1-3 Grayscale color

As you learned in Section 1-2 on page 5, the primary building block for placing shapes onscreen is a pixel coordinate. You politely instructed the computer to draw a shape at a specific location with a specific size. Nevertheless, a fundamental element was missing — color.

In the digital world, precision is required. Saying "Hey, can you make that circle bluish-green?" will not do. Therefore, color is defined with a range of numbers. I'll start with the simplest case: *black and white* or *grayscale*. To specify a value for grayscale, use the following: 0 means black, 255 means white. In between, every other number — 50, 87, 162, 209, and so on — is a shade of gray ranging from black to white. See Figure 1-13.

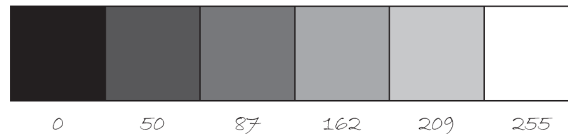


Figure 1-13

> ### Does 0–255 seem arbitrary to you?
>
> Color for a given shape needs to be stored in the computer's memory. This memory is just a long sequence of 0's and 1's (a whole bunch of on or off switches.) Each one of these switches is a *bit*, eight of them together is a *byte*. Imagine if you had eight bits (one byte) in sequence — how many ways can you configure these switches? The answer is (and doing a little research into binary numbers will prove this point) 256 possibilities, or a range of numbers between 0 and 255. Processing will use eight bit color for the grayscale range and 24 bit for full color (eight bits for each of the red, green, and blue color components; see Section 1-4 on page 12).

Understanding how this range works, you can now move to setting specific grayscale colors for the shapes you drew in Section 1-2 on page 5. In Processing, every shape has a `stroke()` or a `fill()` or both. The `stroke()` specifies the color for the outline of the shape, and the `fill()` specifies the color for the interior of that shape. Lines and points can only have `stroke()`, for obvious reasons.

If you forget to specify a color, Processing will use black (0) for the `stroke()` and white (255) for the `fill()` by default. Note that I'm now using more realistic numbers for the pixel locations, assuming a larger window of size 200 × 200 pixels. See Figure 1-14.

```
rect(50, 40, 75, 100);
```

By adding the `stroke()` and `fill()` functions *before* the shape is drawn, you can set the color. It's much like instructing your friend to use a specific pen to draw on the graph paper. You would have to tell your friend *before* he or she starting drawing, not after.

There is also the function `background()`, which sets a background color for the window where shapes will be rendered.

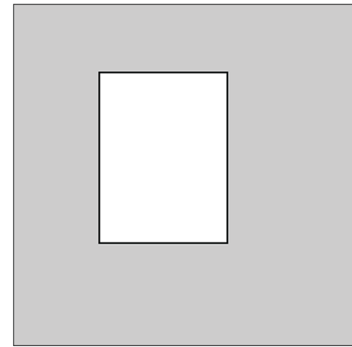


Figure 1-14

Example 1-1. Stroke and fill

```
background(255);
stroke(0);
fill(150);
rect(50, 50, 75, 100);
```

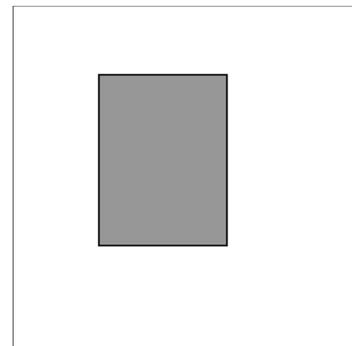


Figure 1-15

`stroke()` or `fill()` can be eliminated with the `noStroke()` or `noFill()` functions. Your instinct might be to say `stroke(0)` for no outline, however, it's important to remember that 0 is not "nothing," but rather denotes the color black. Also, remember not to eliminate both — with `noStroke()` and `noFill()`, nothing will appear!

Example 1-2. noFill()

```
background(255);
stroke(0);
noFill();
ellipse(60, 60, 100, 100);
```

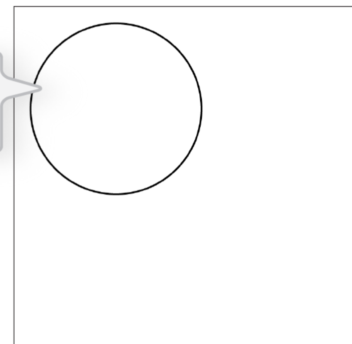


Figure 1-16

When you draw a shape, Processing will always use the most recently specified `stroke()` and `fill()`, reading the code from top to bottom. See Figure 1-17.

```
background(150);
stroke(0);
line(0, 0, 200, 200);
stroke(255);
noFill();
rect(25, 25, 75, 75);
```

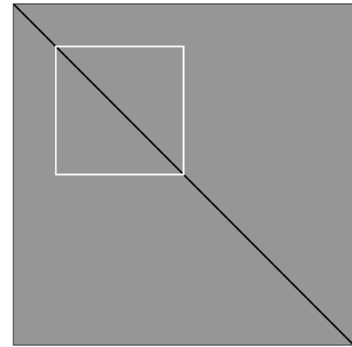


Figure 1-17

Exercise 1-4: Try to guess what the instructions would be for the following screenshot.

____

____

____

____

____

____

1-4 RGB color

A nostalgic look back at graph paper helped you to learn the fundamentals for pixel locations and size. Now that it's time to study the basics of digital color, here's another childhood memory to get you started. Remember finger painting? By mixing three "primary" colors, any color could be generated. Swirling all colors together resulted in a muddy brown. The more paint you added, the darker it got.

Digital colors are also constructed by mixing three primary colors, but it works differently from paint. First, the primaries are different: red, green, and blue (i.e., "RGB" color). And with color on the screen, you're mixing light, not paint, so the mixing rules are different as well.

- Red + green = yellow
- Red + blue = purple
- Green + blue = cyan (blue-green)
- Red + green + blue = white
- No colors = black

This assumes that the colors are all as bright as possible, but of course, you have a range of color available, so some red plus some green plus some blue equals gray, and a bit of red plus a bit of blue equals dark purple.

While this may take some getting used to, the more you program and experiment with RGB color, the more it will become instinctive, much like swirling colors with your fingers. And of course you can't say "Mix some red with a bit of blue"; you have to provide an exact amount. As with grayscale, the individual color elements are expressed as ranges from 0 (none of that color) to 255 (as much as possible), and they are listed in the order red, green, and blue. You will get the hang of RGB color mixing through experimentation, but next I will cover some code using some common colors.

Note that the print version of this book will only show you black and white versions of each Processing sketch, but all sketches can be seen online in full color at *http://learningprocessing.com*. You can also see a color version of the tutorial on the Processing website.

Example 1-3. RGB color

```
background(255);
noStroke();

fill(255, 0, 0);
ellipse(20, 20, 16, 16);

fill(127, 0, 0);
ellipse(40, 20, 16, 16);

fill(255, 200, 200);
ellipse(60, 20, 16, 16);
```

Bright red

Dark red

Pink (pale red).

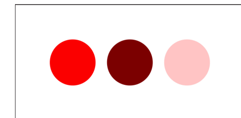


Figure 1-18

Processing also has a color selector to aid in choosing colors. Access this via "Tools" (from the menu bar) → "Color Selector." See Figure 1-19.

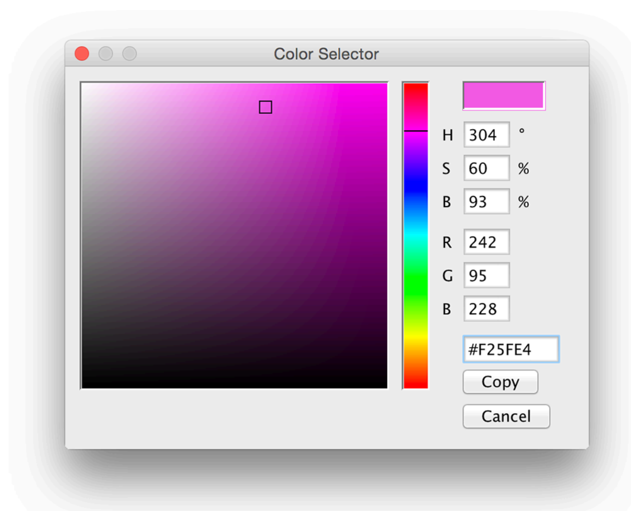


Figure 1-19

Exercise 1-5: Complete the following program. Guess what RGB values to use (you will be able to check your results in Processing after reading the next chapter). You could also use the color selector, shown in Figure 1-19.

```
fill(________,________,________);    Bright blue
ellipse(20, 40, 16, 16);

fill(________,________,________);    Dark purple
ellipse(40, 40, 16, 16);

fill(________,________,________);    Yellow
ellipse(60, 40, 16, 16);
```

Exercise 1-6: What color will each of the following lines of code generate?

```
fill(0, 100, 0);          _____________________________________

fill(100);                _____________________________________

stroke(0, 0, 200);        _____________________________________

stroke(225);              _____________________________________

stroke(255, 255, 0);      _____________________________________

stroke(0, 255, 255);      _____________________________________

stroke(200, 50, 50);      _____________________________________
```

1-5 Color transparency

In addition to the red, green, and blue components of each color, there is an additional optional fourth component, referred to as the color's "alpha." Alpha means opacity and is particularly useful when you want to draw elements that appear partially see-through on top of one another. The alpha values for an image are sometimes referred to collectively as the "alpha channel" of an image.

It's important to realize that pixels are not literally transparent; this is simply a convenient illusion that is accomplished by blending colors. Behind the scenes, Processing takes the color numbers and adds a percentage of one to a percentage of another, creating the optical perception of blending. (If you're interested in programming "rose-colored" glasses, this is where you would begin.)

Alpha values also range from 0 to 255, with 0 being completely transparent (i.e., zero percent opaque) and 255 completely opaque (i.e., 100 percent opaque). Example 1-4 shows a code example that is displayed in Figure 1-20.

Example 1-4. Opacity

```
background(0);
noStroke();

fill(0, 0, 255);          // No fourth argument means 100% opacity.
rect(0, 0, 100, 200);

fill(255, 0, 0, 255);     // 255 means 100% opacity.
rect(0, 0, 200, 40);

fill(255, 0, 0, 191);     // 75% opacity
rect(0, 50, 200, 40);

fill(255, 0, 0, 127);     // 50% opacity
rect(0, 100, 200, 40);

fill(255, 0, 0, 63);      // 25% opacity
rect(0, 150, 200, 40);
```

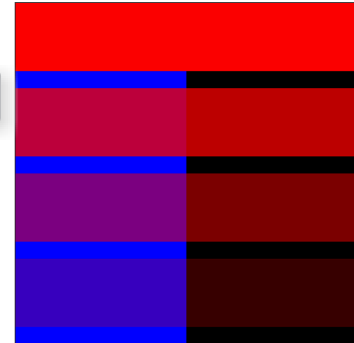


Figure 1-20

1-6 Custom color ranges

RGB color with ranges of 0 to 255 is not the only way you can handle color in Processing. Behind the scenes in the computer's memory, color is *always* talked about as a series of 24 bits (or 32 in the case of colors with an alpha). However, Processing will let you think about color any way you like, and translate any values into numbers the computer understands. For example, you might prefer to think of color as ranging from 0 to 100 (like a percentage). You can do this by specifying a custom `colorMode()`.

```
colorMode(RGB, 100);      // With colorMode() you can set your own color range.
```

The above function says: "OK, I want to think about color in terms of red, green, and blue. The range of RGB values will be from 0 to 100."

Although it's rarely convenient to do so, you can also have different ranges for each color component:

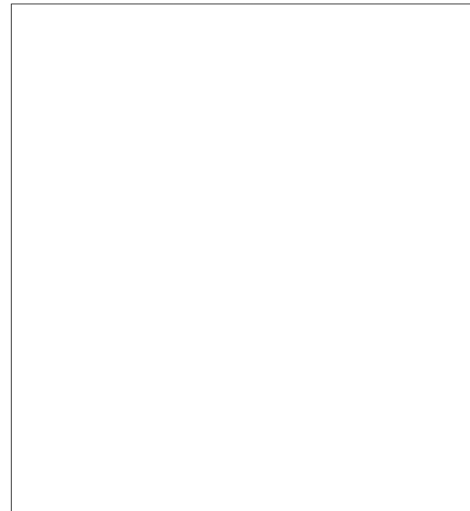
```
colorMode(RGB, 100, 500, 10, 255);
```

Now I am saying "Red values range from 0 to 100, green from 0 to 500, blue from 0 to 10, and alpha from 0 to 255."

Finally, while you will likely only need RGB color for all of your programming needs, you can also specify colors in the HSB (hue, saturation, and brightness) mode. While HSB values also default to a range of 0 to 255, a common set of ranges (with brief explanation) are as follows:

- **Hue** — The shade of color itself (red, blue, orange, etc.) ranging from 0 to 360 (think of 360° on a color "wheel").
- **Saturation** — The vibrancy of the color, 0 to 100 (think of 50%, 75%, etc.).
- **Brightness** — The, well, brightness of the color, 0 to 100.

Exercise 1-7: Design a creature using simple shapes and colors. Draw the creature by hand using only points, lines, rectangles, and ellipses. Then attempt to write the code for the creature, using the Processing commands covered in this chapter: `point()`, `line()`, `rect()`, `ellipse()`, `stroke()`, *and* `fill()`. *In the next chapter, you will have a chance to test your results by running your code in Processing.*



Example 1-5 shows my version of Zoog, with the outputs shown in Figure 1-21.

Example 1-5. Zoog

```
background(255);
ellipseMode(CENTER);
rectMode(CENTER);
stroke(0);
fill(150);
rect(100,100,20,100);
fill(255);
ellipse(100,70,60,60);
fill(0);
ellipse(81,70,16,32);
ellipse(119,70,16,32);
stroke(0);
line(90,150,80,160);
line(110,150,120,160);
```

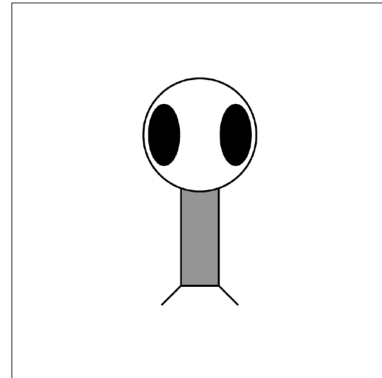


Figure 1-21

The sample answer is my Processing-born being, named Zoog. Over the course of the first nine chapters of this book, I will follow the course of Zoog's childhood. The fundamentals of programming will be demonstrated as Zoog grows up. You will first learn to display Zoog, then to make an interactive Zoog and animated Zoog, and finally to duplicate Zoog in a world of many Zoogs.

I suggest you design your own "thing" (note that there is no need to limit yourself to a humanoid or creature-like form; any programmatic pattern will do) and recreate all of the examples throughout the first nine chapters with your own design. Most likely, this will require you to change only a small portion (the shape rendering part) of each example. This process, however, should help solidify your understanding of the basic elements required for computer programs — *variables*, *conditionals*, *loops*, *functions*, *objects*, and *arrays* — and prepare you for when Zoog matures, leaves the nest, and ventures off into the more advanced topics from Chapter 10 onwards in this book.