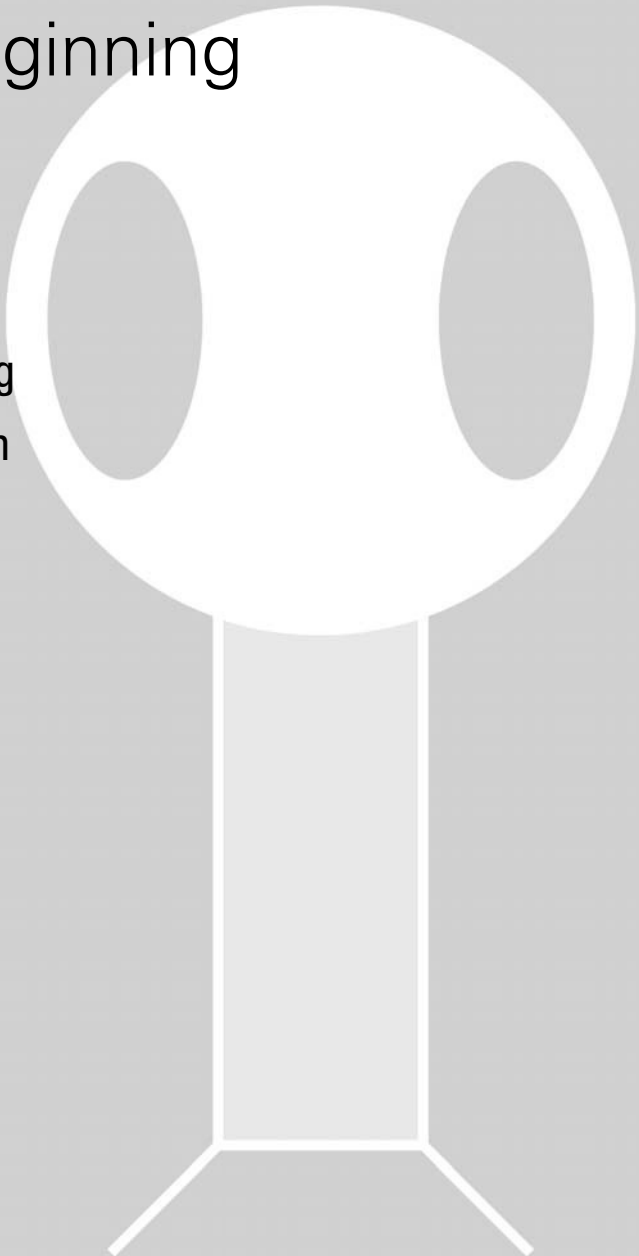
Lesson One

The Beginning



1 Pixels

2 Processing

3 Interaction

This page intentionally left blank

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.

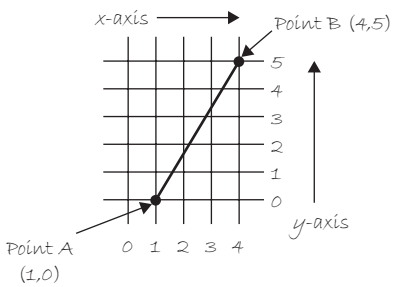


fig. 1.1

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 (0,1) 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.

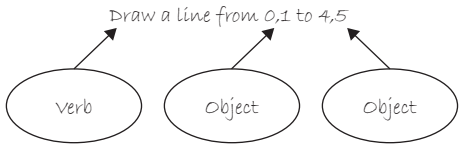


fig. 1.2

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.

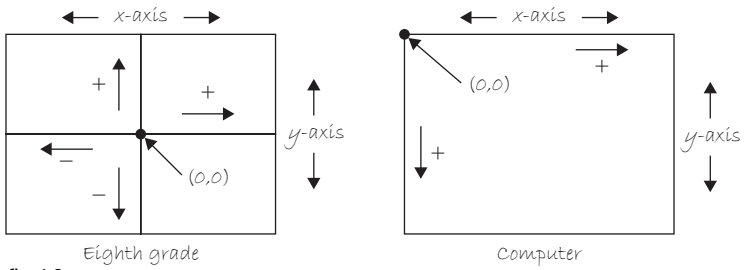


fig. 1.3

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.

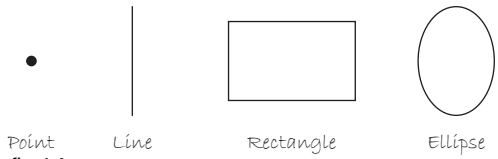


fig. 1.4

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.

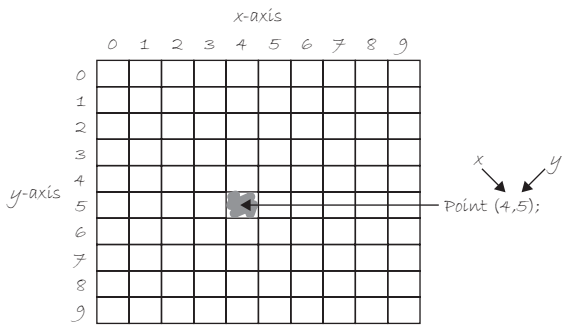


fig. 1.5

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.

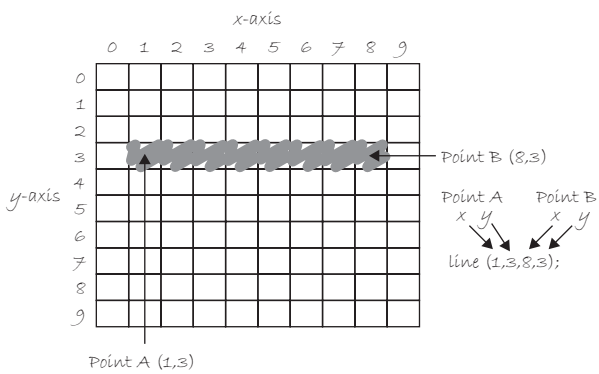


fig. 1.6

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).

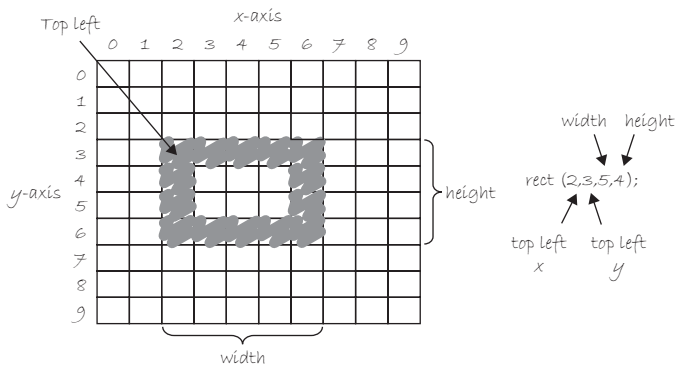


fig. 1.7

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.

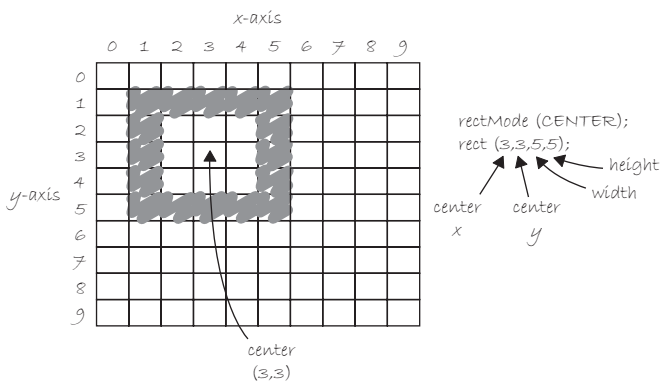


fig. 1.8

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).

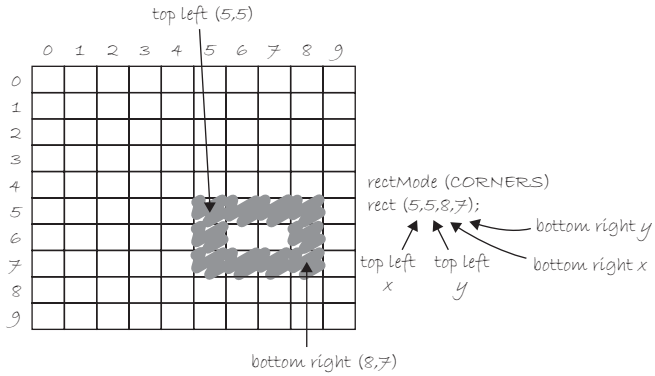


fig. 1.9

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.

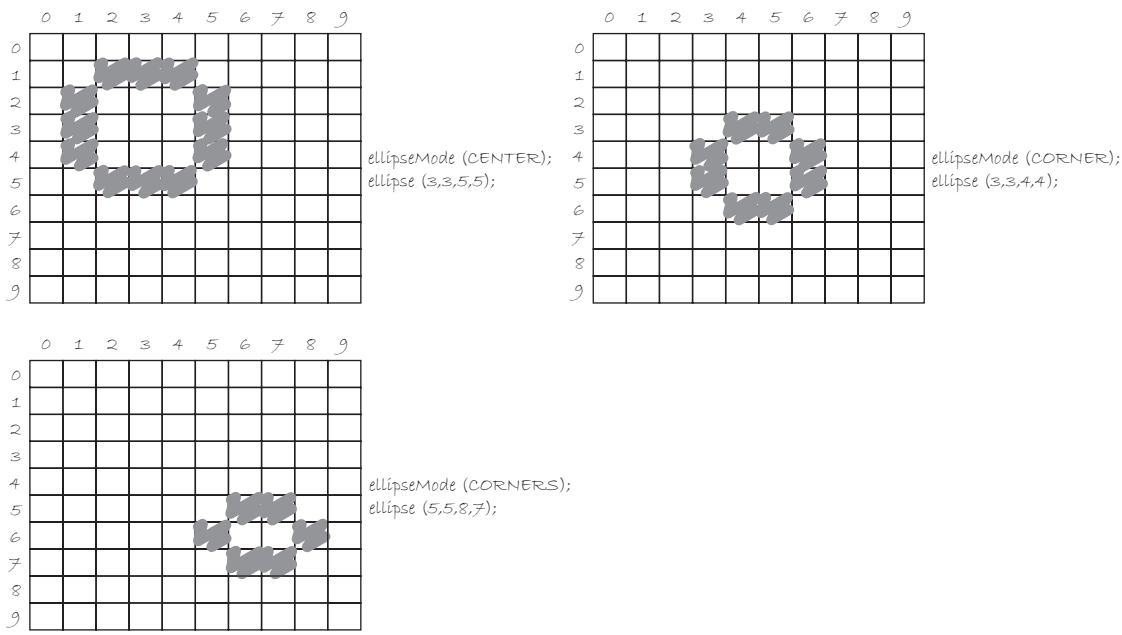


fig. 1.10

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.

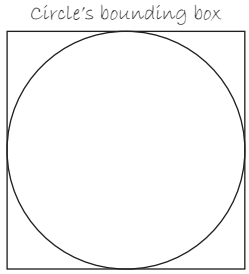


fig. 1.11

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()

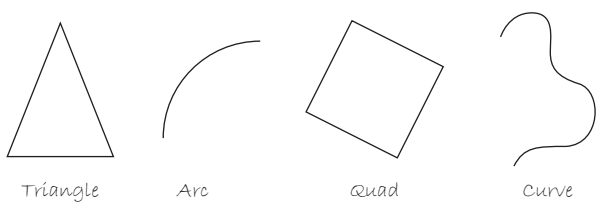
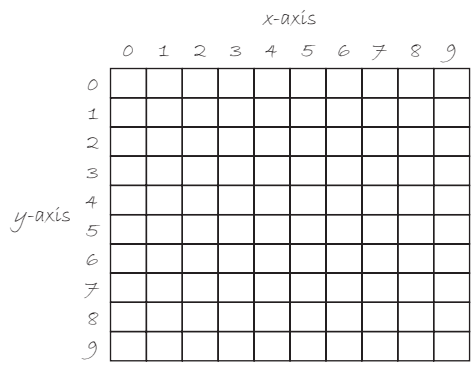


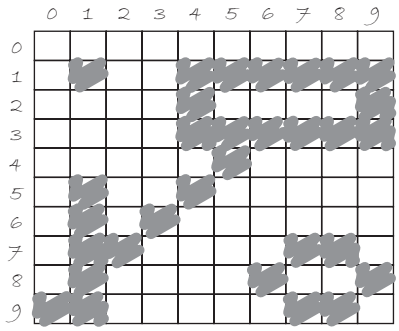
fig. 1.12

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.

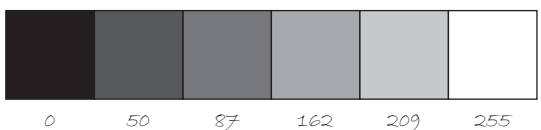


fig. 1.13

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.

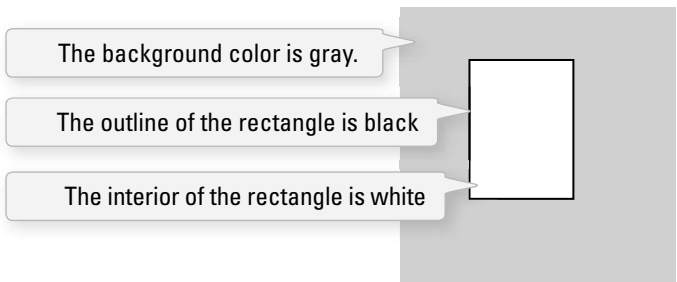


fig. 1.14

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

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);
```

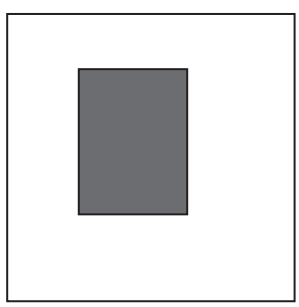


fig. 1.15

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

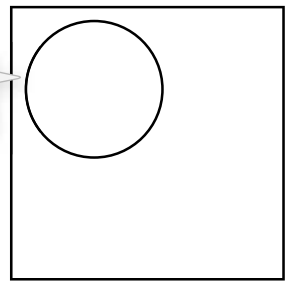


fig. 1.16

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);
```

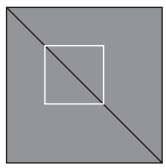
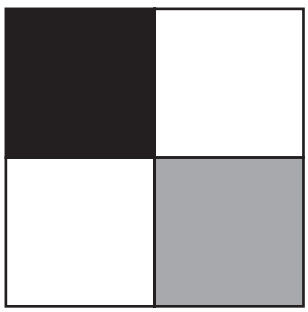


fig. 1.17

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).

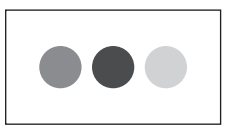


fig. 1.18

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

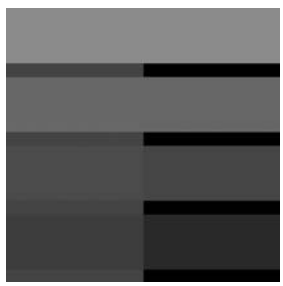


fig. 1.20

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);
```

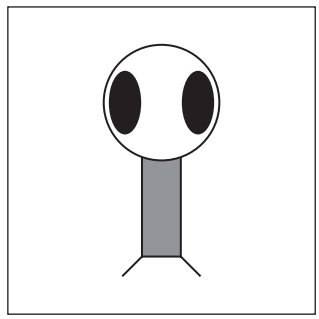


fig. 1.21

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.

2 Processing

"Computers in the future may weigh no more than 1.5 tons."
—Popular Mechanics, 1949

"Take me to your leader."
—Zoog, 2008

In this chapter:
– Downloading and installing *Processing*.
– Menu options.
– A *Processing* "sketchbook."
– Writing code.
– Errors.
– The *Processing* reference.
– The "Play" button.
– Your first sketch.
– Publishing your sketch to the web.

2.1 *Processing* to the Rescue

Now that we conquered the world of primitive shapes and RGB color, we are ready to implement this knowledge in a real world programming scenario. Happily for us, the environment we are going to use is *Processing*, free and open source software developed by Ben Fry and Casey Reas at the MIT Media Lab in 2001. (See this book's introduction for more about *Processing*'s history.)

Processing's core library of functions for drawing graphics to the screen will provide for immediate visual feedback and clues as to what the code is doing. And since its programming language employs all the same principles, structures, and concepts of other languages (specifically Java), everything you learn with *Processing* is *real* programming. It is not some pretend language to help you get started; it has all the fundamentals and core concepts that all languages have.

After reading this book and learning to program, you might continue to use *Processing* in your academic or professional life as a prototyping or production tool. You might also take the knowledge acquired here and apply it to learning other languages and authoring environments. You may, in fact, discover that programming is not your cup of tea; nonetheless, learning the basics will help you become a better-informed technology citizen as you work on collaborative projects with other designers and programmers.

It may seem like overkill to emphasize the *why* with respect to *Processing*. After all, the focus of this book is primarily on learning the fundamentals of computer programming in the context of computer graphics and design. It is, however, important to take some time to ponder the reasons behind selecting a programming language for a book, a class, a homework assignment, a web application, a software suite, and so forth. After all, now that you are going to start calling yourself a computer programmer at cocktail parties, this question will come up over and over again. I need programming in order to accomplish project *X*, what language and environment should I use?

I say, without a shadow of doubt, that for you, the beginner, the answer is *Processing*. Its simplicity is ideal for a beginner. At the end of this chapter, you will be up and running with your first computational design and ready to learn the fundamental concepts of programming. But simplicity is not where *Processing*

ends. A trip through the *Processing* online exhibition (*http://processing.org/exhibition/*) will uncover a wide variety of beautiful and innovative projects developed entirely with *Processing*. By the end of this book, you will have all the tools and knowledge you need to take your ideas and turn them into real world software projects like those found in the exhibition. *Processing* is great both for learning and for producing, there are very few other environments and languages you can say that about.

2.2 How do I get *Processing?*

For the most part, this book will assume that you have a basic working knowledge of how to operate your personal computer. The good news, of course, is that *Processing* is available for free download. Head to *http://www.processing.org/* and visit the download page. If you are a Windows user, you will see two options: "Windows (standard)" and "Windows (expert)." Since you are reading this book, it is quite likely you are a beginner, in which case you will want the standard version. The expert version is for those who have already installed Java themselves. For Mac OS X, there is only one download option. There is also a Linux version available. Operating systems and programs change, of course, so if this paragraph is obsolete or out of date, visit the download page on the site for information regarding what you need.

The *Processing* software will arrive as a compressed file. Choose a nice directory to store the application (usually "c:\Program Files\" on Windows and in "Applications" on Mac), extract the files there, locate the "Processing" executable, and run it.

Exercise 2-1: Download and install Processing.

2.3 The *Processing* Application

The *Processing* development environment is a simplified environment for writing computer code, and is just about as straightforward to use as simple text editing software (such as TextEdit or Notepad) combined with a media player. Each sketch (*Processing* programs are referred to as "sketches") has a filename, a place where you can type code, and some buttons for saving, opening, and running sketches. See Figure 2.1.

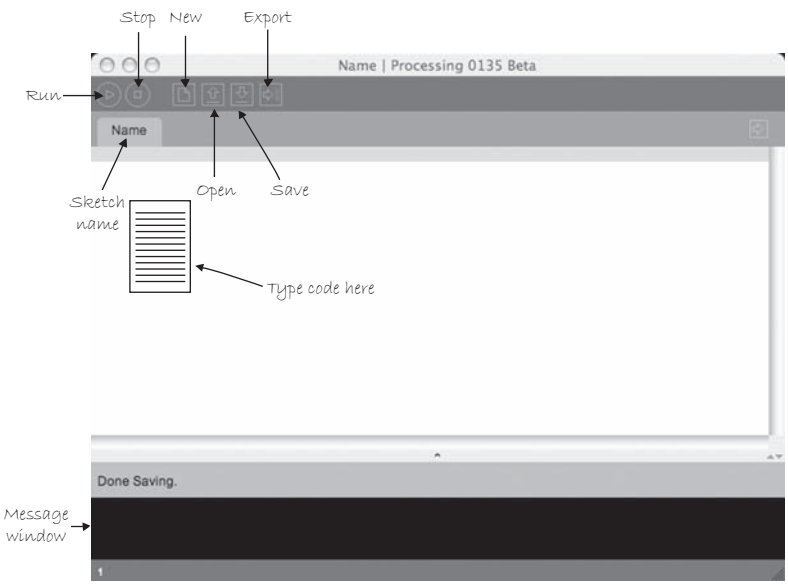


fig. 2.1

To make sure everything is working, it is a good idea to try running one of the *Processing* examples. Go to FILE → EXAMPLES → (pick an example, suggested: Topics → Drawing → ContinuousLines) as shown in Figure 2.2.

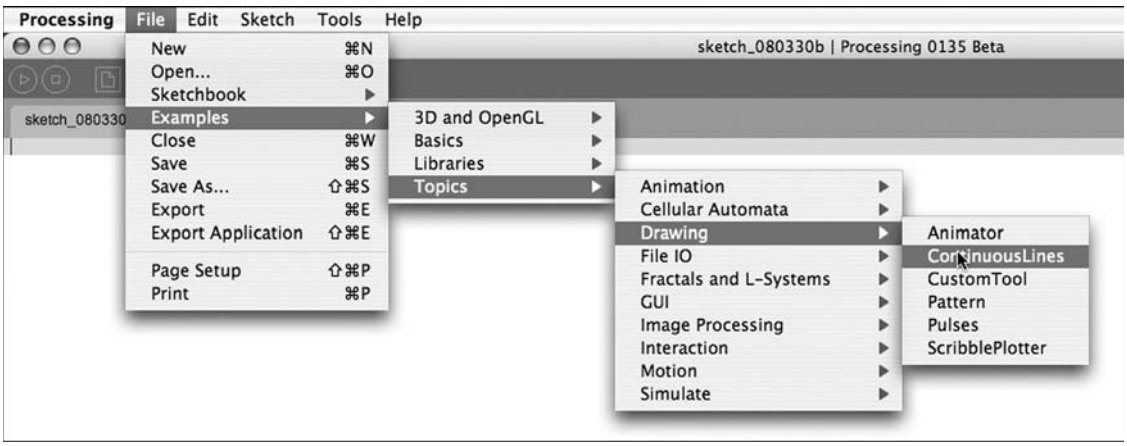


fig. 2.2

Once you have opened the example, click the "run" button as indicated in Figure 2.3. If a new window pops open running the example, you are all set! If this does not occur, visit the online FAQ "Processing won't start!" for possible solutions. The page can be found at this direct link: *http://www.processing.org/faq/bugs.html#wontstart*.

Exercise 2-2: Open a sketch from the Processing *examples and run it.*


fig. 2.3

Processing programs can also be viewed full-screen (known as "present mode" in *Processing*). This is available through the menu option: Sketch → Present (or by shift-clicking the run button). Present will not resize your screen resolution. If you want the sketch to cover your entire screen, you must use your screen dimensions in ***size()***.

2.4 The Sketchbook

Processing programs are informally referred to as *sketches*, in the spirit of quick graphics prototyping, and we will employ this term throughout the course of this book. The folder where you store your sketches is called your "sketchbook." Technically speaking, when you run a sketch in *processing*, it runs as a local application on your computer. As we will see both in this Chapter and in Chapter 18, *Processing* also allows you to export your sketches as web applets (mini-programs that run embedded in a browser) or as platform-specific stand-alone applications (that could, for example, be made available for download).

Once you have confirmed that the *Processing* examples work, you are ready to start creating your own sketches. Clicking the "new" button will generate a blank new sketch named by date. It is a good idea to "Save as" and create your own sketch name. (Note: *Processing* does not allow spaces or hyphens, and your sketch name cannot start with a number.)

When you first ran *Processing*, a default "Processing" directory was created to store all sketches in the "My Documents" folder on Windows and in "Documents" on OS X. Although you can select any directory on your hard drive, this folder is the default. It is a pretty good folder to use, but it can be changed by opening the *Processing* preferences (which are available under the FILE menu).

Each *Processing* sketch consists of a folder (with the same name as your sketch) and a file with the extension "pde." If your *Processing* sketch is named *MyFirstProgram*, then you will have a folder named *MyFirstProgram* with a file *MyFirstProgram.pde* inside. The "pde" file is a plain text file that contains the source code. (Later we will see that *Processing* sketches can have multiple pde's, but for now one will do.) Some sketches will also contain a folder called "data" where media elements used in the program, such as image files, sound clips, and so on, are stored.

Exercise 2-3: Type some instructions from Chapter 1 into a blank sketch. Note how certain words are colored. Run the sketch. Does it do what you thought it would?

2.5 Coding in *Processing*

It is finally time to start writing some code, using the elements discussed in Chapter 1. Let's go over some basic syntax rules. There are three kinds of statements we can write:

- Function calls
- Assignment operations
- Control structures

For now, every line of code will be a function call. See Figure 2.4. We will explore the other two categories in future chapters. Functions have a name, followed by a set of arguments enclosed in parentheses. Recalling Chapter 1, we used functions to describe how to draw shapes (we just called them "commands" or "instructions"). Thinking of a function call as a natural language sentence, the function name is the verb ("draw") and the arguments are the objects ("point 0,0") of the sentence. Each function call must always end with a semicolon. See Figure 2.5.

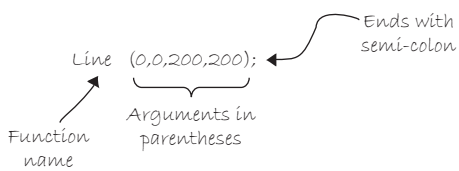


fig. 2.4

We have learned several functions already, including ***background(), stroke(), fill(), noFill (), noStroke(), point(), line(), rect(), ellipse(), rectMode(),*** and ***ellipseMode()***. *Processing* will execute a sequence of functions one by one and finish by displaying the drawn result in a window. We forgot to learn one very important function in Chapter 1, however—***size()***. ***size()*** specifies the dimensions of the window you want to create and takes two arguments, width and height. The ***size()*** function should always be first.

```
size(320,240);
```

Opens a window of width 320 and height 240.

Let's write a first example (see Figure 2.5).

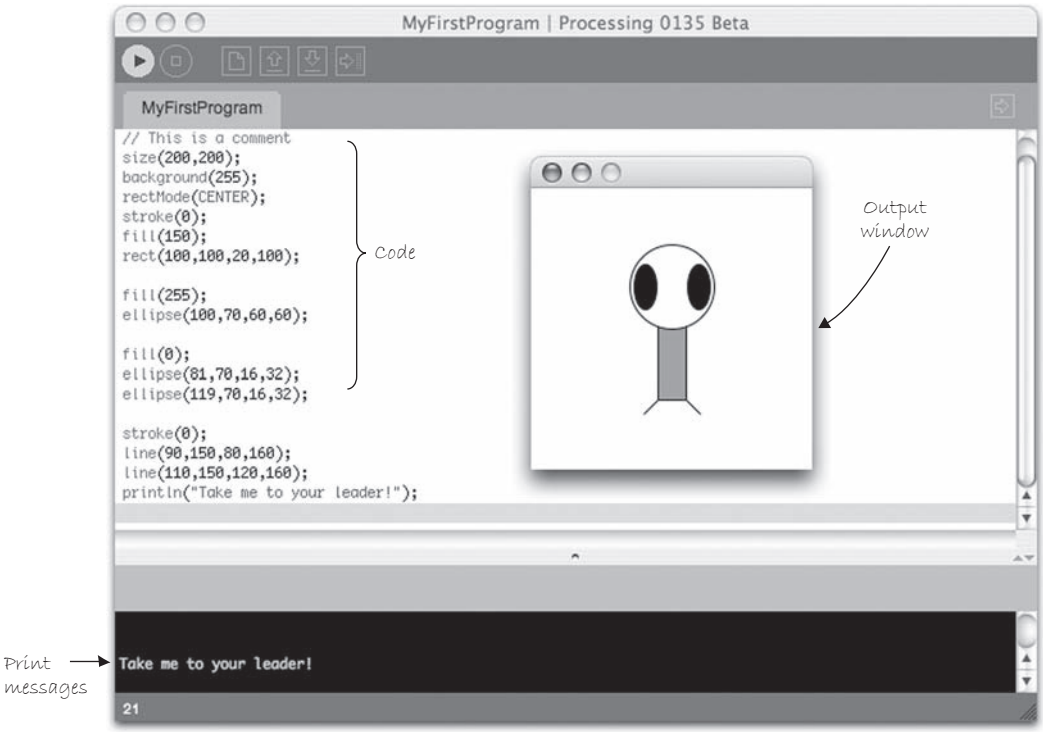


fig. 2.5

There are a few additional items to note.

- The *Processing* text editor will color *known* words (sometimes referred to as "reserved" words or "keywords"). These words, for example, are the drawing functions available in the *Processing* library, "built-in" variables (we will look closely at the concept of *variables* in Chapter 3) and constants, as well as certain words that are inherited from the Java programming language.
- Sometimes, it is useful to display text information in the *Processing* message window (located at the bottom). This is accomplished using the ***println()*** function. ***println()*** takes one argument, a *String* of characters enclosed in quotes (more about *Strings* in Chapter 14). When the program runs, *Processing* displays that *String* in the message window (as in Figure 2.5) and in this case the *String* is "Take me to your leader!" This ability to print to the message window comes in handy when attempting to *debug* the values of variables (see Chapter 12, Debugging).
- The number in the bottom left corner indicates what line number in the code is selected.
- You can write "comments" in your code. Comments are lines of text that *Processing* ignores when the program runs. You should use them as reminders of what the code means, a bug you intend to fix, or a to do list of items to be inserted, and so on. Comments on a single line are created with two forward slashes, //. Comments over multiple lines are marked by /* followed by the comments and ending with */.

```
// This is a comment on one line

/* This is a comment that
spans several lines
of code */
```

A quick word about comments. You should get in the habit right now of writing comments in your code. Even though our sketches will be very simple and short at first, you should put comments in for everything. Code is very hard to read and understand without comments. You do not need to have a comment for every line of code, but the more you include, the easier a time you will have revising and reusing your code later. Comments also force you to understand how code works as you are programming. If you do not know what you are doing, how can you write a comment about it?

Comments will not always be included in the text here. This is because I find that, unlike in an actual program, code comments are hard to read in a book. Instead, this book will often use code "hints" for additional insight and explanations. If you look at the book's examples on the web site, though, comments will always be included. So, I can't emphasize it enough, write comments!

```
//A comment about this code
line(0,0,100,100);
```

A hint about this code!

Exercise 2-4: Create a blank sketch. Take your code from the end of Chapter 1 and type it in the Processing *window. Add comments to describe what the code is doing. Add a* ***println()*** *statement to display text in the message window. Save the sketch. Press the "run" button. Does it work or do you get an error?*

2.6 Errors

The previous example only works because we did not make any errors or typos. Over the course of a programmer's life, this is quite a rare occurrence. Most of the time, our first push of the play button will not be met with success. Let's examine what happens when we make a mistake in our code in Figure 2.6.

Figure 2.6 shows what happens when you have a typo—"elipse" instead of "ellipse" on line 9. If there is an error in the code when the play button is pressed, *Processing* will not open the sketch window, and will instead display the error message. This particular message is fairly friendly, telling us that we probably meant to type "ellipse." Not all *Processing* error messages are so easy to understand, and we will continue to look at other errors throughout the course of this book. An Appendix on common errors in Processing is also included at the end of the book.

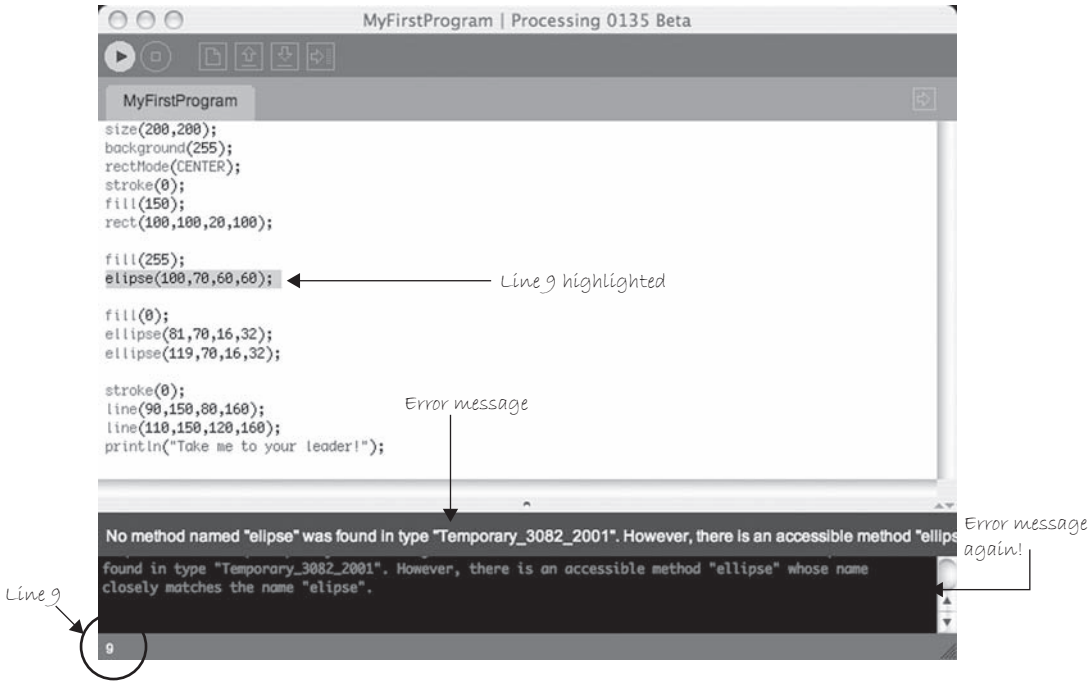


fig. 2.6

Processing *is case sensitive!*

If you type *Ellipse* instead of *ellipse*, that will also be considered an error.

In this instance, there was only one error. If multiple errors occur, *Processing* will only alert you to the first one it finds (and presumably, once that error is corrected, the next error will be displayed at run time). This is somewhat of an unfortunate limitation, as it is often useful to have access to an entire list of errors when fixing a program. This is simply one of the trade-offs we get in a simplified environment such as *Processing*. Our life is made simpler by only having to look at one error at a time, nevertheless we do not have access to a complete list.

This fact only further emphasizes the importance of incremental development discussed in the book's introduction. If we only implement one feature at a time, we can only make one mistake at a time.

Exercise 2-5: Try to make some errors happen on purpose. Are the error messages what you expect?

Exercise 2-6: Fix the errors in the following code.

```
size(200,200);
background();
stroke 255;
fill(150)
rectMode(center);
rect(100,100,50);
```

2.7 The *Processing* Reference

The functions we have demonstrated—***ellipse(), line(), stroke(),*** and so on—are all part of *Processing*'s library. How do we know that "ellipse" isn't spelled "elipse", or that ***rect()*** takes four arguments (an "*x* coordinate," a "*y* coordinate," a "width," and a "height")? A lot of these details are intuitive, and this speaks to the strength of *Processing* as a beginner's programming language. Nevertheless, the only way to know for sure is by reading the online reference. While we will cover many of the elements from the reference throughout this book, it is by no means a substitute for the reference and both will be required for you to learn *Processing*.

The reference for *Processing* can be found online at the official web site (*http://www.processing.org*) under the "reference" link. There, you can browse all of the available functions by category or alphabetically. If you were to visit the page for ***rect()***, for example, you would find the explanation shown in Figure 2.7.

As you can see, the reference page offers full documentation for the function ***rect()***, including:

- **Name**—The name of the function.
- **Examples**—Example code (and visual result, if applicable).
- **Description**—A friendly description of what the function does.
- **Syntax**—Exact syntax of how to write the function.
- **Parameters**—These are the elements that go inside the parentheses. It tells you what kind of data you put in (a number, character, etc.) and what that element stands for. (This will become clearer as we explore more in future chapters.) These are also sometimes referred to as "arguments."
- **Returns**—Sometimes a function sends something back to you when you call it (e.g., instead of asking a function to perform a task such as draw a circle, you could ask a function to add two numbers and *return* the answer to you). Again, this will become more clear later.
- **Usage**—Certain functions will be available for *Processing* applets that you publish online ("Web") and some will only be available as you run *Processing* locally on your machine ("Application").
- **Related Methods**—A list of functions often called in connection with the current function. Note that "functions" in Java are often referred to as "methods." More on this in Chapter 6.

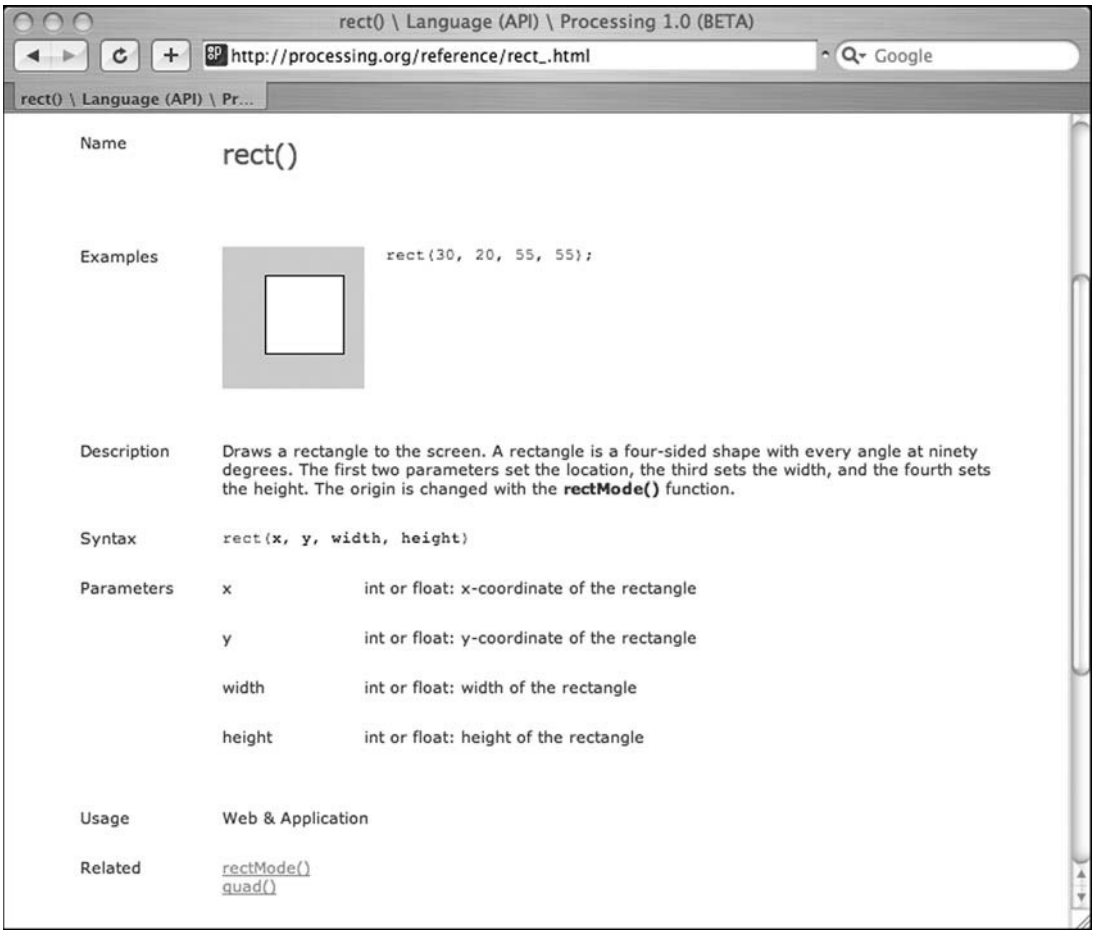


fig. 2.7

Processing also has a very handy "find in reference" option. Double-click on any keyword to select it and go to to HELP → FIND IN REFERENCE (or select the keyword and hit SHIFT+CNTRL+F).

Exercise 2-7: Using the Processing *reference, try implementing two functions that we have not yet covered in this book. Stay within the "Shape" and "Color (setting)" categories.*

Exercise 2-8: Using the reference, find a function that allows you to alter the thickness of a line. What arguments does the function take? Write example code that draws a line one pixel wide, then five pixels wide, then 10 pixels wide.

2.8 The "Play" Button

One of the nice qualities of *Processing* is that all one has to do to run a program is press the "play" button. It is a nice metaphor and the assumption is that we are comfortable with the idea of *playing* animations,

movies, music, and other forms of media. *Processing* programs output media in the form of real-time computer graphics, so why not just *play* them too?

Nevertheless, it is important to take a moment and consider the fact that what we are doing here is not the same as what happens on an iPod or TiVo. *Processing* programs start out as text, they are translated into machine code, and then executed to run. All of these steps happen in sequence when the play button is pressed. Let's examine these steps one by one, relaxed in the knowledge that *Processing* handles the hard work for us.

Step 1. Translate to Java. *Processing* is really Java (this will become more evident in a detailed discussion in Chapter 23). In order for your code to run on your machine, it must first be translated to Java code.

Step 2. Compile into Java byte code. The Java code created in Step 1 is just another text file (with the .java extension instead of .pde). In order for the computer to understand it, it needs to be translated into machine language. This translation process is known as compilation. If you were programming in a different language, such as C, the code would compile directly into machine language specific to your operating system. In the case of Java, the code is compiled into a special machine language known as Java byte code. It can run on different platforms (Mac, Windows, cellphones, PDAs, etc.) as long as the machine is running a "Java Virtual Machine." Although this extra layer can sometimes cause programs to run a bit slower than they might otherwise, being cross-platform is a great feature of Java. For more on how this works, visit http://java.sun.com or consider picking up a book on Java programming (after you have finished with this one).

Step 3. Execution. The compiled program ends up in a JAR file. A JAR is a Java archive file that contains compiled Java programs ("classes"), images, fonts, and other data files. The JAR file is executed by the Java Virtual Machine and is what causes the display window to appear.

2.9 Your First Sketch

Now that we have downloaded and installed *Processing*, understand the basic menu and interface elements, and have gotten familiar with the online reference, we are ready to start coding. As I briefly mentioned in Chapter 1, the first half of this book will follow one example that illustrates the foundational elements of programming: *variables, arrays, conditionals, loops, functions,* and *objects*. Other examples will be included along the way, but following just one will reveal how the basic elements behind computer programming build on each other.

The example will follow the story of our new friend Zoog, beginning with a static rendering with simple shapes. Zoog's development will include mouse interaction, motion, and cloning into a population of many Zoogs. While you are by no means required to complete every exercise of this book with your own alien form, I do suggest that you start with a design and after each chapter, expand the functionality of that design with the programming concepts that are explored. If you are at a loss for an idea, then just draw your own little alien, name it Gooz, and get programming! See Figure 2.8.

Example 2-1: Zoog again

```
size(200,200); // Set the size of the window
background(255); // Draw a black background
smooth();


// Set ellipses and rects to CENTER mode
ellipseMode(CENTER);
rectMode(CENTER);


// Draw Zoog's body
stroke(0);
fill(150);
rect(100,100,20,100);


// Draw Zoog's head
fill(255);
ellipse(100,70,60,60);


// Draw Zoog's eyes
fill(0);
ellipse(81,70,16,32);
ellipse(119,70,16,32);


// Draw Zoog's legs
stroke(0);
line(90,150,80,160);
line(110,150,120,160);
```

The function ***smooth()*** enables "anti-aliasing" which smooths the edges of the shapes. ***no smooth()*** disables anti-aliasing.

Zoog's body.

Zoog's head.

Zoog's eyes.

Zoog's legs.

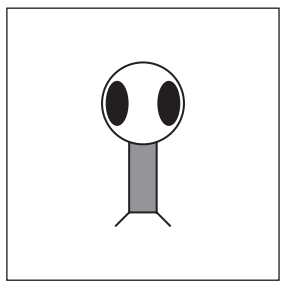


fig. 2.8

Let's pretend, just for a moment, that you find this Zoog design to be so astonishingly gorgeous that you just cannot wait to see it displayed on your computer screen. (Yes, I am aware this may require a fairly significant suspension of disbelief.) To run any and all code examples found in this book, you have two choices:

- Retype the code manually.
- Visit the book's web site (*http://www.learningprocessing.com*), find the example by its number, and copy/paste (or download) the code.

Certainly option #2 is the easier and less time-consuming one and I recommend you use the site as a resource for seeing sketches running in real-time and grabbing code examples. Nonetheless, as you start learning, there is real value in typing the code yourself. Your brain will sponge up the syntax and logic as you type and you will learn a great deal by making mistakes along the way. Not to mention simply running the sketch after entering each new line of code will eliminate any mystery as to how the sketch works.

You will know best when you are ready for copy/paste. Keep track of your progress and if you start running a lot of examples without feeling comfortable with how they work, try going back to manual typing.

*Exercise 2-9: Using what you designed in Chapter 1, implement your own screen drawing, using only 2D primitive shapes—**arc()**, **curve()**, **ellipse()**, **line()**, **point()**, **quad()**, **rect()**, **triangle()**—and basic color functions—**background()**, **colorMode()**, **fill()**, **noFill()**, **noStroke()**, and **stroke()**. Remember to use **size()** to specify the dimensions of your window. Suggestion: Play the sketch after typing each new line of code. Correct any errors or typos along the way.*

2.10 Publishing Your Program

After you have completed a *Processing* sketch, you can publish it to the web as a Java applet. This will become more exciting once we are making interactive, animated applets, but it is good to practice with a simple example. Once you have finished Exercise 2-9 and determined that your sketch works, select FILE → EXPORT.

Note that if you have errors in your program, it will not export properly, so always test by running first!

A new directory named "applet" will be created in the sketch folder and displayed, as shown in Figure 2.9.

fig. 2.9

You now have the necessary files for publishing your applet to the web.

- **index.html**—The HTML source for a page that displays the applet.
- **loading.gif**—An image to be displayed while the user loads the applet (*Processing* will supply a default one, but you can create your own).
- **zoog.jar**—The compiled applet itself.
- **zoog.java**—The translated Java source code (looks like your *Processing* code, but has a few extra things that Java requires. See Chapter 20 for details.)
- **zoog.pde**—Your *Processing* source.

To see the applet working, double-click the "index.html" file which should launch a page in your default web browser. See Figure 2.10. To get the applet online, you will need web server space and FTP software (or you can also use a *Processing* sketch sharing site such as http://www.openprocessing.org). You can find some tips for getting started at this book's web site.

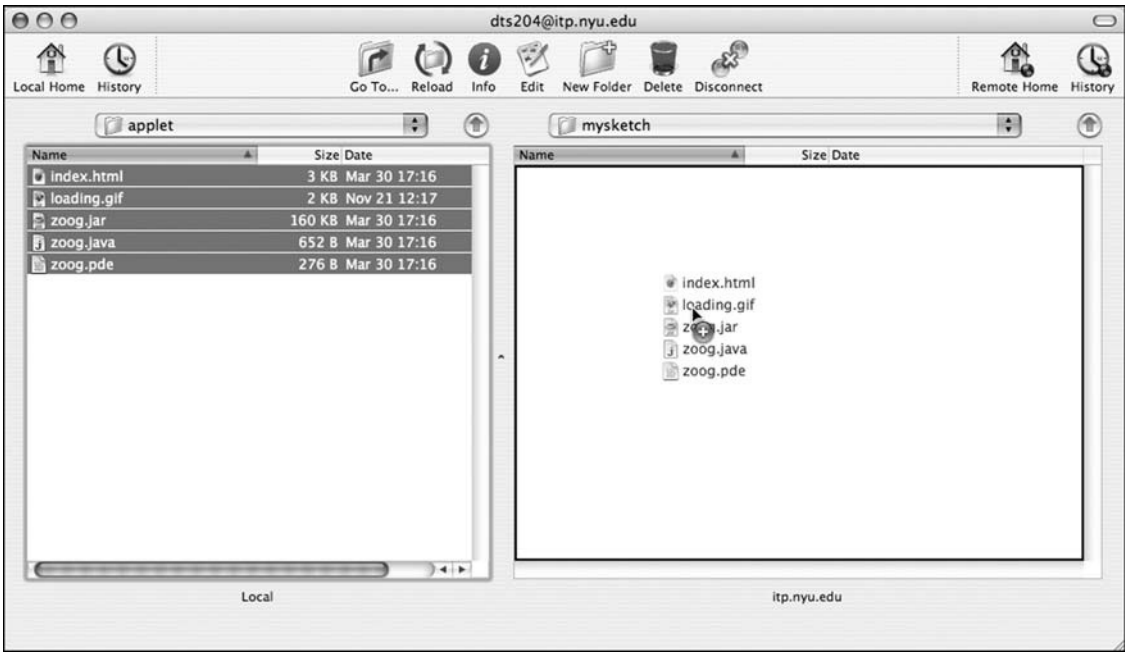


fig. 2.10

*Exercise 2-10: Export your sketch as an applet. View the sketch in the browser (either locally or by uploading).*0000200002

This page intentionally left blank