture.pde

```
import ketai.camera.*;
KetaiCamera cam;
PShape boxShape;
PImage sphereTexture;

void setup() {
  size(displayWidth, displayHeight, P3D);
  orientation(LANDSCAPE);
  noStroke();
  fill(255);

  boxShape = createShape(BOX, height/2);

  cam = new KetaiCamera(this, 640, 480, 30);              // 1
  cam.setCameraID(1);                                     // 2
}

void draw() {
  background(0);

  translate(width/2, height/2, 0);
  rotateY(PI * frameCount / 500);
  shape(boxShape);
}
```

```
void onCameraPreviewEvent()
{
  cam.read();
  boxShape.setTexture(cam);                                  // 3
}

void mousePressed()
{
  if (cam.isStarted())
  {
    cam.stop();
  } else
  {
    cam.start();
  }
}
```

Here's what we need to do to apply the camera preview as a shape texture.

---

1. Create a `KetaiCamera` object `cam` with a resolution of `320` by `240` pixels.
2. Set the camera ID to `1` for the front-facing camera.
3. Apply the `cam` image as a `setTexture()` for the `boxShape`.

Let's test the sketch.

Run the App

Run the sketch on your Android device. When the scene starts up, you'll see the 3D box rotate once every ten seconds. Tap the screen now to start up the camera. As soon as a camera preview image is available, we use it as a texture for our box.

Feel free to change the `BOX` back to a `SPHERE` 3D primitive and observe how the image wraps around the sphere.
Let's now take a closer look at the different types of light sources we can work with in Processing.

Work with Spot and Point Lights

Lighting affects the appearance of all the geometry in a scene, which is why we'll take a closer look now at the various lighting options we have in Processing. Let's create a sketch using three colored spotlights, as shown in Figure 11.3, where we can see how each of the light sources interacts with the surface of the 3D geometry.

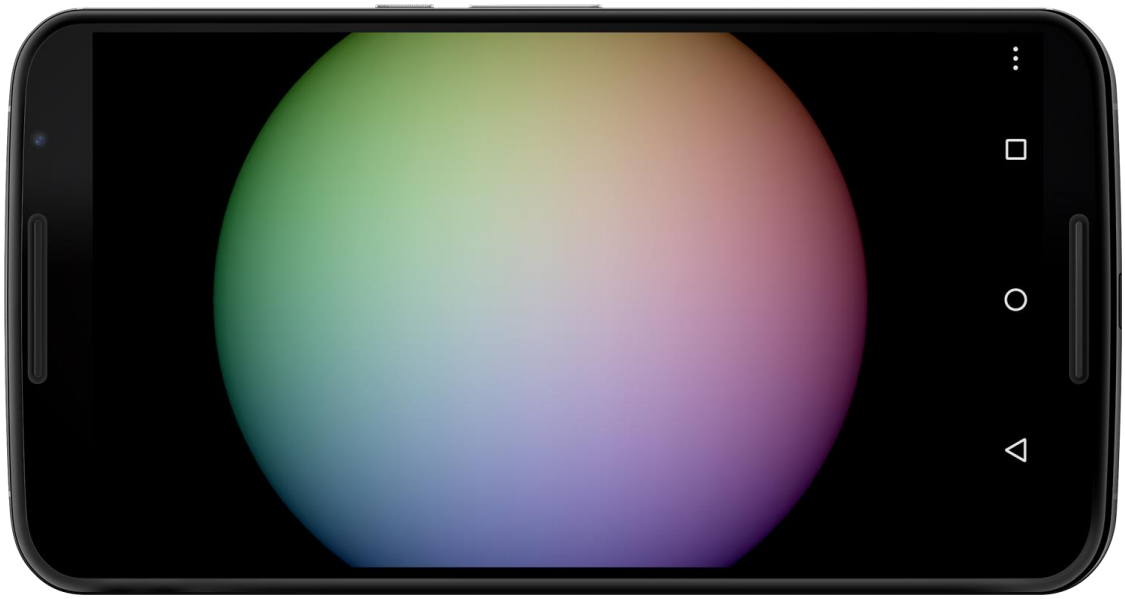


Figure 11.3 - Using spotlights.

The three colored spotlights introduced to the scene (red, green, blue) add up to white in the additive color space.

We'll use the basic light colors red, green, and blue, because those colors mixed together create white light in the additive color space. We'll keep the geometry simple and continue working with our basic sphere shape. But to get a more accurate sphere geometry, we'll increase the render detail of the sphere using the `sphereDetail()` method. It defines the number of vertices of the sphere mesh, set by default to `30` vertices per full `360` circle revolution, and we increase it to `60` vertices per revolution, resulting in one vertex every six degrees (`360` degrees divided by `60` vertices).

The `spotLight()` method we'll use takes eleven parameters and offers the most amount of control compared with other light options in Processing. We can set the light color, position, direction, angle, and concentration using the method. Let's take a look.

```
spotLight(v1, v2, v3, x, y, z, nx, ny, nz, angle, concentration)
```

`v1`	The red or hue value of the light—red in the default `RGB` color mode and hue in the `HSB` color mode
`v2`	The green or saturation value of the light
`v3`	The blue or brightness value of the light
`x`	The horizontal or *x* position of the light
`y`	The vertical or *y* position of the light
`z`	The depth or *z* position of the light

`nx`	The direction of the light along the *x*-axis
`ny`	The direction of the light along the *y*-axis
`nz`	The direction of the light along the *z*-axis
`angle`	The angle of the light cone
`concentration`	The concentration exponent determining the center bias of the spotlight cone

Every object we draw after calling this or any other lighting method in Processing is affected by that light source; objects drawn before the method call are unfazed by the light source. To retain a light source in the scene, we must call the lighting method within `draw()`. If we call the lighting method in `setup()`, the light will only affect the first frame of our app and not the consecutive frames.

We'll place our three colored spotlights slightly off-center while pointing straight ahead at the scene. Each of the lights will hit our sphere object at an individual spot off the sphere's center, and we can observe how the three spotlights interact with the sphere's surface when they mix together. We'll define the spotlight cone angle at `15` degrees, which happens to match a standard lens used in theater lighting, and we'll keep the concentration bias at `0` to achieve the maximum blending effect between the colors for now.

Once we've tested the spotlights, we'll replace them with point lights and compare the difference. Point lights offer less options and control than spotlights do, which makes them arguably easier to use as well. Point lights are omnidirectional light sources emanating equally in all directions from one specified point in 3D space. The `pointLight()` method takes only the light color and position as parameters.

Let's go ahead and write our spotlight program next.

code/Mobile3D/SpotLights/SpotLights.pde

```
PShape sphereShape;
int sSize;

void setup()
{
  size(displayWidth, displayHeight, P3D);
  orientation(LANDSCAPE);
  noStroke();
  fill(204);
  sphereDetail(60);                                    // 1
  sSize = height/2;                                    // 2
  sphereShape = createShape(SPHERE, sSize);
}

void draw()
{
  background(0);
```

```
  translate(width/2, height/2, 0);

  spotLight(255, 0, 0, sSize/4, -sSize/4, 2*sSize, 0, 0, -1, radians(15), 0);   // 3
  spotLight(0, 255, 0, -sSize/4, -sSize/4, 2*sSize, 0, 0, -1, radians(15), 0);  // 4
  spotLight(0, 0, 255, 0, sSize/4, 2*sSize, 0, 0, -1, radians(15), 0);          // 5

//  pointLight(255, 0, 0, sSize/4, -sSize/4, 2*sSize);
//  pointLight(0, 255, 0, -sSize/4, -sSize/4, 2*sSize);
//  pointLight(0, 0, 255, 0, sSize/4, 2*sSize);

  shape(sphereShape);
}
```

Let's take a look at the methods we use for defining the spotlight and sphere detail.

---

1. Increase the number of vertices composing the sphere to `60` for one full `360`-degree revolution.
2. Define a variable for the sphere size called `sSize`.
3. Create a red spotlight pointing straight at the scene, slightly offset to the right and up.
4. Create a green spotlight pointing straight at the scene, slightly offset to the left and up.
5. Create a blue spotlight pointing straight at the scene, slightly offset to the bottom.

Let's test the sketch.

Run the App

Now run the sketch on the device, and see how the increased sphere detail and the three spotlights produce a high-quality color blending effect on the sphere's surface.

Let's take a look at our current frame rate to see how computationally expensive this operation is. Go ahead and add these two lines of code at the end of `draw()`:

```
 if (frameCount%10 == 0) println(frameRate);
```

On the Nexus S, the slowest of the devices tested for this book (), we still get minimum frame rates of `60`, which is the default Processing uses if we don't determine otherwise via the `frameRate()` method. This is true despite the fact that we have multiple hundreds of vertices and polygons at work to create our sphere with increased detail and we superimpose multiple light sources onto the sphere surface.
Let's replace the spotlights now with point light sources and see the difference. They are already present in the Mobile3D/SpotLights/SpotLights.pde, but currently commented out. The color and position of the point light is identical to the spotlight we've used.

Rerun the app and take a look at how the three colored light sources cover the sphere—mixing white in the center of the sphere. The lighting seems identical to the spotlights since we've used a fairly wide spotlight cone earlier and we also did not add a concentration bias. Now change back to the spotlights and decrease the current `15`-degree cone angle to, let's say, `5`, and increase the concentration bias to `1`. You'll see how spotlights offer additional controls over the light cone and concentration in our 3D scene.

Now that we've explored the different types of virtual lights in Processing, let's continue our work on geometry, specifically typography in a 3D space.

Use Custom Fonts and Large Amounts of Text

A typical scenario in a data visualization project is to use text with a custom font alongside graphic elements on the device display. Text labels and large amounts of body text quickly add complexity and a lot of work for our device to render the scene. We can enlist Processing's `OpenGL` capabilities to help keep the frame rate up. Let's create a sketch where we cover the screen dynamically with text positioned along the z (or depth) axis of the 3D scene, depending on where we touch the screen surface, as shown in Figure 11.4. We'll work with custom fonts for this project, both released by Google and available for us to use without restriction.

Figure 11.4 - Text rendered in lights.

Large amounts of text can be rendered with lighting effects using OpenGL.

Processing uses a default font called Lucida Sans for all its modes because it is available on all platforms. We've come quite far in this book without switching to a custom font for our apps, focusing on the particular chapter topics and keeping any lines of code we don't desperately need away from our sketches. Now it's time we learned how to use a custom font for our app.

Processing provides a `PFont` class to us for working with custom fonts loaded into our sketch. We can use it in two ways:

- Using Processing's `createFont()` method, we can load an already installed system font into our app at a defined font size. This way of working with a custom font requires that the font

we'd like to use as the `createFont()` parameter is available on the platform we'll run our app on. To find out what's available on the system, the `PFont` class provides us with a `list()` method, which we can use to print a list of all available system fonts to the console.

- Alternatively, we can use the "Create Font..." dialog in the Processing menu, shown in Figure 11.6 - Create a Processing font, and available under Tools, which allows us to import any font we'd like—and are allowed—to import into our app. Processing opens all the fonts that are installed on our desktop system in a window. We can select the font we'd like, the point size for our import, whether we'd like to "smooth" the font, and how many characters we'd like to load, and it shows a font preview for the selections we've made. For the import, we can give the font a custom name for our sketch, and when we OK the dialog window, the font will be loaded as a Processing font file (`vlw`) into the `data` folder of our sketch. Once we've created the font, we can load it into a `PFont` object in our sketch using the `loadFont()` method.
  Both methods require that we set the current font used to draw `text()` in our sketch to the font we've created or loaded using the `textFont()` method. This is necessary because we could work with two or more fonts in one sketch, and therefore we use the `textFont()` method like we'd also use `fill()` or `stroke()`, this time setting the current text font for all the text we draw after the method call.

Load a System Font

Android has introduced comprehensive typography and app design guidelines with Ice Cream Sandwich to improve the user experience across the myriads of apps available for the OS. The typography guidelines build on a new font family, called Roboto, which we can use without restriction for our apps. If you are running Ice Cream Sandwich or Jelly Bean on your device, you'll have the following Roboto font styles already installed on the system, which we can activate using the `createFont()` method in Processing.
Use the code snippet below to confirm the font list on your device using the `list()` method of the `PFont`class.

```
String[] fontList = PFont.list(); println(fontList);
```

Printed below is the `list()` returned to the console on the Nexus 6 I tested with.

```
[0] "Monospaced-Italic"
[1] "Monospaced"
[2] "Monospaced-Bold"
[3] "Monospaced-BoldItalic"
[4] "Serif-BoldItalic"
[5] "SansSerif-Bold"
[6] "Serif"
[7] "SansSerif"
[8] "SansSerif-BoldItalic"
[9] "Serif-Italic"
[10] "SansSerif-Italic"
[11] "Serif-Bold"
```

We'll first work with Google's current Roboto font and then switch over to an older Google font called Droid Serif to learn how to load a custom font that is not currently installed on the system.

We'll use two point lights, a blue one positioned to the left of the screen and an orange one positioned to the right, so we can see how the text reacts to the light sources placed into the scene when they move closer and further away from the virtual camera.

Let's take a look at the code.

code/Mobile3D/TextLights/TextLights.pde

```
PFont font;                                                  // 1

void setup()
{
  size(displayWidth, displayHeight, P3D);
  orientation(LANDSCAPE);
  noStroke();

  font = createFont("SansSerif", 48);                         // 2
  textFont(font);                                             // 3
  textAlign(CENTER, CENTER);
}

void draw()
{
  background(0);

  pointLight(0, 150, 250, 0, height/2, 500);                  // 4
  pointLight(250, 50, 0, width, height/2, 500);               // 5

  for (int y = 0; y < height; y+=100) {                       // 6
    for (int x = 0; x < width; x+=250) {                      // 7
      float distance = dist(x, y, mouseX, mouseY);            // 8
      float z = map(distance, 0, width, 0, -1000);            // 9
      text(x +","+ y, x, y, z);                               // 10
    }

  }

  if (frameCount%10 == 0)
    println(frameRate);
}
```

Here's what we need to do to place the text dynamically.

---

1. Define a `PFont` variable called `font`.
2. Create the `SansSerif` font from the Roboto font family already installed on the Android device in `48`-point size and assign it to the `font` object.
3. Define `font` as the current font used to draw all `text()` that follows.
4. Place the blue point light to the left edge of the screen, centered vertically.
5. Place the orange point light to the right edge of the screen, centered vertically.
6. Use a `for()` loop to calculate the vertical position `y` of the text, spread evenly across the screen `width`.
7. Use a `for()` loop to calculate the horizontal position `x` of the text, spread evenly across the screen `height`.

8. Calculate the `distance` between the fingertip and the text position on the screen.
9. Map the calculated `distance` to the `z`-axis values ranging from `-1000` to `0`, with text close to the fingertip appearing larger and text at the fingertip position placed at `z` equal to `0`.
10. Draw the text on the screen at the calculated position [`x`, `y`, `z`].

Let's run the sketch.

Run the App

Run the sketch on the device and observe how the screen shows a grid of text labels in the Roboto system font, indicating their individual [x, y] positions on the screen. Because the `mouseX` and `mouseY` constants each default to `0` when the app starts up, you'll see how the text labels in the upper left corner appear larger than the ones in the lower right corner. Move your finger across the touch screen surface and observe how the text closest to the fingertip enlarges and the other text scales proportionally due to their z positions ranging from `0` (close by) to `-1000` (distant and small).

Using Roboto as the default font for our app adheres to Android's typography guidelines. It's generally a good idea, though, for us to choose the best font for the job, which is why we'll look next at loading custom fonts that are not already available in the Android OS.

Load a Custom Font

Different fonts are good for different things, and the limited pixel real estate that is available to us on a mobile screen poses particular typographic challenges. We want to use the best font for a particular task and scale. When we introduce a font other than Roboto to the equation, we'll need to use Processing's `loadFont()`method for loading a custom font into our app. Before we can load the font into the app, we first need to create it via the "Create Font..." tool in the Processing IDE. Let's modify our prior sketch (Mobile3D/TextLights/TextLights.pde) now to use a custom font and make the text we'll display scalable so we can evaluate the font's particular text details at different text sizes, as shown in Figure 11.5.

Figure 11.5 - Use a custom font in Processing.

The illustration shows the custom Droid Serif font at different scales, resized based on finger position on the screen.

This sketch is very similar to the previous one. Let's start by making the font we'll use available to our sketch. Let's use Google's Droid Serif font as our custom font for this sketch: it's available for us to use without restriction and has been used on the Android since 2008. Let's download it from the Font Squirrel website, where we can find many other fonts both for free, for personal use, or for purchase.

Download Droid Serif now at http://www.fontsquirrel.com/fonts/Droid-Serif, and extract the file onto your desktop computer. Follow the Displaying Text tutorial on the Processing website to load the font into your system, and then go back to Processing. Now open "Create Font..." from the Tools menu; a window opens as shown in Figure 11.6. Let's use `48` points for this sketch, and click OK to close the window.

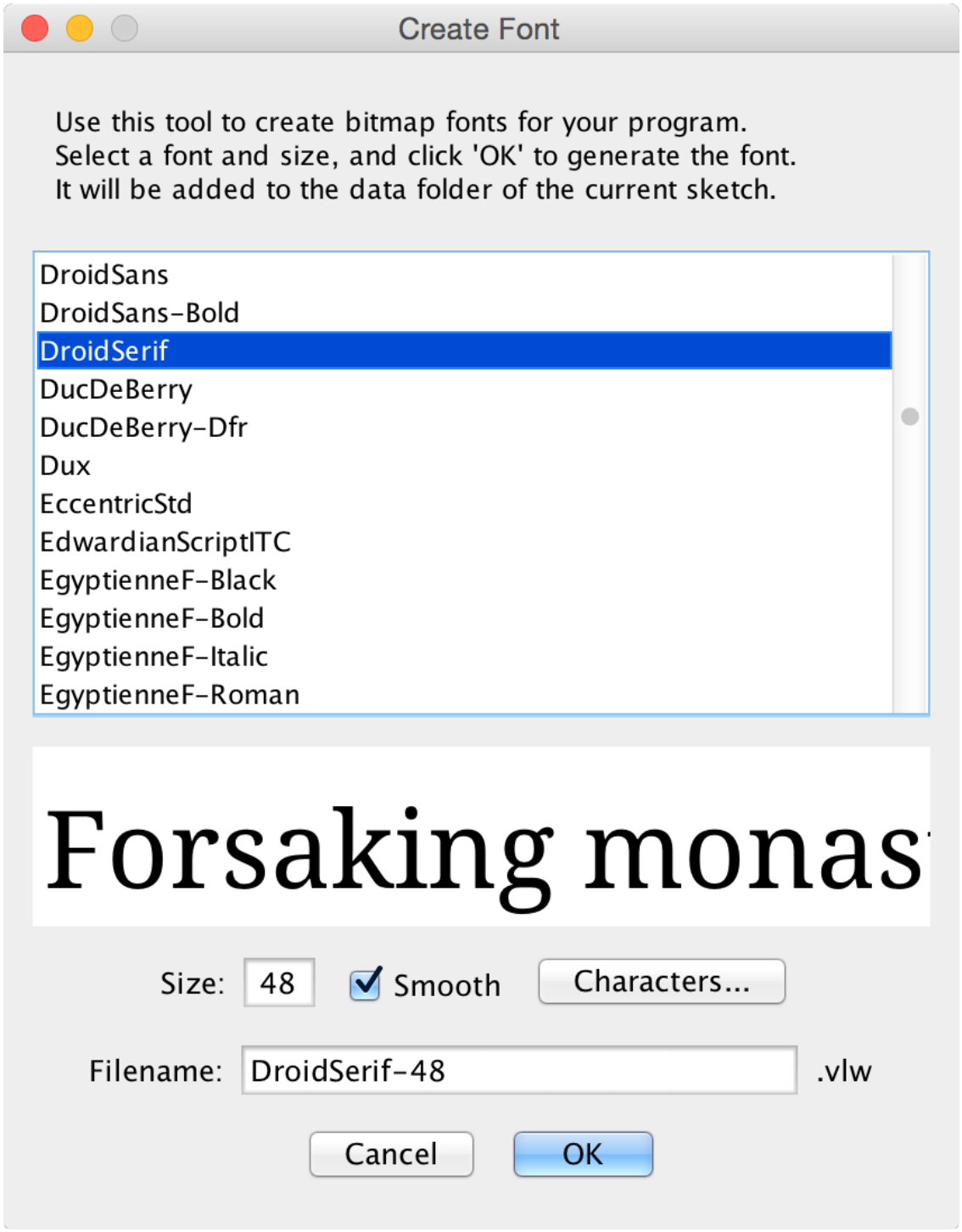


Figure 11.6 - Create a Processing font.

The dialog available under Tools ↦ “Create Font...” creates a vlw file in the sketch data folder.

Processing has just created a custom Droid Serif font for us to use in our sketch, compiling the Processing font file called `DroidSerif-48.vlw` into our sketch’s `data` folder. It contains all the text characters we need, also called glyphs, as bitmap images that are optimally displayed on

the screen at `48` points due to the settings we've chosen when we created the font. Now we are ready to use this custom font in our app.

To display our text at different sizes, let's introduce a variable called `scaleFactor` to set the individual size for each text box. If we map `scaleFactor` to values ranging `[0..48]`, we can use the variable as the parameter for `textSize()` directly. We use `48` points as a maximum, because if we scaled our text beyond that number, we'd quickly see individual characters appear pixelated with jagged edges. After all, we've created the Droid Serif font at `48` points, and because each glyph is rendered as a bitmap image, the same rule applies as for any other pixel image: if you enlarge a pixel image beyond the size and resolution it's optimized for, it appears pixelated and jagged. `OpenGL` will try to compensate by anti-aliasing, but the quality of the text's appearance will always be highest if rendered at the size it has been created initially. Let's look at the code.

code/Mobile3D/LoadFont/LoadFont.pde

```
PFont font;
float scaleFactor = 1;

void setup()
{
  size(displayWidth, displayHeight, P3D);
  orientation(LANDSCAPE);
  noStroke();

  font = loadFont("DroidSerif-48.vlw");                    // 1
  textFont(font);
  textAlign(CENTER, CENTER);
}

void draw()
{
  background(0);

  pointLight(0, 150, 250, 0, height/2, 500);
  pointLight(250, 50, 0, width, height/2, 500);

  for (int y = 0; y < height; y+=100) {
    for (int x = 0; x < width; x+=250) {
      float distance = dist(x, y, mouseX, mouseY);
      float z = map(distance, 0, width, 0, -1000);
      textSize(scaleFactor);                               // 2
      text(x +","+ y, x, y, z);
    }
  }
}

void mouseDragged()
{
  scaleFactor = map(mouseY, 0, height, 4, 72);             // 3
}
```

Here are the steps we need to take to load the custom font.

---

1. Load the font from the font file `DroidSerif-48.vlw` and assign it to the `PFont` object `font`.
2. Set the `textSize()` to the size determined by the vertical position of a fingertip on the screen.

3. Calculate the `scaleFactor` for the text size we use to draw the text.

Let’s test the app next.

Run the App

Run the sketch on your device and move your finger across the screen. You’ll now see the Droid Serif font appear on the screen, illuminated by our two point lights. The vertical position of the finger defines the text size of each individual text block, becoming bigger as you move your finger down and smaller as you move up. You can recognize the text serifs better as you move your finger down the screen and increase the text size. The text behaves the same way as our previous one with the scale difference: the z (depth) of each text box remains defined by its distance to the fingertip.

Now that we know how to load and use custom fonts into our apps, we can use the font that’s best for a specific task and scale.

Wrapping Up

You’ve now learned how to work with geometric shapes, including the 3D primitives sphere and box, and how to illuminate them in a 3D scene. You’ve applied image textures to these shapes (still and moving) and learned to manipulate the properties of spotlights, ambient lights, and point lights. You’ve worked with system fonts and loaded a custom font into a Processing sketch using Processing’s 3D renderer, which takes advantage of the device’s `OpenGL` hardware acceleration.
If you've mastered this introduction to 3D graphics and OpenGL, you should now be ready to build and control basic 3D scenes and manipulate their elements. There is more to explore, though, when it comes to 3D geometry and interactive 3D apps. Let's dive a little deeper into the subject in the next chapter, where we'll work with Scalable Vector Graphics and 3D Objects and use the Android gyro sensor and the camera as interactive interfaces for controlling a 3D scene.

12. Working with Shapes and 3D Objects

With the basic understanding we've gained running 3D sketches on the Android in Chapter 11, Introducing 3D Graphics with OpenGL, we're now ready to tackle some more advanced tasks: working with shapes and objects, generating our own geometric figures, and shifting the point of view of the virtual camera that determines how our scene is rendered. Once we've mastered those tasks, we'll know all we need to create interactive 3D scenes and games and to organize information three-dimensionally.

Processing's features for handling shapes and other figures are quite extraordinary. The `PShape` class—which we'll use throughout the chapter for much of this work&—makes it easy for us to work with Scalable Vector Graphics (`SVG`) and 3D Object (`OBJ`) files and to record vertices that define custom shapes and figures algorithmically. `PShape` leverages the `OpenGL` hardware acceleration found on most recent Android phones and tablets and is a great example of how we can tackle complex operations with just one class and a few lines of Processing code.

To show the support that `PShape` provides for handling `SVG` files, we'll start with a US map saved in that format, and then we'll give users the ability to zoom and pan over its features. Because this format is based on vectors and vertices, users won't lose graphics detail or quality as they zoom the map. We'll also see how to modify the `SVG` file to highlight typical tossup states during recent presidential elections.

To demonstrate how we can use `PShape` to manipulate 3D objects, we'll load a model of One World Trade Center from an Object file, including its materials and textures, display it on the touch screen, and then rotate and zoom the figure using multitouch gestures. We'll also create a figure algorithmically and construct a 3D Möbius strip using individual vertices that we record. We'll learn how to store the information we need to draw the figure in the `GPU`'s memory and thereby radically increase the speed with which it's rendered as well as the number of times per second the image is refreshed.

To give users a way to interact with the Möbius strip, we'll introduce and use the built-in gyro sensor, a component that now ships with the latest Androids. The gyro makes a useful input device, and we'll use it to rotate the figure on the screen by rotating the device itself. We'll conclude the chapter by bringing several of these new features together in a single application that also makes use of Android's ability to recognize faces. We'll use our gaze to control the point of view of the virtual camera. The scene consists of Earth and the Moon&—revisiting

code from Apply an Image Texture, and face recognition features we've explored in Detect Faces.

Let's look first at the classes and methods Processing provides to us to work with shapes and 3D objects—we'll use them throughout the chapter.

Working with the PShape Class

In this chapter, we'll use Processing's `PShape` features for all the projects we'll create. We can use the class to load 2D vector shape files (`svg`) and 3D object files (`obj`) and work with 3D vertices generated algorithmically. Let's take a look at the methods we'll use to load and create the 3D scenes in this chapter.

`loadShape()`	A Processing method to load a Scalable Vector Graphic, or `svg`, file into a `PShape` object
`beginShape()`	A Processing method to start recording a shape using vertices—we can connect vertices with the following modes: `POINTS`, `LINES`, `TRIANGLES`, `TRIANGLE_FAN`, `TRIANGLE_STRIP`, `QUADS`, and `QUAD_STRIP`.
`endShape()`	A Processing method to stop recording a shape using vertices
`vertex()`	A Processing method to add a vertex point to a shape using either *x* and *y* values or *x*, *y*, and *z*values for two and three dimensions, respectively—it takes only two vertices to create a shape, but we can add thousands and are only limited by the memory installed in our device. Vertices are connected with straight lines. To create curves, use the `bezierVertex()` or `curveVertex()`instead.
`createShape()`	A Processing method to load a 3D primitive or vertices into a `PShape`—the method can also handle parameters for the 3D primitives `BOX` and `SPHERE`. It also mirrors the `beginShape()`method for recording vertices into a `PShape` object and is used in conjunction with `end()`-to-end recording.
`camera()`	A Processing method to define the camera viewpoint (where the camera is looking and how the camera is facing)—we use it to navigate a 3D scene while keeping an eye on the particular spot we've defined.

Working with SVG Graphics and Maps

In addition to the fonts that are used to render text, images such as icons, line art, and maps also depend on outlines that remain accurate and legible at different scales. The pixel-based bitmap images that we've worked with so far look best if they are presented with their original size and resolution for a particular device screen. But in scenarios where we'd like to zoom images or graphic elements and we'd like to be independent of the variation in screen resolution found on different devices, we are best served by using vector graphics wherever we can.

A picture taken with a photo camera will never be accurately represented by a vector graphic. However, text, icons, symbols, and maps are good candidates because they typically use outlines and a limited number of colors. `SVG` is a popular `XML`-based file format for saving vectors, text, and shapes. It can be displayed on virtually any web browser and handled by most image processing applications.

`SVG`s are great when we need accurate detail and precise outlines at varying scales. Maps typically contain paths representing geography, city infrastructure, state lines, country borders, nations, or continents. What's more, we understand maps as images at a particular scale, providing overview and a level of detail for this scale, and we expect that we can adjust the scale seamlessly for maps with digital devices. To accomplish this goal, we can either keep multiple bitmaps in store to adjust for the shifts in scale, or we can use `SVG`files that do not lose accuracy or quality when they scale.

Maps balance abstraction and detail: we hide a map's details in order to provide a clear overview of an area, but we must add them as a user dives deeper into a particular region of a map. Shifts in scale always present challenges because we can't load everything we need at once, and we can't predict what a user might want to see in more detail at a given moment. So when we work with digital maps, we need to maintain a balance between abstraction and detail and work within the limits of resolution, graphics processing power, and storage of the phones and tablets in the Android universe. With Google Maps and Google Earth, we experience this balance literally as we watch the successive loading process for added detail.

`SVG` images are not immune to the challenges of rendering large amounts of data. For example, when we work with hundreds of words of text contained in an `SVG` file, we'll see our frame rate drop quickly because the graphics processor needs to calculate dozens of vector points at different scales for each character. It's the reason why large amounts of text are typically rendered as bitmap images made for one size, and we've explored this approach already in Use Custom Fonts and Large Amounts of Text. However, when we work with a headline or a few text labels, we definitely can work with `SVG` text and change scale, color, or rotation dynamically without sacrificing text legibility or image quality.

Let's turn now to the first project in this chapter, where we'll learn how to display a map of the United States that has been stored in an SVG file.

Map the United States

For our next project, we'll work with a US map saved as an `SVG` file, and we'll load it as a file asset into our sketch. The `SVG` file contains a blank map showing all US states and borders and is available on Wikipedia in the public domain. The `XML` code in the `SVG` file contains vertex style definitions to determine how to connect those vertex points, including stroke, thickness, and fill color.

We'll implement this project in two steps. First we load the map using `loadShape()`, and then we'll draw it on the Android screen with Processing's `shape()` method. Second, we'll alter the fill color of states that political poll takers typically consider as swing states during recent presidential elections and draw them as purple shapes on the screen, as shown in Figure 12.0. In both cases, to zoom and pan the map we'll return to the multitouch gestures we learned in Chapter 2, Working with the Touch Screen Display.

Load the `SVG` File

Like all file assets we use in Processing, we'll put our `SVG` file into the sketch's `data` folder and load it with its file name using our `loadShape()` method.

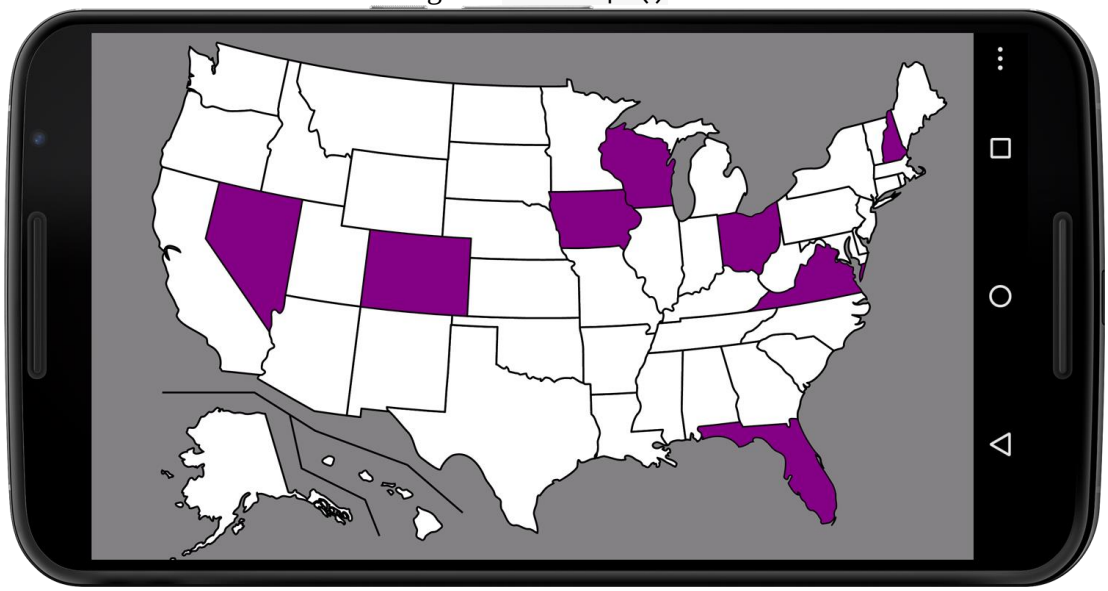


Figure 12.0 - Scalable Vector Graphic map of the United States.

The image shows a US map with typical toss-up states. Line detail remains accurate even if we zoom into the map.

Let's take a look at the code for this sketch.

code/ShapesObjects/ScalableVectorGraphics/ScalableVectorGraphics.pde

```
import ketai.ui.*;
import android.view.MotionEvent;
```

```
KetaiGesture gesture;

PShape us;                                                  // 1
float scaleFactor = 3;

void setup() {
  orientation(LANDSCAPE);
  gesture = new KetaiGesture(this);                         // 2

  us = loadShape("Blank_US_map_borders.svg");               // 3
  shapeMode(CENTER);                                        // 4
}

void draw() {
  background(128);

  translate(width/2, height/2);
  scale(scaleFactor);                                       // 5
  shape(us);                                                // 6
}

void onPinch(float x, float y, float d)
{
  scaleFactor += d/100;                                     // 7
  scaleFactor = constrain(scaleFactor, 1, 10);              // 8
}

public boolean surfaceTouchEvent(MotionEvent event) {
  super.surfaceTouchEvent(event);
  return gesture.surfaceTouchEvent(event);
}
```

Here are the steps we take to load and display the `SVG` file.

1. Define a `PShape` variable called `us` to store the `SVG` file.
2. Create a `KetaiGesture` class so we can scale the map using a pinch gesture.
3. Load the Scalable Vector Graphic containing a blank US map.
4. Set the `shapeMode()` to center so we can scale the map around its center.
5. Set the matrix `scale()` to the current `scaleFactor`.
6. Draw the `SVG` map of the United States.
7. Integrate the pinch distance `d` to increase or decrease the `scaleFactor`.
8. Constrain the `scaleFactor` to a minimum of `1` of the original map scale and a maximum of `10` times its scale.

Let's test the app.

Run the App

Run the app on the device. You'll see the `SVG` US map appear centered on the screen due to our call on `shapeMode(CENTER)`. Use the pinch gesture to scale the map, and keep zooming in while observing the line detail of the state borders saved in the file. The image does not become pixelated and the lines remain accurate.

Now that we've learned how to load, display, and scale an `SVG` file on the Android device, let's take it one step further and modify the properties of the shapes it contains.

Manipulate Shapes Within the SVG File

Now for this second part of our mapping project, we want to take advantage of the fact that `SVG` files can contain individual shapes that we can work with. In the `XML` hierarchy of our US map `SVG`, each state shape is labeled with a two-letter abbreviation, such as “fl” for Florida, which we can call to manipulate the shape’s style definitions. To verify, open the `SVG` map in your favorite photo or vector editor and you’ll see that the image consists of individual vector shapes grouped into individual layers and folders that you can edit as you wish.

For this project, we’ll highlight the eight states typically considered toss-up states during recent presidential elections. We’ll need to single out each of those states in our `SVG` file, overwrite its style definition, and set its fill to purple before we draw it.

We’ll use the `getChild()` method to find individual state shapes using the two-letter abbreviations saved in the `SVG` file. We’ll store the state abbreviations we are looking for in `String` array, namely “co” (Colorado), “fl” (Florida), “ia” (Iowa), “nh” (New Hampshire), “nv” (Nevada), “oh” (Ohio), “va” (Virginia), and “wi” (Wisconsin). If we find a matching abbreviation in the `SVG` file, we’ll grab the shape and assign it to a `PShape` array we’ll provide. To change the shape’s color to purple, we’ll use `disableStyle()` to ignore the style definitions included in the file and replace them with our own. We’ll also reuse the `mouseDragged()` method to move the map horizontally and vertically so that we can browse the whole map while being zoomed in at a state level.

Let’s take a look at the code we’ve modified based on earlier ShapesObjects/ScalableVectorGraphics/ScalableVectorGraphics.pde.

code/ShapesObjects/ScalableVectorGraphicsChild/ScalableVectorGraphicsChild.pde

```
import ketai.ui.*;
import android.view.MotionEvent;

KetaiGesture gesture;
PShape us;
String tossups[] = {
  "co", "fl", "ia", "nh", "nv", "oh", "va", "wi"          // 1
};
PShape[] tossup = new PShape[tossups.length];             // 2
float scaleFactor = 3;
int x, y;

void setup() {
  orientation(LANDSCAPE);
  gesture = new KetaiGesture(this);

  us = loadShape("Blank_US_map_borders.svg");
  shapeMode(CENTER);
  for (int i=0; i<tossups.length; i++)
  {
    tossup[i] = us.getChild(tossups[i]);                  // 3
    tossup[i].disableStyle();                             // 4
  }
  x = width/2;
```

```
  y = height/2;
}
void draw() {
  background(128);

  translate(x, y);                                    // 5
  scale(scaleFactor);                                 // 6

  shape(us);
  for (int i=0; i<tossups.length; i++)
  {
    fill(128, 0, 128);                                // 7
    shape(tossup[i]);                                 // 8
  }
}
void onPinch(float x, float y, float d)               // 9
{
  scaleFactor += d/100;
  scaleFactor = constrain(scaleFactor, 1, 10);
  println(scaleFactor);
}
void mouseDragged()                                   // 10
{
  if (abs(mouseX - pmouseX) < 50)
    x += mouseX - pmouseX;
  if (abs(mouseY - pmouseY) < 50)
    y += mouseY - pmouseY;
}
public boolean surfaceTouchEvent(MotionEvent event) {
  super.surfaceTouchEvent(event);
  return gesture.surfaceTouchEvent(event);
}
```

Here are the additional steps we take to highlight potential toss-up states.

---

1. Create a `String` array containing the two-letter abbreviations of toss-up states.
2. Create a `PShape` array called `tossup` of the same length as the `tossups` `String` array.
3. Assign a child shape in the `us` map to the `tossup` `PShape` array.
4. Disable the color and opacity style found in the `SVG`.
5. Move to the `x` and `y` location.
6. Scale the matrix to our calculated `scaleFactor`.
7. Set the new `fill()` color to purple.
8. Draw the individual swing states.

9. Use the `KetaiGesture` callback method `onPinch()` to calculate the map `scaleFactor`.
10. Use the `mouseDragged()` callback method to set the horizontal and vertical position of the map on the device screen.

    Let’s run the app.

Run the App

Run the modified sketch on the device again. This time the app starts up showing “purple states,” and we can still zoom into the map and move it horizontally and vertically.

This completes our mapping project and our investigation of Scalable Vector Graphics.

Now that we've learned how to work with shapes and vertices contained in a Scalable Vector Graphic, it's time we looked at another file type comparable to `SVG` and used for three-dimensional objects—the Object file format.

Display an Architectural Model Loaded from an Object File

For this project, we'll work with three-dimensional coordinates for vertices that define a figure contained in an Object file, as well as material definitions and image textures linked from that file. We'll use a model of One World Trade Center, also known as Freedom Tower, the lead building of the World Trade Center complex planned by architect Daniel Libeskind and designed by David Childs (to be completed in 2013). The model contains a 3D geometric figure for the main building's architecture and some image textures, shown here:

Figure 12.1 - Displaying an Object file.

The Object file contains vertices and links to materials and textures for displaying One World Trade Center as a 3D PShape object.

Working in 3D, Object (`obj`) is a very popular and versatile file format. We can use an `OBJ` as a self-contained 3D asset and load it into a 3D app on our Android. All the textures for our

figure are already predefined in the `OBJ` file, making it fairly easy to handle in Processing with our familiar `PShape` class. Yes, it handles `OBJ` files as well.

Object files are not `XML`-based in their organizational structure, but they still contain data segments with coordinates for the vertices that define the figure and data segments that link to assets such as materials and textures to the file. The model we'll work with was loaded from Google Sketchup's 3D warehouse and converted into the Object format using Autodesk Maya.

We now have a file called `OneWTC.obj`, a linked material file of the same name, `OneWTC.mtl`, and twelve `JPEG` images named `texture0.jpg`...`texture12.jpg` in our sketch `data` folder.

The code for this project is fairly concise and very similar in structure to our `SVG` map project ShapesObjects/ScalableVectorGraphics/ScalableVectorGraphics.pde. We'll first load our Object file from the data folder into the sketch and display it using the `shape()` method. Then we use the `onPinch()` method to allow for scaling the object and `mouseDragged()` for rotating the building and moving it up and down vertically.

Here's the code.

code/ShapesObjects/ObjectFiles/ObjectFiles.pde

```
import ketai.ui.*;
import android.view.MotionEvent;

KetaiGesture gesture;

PShape wtc;
int r, y;
float scaleFactor = .04;

void setup() {
  size(displayWidth, displayHeight, P3D);
  orientation(PORTRAIT);
  gesture = new KetaiGesture(this);
  noStroke();
  wtc = loadShape("OneWTC.obj");                          // 1
  y = height;
}

void draw() {
  background(0);
  lights();                                               // 2

  translate(width/2, y);                                  // 3
  scale(scaleFactor);                                     // 4
  rotateX(PI);                                            // 5
  rotateY(radians(r));                                    // 6

  shape(wtc);                                             // 7
}

void onPinch(float x, float y, float d)                   // 8
{
  scaleFactor += d/5000;
  scaleFactor = constrain(scaleFactor, 0.01, .3);
  println(scaleFactor);
}

void mouseDragged()                                       // 9
```

```
{
  if (abs(mouseX - pmouseX) < 50)
    r += mouseX - pmouseX;
  if (abs(mouseY - pmouseY) < 50)
    y += mouseY - pmouseY;
}

public boolean surfaceTouchEvent(MotionEvent event) {
  super.surfaceTouchEvent(event);
  return gesture.surfaceTouchEvent(event);
}
```

1. Load the Object file into a `PShape` variable called `wtc` using `loadShape()`.
2. Switch on the default `lights()`.
3. Move the matrix horizontally to the center of the screen and vertically to the position `y`, determined by moving a finger across the screen.
4. Scale the matrix to the `scaleFactor` determined by our pinch gesture.
5. Rotate the building around the x-axis so it appears upright in the `PORTRAIT` mode and not upside-down.
6. Rotate the building around the x-axis so we can look at all its sides when we drag one finger horizontally.

7. Draw the `wtc` Object file.
8. Calculate the object's `scaleFactor` using the `onPinch()` `KetaiGesture` callback method.
9. Determine the vertical position `y` of the building on the screen and its rotation `r` using Processing's `mousePressed()` method.
   Let's test the app now.

Run the App

Now run the app on the device. When the 3D scene starts up, the tall One World Trade Center building will appear fullscreen in `PORTRAIT` mode. Move your finger across the screen horizontally to rotate the building. It's a fairly demanding model for the graphics processor on the Android, and initially shown at 4% of its original scale, so the frame rate is not as high as for most other projects we've worked with so far. Pinch to scale the building, and move your finger across the screen to rotate the building or move it up and down.
Now that we've looked at 3D primitives, Scalable Vector Graphics, and Object files, the missing piece is how to create a figure from scratch using individual vertex points and algorithms.

Create a M&#xFbius Shape and Control It Using the Gyroscope

In our next project, we'll generate a figure from scratch and use a for loop and a sequence of translations and rotations to create a Möbius strip based on individual vertices that we record. Topologically speaking, the Möbius strip is an interesting example of a 3D figure that does not have a determinable surface area. Practically, it's pretty simple to understand and easy to create using just a piece of paper. If you'd like to try, cut or fold a piece of paper into a strip at least five times as long as it is wide. Hold both ends, and twist one of them 180 degrees while holding the other stationary. Now connect the two ends with a piece of tape and you've got yourself a Möbius strip. It's basically a ring twisted by half a revolution, as shown in Figure 12.2.

Figure 12.2 - Control a Möbius strip using the gyro.

The image shows a PShape object composed of individual vertices that are illuminated by blue ambient light and a white point light source positioned in the center of the strip.

You can confirm that a Möbius strip has only one side by taking a pen and drawing a continuous line in the center of your paper strip until you've reached the beginning of your line again. You'll need two revolutions to get there, because the Möbius strip is a surface with only one side and only one border. In this project, we'll create this shape using custom vertices. To navigate the 3D scene with our Möbius strip by just rotating our Android device, we'll use a sensor that we haven't yet given the attention it deserves: the gyroscope sensor.

Introducing the Gyroscope Sensor

The Nintendo Wii was the first device to introduce millions of users to gesture-based games, but now similar apps can be purchased for phones and tablets as well. At its heart is an

onboard gyroscope sensor. The gyroscope sensor was introduced in 2010 with the iPhone 4 (June 2010), followed by the Samsung Galaxy S (July 2010). Augmented reality (`AR`) applications are especially able to take advantage of the precise pitch, roll, and yaw angles the sensor provides in real time. The gyro is able to determine the device rotation around gravity, for example, when we point the device camera toward the horizon and we rotate it following the horizon line—the accelerometer cannot help us there. Apple's "Synchronized, Interactive Augmented Reality Displays for Multifunction Devices" patent from July 2011 suggests that AR will have an increasing role in the mobile space.

The gyro provides us with information about an Android's rotation, reporting so-called angular rates in degrees for the x-, y-, and z-axes. We can access these readings with the `KetaiSensor` class and the `onGyroscopeEvent(x, y, z)` callback method. The angular rate is a positive or negative floating point value reflecting the change for each axis since we last read it. We'll use the standard convention and refer to the rotation around the device's x-axis as pitch, around the y-axis as roll, and around the z-axis as yaw.

Unlike the accelerometer, the gyro is independent of g-force, unfazed by device shakes, and very responsive to device rotation. It's the best sensor for games and 3D scenes that are controlled by moving and rotating the device. The sensor is less ubiquitous than the accelerometer, but it is now found in most Android phones and tablets.

When a device sits on a table, its gyro reports `+-0.000` degrees for pitch, roll, and yaw. To calculate how much a device has rotated, we just need to integrate the values for each of its axes, or in other words, add them all up. Once we've done that we'll know the pitch, roll, and yaw of the device in degrees. The gyro is not aware of how the device is oriented relative to the ground (g-force) or to magnetic north. For that, we'd need to enlist the onboard accelerometer and the magnetic field sensor. But while that information might be critical for a navigation and/or augmented reality app, it's more than we need to know for this one.

A limitation of the gyro is that over time it's susceptible to drift. If we integrate all gyro values while the device is sitting still on the table, we should receive `0.000` degrees for all three axes. However, the values slowly drift after just a few seconds and significantly after a few minutes because we integrate fractions of floating point values many thousands of times per minute, and the sum of all those tiny values won't add up perfectly to `0.000` degrees. For gaming or 3D apps this drift is less relevant, as instantaneous feedback on device rotation outweighs accuracy over extended periods of time. In scenarios where this drift causes problems, the accelerometer can be used to correct the drift.

Record the Vertices for a Möbius Shape

To create the Möbius shape, we'll use an approach that resembles the one we used to create a paper model. To set this up, we'll use a `for` loop and a series of translations and rotations in

three different matrices to determine the individual vertex points we'll use to draw the Möbius shape in three-dimensional space. We've used these methods earlier in Detect Multitouch Gestures, and elsewhere.

As with our paper model, we'll choose a center point around which to rotate our band. While we rotate the band one full revolution around the z-axis, we also rotate it by `180` degrees around the y-axis, or half a revolution. Once we've assembled all the vertex points we need to define the two edges of our band in `setup()`, we can draw the resulting shape in `draw()` using the `beginShape()` and `endShape()` methods.

As the reference for `beginShape()` points out, "Transformations such as `translate()`, `rotate()`, and `scale()` do not work within `beginShape()`," so as a workaround, we take each of our vertex points through multiple transformations and look up their final position within our 3D scene, the so-called model space. The model space x, y, and z coordinates, which we'll get using `modelX()`, `modelY()`, and `modelZ()`, gives us the final absolute position of our vertex in the scene, and we'll take that position and write it to a `PVector`array list that we can then use to `draw()` our shape.

In `draw()`, we'll use the `beginShape(QUAD_STRIP)`, `vertex()`, and `endShape(CLOSE)` methods to assemble the shape, which we draw using the `QUAD_STRIP` mode, where each of the vertices are connected, and we'll end up with a closed strip surface to which we can assign a fill color. Using both a blue `ambientLight()`, as well as a white `pointLight()` located in the Möbius's center, we'll get a good impression of the three-dimensional shape surface. Because we'll use the device's built-in gyro to rotate the Möbius shape in 3D space around its center point, we'll get to look at all sides by just rotating our Android phone or tablet around its x- (pitch), y- (roll), and z- (yaw) axes.

Let's take a look at the code.

code/ShapesObjects/Moebius/Moebius.pde

```
import ketai.sensors.*;

KetaiSensor sensor;
float rotationX, rotationY, rotationZ;
float roll, pitch, yaw;

ArrayList<PVector> points = new ArrayList<PVector>();     // 1

void setup()
{
  size(displayWidth, displayHeight, P3D);
  orientation(PORTRAIT);

  sensor = new KetaiSensor(this);
  sensor.start();

  noStroke();
  int sections = 3600;                                     // 2

  for (int i=0; i<=sections; i++)                          // 3
  {
```

```
    pushMatrix();                                          // 4
    rotateZ(radians(map(i, 0, sections, 0, 360)));         // 5
    pushMatrix();                                          // 6
    translate(width/2, 0, 0);                              // 7
    pushMatrix();                                          // 8
    rotateY(radians(map(i, 0, sections, 0, 180)));         // 9
    points.add(
      new PVector(modelX(0, 0, 100), modelY(0, 0, 100), modelZ(0, 0, 100))     // 10
    );
    points.add(
      new PVector(modelX(0, 0, -100), modelY(0, 0, -100), modelZ(0, 0, -100))  // 11
    );
    popMatrix();
    popMatrix();
    popMatrix();
  }
}

void draw()
{
  background(0);
  ambientLight(0, 0, 128);                                 // 12
  pointLight(255, 255, 255, 0, 0, 0);                      // 13

  pitch += rotationX;                                      // 14
  roll += rotationY;                                       // 15
  yaw += rotationZ;                                        // 16

  translate(width/2, height/2, 0);
  rotateX(pitch);
  rotateY(-roll);
  rotateZ(yaw);

  beginShape(QUAD_STRIP);                                  // 17
  for (int i=0; i<points.size(); i++)
  {
    vertex(points.get(i).x, points.get(i).y, points.get(i).z);  // 18
  }
  endShape(CLOSE);

  if (frameCount % 100 == 0)
    println(frameRate);
}

void onGyroscopeEvent(float x, float y, float z)           // 19
{
  rotationX = radians(x);
  rotationY = radians(y);
  rotationZ = radians(z);
}

void mousePressed()
{
  pitch = roll = yaw = 0;                                   // 20
}
```

Let’s take a look at the steps we take to create the figure and control the scene using the gyro.

---

1. Create an `ArrayList` of type `PVector` to store all the vertices for our Möbius strip.
2. Define the number of sections, or quads, used to draw the strip.
3. Use a for loop to calculate the x, y, and z positions for each vertex used to define the Möbius strip.

4. Add a new matrix on the matrix stack for our first rotation.

5. Rotate by one degree around the z-axis to complete a full `360`-degree revolution after `3600` iterations, as defined by `sections`.
6. Add a new matrix on the matrix stack for our next transformation.

7. Move the vertex along the x-axis by half the screen `height`, representing the radius of our Möbius strip.
8. Add another matrix on the stack for our next rotation.

9. Rotate around the y-axis by `180` degrees for the twist along the Möbius strip.
10. Set the first vertex point for the current quad element model position in the Möbius strip, displaced by `50`pixels along the z-axis with regard to the current matrix's origin.
11. Set the first vertex point for the current quad element model position in the Möbius strip, displaced by `-50` pixels along the z-axis with regard to the current matrix's origin.
12. Set a blue ambient light source using the `ambientLight()` method.
13. Set a white point light source in the center of the Möbius strip using `pointLight()`.
14. Enumerate the gyroscope `x` values to calculate the pitch of the device since the app started.
15. Enumerate the gyroscope `y` values to calculate the pitch of the device.
16. Enumerate the gyroscope `z` values to calculate the pitch of the device.
17. Begin recording the `QUAD_STRIP` vertices that make up the Möbius strip.
18. End recoding vertices for the strip, and close the connection to the first recorded vertex using `CLOSE`.
19. Receive the gyroscope sensor values for the `x`-, `y`-, and `z`-axes and assign the `radians()`equivalent of their degree values to `rotationX`, `rotationY`, and `rotationZ`.
20. Reset the `pitch`, `roll`, and `yaw` to `0` so we can reset the rotation of the Möbius strip in 3D space by tapping the device screen.
    Let's test the app.

Run the App

Run the sketch on the device. The Möbius strip begins lying "flat" in front of our camera. Because the gyro takes care of rotating the Möbius shape, it slowly drifts if we don't do anything. Pick up the Android phone or tablet and rotate it up and down along the x-axis, left and right along the y-axis, and finally flat around the z-axis. The Möbius strip counters this device movement in virtual space, and we are able to take a closer look at the figure. Tap the screen to reset the strip to right where we started.

Now let's take a look at how we are doing on graphics performance for this scene, consisting of 7200 vertices, or `3600` `sections` times 2 vertices for the band. I'm using the Google Nexus 6 phone for this test. Look at the `frameRate` printed to the Processing console, and you'll see that the sketch is running at a default `60`frames per second, which is good.
Let's multiply the number of `sections` by 10 and look again. Go ahead and change the `sections` value in the code from `3600` to `36000` and rerun the sketch on the device. On the Nexus 6, the frame rate drops to about `12` frames per second.
This drop in the frame rate can be avoided if we take advantage of Processing's `OpenGL` support for retaining a shape in the `GPU`'s memory. Let's take a look at an alternative method to draw our Möbius shape using the `PShape` class.

Use GPU Memory to Improve Frame Rate

Keeping vertices and textures in the `GPU`'s memory becomes very useful when we deal with more complex figures that do not change over time. Processing's `PShape` object allows us to create a shape from scratch using the `createShape()` method, which works much like `beginShape()`. In this modified sketch based on ShapesObjects/Moebius/Moebius.pde, we create the Möbius strip as a `PShape` in setup, and then we use the `shape()` method to draw the shape on the screen. All the transformations remain the same. We keep the `QUAD_STRIP` drawing mode, and we close the shape we create using the `end(CLOSE)` method.
Let's examine the changes we've made in the code.

code/ShapesObjects/MoebiusRetained/MoebiusRetained.pde

```
import ketai.sensors.*;

KetaiSensor sensor;
float rotationX, rotationY, rotationZ;
float roll, pitch, yaw;

PShape moebius;                                          // 1

void setup()
{
  size(displayWidth, displayHeight, P3D);
  orientation(PORTRAIT);

  sensor = new KetaiSensor(this);
  sensor.start();

  noStroke();
  int sections = 36000;

  moebius = createShape();                               // 2
  moebius.beginShape(QUAD_STRIP);
  for (int i=0; i<=sections; i++)
  {
    pushMatrix();
    rotateZ(radians(map(i, 0, sections, 0, 360)));
```

```
    pushMatrix();
    translate(width/2, 0, 0);
    pushMatrix();
    rotateY(radians(map(i, 0, sections, 0, 180)));
    moebius.vertex(modelX(0, 0, 100), modelY(0, 0, 100), modelZ(0, 0, 100));     // 3
    moebius.vertex(modelX(0, 0, -100), modelY(0, 0, -100), modelZ(0, 0, -100));  // 4
    popMatrix();
    popMatrix();
    popMatrix();
  }
  moebius.endShape(CLOSE);                                  // 5
}

void draw()
{
  background(0);

  ambientLight(0, 0, 128);
  pointLight(255, 255, 255, 0, 0, 0);

  pitch += rotationX;
  roll += rotationY;
  yaw += rotationZ;

  translate(width/2, height/2, 0);
  rotateX(pitch);
  rotateY(-roll);
  rotateZ(yaw);

  shape(moebius);                                           // 6

  if (frameCount % 100 == 0)
    println(frameRate);
}

void onGyroscopeEvent(float x, float y, float z)
{
  rotationX = radians(x);
  rotationY = radians(y);
  rotationZ = radians(z);
}

void mousePressed()
{
  pitch = roll = yaw = 0;
}
```

Let’s take a look at the steps we take to record the shape into a `PShape` object.

1. Create a `PShape` variable called `moebius` to record vertex points into.
2. Create the `QUAD_STRIP` `PShape` object `moebius` using the `createShape()` method.
3. Add our first strip `vertex()` to the `moebius` `PShape`.
4. Add the second strip `vertex()` to the `moebius` `PShape`.
5. `CLOSE` the `PShape` object using the `end()` method.
6. Draw the `moebius` strip using Processing’s `shape()` method.

Let’s test the app.

Run the App

Run the sketch on the device. We've also used `36000` `sections` as we've done in the code before, but the frame rate is back up to `60` frames per second. The `PShape` class and its ability to leverage `OpenGL` is definitely one of the main improvements since Processing 2.0.
You are now able to create figures from scratch by recording the coordinates for the vertices that define the figure. Finally, let's explore how to change the viewpoint of our virtual camera in a 3D scene.

Control a Virtual Camera with Your Gaze

For this final chapter project, we'll implement an experimental and lesser-known method to interact with a 3D scene&—using our gaze to rotate a constellation of planets containing Earth and the Moon. By looking at the scene displayed on our device screen from a specific angle, we can control the rotation of Earth and the Moon around the x- and y-axes, as illustrated in Figure 12.3. Similar to a scenario where we look around a fixed object by moving our head slightly sideways, we cause Earth to rotate and reveal the continents located on its sides accordingly.

Figure 12.3 - Control the camera location via gaze detection.

The 3D scene containing Earth and the Moon is controlled by the relative location of our eyes looking at the device camera.

We'll place Earth at the center of the scene, since the Moon rotates around Earth in this two-body system. To navigate the 3D scene, we move the camera this time, and not the figure. Earlier, we've rotated the 3D primitives we've worked with around the scene's y-axis and left the virtual camera where it is by default. This time, we are moving the `camera()` while

keeping it continuously pointed at the center of our scene, where we placed Earth&—independent of our camera's viewpoint. As a result, we'll always keep an eye on Earth. Compared with the `NASA` texture image we used in Apply an Image Texture, we'll use a higher resolution version of the same image for more accurate detail. We'll use the maximum resolution the `PShape` can handle&—`2048` pixels wide or high.

The Moon is about `3.7` times smaller than Earth and located roughly `110` times its diameter from Earth. We'll use these ratios to add a white sphere to our scene and then cause it to revolve around our home planet. Because the Moon is quite far from Earth, both in reality and in our 3D scene, we'll hardly catch it in our camera view. Feel free to place it closer when we test the sketch.

To implement this sketch, we'll translate our coordinate system to the center of the screen located at `[width/2, height/2, 0]`. Then we place Earth at this new center, `[0, 0, 0]`. Finally we rotate the Moon around Earth and translate it `110` times its diameter away from Earth.

Our camera is located at `[0, 0, height]`, which means it's also centered horizontally and vertically on the screen but at a z distance of `height` away from the center of Earth's sphere. With a sphere size of Earth set to `height/3`, both the camera distance and the display size of Earth are defined in relation to the screen height, keeping the scene proportional and independent of the Android device we are using.

To control the scene, we'll use Ketai's `KetaiSimpleFace` class, which can recognize the midpoint and distance between the eyes of a face detected by the front-facing device camera&—as we've done already in Detect Faces. We'll assume only one face in the scene less than an arm's length away from the device, which is why we set the maximum number of faces to be recognized to `1`. If we find a face, we'll use the x and y`location` of the face to calculate the position of our camera in the scene. If we don't find a face, we'll fall back to a touch screen input, where we use the `mouseX` and `mouseY` position of the fingertip instead. Let's take a look at the code.

code/ShapesObjects/LiveFaceDetectionAndroid/LiveFaceDetectionAndroid.pde

```
import ketai.camera.*;
import ketai.cv.facedetector.*;

int MAX_FACES = 1;
KetaiSimpleFace[] faces = new KetaiSimpleFace[MAX_FACES];
KetaiCamera cam;
PVector camLocation = new PVector();                    // 1
PShape sphereShape;
PImage sphereTexture;

void setup() {
  size(displayWidth, displayHeight, P3D);
  orientation(LANDSCAPE);
  stroke(255, 50);
  sphereTexture = loadImage("earth_lights.jpg");
  sphereDetail(36);                                     // 2
```

```
  sphereShape = createShape(SPHERE, height/2);
  sphereShape.setTexture(sphereTexture);

  cam = new KetaiCamera(this, 320, 240, 24);
  cam.setCameraID(1);
}

void draw() {
  if (cam.isStarted())
    background(50);
  else
    background(0);

  translate(width/2, height/2, 0);
  camera(camLocation.x, camLocation.y, height, // eyeX, eyeY, eyeZ  // 3
  0.0, 0.0, 0.0, // centerX, centerY, centerZ
  0.0, 1.0, 0.0); // upX, upY, upZ
  noStroke();

  faces = KetaiFaceDetector.findFaces(cam, MAX_FACES);    // 4

  for (int i=0; i < faces.length; i++) {
    //reverse the "face-mapping" correcting mirrored camera image
    camLocation.x = map(faces[i].location.x, 0, cam.width, width/2, -width/2);   // 5
    camLocation.y = map(faces[i].location.y, 0, cam.height, -height/2, height/2);  // 6
  }

  if (!cam.isStarted()) {
    camLocation.x = map(mouseX, 0, width, -width/2, width/2);  // 7
    camLocation.y = map(mouseY, 0, height, -height/2, height/2);
  }
  shape(sphereShape);

  fill(255);
  rotateY(PI * frameCount / 500);                            // 8
  translate(0, 0, -height/2 / 3.7 * 2 * 110);                // 9
  sphere(height/2 / 3.7);                                    // 10

  if (cam.isStarted())
    image(cam, 0, 0);
}

void onCameraPreviewEvent() {
  cam.read();
}

void keyPressed() {
  if (key == CODED) {
    if (keyCode == MENU) {                                   // 11
      if (cam.isStarted())
        cam.stop();
      else
        cam.start();
    }
  }
}
```

Let’s take a look at the steps we need to control the scene with our gaze.

---

1. Define a `PVector` `camLocation` to keep track of the camera position within the scene we’ll calculate.
2. Increase the number of vertices per full `360`-degree revolution around the sphere to `36`, or one tessellation per 10 degrees longitude.
3. Set the `camera()` viewpoint to our calculated location, looking at the exact center of the scene.

4. Detect up to one face in the camera image.
5. Map the horizontal position of the `PVector` `camLocation` from the `cam` preview width to the screen `width`.
6. Map the vertical position of the `PVector` `camLocation` from the `cam` preview height to the screen `height`.
7. Set the `camLocation` to the fingertip if we are not using gaze control.
8. Rotate the Moon slowly around Earth, one revolution per 10 seconds at an assumed device frame rate of `60` frames per second.
9. Place the white Moon sphere at its relative distance to Earth (about 110 times its diameter).
10. Draw the Moon's sphere.
11. Start face detection with the Menu button.

Let's test the app next.

Run the App

Run the sketch on the device. When it starts up, you'll see a sphere drawn on a black background and painted with the NASA-provided image of Earth that we used earlier in this chapter. The camera will not yet have started, and until it is activated, it won't recognize faces.

Move your finger to the center of the screen and observe a view straight ahead at Earth's sphere. Move it left across the screen and you'll move the virtual camera to the left in the scene, move it up and you'll move the camera up, and so on. We get an impression of the movement we can expect when we start the face tracker.

Now press the menu key to start up the camera and begin recognizing faces, indicated by the text printed to the console but also by switching the black background to dark gray. Hold the device comfortably, as you would when you look straight at the screen. Now move your head sideways and see Earth reveal its sides; move your head up and down and see Earth reveal its poles.

Because we are moving the camera and do not rotate Earth, we are getting the side effect that Earth seems to scale—it doesn't. Instead, we are observing what in the motion picture world would be called a tracking shot, where a camera is mounted on a dolly on a track. In our scenario, the track is straight, so we are actually moving further away from Earth as we move left and right, up and down. We could mitigate this effect by putting our camera on a "circular track"; however, it would be a less dynamic "shot" as well.

This completes our exploration of 3D apps in Processing.

Wrapping Up

You are now able to create 3D apps in Processing using `OpenGL`'s hardware acceleration to render shapes, text, objects, textures, lights, and cameras. Each one of those subjects deserves further exploration. The basic principles of creating or loading shapes and objects, using image textures, working with different types of lights, and animating camera movements also remain the same for more complex 3D apps.

Let's apply our knowledge about hardware-accelerated graphics now in our last chapter, where we'll develop cross-platform apps running in the `HTML5` browsers installed on all smart phone devices shipped today.