Preface

We created Processing to make programming interactive graphics easier. We were frustrated with how difficult it was to write this type of software with the programming languages we usually used (Java and C++), and were inspired by how simple it was to write interesting programs with the languages of our childhood (Logo and BASIC). We were most influenced by Design By Numbers (DBN), a language we were maintaining and teaching at the time (and which was created by our research advisor, John Maeda).

Processing was born in spring 2001 as a brainstorming session on a sheet of paper. Our goal was to make a way to sketch (prototype) the type of software we were working on, which was almost always full-screen and interactive. We were searching for a better way to test our ideas easily in code, rather than just talking about them or spending too much time programming them in C++. Our other goal was to make a language for teaching design and art students how to program and to give more technical students an easier way to work with graphics. The combination is a positive departure from the way programming is usually taught. We begin by focusing on graphics and interaction rather than on data structures and text console output.

Processing experienced a long childhood; it was alpha software from August 2002 to April 2005 and then public beta software until November 2008. During this time, it was used continuously in classrooms and by thousands of people around the world. The language, software environment, and curricula around the project were revised continuously during this time. Many of our original decisions about the language were reinforced and many were changed. We developed a system of software extensions, called *libraries*, that have allowed people to expand Processing into many unforeseen and amazing directions. (There are now over 100 libraries.)

In fall 2008, we launched the 1.0 version of the software. After seven years of work, the 1.0 launch signified stability for the language. We launched the 2.0 release in spring 2013 to make the software faster. The 2.0 releases introduced better OpenGL integration, GLSL shaders, and faster video playback with GStreamer. The 3.0 releases in 2015 make programming in Processing easier with a new interface and error checking while programming.

Now, fourteen years after its origin, Processing has grown beyond its original goals, and we've learned how it can be useful in other contexts. Accordingly, this book is written for a new audience—casual programmers, hobbyists, and anyone who wants to explore what Processing can do without getting lost in the details of a huge textbook. We hope you'll have fun and be inspired to continue programming. This book is just the start.

While we (Casey and Ben) have been guiding the Processing ship through the waters for the last twelve years, we can't overstate that Processing is a community effort. From writing libraries that extend the software to posting code online and helping others learn, the community of people who use Processing has pushed it far beyond its initial conception. Without this group effort, Processing would not be what it is today.

How This Book Is Organized

The chapters in this book are organized as follows:

- Chapter 1: Learn about Processing.
- Chapter 2: Create your first Processing program.
- Chapter 3: Define and draw simple shapes.
- Chapter 4: Store, modify, and reuse data.
- Chapter 5: Control and influence programs with the mouse and the keyboard.
- Chapter 6: Transform the coordinates.
- Chapter 7: Load and display media including images, fonts, and vector files.
- Chapter 8: Move and choreograph shapes.
- Chapter 9: Build new code modules.
- Chapter 10: Create code modules that combine variables and functions.
- Chapter 11: Simplify working with lists of variables.
- Chapter 12: Load and visualize data.
- Chapter 13: Learn about 3D, PDF export, computer vision, and reading data from an Arduino board.

Who This Book Is For

This book is written for people who want a casual and concise introduction to computer programming, who want to create images and simple interactive programs. It's for people who want a jump-start on understanding the thousands of free Processing code examples and reference materials available online. *Getting Started with Processing* is not a programming textbook; as the title suggests, it will get you started. It's for teenagers, hobbyists, grandparents, and everyone in between.

This book is also appropriate for people with programming experience who want to learn the basics of interactive computer graphics. *Getting Started with Processing* contains techniques

that can be applied to creating games, animation, and interfaces.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip, suggestion, or general note.

This element indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from Make: books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a sig-

nificant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: *"Getting Started with Processing* by Casey Reas and Ben Fry. Copyright 2015 Casey Reas and Ben Fry, 978-1-457-18708-7."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *bookpermissions@makermedia.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like Maker Media, O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

Maker Media, Inc.
1160 Battery Street East, Suite 125
San Francisco, California 94111
800-998-9938 (in the United States or Canada)
http://makermedia.com/contact-us/

Make: unites, inspires, informs, and entertains a growing community of resourceful people who undertake amazing projects in their backyards, basements, and garages. Make: celebrates your right to tweak, hack, and bend any technology to your will. The Make: audience continues to be a growing culture and community that believes in bettering ourselves, our environment, our educational system—our entire world. This is much more than an audience, it's a worldwide movement that Make: is leading—we call it the Maker Movement.

For more information about Make:, visit us online:

Make: magazine: *http://makezine.com/magazine/*
Maker Faire: *http://makerfaire.com*
Makezine.com: *http://makezine.com*
Maker Shed: *http://makershed.com/*

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at: *http://shop.oreilly.com/product/0636920031406.do*

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

Acknowledgments

For the first and second editions of this book, we thank Brian Jepson for his great energy, support, and insight. For the first edition, Nancy Kotary, Rachel Monaghan, and Sumita Mukherji gracefully carried the book to the finish line. Tom Sgouros made a thorough edit of the book and David Humphrey provided an insightful technical review.

We can't imagine this book without Massimo Banzi's *Getting Started with Arduino* (Maker Media). Massimo's excellent book is the prototype.

A small group of individuals has, for years, contributed essential time and energy to Processing. Dan Shiffman is our partner in the Processing Foundation, the 501(c)(3) organization that supports the Processing software. Much of the core code for Processing 2.0 and 3.0 has come from the sharp minds of Andres Colubri and Manindra Moharana. Scott Murray, Jamie Kosoy, and Jon Gacnik have built a wonderful web infrastructure for the project. James Grady is rocking the 3.0 user interface. We thank Florian Jenett for his years of diverse work on the project including the forums, website, and design. Elie Zananiri and Andreas Schlegel have created the infrastructure for building and documenting contributed libraries and have spent countless hours curating the lists. Many others have contributed significantly to the project; the precise data is available at *https://github.com/processing*.

The Processing 1.0 release was supported by Miami University and Oblong Industries. The Armstrong Institute for Interactive Media Studies at Miami University funded the Oxford Project, a series of Processing development workshops. These workshops were made possible through the hard work of Ira Greenberg. These four-day meetings in Oxford, Ohio, and Pittsburgh, Pennsylvania, enabled the November 2008 launch of Processing 1.0. Oblong Industries funded Ben Fry to develop Processing during summer 2008; this was essential to the release.

The Processing 2.0 release was facilitated by a development workshop sponsored by New York University's Interactive Telecommunication Program. The work on Processing 3.0 was generously sponsored by the Emergent Digital Practices program at the University of Denver. We thank Christopher Colemen and Laleh Mehran for the essential support.

This book grew out of teaching with Processing at UCLA. Chandler McWilliams has been instrumental in defining these classes. Casey thanks the undergraduate students in the Department of Design Media Arts at UCLA for their energy and enthusiasm. His teaching assistants have been great collaborators in defining how Processing is taught. Hats off to Tatsuya Saito, John Houck, Tyler Adams, Aaron Siegel, Casey Alt, Andres Colubri, Michael Kontopoulos, David Elliot, Christo Allegra, Pete Hawkes, and Lauren McCarthy.

Through founding the Aesthetics and Computation Group (1996–2002) at the MIT Media Lab, John Maeda made all of this possible.

1/Hello

Processing is for writing software to make images, animations, and interactions. The idea is to write a single line of code, and have a circle show up on the screen. Add a few more lines of code, and the circle follows the mouse. Another line of code, and the circle changes color when the mouse is pressed. We call this *sketching* with code. You write one line, then add another, then another, and so on. The result is a program created one piece at a time.

Programming courses typically focus on structure and theory first. Anything visual—an interface, an animation—is considered a dessert to be enjoyed only after finishing your vegetables, usually several weeks of studying algorithms and methods. Over the years, we've watched many friends try to take such courses and drop out after the first lecture or after a long, frustrating night before the first assignment deadline. What initial curiosity they had about making the computer work for them was lost because they couldn't see a path from what they had to learn first to what they wanted to create.

Processing offers a way to learn programming through creating interactive graphics. There are many possible ways to teach coding, but students often find encouragement and motivation in immediate visual feedback. Processing's capacity for providing that feedback has made it a popular way to approach pro-

gramming, and its emphasis on images, sketching, and community is discussed in the next few pages.

Sketching and Prototyping

Sketching is a way of thinking; it's playful and quick. The basic goal is to explore many ideas in a short amount of time. In our own work, we usually start by sketching on paper and then moving the results into code. Ideas for animation and interactions are usually sketched as storyboards with notations. After making some software sketches, the best ideas are selected and combined into prototypes (Figure 1-1). It's a cyclical process of making, testing, and improving that moves back and forth between paper and screen.

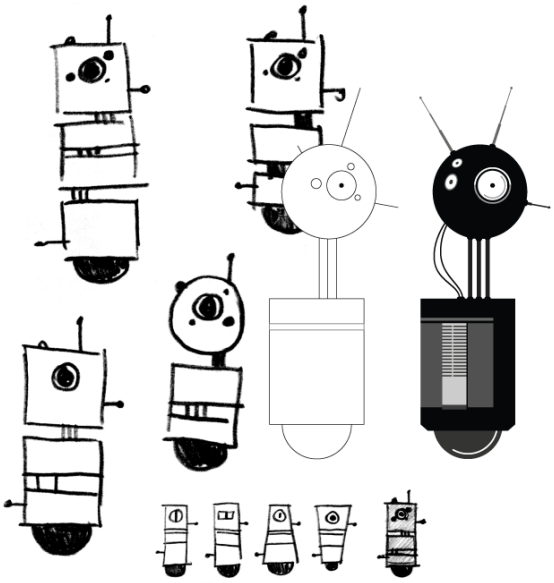


Figure 1-1. *As drawings move from sketchbook to screen, new possibilities emerge*

Flexibility

Like a software utility belt, Processing consists of many tools that work together in different combinations. As a result, it can

be used for quick hacks or for in-depth research. Because a Processing program can be as short as one line or as long as thousands, there's room for growth and variation. More than 100 libraries extend Processing even further into domains including sound, computer vision, and digital fabrication (Figure 1-2).

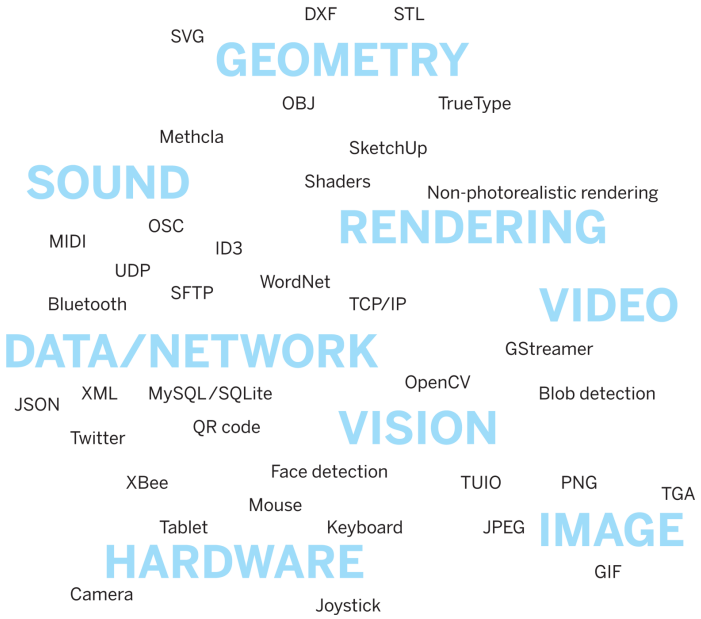


Figure 1-2. *Many types of information can flow in and out of Processing*

Giants

People have been making pictures with computers since the 1960s, and there's much to be learned from this history. For example, before computers could display to CRT or LCD screens, huge plotter machines (Figure 1-3) were used to draw images. In life, we all stand on the shoulders of giants, and the titans for Processing include thinkers from design, computer graphics, art, architecture, statistics, and the spaces between. Have a look at Ivan Sutherland's *Sketchpad* (1963), Alan Kay's *Dynabook* (1968), and the many artists featured in Ruth Leavitt's *Artist and Computer* (Harmony Books, 1976). The ACM SIG-

GRAPH and Ars Electronica archives provide fascinating glimpses into the history of graphics and software.

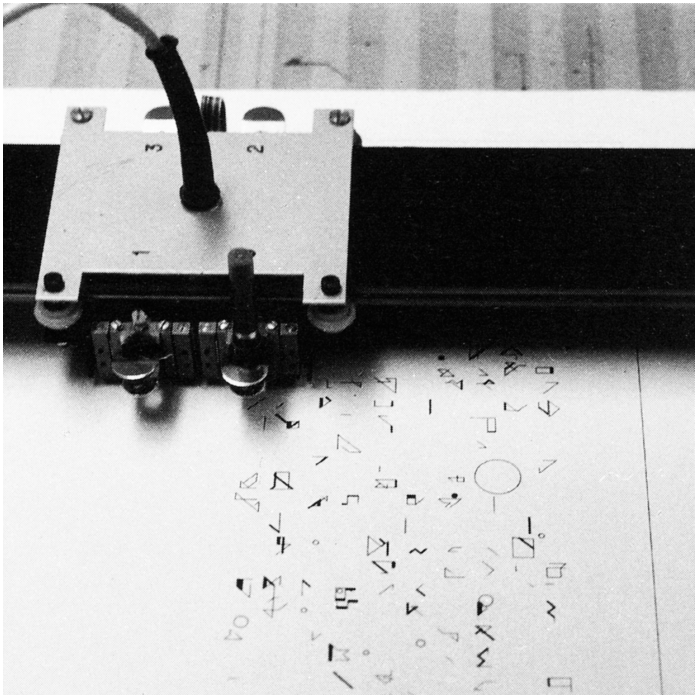


Figure 1-3. *Drawing demonstration by Manfred Mohr at Musée d'Art Moderne de la Ville de Paris using the Benson plotter and a digital computer on May 11, 1971. (photo by Rainer Mürle, courtesy bitforms gallery, New York)*

Family Tree

Like human languages, programming languages belong to families of related languages. Processing is a dialect of a programming language called Java; the language syntax is almost identical, but Processing adds custom features related to graphics and interaction (Figure 1-4). The graphic elements of Processing are related to PostScript (a foundation of PDF) and OpenGL (a 3D graphics specification). Because of these shared features, learning Processing is an entry-level step to programming in other languages and using different software tools.

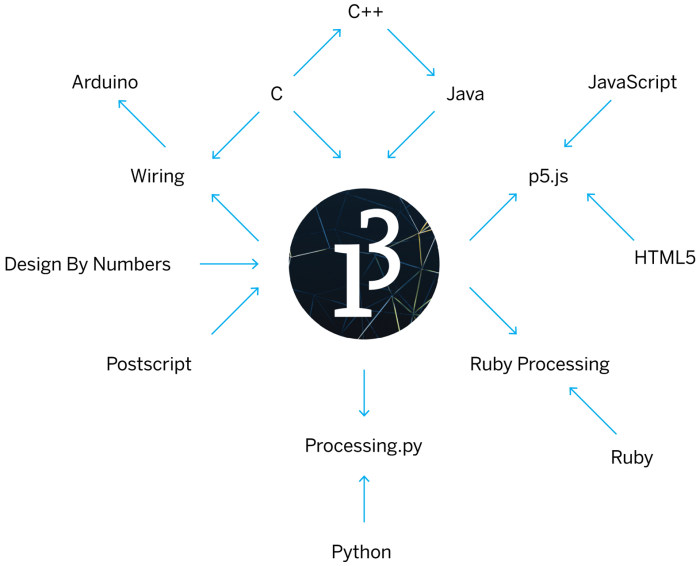


Figure 1-4. *Processing has a large family of related languages and programming environments*

Join In

Thousands of people use Processing every day. Like them, you can download Processing without cost. You even have the option to modify the Processing code to suit your needs. Processing is a *FLOSS* project (that is, *free/libre/open source software*), and in the spirit of community, we encourage you to participate by sharing your projects and knowledge online at Processing.org and at the many social networking sites that host Processing content. These sites are linked from the Processing.org website.

2/Starting to Code

To get the most out of this book, you need to do more than just read the words. You need to experiment and practice. You can't learn to code just by reading about it—you need to do it. To get started, download Processing and make your first sketch.

Start by visiting *http://processing.org/download* and selecting the Mac, Windows, or Linux version, depending on what machine you have. Installation on each machine is straightforward:

- On Windows, you'll have a *.zip* file. Double-click it, and drag the folder inside to a location on your hard disk. It could be *Program Files* or simply the desktop, but the important thing is for the *processing* folder to be pulled out of that *.zip* file. Then double-click *processing.exe* to start.
- The Mac OS X version is a *.zip* file. Double-click it, and drag the Processing icon to the *Applications* folder. If you're using someone else's machine and can't modify the *Applications* folder, just drag the application to the desktop. Then double-click the Processing icon to start.
- The Linux version is a *.tar.gz* file, which should be familiar to most Linux users. Download the file to your home directory, then open a terminal window, and type:

```
tar xvfz processing-xxxx.tgz
```

(Replace *xxxx* with the rest of the file's name, which is the version number.) This will create a folder named *processing-3.0* or something similar. Then change to that directory:

```
cd processing-xxxx
```

and run it:

```
./processing
```

With any luck, the main Processing window will now be visible (Figure 2-1). Everyone's setup is different, so if the program didn't start, or you're otherwise stuck, visit the troubleshooting page for possible solutions.

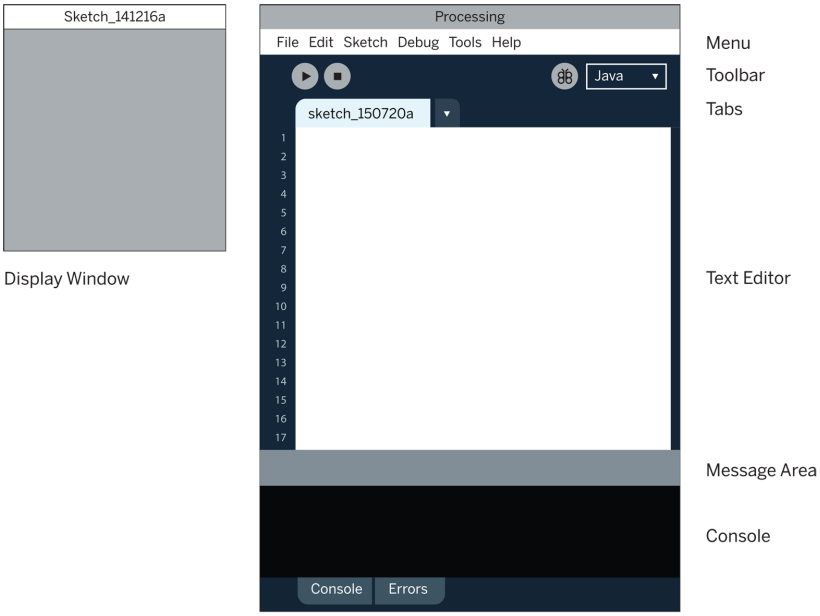


Figure 2-1. *The Processing Development Environment*

Your First Program

You're now running the Processing Development Environment (or PDE). There's not much to it; the large area is the Text Editor, and there's two buttons across the top; this is the Toolbar. Below the editor is the Message Area, and below that is the Console. The Message Area is used for one-line messages, and the Console is used for more technical details.

Example 2-1: Draw an Ellipse

In the editor, type the following:

```
ellipse(50, 50, 80, 80);
```

This line of code means "draw an ellipse, with the center 50 pixels over from the left and 50 pixels down from the top, with a width and height of 80 pixels." Click the Run button the (triangle button in the Toolbar).

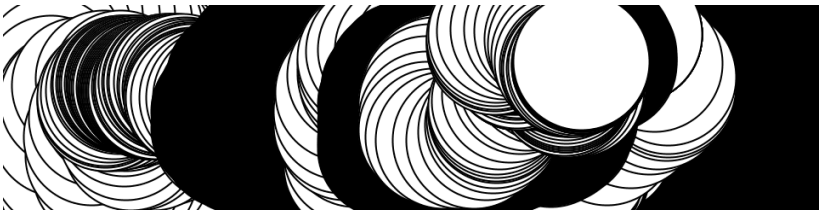
If you've typed everything correctly, you'll see a circle on your screen. If you didn't type it correctly, the Message Area will turn red and complain about an error. If this happens, make sure that you've copied the example code exactly: the numbers should be contained within parentheses and have commas between each of them, and the line should end with a semicolon.

One of the most difficult things about getting started with programming is that you have to be very specific about the syntax. The Processing software isn't always smart enough to know what you mean, and can be quite fussy about the placement of punctuation. You'll get used to it with a little practice.

Next, we'll skip ahead to a sketch that's a little more exciting.

Example 2-2: Make Circles

Delete the text from the last example, and try this one:



```
void setup() {
  size(480, 120);
}

void draw() {
  if (mousePressed) {
    fill(0);
  } else {
```

```
    fill(255);
  }
  ellipse(mouseX, mouseY, 80, 80);
}
```

This program creates a window that is 480 pixels wide and 120 pixels high, and then starts drawing white circles at the position of the mouse. When a mouse button is pressed, the circle color changes to black. We'll explain more about this program later. For now, run the code, move the mouse, and click to see what it does. While the sketch is running, the Run button will change to a square "stop" icon, which you can click to halt the sketch.

Show

If you don't want to use the buttons, you can always use the Sketch menu, which reveals the shortcut Ctrl-R (or Cmd-R on the Mac) for Run. The Present option clears the rest of the screen when the program is run to present the sketch all by itself. You can also use Present from the Toolbar by holding down the Shift key as you click the Run button. See Figure 2-2.

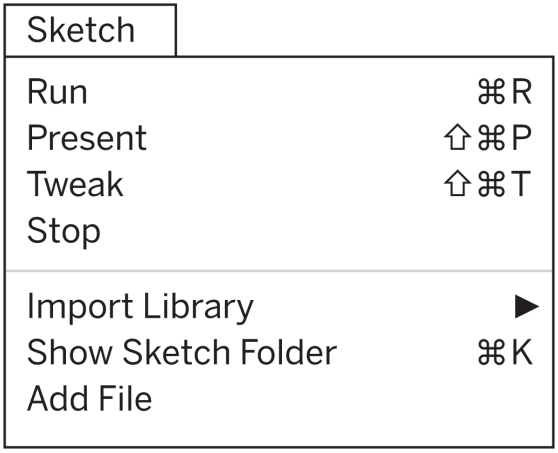


Figure 2-2. *A Processing sketch is displayed on screen with Run and Present. The Present option clears the entire screen before running the code for a cleaner presentation.*

Save and New

The next command that's important is Save. You can find it under the File menu. By default, your programs are saved to the "sketchbook," which is a folder that collects your programs for easy access. Select the Sketchbook option in the File menu to bring up a list of all the sketches in your sketchbook.

It's always a good idea to save your sketches often. As you try different things, keep saving with different names, so that you can always go back to an earlier version. This is especially helpful if—no, *when*—something breaks. You can also see where the sketch is located on your computer with the Show Sketch Folder command under the Sketch menu.

You can create a new sketch by selecting the New option from the File menu. This will create a new sketch in its own window.

Share

Processing sketches are made to be shared. The Export Application option in the File menu will bundle your code into a single folder. Export Application creates an application for your choice of Mac, Windows, and/or Linux. This is an easy way to make self-contained, double-clickable versions of your projects that can run full screen or in a window.

The application folders are erased and re-created each time you use the Export Application command, so be sure to move the folder elsewhere if you do not want it to be erased with the next export.

Examples and Reference

Learning how to program involves exploring lots of code: running, altering, breaking, and enhancing it until you have reshaped it into something new. With this in mind, the Processing software download includes dozens of examples that demonstrate different features of the software.

To open an *example*, select Examples from the File menu and double-click an example's name to open it. The examples are grouped into categories based on their function, such as Form, Motion, and Image. Find an interesting topic in the list and try an example.

All of the examples in this book can be downloaded and run from the Processing Development Environment. Open the examples through the File menu, then click Add Examples to open the list of example packages available to download. Select the *Getting Started with Processing* package and click Install to download.

When looking at code in the editor, you'll see that functions like `ellipse()` and `fill()` have a different color from the rest of the text. If you see a function that you're unfamiliar with, select the text, and then click "Find in Reference" from the Help menu. You can also right-click the text (or Ctrl-click on a Mac) and choose "Find in Reference" from the menu that appears. This will open a web browser and show the reference for that function. In addition, you can view the full documentation for the software by selecting Reference from the Help menu.

The *Processing Reference* explains every code element with a description and examples. The *Reference* programs are much shorter (usually four or five lines) and easier to follow than the longer code found in the *Examples* folder. We recommend keeping the *Reference* open while you're reading this book and while you're programming. It can be navigated by topic or alphabetically; sometimes it's fastest to do a text search within your browser window.

The *Reference* was written with the beginner in mind; we hope that we've made it clear and understandable. We're grateful to the many people who've spotted errors over the years and reported them. If you think you can improve a reference entry or you find a mistake, please let us know by clicking the link at the top of each reference page.

3/Draw

At first, drawing on a computer screen is like working on graph paper. It starts as a careful technical procedure, but as new concepts are introduced, drawing simple shapes with software expands into animation and interaction. Before we make this jump, we need to start at the beginning.

A computer screen is a grid of light elements called *pixels*. Each pixel has a position within the grid defined by coordinates. In Processing, the *x* coordinate is the distance from the left edge of the Display Window and the *y* coordinate is the distance from the top edge. We write coordinates of a pixel like this: (x, y). So, if the screen is 200×200 pixels, the upper-left is (0, 0), the center is at (100, 100), and the lower-right is (199, 199). These numbers may seem confusing; why do we go from 0 to 199 instead of 1 to 200? The answer is that in code, we usually count from 0 because it's easier for calculations that we'll get into later.

The Display Window

The Display Window is created and images are drawn inside through code elements called *functions*. Functions are the basic building blocks of a Processing program. The behavior of a function is defined by its *parameters*. For example, almost every Processing program has a `size()` function to set the width and height of the Display Window. (If your program doesn't have a `size()` function, the dimension is set to 100×100 pixels.)

Example 3-1: Draw a Window

The `size()` function has two parameters: the first sets the width of the window and the second sets the height. To draw a window that is 800 pixels wide and 600 high, type:

```
size(800, 600);
```

Run this line of code to see the result. Put in different values to see what's possible. Try very small numbers and numbers larger than your screen.

Example 3-2: Draw a Point

To set the color of a single pixel within the Display Window, we use the `point()` function. It has two parameters that define a position: the *x* coordinate followed by the *y* coordinate. To draw a little window and a point at the center of the screen, coordinate (240, 60), type:

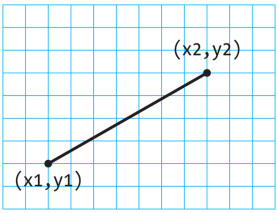

```
size(480, 120);
point(240, 60);
```

Try to write a program that puts a point at each corner of the Display Window and one in the center. Try placing points side by side to make horizontal, vertical, and diagonal lines.

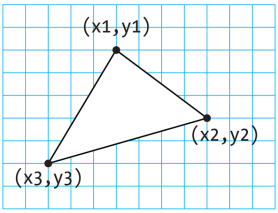
Basic Shapes

Processing includes a group of functions to draw basic shapes (see Figure 3-1). Simple shapes like lines can be combined to create more complex forms like a leaf or a face.

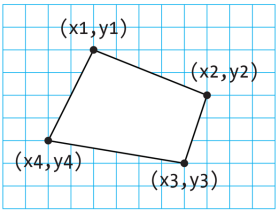
To draw a single line, we need four parameters: two for the starting location and two for the end.



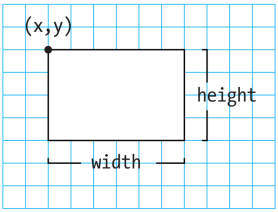
line(x1, y1, x2, y2)



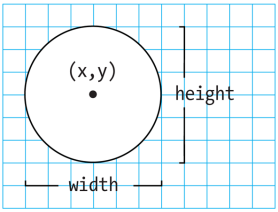
triangle(x1, y1, x2, y2, x3, y3)



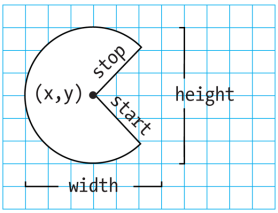
quad(x1, y1, x2, y2, x3, y3, x4, y4)



rect(x, y, width, height)



ellipse(x, y, width, height)

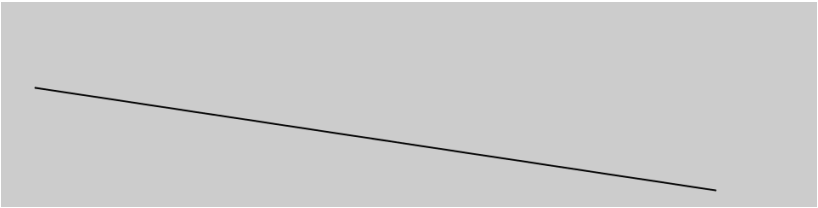


arc(x, y, width, height, start, stop)

Figure 3-1. *Shapes and their coordinates*

Example 3-3: Draw a Line

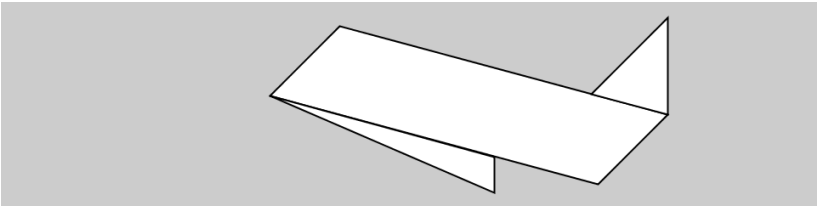
To draw a line between coordinate (20, 50) and (420, 110), try:



```
size(480, 120);
line(20, 50, 420, 110);
```

Example 3-4: Draw Basic Shapes

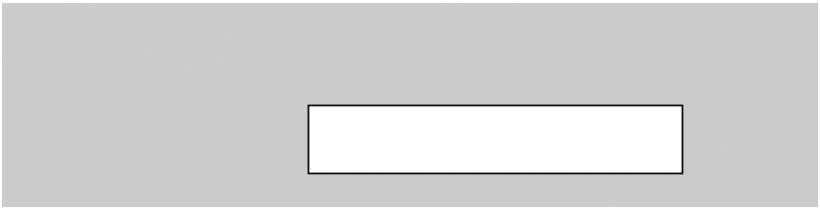
Following this pattern, a triangle needs six parameters and a quadrilateral needs eight (one pair for each point):



```
size(480, 120);
quad(158, 55, 199, 14, 392, 66, 351, 107);
triangle(347, 54, 392, 9, 392, 66);
triangle(158, 55, 290, 91, 290, 112);
```

Example 3-5: Draw a Rectangle

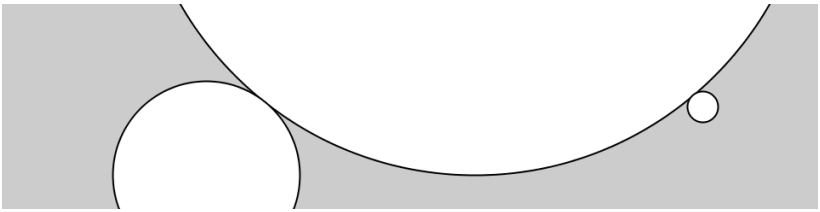
Rectangles and ellipses are both defined with four parameters: the first and second are for the *x* and *y* coordinates of the anchor point, the third for the width, and the fourth for the height. To make a rectangle at coordinate (180, 60) with a width of 220 pixels and height of 40, use the `rect()` function like this:



```
size(480, 120);
rect(180, 60, 220, 40);
```

Example 3-6: Draw an Ellipse

The *x* and *y* coordinates for a rectangle are the upper-left corner, but for an ellipse they are the center of the shape. In this example, notice that the *y* coordinate for the first ellipse is outside the window. Objects can be drawn partially (or entirely) out of the window without an error:

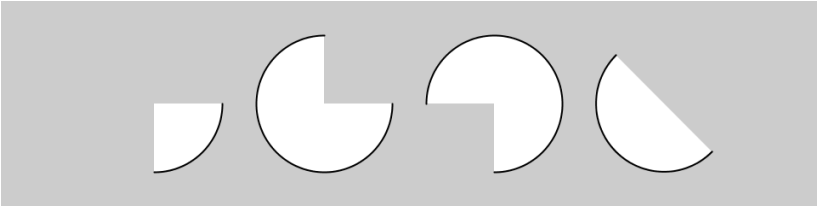


```
size(480, 120);
ellipse(278, -100, 400, 400);
ellipse(120, 100, 110, 110);
ellipse(412, 60, 18, 18);
```

Processing doesn't have separate functions to make squares and circles. To make these shapes, use the same value for the `width` and the `height` parameters to `ellipse()` and `rect()`.

Example 3-7: Draw Part of an Ellipse

The `arc()` function draws a piece of an ellipse:



```
size(480, 120);
arc(90, 60, 80, 80, 0, HALF_PI);
arc(190, 60, 80, 80, 0, PI+HALF_PI);
arc(290, 60, 80, 80, PI, TWO_PI+HALF_PI);
arc(390, 60, 80, 80, QUARTER_PI, PI+QUARTER_PI);
```

The first and second parameters set the location, the third and fourth set the width and height. The fifth parameter sets the angle to start the arc, and the sixth sets the angle to stop. The angles are set in radians, rather than degrees. Radians are angle measurements based on the value of pi (3.14159). Figure 3-2 shows how the two relate. As featured in this example, four radian values are used so frequently that special names for them were added as a part of Processing. The values `PI`, `QUARTER_PI`, `HALF_PI`, and `TWO_PI` can be used to replace the radian values for 180°, 45°, 90°, and 360°.

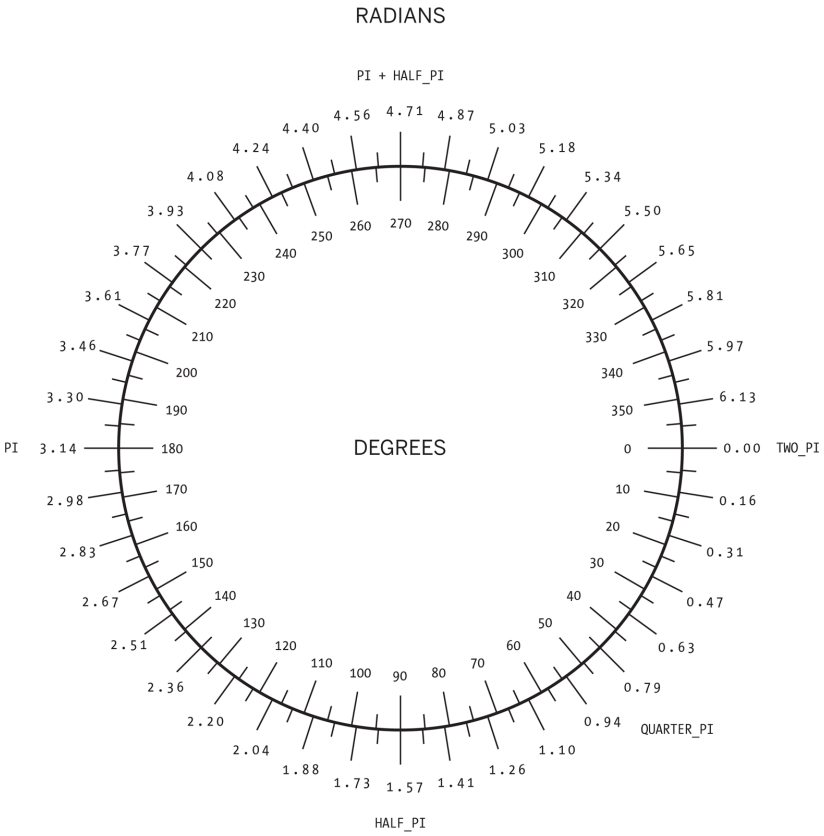


Figure 3-2. *Radians and degrees are two ways to measure an angle. Degrees move around the circle from 0 to 360, while radians measure the angles in relation to pi, from 0 to approximately 6.28.*

Example 3-8: Draw with Degrees

If you prefer to use degree measurements, you can convert to radians with the `radians()` function. This function takes an angle in degrees and changes it to the corresponding radian value. The following example is the same as Example 3-7 on page 18, but it uses the `radians()` function to define the start and stop values in degrees:

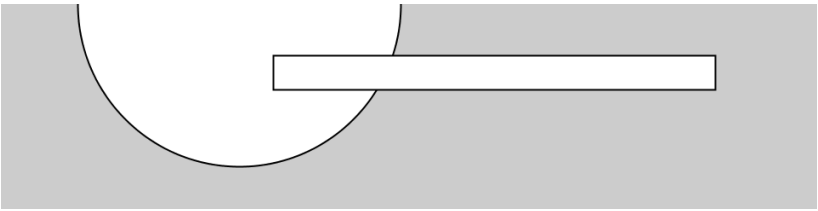
```
size(480, 120);
arc(90, 60, 80, 80, 0, radians(90));
```

```
arc(190, 60, 80, 80, 0, radians(270));
arc(290, 60, 80, 80, radians(180), radians(450));
arc(390, 60, 80, 80, radians(45), radians(225));
```

Drawing Order

When a program runs, the computer starts at the top and reads each line of code until it reaches the last line and then stops. If you want a shape to be drawn on top of all other shapes, it needs to follow the others in the code.

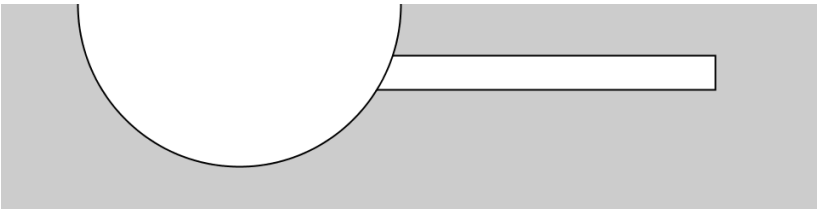
Example 3-9: Control Your Drawing Order



```
size(480, 120);
ellipse(140, 0, 190, 190);
// The rectangle draws on top of the ellipse
// because it comes after in the code
rect(160, 30, 260, 20);
```

Example 3-10: Put It in Reverse

Modify by reversing the order of `rect()` and `ellipse()` to see the circle on top of the rectangle:



```
size(480, 120);
rect(160, 30, 260, 20);
// The ellipse draws on top of the rectangle
```

```
// because it comes after in the code
ellipse(140, 0, 190, 190);
```

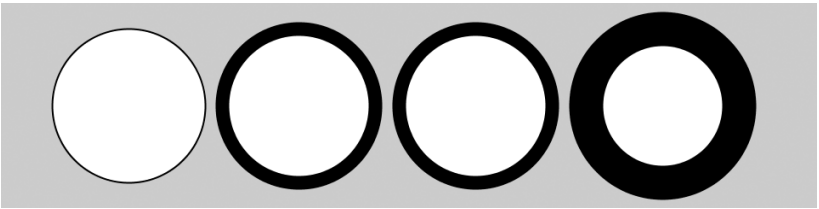
You can think of it like painting with a brush or making a collage. The last element that you add is what's visible on top.

Shape Properties

The most basic and useful shape properties are stroke weight, the way the ends (caps) of lines are drawn, and how the corners of shapes are displayed.

Example 3-11: Set Stroke Weight

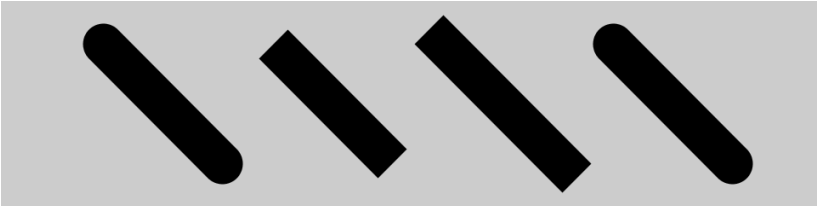
The default stroke weight is a single pixel, but this can be changed with the `strokeWeight()` function. The single parameter to `strokeWeight()` sets the width of drawn lines:



```
size(480, 120);
ellipse(75, 60, 90, 90);
strokeWeight(8);  // Stroke weight to 8 pixels
ellipse(175, 60, 90, 90);
ellipse(279, 60, 90, 90);
strokeWeight(20);  // Stroke weight to 20 pixels
ellipse(389, 60, 90, 90);
```

Example 3-12: Set Stroke Caps

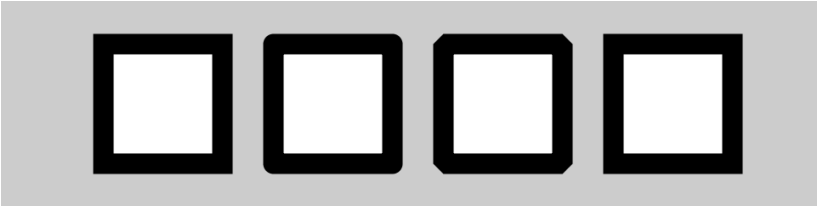
The `strokeCap()` function changes how lines are drawn at their endpoints. By default, they have rounded ends:



```
size(480, 120);
strokeWeight(24);
line(60, 25, 130, 95);
strokeCap(SQUARE);   // Square the line endings
line(160, 25, 230, 95);
strokeCap(PROJECT);  // Project the line endings
line(260, 25, 330, 95);
strokeCap(ROUND);    // Round the line endings
line(360, 25, 430, 95);
```

Example 3-13: Set Stroke Joins

The `strokeJoin()` function changes the way lines are joined (how the corners look). By default, they have pointed (mitered) corners:



```
size(480, 120);
strokeWeight(12);
rect(60, 25, 70, 70);
strokeJoin(ROUND);   // Round the stroke corners
rect(160, 25, 70, 70);
strokeJoin(BEVEL);   // Bevel the stroke corners
rect(260, 25, 70, 70);
strokeJoin(MITER);   // Miter the stroke corners
rect(360, 25, 70, 70);
```

When any of these attributes are set, all shapes drawn afterward are affected. For instance, in Example 3-11 on page 21, notice how the second and third circles both have the same stroke weight, even though the weight is set only once before both are drawn.

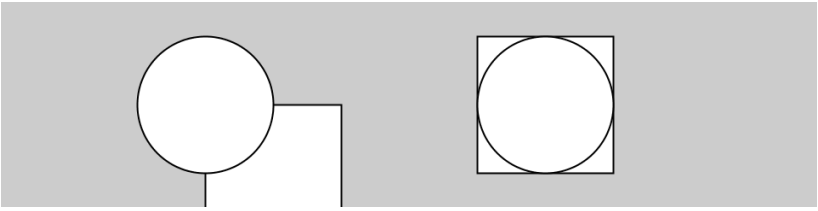
Drawing Modes

A group of functions with "mode" in their name change how Processing draws geometry to the screen. In this chapter, we'll look at `ellipseMode()` and `rectMode()`, which help us to draw

ellipses and rectangles, respectively; later in the book, we'll cover `imageMode()` and `shapeMode()`.

Example 3-14: On the Corner

By default, the `ellipse()` function uses its first two parameters as the *x* and *y* coordinate of the center and the third and fourth parameters as the width and height. After `ellipseMode(CORNER)` is run in a sketch, the first two parameters to `ellipse()` then define the position of the upper-left corner of the rectangle the ellipse is inscribed within. This makes the `ellipse()` function behave more like `rect()` as seen in this example:



```
size(480, 120);
rect(120, 60, 80, 80);
ellipse(120, 60, 80, 80);
ellipseMode(CORNER);
rect(280, 20, 80, 80);
ellipse(280, 20, 80, 80);
```

You'll find these "mode" functions in examples throughout the book. There are more options for how to use them in the *Processing Reference*.

Color

All the shapes so far have been filled white with black outlines, and the background of the Display Window has been light gray. To change them, use the `background()`, `fill()`, and `stroke()` functions. The values of the parameters are in the range of 0 to 255, where 255 is white, 128 is medium gray, and 0 is black. Figure 3-3 shows how the values from 0 to 255 map to different gray levels.

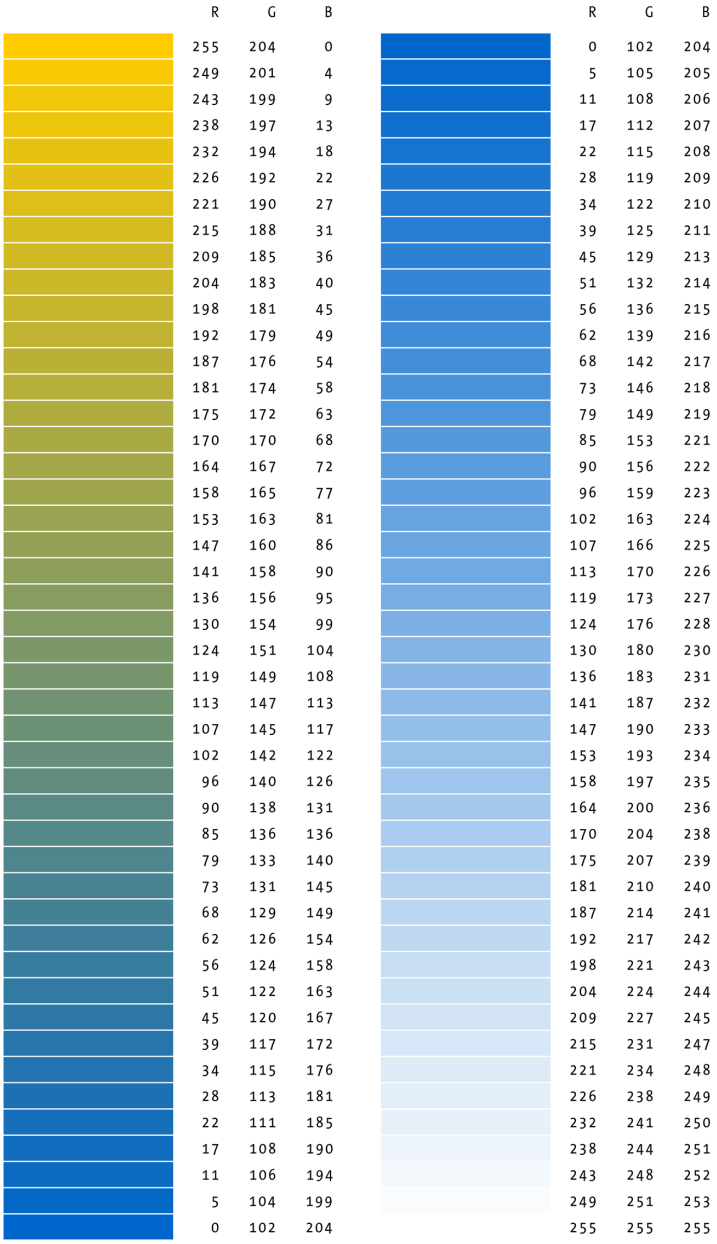
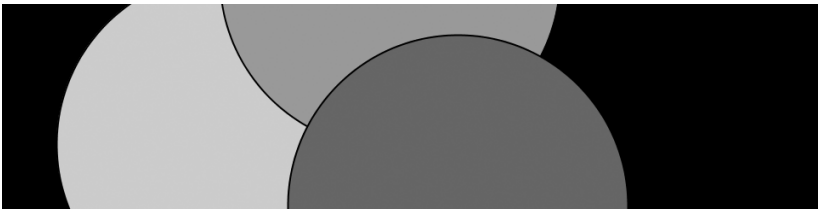


Figure 3-3. *Colors are created by defining RGB (red, green, blue) values*

Example 3-15: Paint with Grays

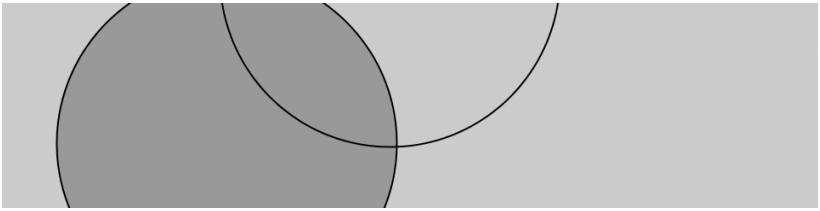
This example shows three different gray values on a black background:



```
size(480, 120);
background(0);               // Black
fill(204);                   // Light gray
ellipse(132, 82, 200, 200);  // Light gray circle
fill(153);                   // Medium gray
ellipse(228, -16, 200, 200); // Medium gray circle
fill(102);                   // Dark gray
ellipse(268, 118, 200, 200); // Dark gray circle
```

Example 3-16: Control Fill and Stroke

You can disable the stroke so that there's no outline by using `noStroke()`, and you can disable the fill of a shape with `noFill()`:



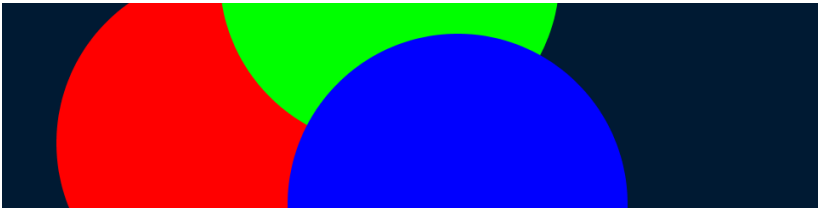
```
size(480, 120);
fill(153);                   // Medium gray
ellipse(132, 82, 200, 200);  // Gray circle
noFill();                    // Turn off fill
ellipse(228, -16, 200, 200); // Outline circle
noStroke();                  // Turn off stroke
ellipse(268, 118, 200, 200); // Doesn't draw!
```

Be careful not to disable the fill and stroke at the same time, as we've done in the previous example, because nothing will draw to the screen.

Example 3-17: Draw with Color

To move beyond grayscale values, you use three parameters to specify the red, green, and blue components of a color.

Run the code in Processing to reveal the colors:



```
size(480, 120);
noStroke();
background(0, 26, 51);         // Dark blue color
fill(255, 0, 0);               // Red color
ellipse(132, 82, 200, 200);    // Red circle
fill(0, 255, 0);               // Green color
ellipse(228, -16, 200, 200);   // Green circle
fill(0, 0, 255);               // Blue color
ellipse(268, 118, 200, 200);   // Blue circle
```

This is referred to as RGB color, which comes from how computers define colors on the screen. The three numbers stand for the values of red, green, and blue, and they range from 0 to 255 the way that the gray values do. Using RGB color isn't very intuitive, so to choose colors, use Tools→Color Selector, which shows a color palette similar to those found in other software (see Figure 3-4). Select a color, and then use the R, G, and B values as the parameters for your `background()`, `fill()`, or `stroke()` function.

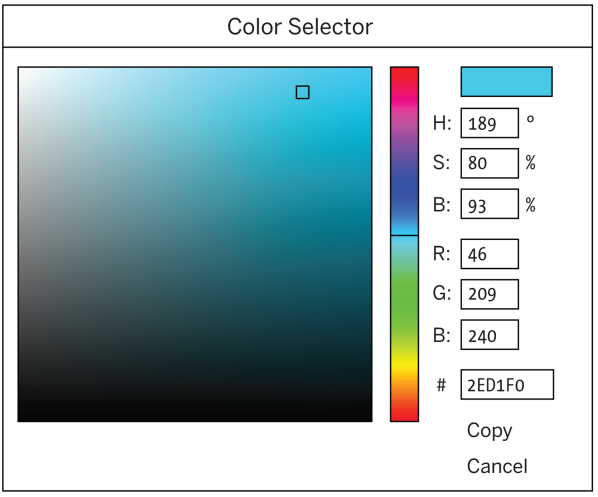
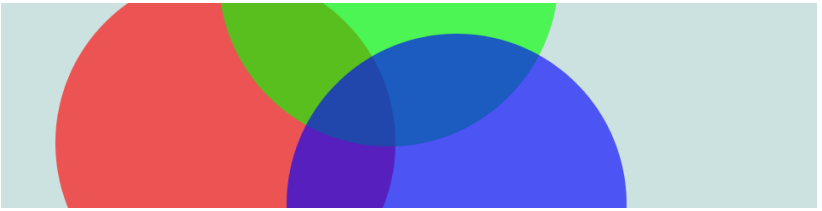


Figure 3-4. *Processing Color Selector*

Example 3-18: Set Transparency

By adding an optional fourth parameter to `fill()` or `stroke()`, you can control the transparency. This fourth parameter is known as the *alpha* value, and also uses the range 0 to 255 to set the amount of transparency. The value 0 defines the color as entirely transparent (it won't display), the value 255 is entirely opaque, and the values between these extremes cause the colors to mix on screen:



```
size(480, 120);
noStroke();
background(204, 226, 225);    // Light blue color
fill(255, 0, 0, 160);         // Red color
ellipse(132, 82, 200, 200);   // Red circle
fill(0, 255, 0, 160);         // Green color
ellipse(228, -16, 200, 200);  // Green circle
```

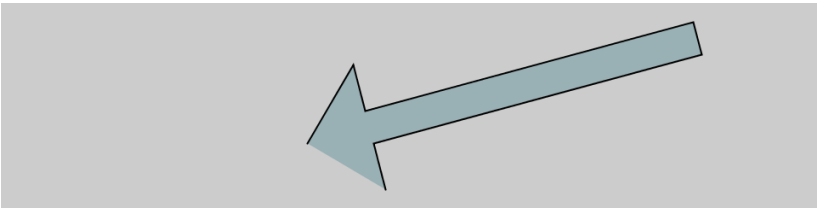
```
fill(0, 0, 255, 160);         // Blue color
ellipse(268, 118, 200, 200);  // Blue circle
```

Custom Shapes

You're not limited to using these basic geometric shapes—you can also define new shapes by connecting a series of points.

Example 3-19: Draw an Arrow

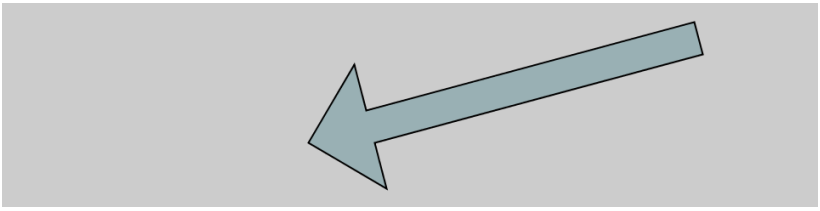
The `beginShape()` function signals the start of a new shape. The `vertex()` function is used to define each pair of *x* and *y* coordinates for the shape. Finally, `endShape()` is called to signal that the shape is finished:



```
size(480, 120);
beginShape();
fill(153, 176, 180);
vertex(180, 82);
vertex(207, 36);
vertex(214, 63);
vertex(407, 11);
vertex(412, 30);
vertex(219, 82);
vertex(226, 109);
endShape();
```

Example 3-20: Close the Gap

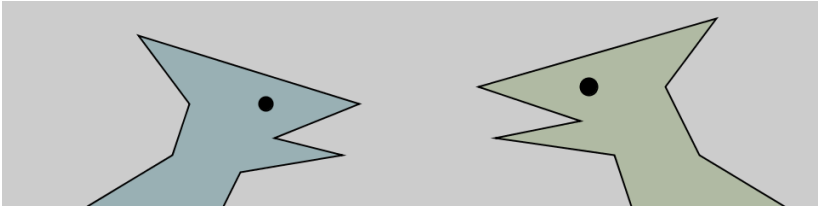
When you run Example 3-19 on page 28, you'll see the first and last point are not connected. To do this, add the word `CLOSE` as a parameter to `endShape()`, like this:



```
size(480, 120);
beginShape();
fill(153, 176, 180);
vertex(180, 82);
vertex(207, 36);
vertex(214, 63);
vertex(407, 11);
vertex(412, 30);
vertex(219, 82);
vertex(226, 109);
endShape(CLOSE);
```

Example 3-21: Create Some Creatures

The power of defining shapes with `vertex()` is the ability to make shapes with complex outlines. Processing can draw thousands and thousands of lines at a time to fill the screen with fantastic shapes that spring from your imagination. A modest but more complex example follows:



```
size(480, 120);

// Left creature
fill(153, 176, 180);
beginShape();
vertex(50, 120);
vertex(100, 90);
vertex(110, 60);
vertex(80, 20);
vertex(210, 60);
```

```
vertex(160, 80);
vertex(200, 90);
vertex(140, 100);
vertex(130, 120);
endShape();
fill(0);
ellipse(155, 60, 8, 8);

// Right creature
fill(176, 186, 163);
beginShape();
vertex(370, 120);
vertex(360, 90);
vertex(290, 80);
vertex(340, 70);
vertex(280, 50);
vertex(420, 10);
vertex(390, 50);
vertex(410, 90);
vertex(460, 120);
endShape();
fill(0);
ellipse(345, 50, 10, 10);
```

Comments

The examples in this chapter use double slashes (//) at the end of a line to add comments to the code. Comments are parts of the program that are ignored when the program is run. They are useful for making notes for yourself that explain what's happening in the code. If others are reading your code, comments are especially important to help them understand your thought process.

Comments are also especially useful for a number of different options, such as when trying to choose the right color. So, for instance, I might be trying to find just the right red for an ellipse:

```
size(200, 200);
fill(165, 57, 57);
ellipse(100, 100, 80, 80);
```

Now suppose I want to try a different red, but don't want to lose the old one. I can copy and paste the line, make a change, and then "comment out" the old one:

```
size(200, 200);
//fill(165, 57, 57);
fill(144, 39, 39);
ellipse(100, 100, 80, 80);
```

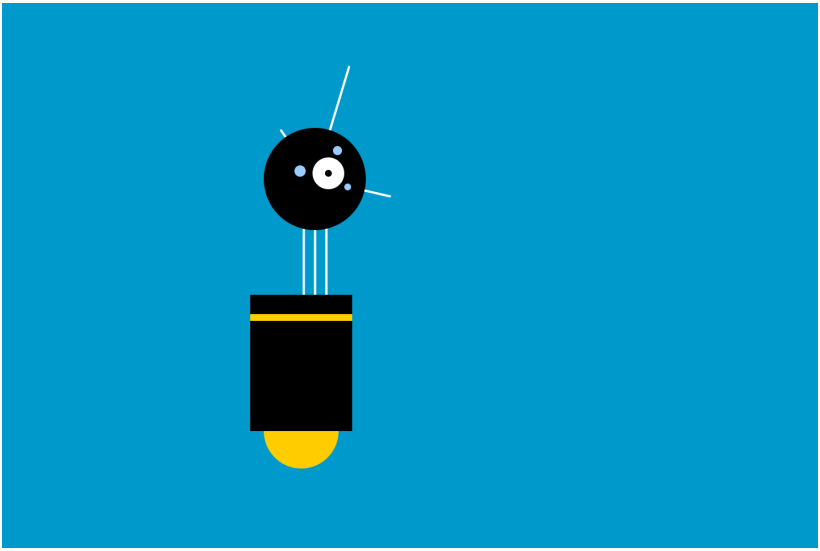
Placing // at the beginning of the line temporarily disables it. Or I can remove the // and place it in front of the other line if I want to try it again:

```
size(200, 200);
fill(165, 57, 57);
//fill(144, 39, 39);
ellipse(100, 100, 80, 80);
```

As you work with Processing sketches, you'll find yourself creating dozens of iterations of ideas; using comments to make notes or to disable code can help you keep track of multiple options.

As a shortcut, you can also use Ctrl-/ (Cmd-/ on the Mac) to add or remove comments from the current line or a selected block of text. You can also comment out many lines at a time with the alternative comment notation introduced in "Comments" on page 203.

Robot 1: Draw



This is P5, the Processing Robot. There are 10 different programs to draw and animate him in the book—each one explores a different programming idea. P5's design was inspired by Sputnik I (1957), Shakey from the Stanford Research Institute (1966–1972), the fighter drone in David Lynch's *Dune* (1984), and HAL 9000 from *2001: A Space Odyssey* (1968), among other robot favorites.

The first robot program uses the drawing functions introduced in this chapter. The parameters to the `fill()` and `stroke()` functions set the gray values. The `line()`, `ellipse()`, and `rect()` functions define the shapes that create the robot's neck, antennae, body, and head. To get more familiar with the functions, run the program and change the values to redesign the robot:

```
size(720, 480);
strokeWeight(2);
background(0, 153, 204);     // Blue background
ellipseMode(RADIUS);

// Neck
stroke(255);                 // Set stroke to white
line(266, 257, 266, 162);    // Left
```

```
line(276, 257, 276, 162);   // Middle
line(286, 257, 286, 162);   // Right

// Antennae
line(276, 155, 246, 112);   // Small
line(276, 155, 306, 56);    // Tall
line(276, 155, 342, 170);   // Medium

// Body
noStroke();                 // Disable stroke
fill(255, 204, 0);          // Set fill to orange
ellipse(264, 377, 33, 33);  // Antigravity orb
fill(0);                    // Set fill to black
rect(219, 257, 90, 120);    // Main body
fill(255, 204, 0);          // Set fill to yellow
rect(219, 274, 90, 6);      // Yellow stripe

// Head
fill(0);                    // Set fill to black
ellipse(276, 155, 45, 45);  // Head
fill(255);                  // Set fill to white
ellipse(288, 150, 14, 14);  // Large eye
fill(0);                    // Set fill to black
ellipse(288, 150, 3, 3);    // Pupil
fill(153, 204, 255);        // Set fill to light blue
ellipse(263, 148, 5, 5);    // Small eye 1
ellipse(296, 130, 4, 4);    // Small eye 2
ellipse(305, 162, 3, 3);    // Small eye 3
```