CHAPTER 2

The Processing Language

If you are not familiar with the Processing language, read this chapter for an introduction to how to create 2D shapes, use geometry transformations and color, and handle touchscreen input in Processing. The chapter ends with a step-by-step example of a drawing sketch, which we will use in Chapter 3 to learn how to export and upload an app created with Processing to the Play Store.

A Programming Sketchbook for Artists and Designers

As we learned in the first chapter, the Processing language, together with the Processing Development Environment (PDE), makes it easier for users who are new to programming to start creating interactive graphics. The language has been designed to be minimal and simple for learning and yet expressive enough to create many kinds of code-based projects, including generative art, data visualization, sound art, film, and performance. It includes around 200 functions across different categories: drawing, interaction, typography, etc., as well as several classes that help with the handling of form, color, and data.

The creators of Processing were also inspired by the concept of sketching: when we have an idea and use a paper sketchbook to draw it down, we often create many iterations or variations of the original idea as we refine it or make changes. Processing supports a similar way of working with computer code by making it easy for us to obtain visual feedback from the code so we can quickly iterate and refine a programming idea. This is why you will see Processing described as a "coding sketchbook" in many places. The next section describes the basic structure in Processing that allows us to sketch with code by getting animated graphics to the computer screen.

The Structure of a Processing Sketch

Most of the time, we want our Processing sketches to run nonstop to animate graphics on the screen and keep track of user input. This means that instead of running our code just one time, Processing will continue running it repeatedly so that we can animate the visual output by programming how it changes from one frame to the next and how it's affected by any interaction of the user with the computer (desktop, laptop, phone, etc.). Because of that, we call these "interactive sketches" in contrast with "static sketches" that run just one time (we can create these with Processing too, but for now, we will focus on interactive sketches).

Note All the code examples in this chapter can be run in the Java or the Android mode, since they don't rely on any features of the Processing language specific to either mode. This chapter also assumes some basic knowledge of programming, including conditions (if/else), loops (for/while), use of variables, and organizing our code with functions, so we will not go over those concepts.

A. Colubri, *Processing for Android*, https://doi.org/10.1007/978-1-4842-9585-4_2

The structure of an interactive sketch in Processing is uncomplicated: it requires two functions, one called `setup()` and the other, draw(). The setup() function will contain the code that we need to run only one time at the beginning to set things up, and the draw() function will contain the code that generates the visual output of our sketch and Processing will run every time it has to draw or "render" a new frame on the screen of the computer, typically 60 times per second. Let's use these functions to create our first animated sketch in Processing!

A simple animation could be moving a vertical line from left to right across the screen. We do just that in Listing 2-1, where we have both setup() and draw(). In the setup() function, we run a few initial tasks: first, we set the size of the output window or "canvas" where Processing will draw to. In this case, we are initializing the canvas with fullScreen(), which makes it as large as the entire screen of the computer (or phone). Then we set the parameters of our line with `strokeWeight(2)`, which sets the line weight (or thickness) to 2 pixels, and `stroke(255)`, which sets the line color to white (we will go over the use of numbers to set color values later in this chapter). Finally, we set the value of the variable x, declared at the top of the code, to zero. This variable holds the horizontal position of the line, and setting to zero means placing it on the left side of the screen (also later in the chapter we will discuss about screen coordinates in more detail). In the draw() function, we start by setting the color of the entire canvas to a tone of gray with background(50) and then draw the line with `line(x, 0, x, height)`. The line() function takes four parameters: the first two (x, 0) correspond to the first point of the line, and the last two (x, height), to the second point, so by calling this function, we get Processing to draw a line from (x, 0) to (x, height). The word "height" is a "constant" in Processing that always represents the height of the canvas and therefore cannot be used as a variable name by us (the Processing language includes several more constants like this; we will learn about some of them later in this and following chapters). Because the two points have the same x coordinate, we will get a vertical line running between the top (0) and the bottom (height) of the screen. The last two lines of code in the draw() function control the animation: by doing `x = x + 1`, we are increasing the value of the horizontal position of our vertical line by one unit; this means that the line will move to the right one pixel per frame. If we keep adding 1 to x enough times, its value will eventually be larger than the width of our canvas, and the line will no longer be visible. In that case, we put the line back at the left edge of the canvas by checking if x is larger than the width and resetting to zero with `if (width < x) x = 0`. The output of this sketch running on a phone is shown in Figure 2-1.

Listing 2-1. A sketch that draws a vertical line moving horizontally across the screen

```
int x;

void setup() {
  fullScreen();
  strokeWeight(2);
  stroke(255);
  x = 0;
}
void draw() {
  background(50);
  line(x, 0, x, height);
  x = x + 1;
  if (width < x) x = 0;
}
```

Figure 2-1. *Output of the animated line sketch, running on a Pixel 4a phone*

This continuous sequence of calls to the `draw()` function that Processing does automatically in any interactive sketch is called the "animation loop." Processing calls the `draw()` function during the animation loop at a default frame rate of 60 frames per second; however, we can change this default using the function `frameRate()`. For instance, if we add `frameRate(1)` in `setup()`, then the animation loop will draw 1 frame per second.

Sometimes, we may need to stop the animation loop at some point. We can use the `noLoop()` function for doing this and then the `loop()` function to resume the animation afterward. Processing also has a boolean (logical) constant named `looping`, which is true or false depending on whether the sketch is running the animation loop. We can build on top of our previous sketch to pause/resume the line movement if the user clicks the mouse (on a desktop/laptop) or taps the touchscreen (on a phone), as shown in Listing 2-2. This code introduces another new function from the Processing language called mousePressed(), where we can put code that should be run only when the user presses the mouse (or taps the touchscreen). We will see a few more interaction-handling examples in this chapter and then will deep-dive into touchscreen interaction more specific for mobile development in Chapter 5.

Listing 2-2. Pausing/resuming the animation loop

```
int x = 0;

void setup() {
  fullScreen();
  strokeWeight(2);
  stroke(255);
}

void draw() {
  background(50);
  line(x, 0, x, height);
  x = x + 1;
  if (width < x) x = 0;

}
```

```
void mousePressed() {
  if (looping) {
    noLoop();
  } else {
    loop();
  }
}
```

We can also write static sketches without setup/draw, which are useful if we only want to create a fixed composition that does not need to be animated. Processing runs the code in these sketches only one time. Listing 2-3 contains a simple static sketch that draws the white circle in Figure 2-2. Here, we use the function ellipse() to draw a circle centered at coordinates (400, 400) and width and height equal to 300 pixels (by setting different width and height values, we would get an ellipse of any proportions we like.) We use the fill() function to paint the interior of the circle with a purple color. We also use the size() function, which allows us to set the width and height of the output canvas. We apply this function instead of fullScreen() if we only want to use an area of the device's screen for drawing and interaction. Processing will paint the pixels outside of the drawing area with a light gray color, as seen in Figure 2-2, and we will not be able to change this color. Tapping on this area will not result in any interaction events with the sketch.

Listing 2-3. Static sketch without setup() and draw() functions

```
size(1000, 1000);
background(255);
fill(150, 100, 250);
ellipse(500, 500, 400, 400);
```

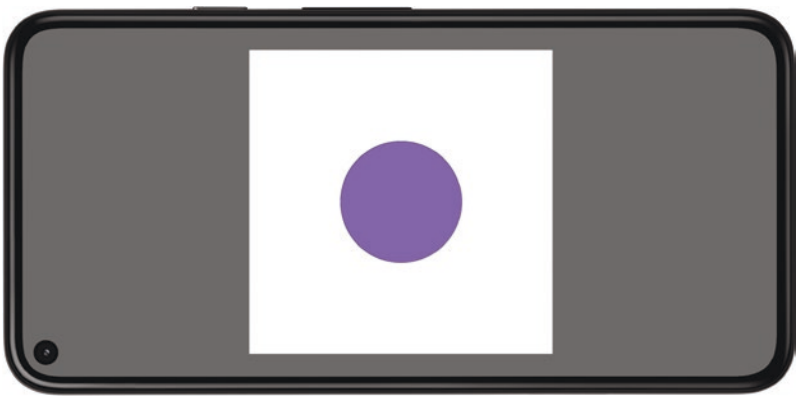

Figure 2-2. *Output of the static sketch in Listing 2-3*

Drawing with Code

We just saw in the examples from the previous section how the Processing language allows us to "draw with code." Even though these examples were very simple, they already point to some general concepts for code-based drawing in Processing. First, we need to specify the coordinates of the elements we want to draw on the screen. Second, there are functions, such as `line()` and `ellipse()`, that allow us to draw various shapes

by setting numerical values that define their form, size, and position. Third, we can set the visual "style" of these shapes (e.g., stroke and fill color) by using functions like `fill()`, `stroke()`, and `strokeWeight()`. In the next sections, we will delve into these techniques in more depth to learn how to draw different kinds of shapes.

Coordinates

One of the most important concepts in code-based drawing is knowing how to locate points on the computer screen using numeric coordinates. Processing draws the graphical output of our code into a rectangular grid of pixels, numbered from 0 to width-1 along the horizontal direction (the x axis) and 0 to height-1 along the vertical direction (the y axis), illustrated in Figure 2-3. As we saw before, we can set the precise width and height of this grid by using the size() function with our desired values as arguments. If we use the fullScreen() function instead, then the width and height of the grid will be automatically set to the width and height of the entire screen of our device.

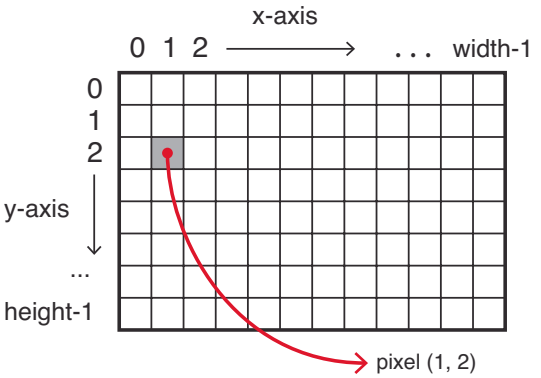

Figure 2-3. *Diagram of the screen's pixels*

The X coordinates run from left to right, while the Y coordinates run from top to bottom. So the pixel (0, 0) represents the upper left corner of the screen, and the pixel (width-1, height-1) represents the lower right corner. The arguments of most of the 2D drawing functions in Processing refer to the pixel coordinates of the screen. Processing also offers two internal constants, conveniently called width and height, that hold the values we set in the size() function or determined automatically when using fullScreen(). The following "toy" sample code, where we set a very small output size so we can see each single pixel, would produce the output in Figure 2-4 (although the actual output of running this code on a computer would be so small that we would likely not be able to differentiate its parts):

```
size(12, 8);
stroke(200, 0, 0);
fill(100, 200, 100);
rect(2, 1, width - 1, height - 2);
```

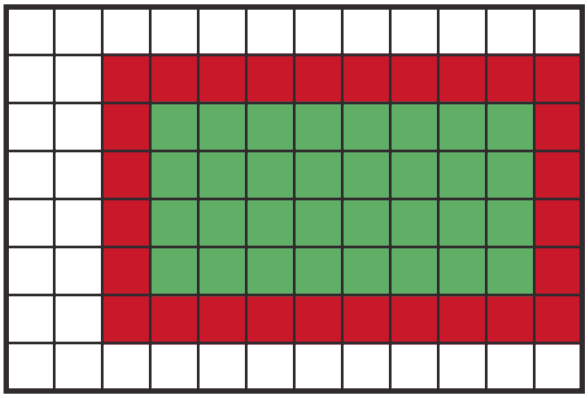

Figure 2-4. *Pixels covered by a stroked rectangle in Processing*

We should set the size of the shapes we draw with Processing according to the constraints of the screen size. In general, it is recommended to use the `width` and `height` constants when referring to the size of the screen instead of entering predetermined values, because the width and height of our output will likely change between devices, especially if we use the fullScreen() function. Even when we set the width and height in the size() function, if we later change the arguments of size(), we would then need to go through the code updating any references to the original width and height values. This won't be necessary if we use the width and height constants. The code in Listing 2-4 provides a demonstration of this technique. With this code, we generate a rectangular checkboard of 4×4 by drawing rectangles that have size equal to a quarter of the width and height of the output canvas. The result is shown in Figure 2-5, as it would appear on a Pixel 4a phone. If we change the values in size(1000, 1000) to something else, the result would still be a grid of 4×4 rectangles, because those rectangles will be resized automatically according to the value of width and height.

Listing 2-4. Using screen coordinates

```
size(1000, 1000);
noStroke();
fill(255);
background(0);
rect(0, 0, width/4, height/4);
rect(width/2, 0, width/4, width/4);
rect(width/4, height/4, width/4, height/4);
rect(3*width/4, height/4, width/4, height/4);
rect(0, height/2, width/4, height/4);
rect(width/2, height/2, width/4, width/4);
rect(width/4, 3*height/4, width/4, height/4);
rect(3*width/4, 3*height/4, width/4, height/4);
```

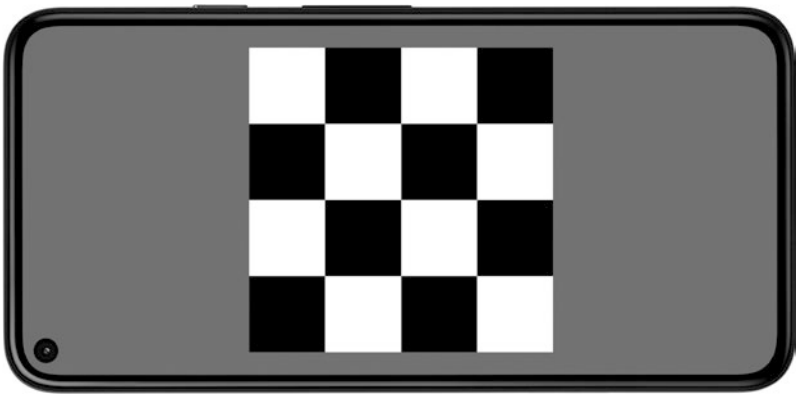

Figure 2-5. *Using screen coordinates to draw a grid of rectangles*

Form

All the shapes we draw in Processing have a form in two or three dimensions. One way to think of a shape is as a perimeter or boundary that separates the inside of the shape from the outside. We already saw that Processing includes functions to draw common shapes such as rectangles or ellipses by providing the coordinates of their center and size. Processing will automatically construct the perimeter of these shapes for us. But to construct the perimeter of an arbitrary shape in Processing, we need to explicitly provide the coordinates of the vertices alongside the perimeter. We can do this by listing the vertices between the `beginShape()` and `endShape()` functions as shown in Listing 2-5, whose output is presented in Figure 2-6.

Listing 2-5. Using beginShape() and endShape()

```
size(600, 300);

strokeWeight(2);

// Draw a rectangle in the bottom of the screen
beginShape();
vertex(5, 250);
vertex(590, 250);
vertex(590, 290);
vertex(5, 290);
endShape(CLOSE);

// Draw a star-like shape in right side
beginShape();
vertex(330, 25);
vertex(390, 90);
vertex(510, 10);
vertex(460, 120);
vertex(510, 270);
vertex(410, 180);
vertex(310, 270);
vertex(360, 150);
endShape(CLOSE);
```

```
// Draw a small triangle right above the star shape
beginShape();
vertex(350, 30);
vertex(410, 30);
vertex(390, 75);
endShape();

ellipse(70, 80, 70, 70);
```

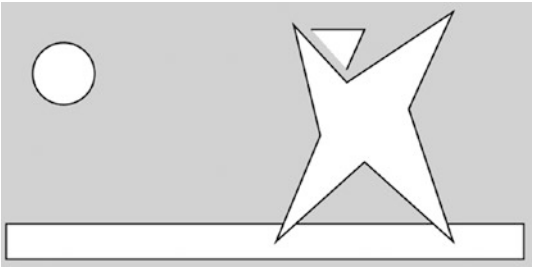

Figure 2-6. *Composition created by drawing several shapes*

We can learn a few important lessons from this example. First, shapes are drawn on top of each other according to the order they appear in the code; this is why the rectangle in the bottom, which is the first shape to be drawn, is partially obstructed by the star-like shape, which is drawn afterwards. Second, shapes can be open or closed. When a shape is open, the stroke line that goes around its perimeter will not connect the last and the first vertices, as we can see in the triangle shape. But if we add the argument CLOSE in the endShape() function, which is the case for the two first shapes, the stroke line will wrap around and result in an uninterrupted boundary delineating the shape. Third, we can combine common (also called "primitive") shapes drawn with functions like ellipse() or rect() with arbitrary shapes drawn with beginShape/endShape.

By applying some basic trigonometry, we can draw shapes that are regular polygons. The vertices alongside the perimeter of a regular polygon have coordinates `x = r * cos(a)` and `y = r * sin(a)`, where r is the radius of the circumference, sin() and cos() are the sine and cosine functions, and a is an angle between 0 and 360 degrees (for a great intro/refresher about trigonometry, check Chapter 3 from Daniel Shiffman's book *The Nature of Code*, available online at https://natureofcode.com/book/chapter-3-oscillation/). For example, in Listing 2-6, we draw three regular polygons: a square, a hexagon, and an octagon.

Listing 2-6. Drawing regular polygons

```
size(900, 300);

fill(200, 100, 100);
beginShape();
for (float angle = 0; angle <= TWO_PI; angle += TWO_PI/4) {
  float x = 150 + 100 * cos(angle);
  float y = 150 + 100 * sin(angle);
  vertex(x, y);
}
endShape(CLOSE);
```

```
fill(100, 200, 100);
beginShape();
for (float angle = 0; angle <= TWO_PI; angle += TWO_PI/6) {
  float x = 450 + 100 * cos(angle);
  float y = 150 + 100 * sin(angle);
  vertex(x, y);
}
endShape(CLOSE);

fill(100, 100, 200);
beginShape();
for (float angle = 0; angle <= TWO_PI; angle += TWO_PI/8) {
  float x = 750 + 100 * cos(angle);
  float y = 150 + 100 * sin(angle);
  vertex(x, y);
}
endShape(CLOSE);
```

Here, we use a for loop to iterate over the angle that gives us the polygon vertices alongside its perimeter. In Processing, because it's built on top of the Java language, we can use all the control structures from Java (if/else, for, etc.) that we need for code-based drawing. So in this loop, we increase the angle variable by a fraction of 2π (which is 360 degrees expressed in radians and represented in Processing by the TWO_PI constant) until we go around the entire perimeter of the polygon, calculating its vertices with the cos/sin formula at each value of the angle. We can see the output of this code in Figure 2-7.

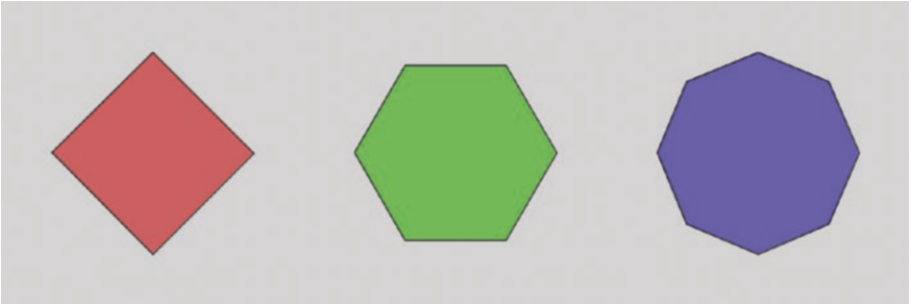

Figure 2-7. *Output of the polygon code example*

Color

Color is a fundamental element of visual design, and Processing provides many functions to let us set the color of the interior of our shapes (the fill color) and their edges (the stroke color), in addition to the background color of the entire output canvas.

By default, we can set colors using RGB (red, green, and blue) values between 0 and 255, as illustrated in the code of Listing 2-7 and its output in Figure 2-8. As we saw at the beginning of this chapter, if we pass a single value to the background(), fill(), or stroke() function, we will get a gray color (with the extremes of black and white when the value is 0 and 255, respectively). This is equivalent to passing the same number (e.g., (0, 0, 0)) as the three RGB values.

Listing 2-7. Setting fill and stroke colors using RGB values

```
size(600, 300);
strokeWeight(5);
fill(214, 87, 58);
stroke(53, 124, 115);
rect(10, 10, 180, 280);
stroke(115, 48, 128);
fill(252, 215, 51);
rect(210, 10, 180, 280);
stroke(224, 155, 73);
fill(17, 76, 131);
rect(410, 10, 180, 280);
```

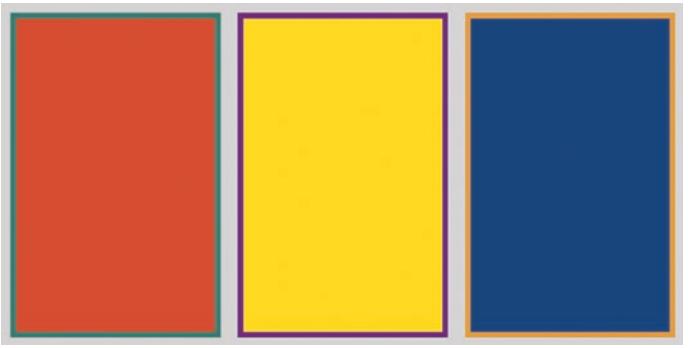

Figure 2-8. *Output of setting stroke and fill RGB colors*

We can also set an optional fourth parameter in the stroke() and fill() functions. This parameter represents the transparency or "alpha value" of the color and allows us to draw semitransparent shapes, as it is demonstrated in Listing 2-8, with its output in Figure 2-9. If we are using a single number to paint with a gray hue, Processing will understand the second parameter in stroke() or fill() as the alpha value.

Listing 2-8. Using color transparency

```
size(600, 600);

background(255);

fill(0, 0, 255);
ellipse(525, 300, 150, 150);

fill(0, 255, 0, 120);
ellipse(450, 300, 270, 270);

fill(255, 0, 0, 50);
ellipse(300, 300, 500, 500);

fill(255, 0);
ellipse(300, 300, 600, 600);
```

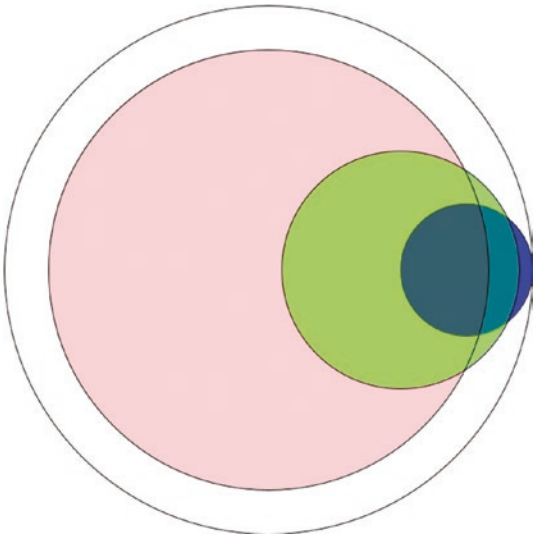

Figure 2-9. *Output of setting alpha values in fill()*

As we have seen so far, we can draw shapes of different kinds (points, lines, polygons) and set not only their fill and stroke color but other style attributes or parameters as well. We used the stroke weight before, but there are many more attributes we will learn about later. We can think of these attributes as "style parameters" that once they are set, they affect everything drawn afterward. For example, each circle in Listing 2-9 has a different fill color, but if we comment out the second fill() call, then the first and second circles will be red, since the fill color set at the beginning affects the first two ellipse calls. Figure 2-10 shows the outputs of this sketch in these two situations, with the left half beginning the output with the second fill() enabled and the right half showing the output of the second fill() commented out.

Listing 2-9. Setting the fill color attribute

```
size(460, 200);
strokeWeight(5);
fill(200, 0, 0);
stroke(255);
ellipse(100, 100, 190, 190);
fill(255, 0, 200); // Comment this line out to make second circle red
stroke(0);
ellipse(250, 100, 100, 100);
fill(0, 200, 200);
ellipse(380, 100, 150, 150);
```

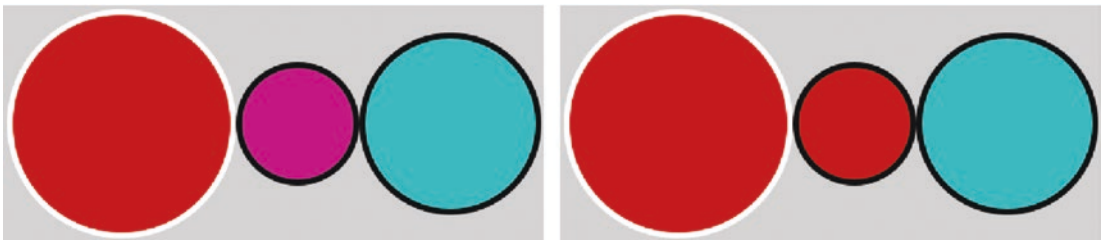

Figure 2-10. *Effect of the calling order of the fill() function*

Even though we can create almost any imaginable color using the RGB values, it can be hard to find the right combination of numbers for the color we need. Processing includes a handy Color Selector tool to help us pick a color interactively, which we can then copy into our sketches as RGB values. The Color Selector is available, alongside any other installed tool, under the "Tools" menu in the PDE (Figure 2-11).

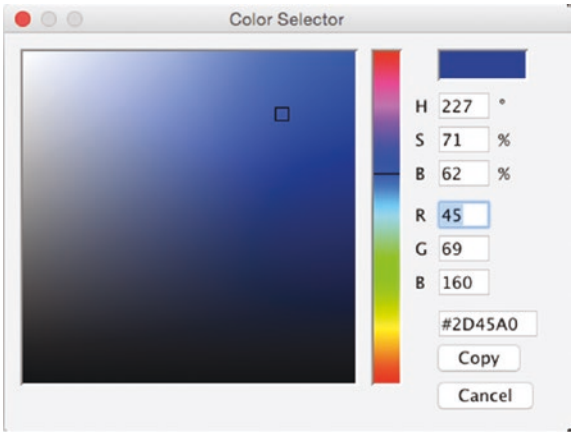

Figure 2-11. *Color Selector tool*

■ **Note** We can also specify colors in hexadecimal (hex) format, often used in web development, that is, fill(#FF0000) or stroke(#FFFFFF). The Code Selector tool provides the selected color in hex format as well.

Applying Geometric Transformations

So far, we have learned how to draw shapes and paint them with fill and stroke colors. In addition to all of this, Processing allows us to move our shapes around and change their size by applying translations, rotations, and scaling transformations (Figure 2-12).

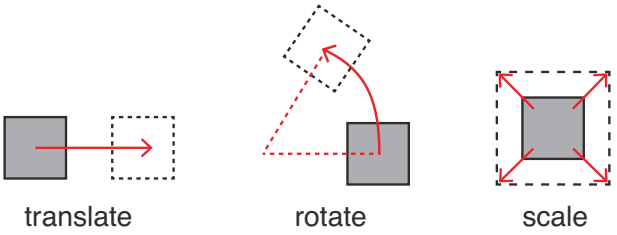

Figure 2-12. *The three types of geometric transformations*

Listings 2-10, 2-11, and 2-12 show applications of the translate(), rotate(), and scale() functions that we can use to apply translations, rotations, and scaling to any of the shapes we draw in our code. In all of them, we put the background() function in setup(); in this way, the canvas is cleared only once at the beginning, and so we can see the traces of the shapes as they move across the screen in each frame. The outputs of these examples are in Figure 2-13.

Listing 2-10. Translation example

```
float x = 0;

void setup() {
  size(400, 400);
  background(150);
}

void draw() {
  translate(x, 0);
  rect(0, 0, 300, 300);
  x += 2;
  if (width < x) x = -300;
}
```

Listing 2-11. Rotation example

```
float angle = 0;

void setup() {
  size(400, 400);
  background(150);
}

void draw() {
  rotate(angle);
  rect(0, 0, 300, 300);
  angle += 0.01 * PI;
}
```

Listing 2-12. Scaling example

```
f float s = 0;

void setup() {
  size(400, 400);
  background(150);
}

void draw() {
  scale(s);
  rect(0, 0, 10, 10);
  s += 1;
  if (100 < s) s = 0;
}
```

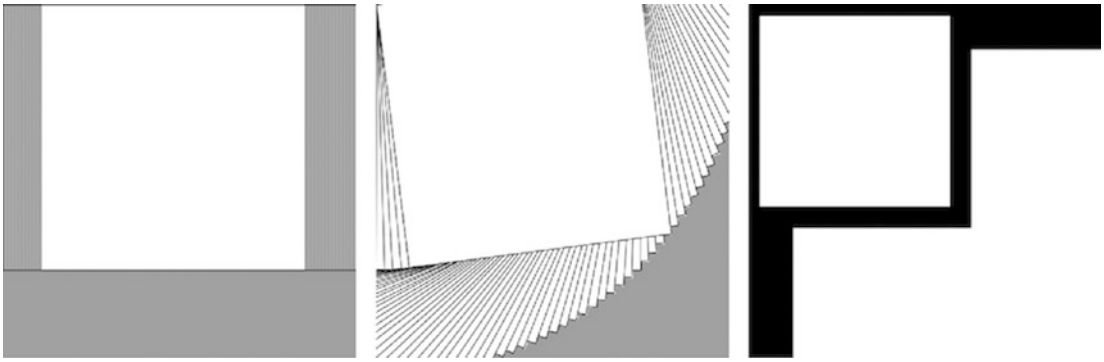

Figure 2-13. *Outputs of the translation (left), rotation (center), and scale (right) examples*

The result of applying a single translation, rotation, or scaling transformation is easy to understand; however, we may find harder to predict the effect of many transformations one after another. A way that could be helpful to think about geometric transformations is that they change the entire coordinate system in the Processing canvas. For instance, if we apply a translation of 20 units along the x axis and 30 units along the y axis and then apply a rotation, the shapes will rotate around point (20, 30) instead of (0, 0), which is the default center for 2D rotations. But if the rotation is applied before the translation, then the X and Y axes will be rotated, and the translation will be applied along the rotated axes. As a result of this, if we draw a shape at the end of a sequence of translate(), rotate(), and scale() calls, its final position will be different if we modify the order of the transformations. Figure 2-14 illustrates this situation with a rectangle drawn after translate() and rotate() calls in different orders. Even though this concept can be difficult to fully grasp at the beginning, it will become clearer with practice.

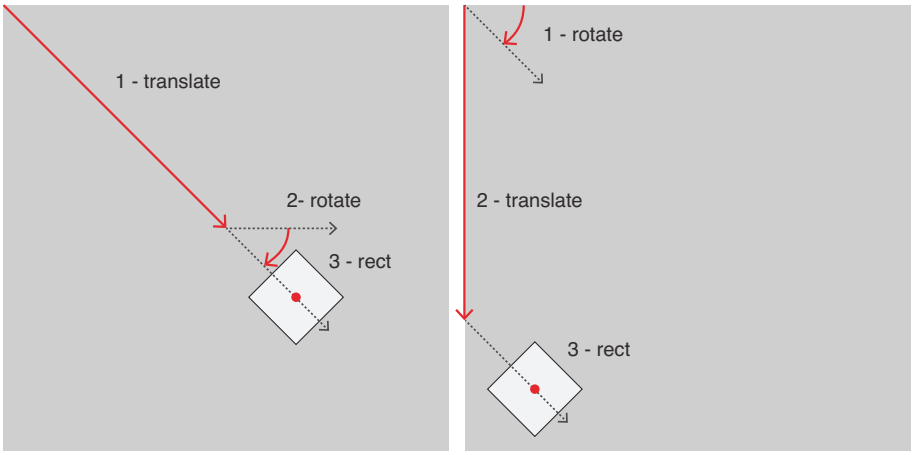

Figure 2-14. *The order of geometric transformations affects the result*

We can save the current transformation "state" with the `pushMatrix()` function and restore it with the corresponding `popMatrix()` function. We must always use these two functions in pairs, which allow us to create complex relative movements by setting transformations only to specific subsets of the shapes. For example, Listing 2-13 generates an animation of an ellipse and square rotating around a larger square placed at the center of the screen, with the smaller square also rotating around its own center. Figure 2-15 shows a snapshot of this animation.

Listing 2-13. Using pushMatrix() and popMatrix()

```
float angle;

void setup() {
  size(400, 400);
}

void draw() {
  background(170);
  translate(width/2, height/2);
  rotate(angle);
  rect(0, 0, 70, 70);
  pushMatrix();
    line(0, 0, 130, 0);
    translate(130, 0);
    rotate(2 * angle);
    rect(0, 0, 30, 30);
    pushMatrix();
      rotate(angle);
      line(0, 0, 50, 0);
      ellipse(50, 0, 15, 15);
    popMatrix();
```

```
  popMatrix();
  rotate(angle);
  line(0, 0, 0, 180);
  translate(0, 180);
  ellipse(0, 0, 30, 30);
  angle += 0.01;
}
```

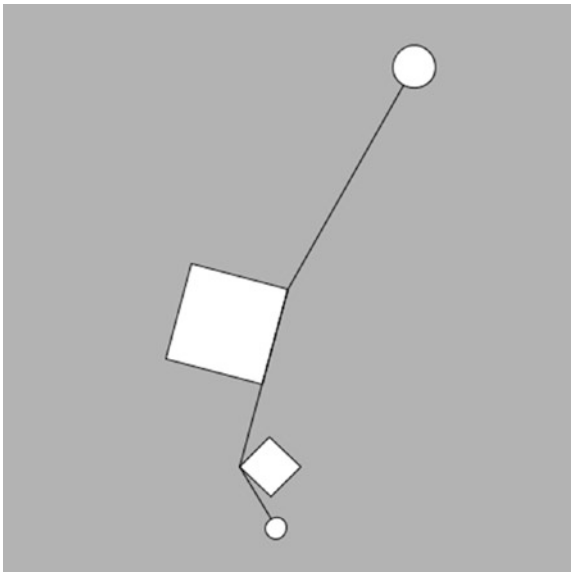

Figure 2-15. *Using pushMatrix() and popMatrix() to keep transformations separate*

■ **Note** For readability of the code, we can consider indenting the code inside push and popMatrix(); many people use this style to know where a set of transformations start and end. But this is completely optional.

Responding to User Input

There are many ways to capture user input into our sketch. Keyboard and mouse (touchscreen in the case of mobile phones) are the most common. Processing provides several built-in variables and functions to respond to user input. For example, the variables `mouseX` and `mouseY` give us the position of the mouse pointer or touchscreen tap. In the Android mode, these variables represent the position of the first touch point on the screen (Processing also supports multitouch interaction, which is covered in Chapter 5). Both mouseX and mouseY are complemented with `mousePressed`, which indicates whether the mouse/touchscreen is being pressed. Using these variables, we can create a drawing sketch with very little code, like the one in Listing 2-14, where we simply draw a semitransparent ellipse at the (mouseX, mouseY) position. Its output on a phone would look like the one shown in Figure 2-16.

Listing 2-14. A free-hand drawing sketch using circles

```
void setup() {
  fullScreen();
  noStroke();
  fill(255, 100);
}

void draw() {
  if (mousePressed) {
    ellipse(mouseX, mouseY, 50, 50);
  }
}
```

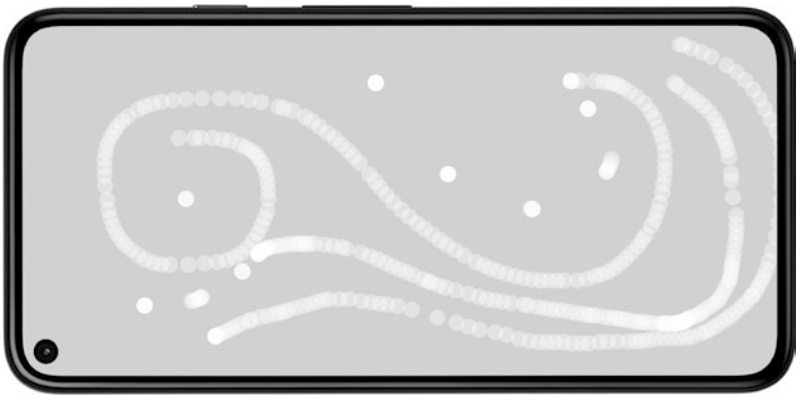

Figure 2-16. *Drawing with ellipses*

While mouseX/Y stores the current position of the mouse/touch, Processing also provides the variables `pmouseX` and `pmouseY`, which store the previous position. By connecting the pmouseX/Y coordinates with the current ones in mouseX/Y, we can draw continuous lines that follow the movement of the mouse or touch pointer. Listing 2-15 illustrates this technique, with its output in Figure 2-17.

Listing 2-15. Using current and previous mouse positions

```
void setup() {
fullScreen();
strokeWeight(5);
stroke(255, 100);
}

void draw() {
  if (mousePressed) {
    line(pmouseX, pmouseY, mouseX, mouseY);
  }
}
```

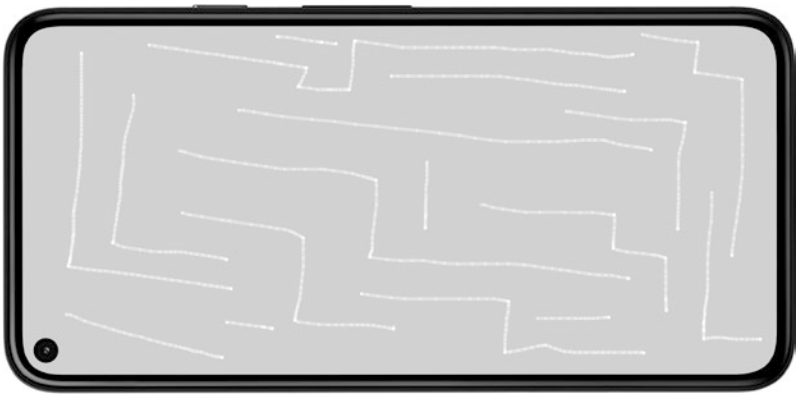

Figure 2-17. *Output of the drawing sketch where current and previous mouse positions are connected to create continuous lines*

We have one last example of a drawing sketch in Listing 2-16. Here, we also add a circle at the current mouse/touch pointer positions. This results in a line that resembles a chain made of connected links (Figure 2-18). The possibilities are virtually endless; it's only up to our imagination to come up with cool ways in which we can convert user input into interactive graphics using code!

Listing 2-16. Another free-hand drawing sketch

```
void setup() {
  fullScreen();
  strokeWeight(2);
  stroke(0);
  fill(255);
}

void draw() {
  if (mousePressed) {
    line(pmouseX, pmouseY, mouseX, mouseY);
    ellipse(mouseX, mouseY, 20, 20);
  }
}
```

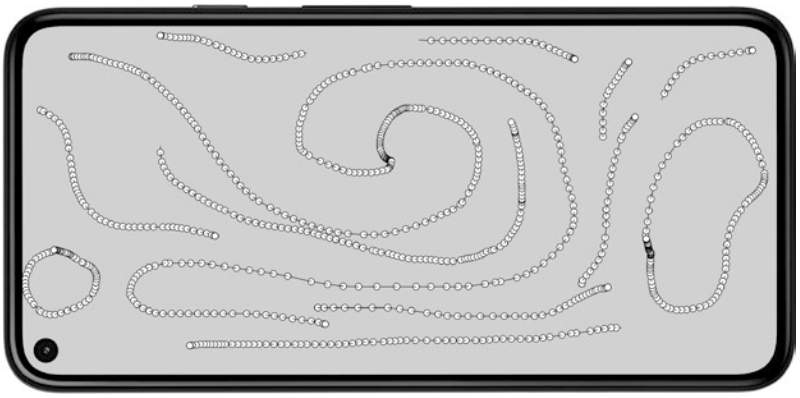

Figure 2-18. *Combining continuous lines with ellipses in drawing sketch*

Coding a "Vine Drawing" App

Our goal in this final section is to code a drawing app that incorporates code-generated shapes into hand-drawn lines. One possibility is to augment the scaffold provided by the lines with shapes that resemble growing vegetation, such as vines and leaves. Instead of trying to arrive to a fully organic look, which may be hard to do using only the functions we learned so far, we could limit our drawing to regular shapes that still suggest vegetation through color and randomness. Some sketching with pen and paper (Figure 2-19) may also help us to explore some visual ideas.

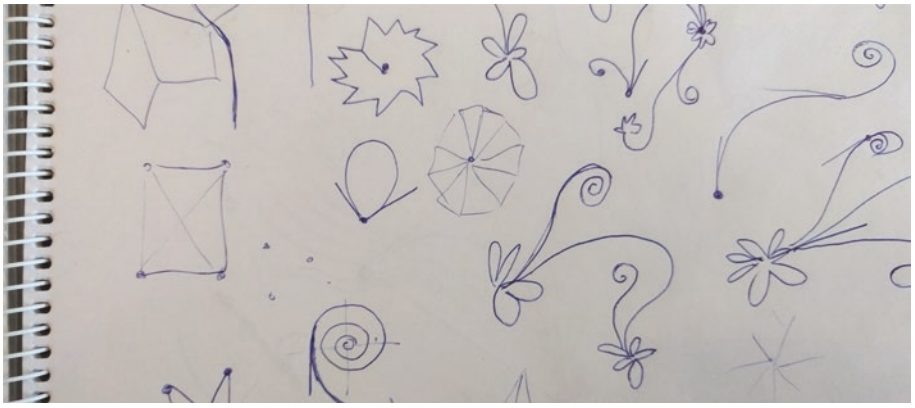

Figure 2-19. *Sketches for the vine drawing app*

We could draw leaves as clusters of regular polygons with begin/endShape, sprouting from random positions alongside the hand-drawn lines representing the main branches in the vine. Here, the random() function in Processing will be very useful to add some randomness to our shapes, so the resulting leaves do not look too regular. This function returns a floating-point (i.e., with decimal digits) random number between a minimum and a maximum value. For example, calling random(10, 20) would give us a random number between 10 and 20, such as 16.8. Every time we run random(), we will get a different result.

Let's draw a bunch with a random number of leaves. Also, the polygon representing each leaf could have a random number of sides. The code in Listing 2-17 contains code that does these two things, and Figure 2-20 shows the several different outputs from this code.

Listing 2-17. Drawing randomized leaves with polygons

```
size(600, 600);
translate(width/2, height/2);
int numLeaves = int(random(4, 8));
for (int i = 0; i < numLeaves; i++) {
  pushMatrix();
    float leafAngle = random(0, TWO_PI);
    float leafLength = random(100, 250);
    rotate(leafAngle);
    line(0, 0, leafLength, 0);
    translate(leafLength, 0);
    pushStyle();
      noStroke();
      fill(255, 190);
      float r = random(50, 100);
      beginShape();
      int numSides = int(random(4, 8));
      for (float angle = 0; angle <= TWO_PI; angle += TWO_PI/numSides) {
        float x = r * cos(angle);
        float y = r * sin(angle);
        vertex(x, y);
      }
      endShape();
    popStyle();
  popMatrix();
}
```

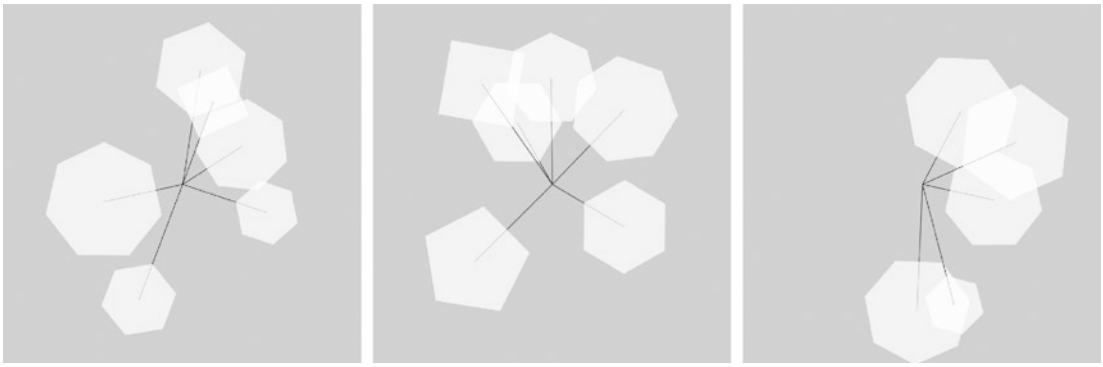

Figure 2-20. *Different outputs of the leaf drawing sketch*

This code gives us a reasonable range of variability in the output, but without being completely random. There are a couple of new things to note in this code. First, some of the random numbers need to be integer, for example, the number of leaves. So in that case, we can use the int() function in Processing to convert the

floating-point result from random() into a whole number. Second, we are using another pair of functions called pushStyle() and popStyle(). Like what push/popMatrix does with transformations, push/popStyle keeps changes in style (e.g., fill and line color) separate so the changes do not affect any shapes drawn outside the push/pop block.

We already explored a few options for hand-drawn lines in the previous section. Listing 2-16 produced an interesting output by combining continuous lines with circles. If we set the lines and leaves to have bark (brownish) and green colors while also introducing some random variability, we may be able to get a complete vine drawing sketch. To do so, we could just take the code in Listing 2-17 and copy it inside a user-defined function that gets called every time we need a bunch of leaves alongside the hand-drawn lines.

One final question is when to draw the leaves. In order to keep our sketch simple, we could make this step automatic, for example, by using the random() function, but applying randomness all the time is not always a good idea because the output would feel too random. It's important that we take purposeful decisions in our code and do not leave everything to chance! We could try to use the user input so that if the mouse or touch pointer is moving quickly (e.g., the vine is growing fast), then new leaves are added to the drawing. An easy way to do this is calculating the distance between the current and previous mouse positions using the dist() function in Processing just like so: `dist(pmouseX, pmouseY, mouseX, mouseY)`. Listing 2-18 puts all this together, and Figure 2-21 shows a sample drawing made with the sketch.

Listing 2-18. Full vine drawing sketch

```
void setup() {
  fullScreen();
  strokeWeight(2);
  stroke(121, 81, 49, 150);
  fill(255);
  background(255);
}

void draw() {
  if (mousePressed) {
    line(pmouseX, pmouseY, mouseX, mouseY);
    ellipse(mouseX, mouseY, 13, 13);
    if (30 < dist(pmouseX, pmouseY, mouseX, mouseY)) {
      drawLeaves();
    }
  }
}

void drawLeaves() {
  int numLeaves = int(random(2, 5));
  for (int i = 0; i < numLeaves; i++) {
    float leafAngle = random(0, TWO_PI);
    float leafLength = random(20, 100);
    pushMatrix();
      translate(mouseX, mouseY);
      rotate(leafAngle);
      line(0, 0, leafLength, 0);
      translate(leafLength, 0);
      pushStyle();
        noStroke();
        fill(random(170, 180), random(200, 230), random(80, 90), 190);
        float r = random(20, 50);
```

```
        beginShape();
        int numSides = int(random(4, 8));
        for (float angle = 0; angle <= TWO_PI; angle += TWO_PI/numSides) {
          float x = r * cos(angle);
          float y = r * sin(angle);
          vertex(x, y);
        }
        endShape();
      popStyle();
    popMatrix();
  }
}
```

Figure 2-21. *Output of the vine drawing sketch*

Summary

We have now a basic knowledge of the Processing language that covers how to draw shapes, set colors, apply transformations, and handle user interaction through the mouse or touchscreen. Even though we learned only a small fraction of all the functionality available in Processing, what we saw in this chapter should give us enough tools to start exploring code-based drawing and to make our own interactive sketches and run them as Android apps.