1 Pixels

"A journey of a thousand miles begins with a single step."
—Lao-tzu

In this chapter:
– Specifying pixel coordinates.
– Basic shapes: point, line, rectangle, ellipse.
– Color: grayscale, "RGB."
– Color transparency.

Note that we are not doing any programming yet in this chapter! We are just dipping our feet in the water and getting comfortable with the idea of creating onscreen graphics with text-based commands, that is, "code"!

1.1 Graph Paper

This book will teach you how to program in the context of computational media, and it will use the development environment *Processing* (*http://www.processing.org*) as the basis for all discussion and examples. But before any of this becomes relevant or interesting, we must first channel our eighth grade selves, pull out a piece of graph paper, and draw a line. The shortest distance between two points is a good old fashioned line, and this is where we begin, with two points on that graph paper.

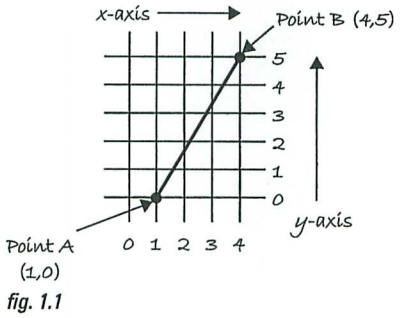


fig. 1.1

Figure 1.1 shows a line between point A (1,0) and point B (4,5). If you wanted to direct a friend of yours to draw that same line, you would give them a shout and say "draw a line from the point one-zero to the point four-five, please." Well, for the moment, imagine your friend was a computer and you wanted to instruct this digital pal to display that same line on its screen. The same command applies (only this time you can skip the pleasantries and you will be required to employ a precise formatting). Here, the instruction will look like this:

```
line(1,0,4,5);
```

Congratulations, you have written your first line of computer code! We will get to the precise formatting of the above later, but for now, even without knowing too much, it should make a fair amount of sense. We are providing a *command* (which we will refer to as a "function") for the machine to follow entitled "line." In addition, we are specifying some *arguments* for how that line should be drawn, from point

A (1,0) to point B (4,5). If you think of that line of code as a sentence, the *function* is a *verb* and the *arguments* are the *objects* of the sentence. The code sentence also ends with a semicolon instead of a period.

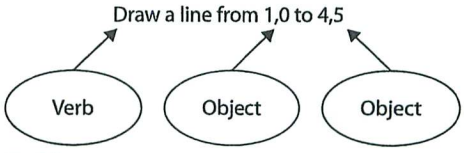


fig. 1.2

The key here is to realize that the computer screen is nothing more than a *fancier* piece of graph paper. Each pixel of the screen is a coordinate—two numbers, an "*x*" (horizontal) and a "*y*" (vertical)—that determine the location of a point in space. And it is our job to specify what shapes and colors should appear at these pixel coordinates.

Nevertheless, there is a catch here. The graph paper from eighth grade ("Cartesian coordinate system") placed (0,0) in the center with the *y*-axis pointing up and the *x*-axis pointing to the right (in the positive direction, negative down and to the left). The coordinate system for pixels in a computer window, however, is reversed along the *y*-axis. (0,0) can be found at the top left with the positive direction to the right horizontally and down vertically. See Figure 1.3.

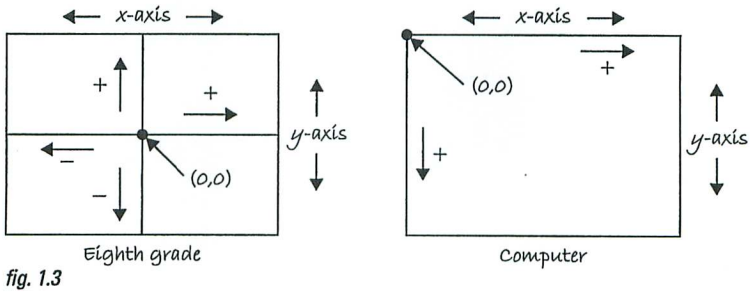


fig. 1.3

Exercise 1-1: Looking at how we wrote the instruction for line "line(1,0,4,5);" how would you guess you would write an instruction to draw a rectangle? A circle? A triangle? Write out the instructions in English and then translate it into "code."

English: _____

Code: _____

English: _____

Code: _____

English: _____

Code: _____

Come back later and see how your guesses matched up with how Processing *actually works.*

1.2 Simple Shapes

The vast majority of the programming examples in this book will be visual in nature. You may ultimately learn to develop interactive games, algorithmic art pieces, animated logo designs, and (insert your own category here) with *Processing*, but at its core, each visual program will involve setting pixels. The simplest way to get started in understanding how this works is to learn to draw primitive shapes. This is not unlike how we learn to draw in elementary school, only here we do so with code instead of crayons.

Let's start with the four primitive shapes shown in Figure 1.4.

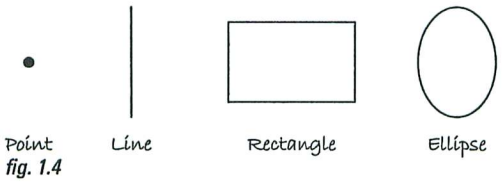


fig. 1.4

For each shape, we will ask ourselves what information is required to specify the location and size (and later color) of that shape and learn how *Processing* expects to receive that information. In each of the diagrams below (Figures 1.5 through 1.11), assume a window with a width of 10 pixels and height of 10 pixels. This isn't particularly realistic since when we really start coding we will most likely work with much larger windows (10 × 10 pixels is barely a few millimeters of screen space). Nevertheless for demonstration purposes, it is nice to work with smaller numbers in order to present the pixels as they might appear on graph paper (for now) to better illustrate the inner workings of each line of code.

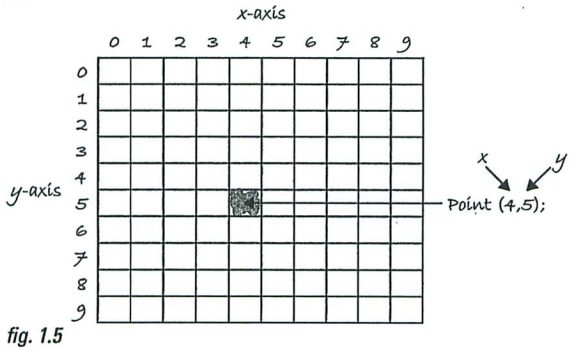


fig. 1.5

A point is the easiest of the shapes and a good place to start. To draw a point, we only need an x and y coordinate as shown in Figure 1.5. A line isn't terribly difficult either. A line requires two points, as shown in Figure 1.6.

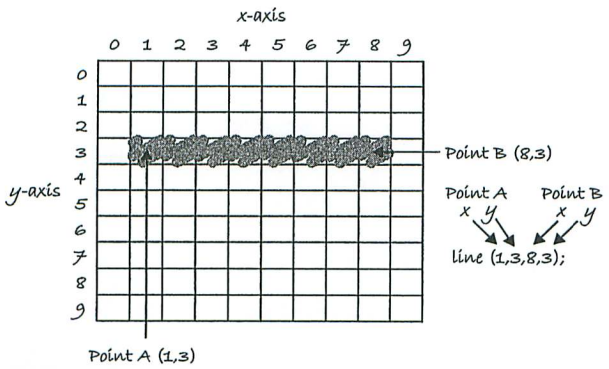


fig. 1.6

Once we arrive at drawing a rectangle, things become a bit more complicated. In *Processing*, a rectangle is specified by the coordinate for the top left corner of the rectangle, as well as its width and height (see Figure 1.7).

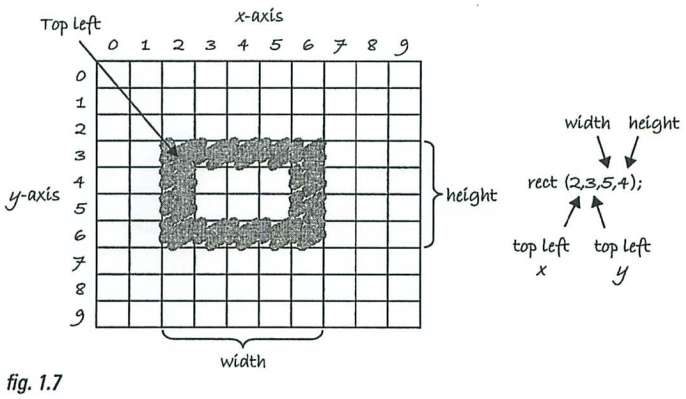


fig. 1.7

However, a second way to draw a rectangle involves specifying the centerpoint, along with width and height as shown in Figure 1.8. If we prefer this method, we first indicate that we want to use the "CENTER" mode before the instruction for the rectangle itself. Note that *Processing* is case-sensitive. Incidentally, the default mode is "CORNER," which is how we began as illustrated in Figure 1.7.

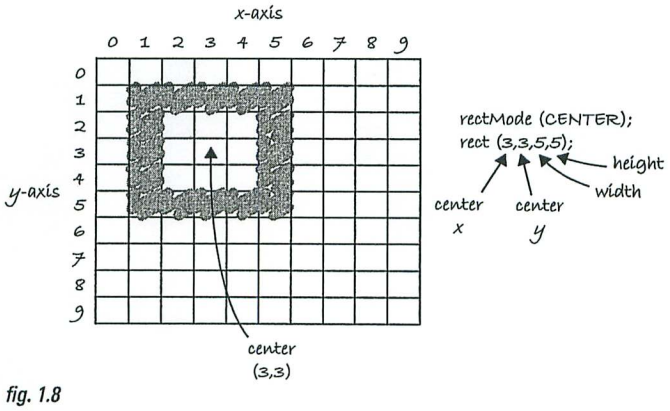


fig. 1.8

Finally, we can also draw a rectangle with two points (the top left corner and the bottom right corner). The mode here is "CORNERS" (see Figure 1.9).

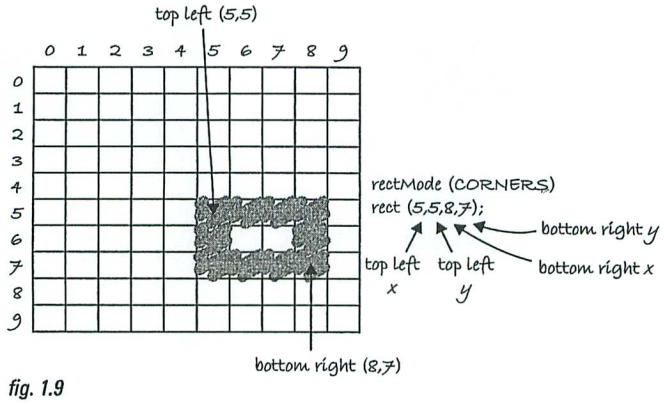


fig. 1.9

Once we have become comfortable with the concept of drawing a rectangle, an ellipse is a snap. In fact, it is identical to ***rect()*** with the difference being that an ellipse is drawn where the bounding box[1] (as shown in Figure 1.11) of the rectangle would be. The default mode for ***ellipse()*** is "CENTER", rather than "CORNER" as with ***rect()***. See Figure 1.10.

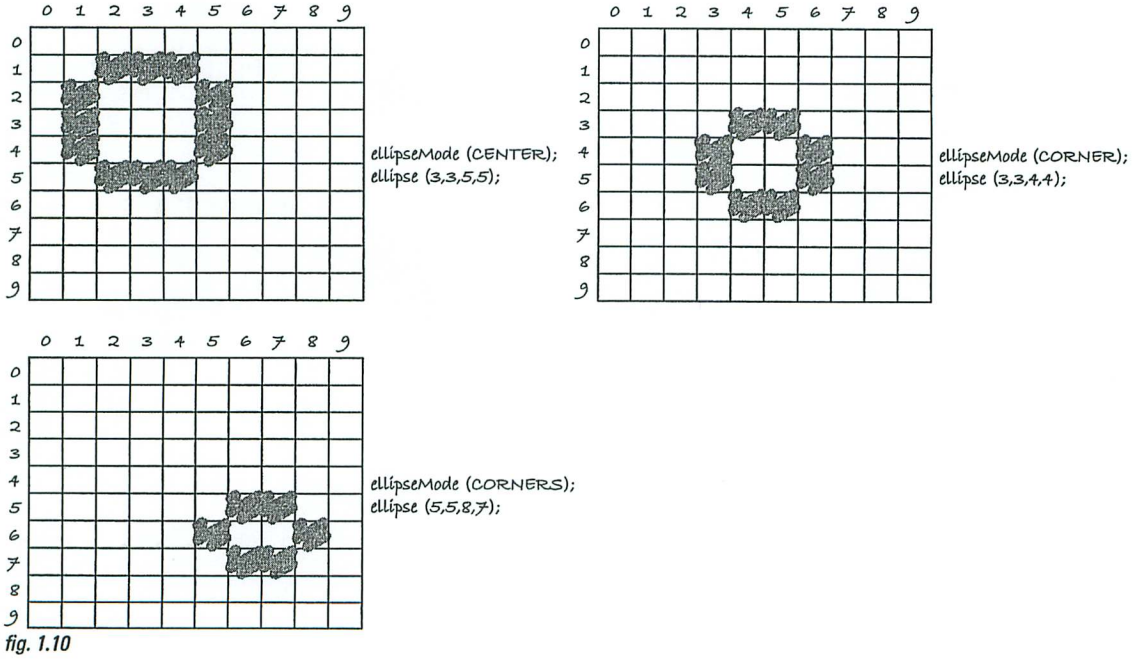


fig. 1.10

It is important to acknowledge that in Figure 1.10, the ellipses do not look particularly circular. *Processing* has a built-in methodology for selecting which pixels should be used to create a circular shape. Zoomed in like this, we get a bunch of squares in a circle-like pattern, but zoomed out on a computer screen, we get a nice round ellipse. Later, we will see that *Processing* gives us the power to develop our own

[1]A bounding box of a shape in computer graphics is the smallest rectangle that includes all the pixels of that shape. For example, the bounding box of a circle is shown in Figure 1.11.

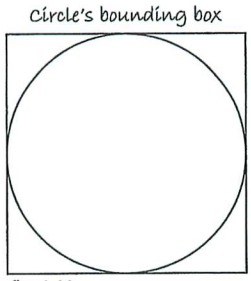


fig. 1.11

algorithms for coloring in individual pixels (in fact, we can already imagine how we might do this using "point" over and over again), but for now, we are content with allowing the "ellipse" statement to do the hard work.

Certainly, point, line, ellipse, and rectangle are not the only shapes available in the *Processing* library of functions. In Chapter 2, we will see how the *Processing* reference provides us with a full list of available drawing functions along with documentation of the required arguments, sample syntax, and imagery. For now, as an exercise, you might try to imagine what arguments are required for some other shapes (Figure 1.12):

triangle()
arc()
quad()
curve()

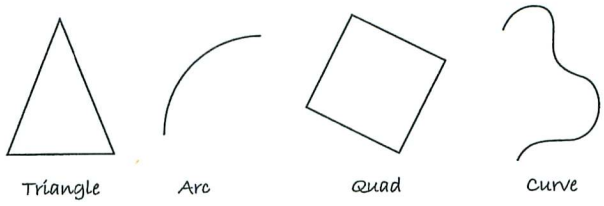
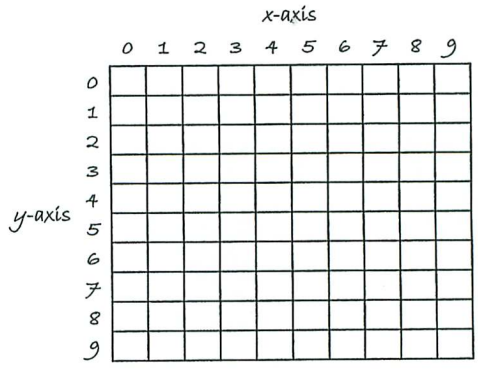


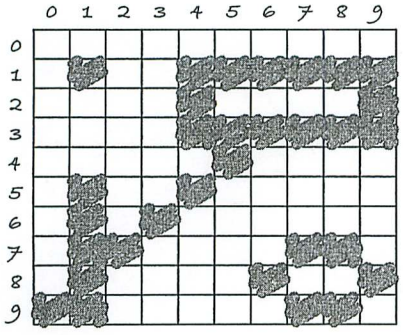
fig. 1.12

Exercise 1-2: Using the blank graph below, draw the primitive shapes specified by the code.

```
line(0,0,9,6);
point(0,2);
point(0,4);
rectMode(CORNER);
rect(5,0,4,3);
ellipseMode(CENTER);
ellipse(3,7,4,4);
```



Exercise 1-3: Reverse engineer a list of primitive shape drawing instructions for the diagram below.



Note: There is more than one correct answer!

1.3 Grayscale Color

As we learned in Section 1.2, 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. Let's start with the simplest case: *black and white* or *grayscale*. In grayscale terms, we have 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.

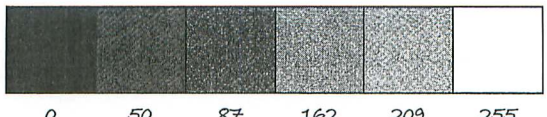


fig. 1.13

Does 0–255 seem arbitary 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 we had eight bits (one byte) in sequence—how many ways can we 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. We will use eight bit color for our grayscale range and 24 bit for full color (eight bits for each of the red, green, and blue color components; see Section 1.4).

Understanding how this range works, we can now move to setting specific grayscale colors for the shapes we drew in Section 1.2. In *Processing*, every shape has a ***stroke()*** or a ***fill()*** or both. The ***stroke()*** is the outline of the shape, and the ***fill()*** is the interior of that shape. Lines and points can only have ***stroke()***, for obvious reasons.

If we forget to specify a color, *Processing* will use black (0) for the ***stroke()*** and white (255) for the ***fill()*** by default. Note that we are 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);
```

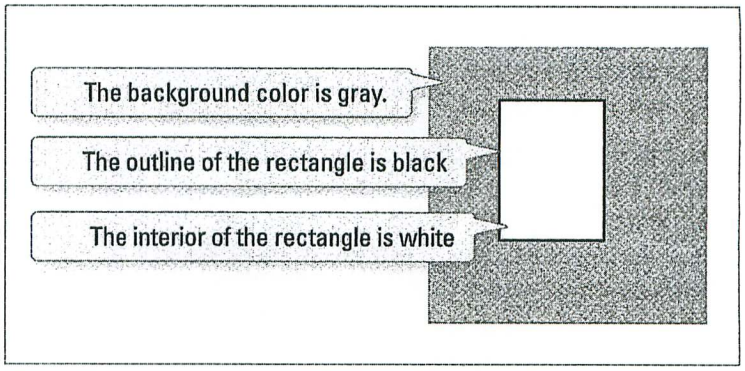


fig. 1.14

By adding the ***stroke()*** and ***fill()*** functions *before* the shape is drawn, we can set the color. It is 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.

Example 1-1: Stroke and fill

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

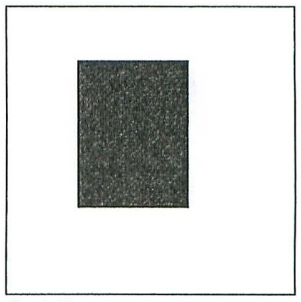


fig. 1.15

stroke() or ***fill()*** can be eliminated with the ***noStroke()*** or ***noFill()*** functions. Our instinct might be to say "***stroke(0)***" for no outline, however, it is 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);
```

nofill() leaves the shape with only an outline

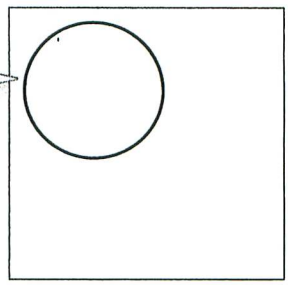


fig. 1.16

If we draw two shapes at one time, *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(0,0,100,100);
stroke(255);
noFill();
rect(25,25,50,50);
```

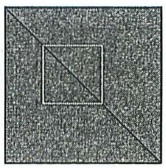
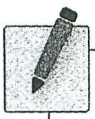
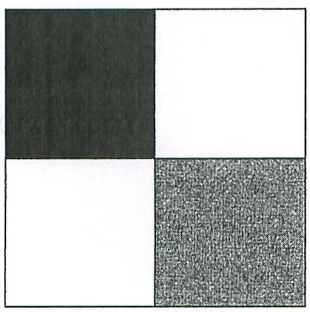


fig. 1.17



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 us learn the fundamentals for pixel locations and size. Now that it is time to study the basics of digital color, we search for another childhood memory to get us 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 are 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 R, G, and B. You will get the hang of RGB color mixing through experimentation, but next we will cover some code using some common colors.

Note that this book will only show you black and white versions of each *Processing* sketch, but everything is documented online in full color at *http://www.learningprocessing.com* with RGB color diagrams found specifically at: *http://learningprocessing.com/color*.

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).

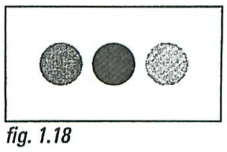


fig. 1.18

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.

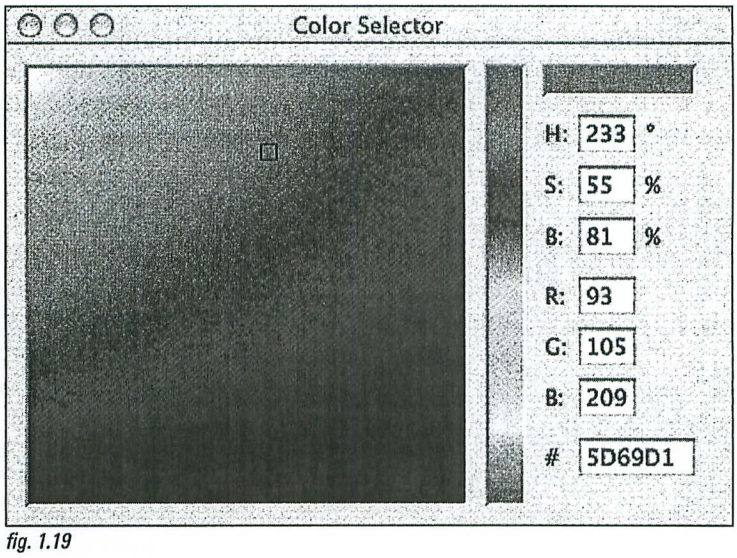
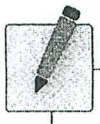


fig. 1.19

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(_____,_____,_____);
ellipse(20,40,16,16);

fill(_____,_____,_____);
ellipse(40,40,16,16);

fill(_____,_____,_____);
ellipse(60,40,16,16);
```

Bright blue

Dark purple

Yellow

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 transparency 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 is 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 are 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., 0% opaque) and 255 completely opaque (i.e., 100% opaque). Example 1-4 shows a code example that is displayed in Figure 1.20.

Example 1-4: Alpha transparency

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

fill(0,0,255);
rect(0,0,100,200);

fill(255,0,0,255);
rect(0,0,200,40);

fill(255,0,0,191);
rect(0,50,200,40);

fill(255,0,0,127);
rect(0,100,200,40);

fill(255,0,0,63);
rect(0,150,200,40);
```

No fourth argument means 100% opacity.

255 means 100% opacity.

75% opacity

50% opacity

25% opacity

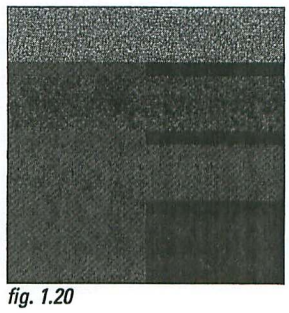


fig. 1.20

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 us think about color any way we like, and translate our 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, we 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 is rarely convenient to do so, you can also have different ranges for each color component:

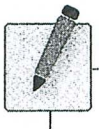
```
colorMode(RGB,100,500,10,255);
```

Now we are saying "Red values go 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. Without getting into too much detail, HSB color works as follows:

- **Hue**—The color type, ranges from 0 to 360 by default (think of 360° on a color "wheel").
- **Saturation**—The vibrancy of the color, 0 to 100 by default.
- **Brightness**—The, well, brightness of the color, 0 to 100 by default.

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(), lines(), 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

```
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);
```

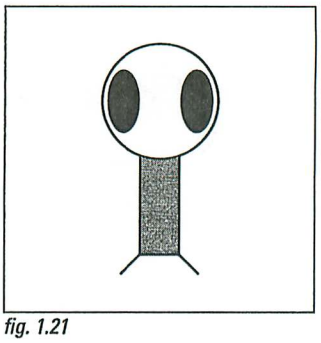


fig. 1.21

The sample answer is my *Processing*-born being, named Zoog. Over the course of the first nine chapters of this book, we will follow the course of Zoog's childhood. The fundamentals of programming will be demonstrated as Zoog grows up. We 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 only change 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 on in this book.