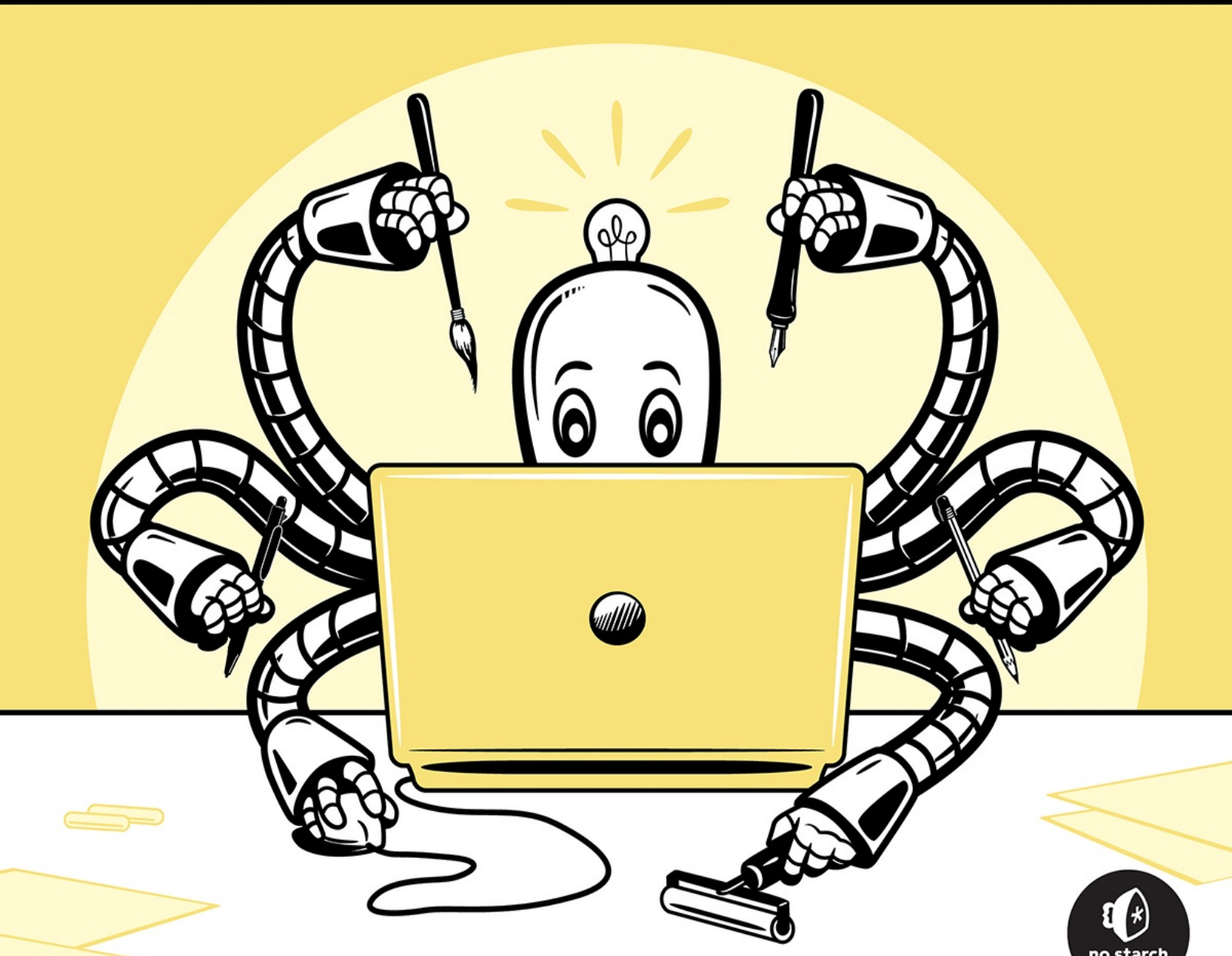
AN ARTIST'S GUIDE TO PROGRAMMING

A GRAPHICAL INTRODUCTION

JIM PARKER



no starch press

CONTENTS IN DETAIL

AN ARTIST'S GUIDE TO PROGRAMMING

A Graphical Introduction

Jim Parker

no starch press

San Francisco

First printing

25 24 23 22 1 2 3 4 5 6 7 8 9

ISBN-13: 978-1-7185-0164-5 (print)
ISBN-13: 978-1-7185-0165-2 (ebook)

Publisher: William Pollock
Production Manager: Rachel Monaghan
Production Editor: Paula Williamson
Developmental Editors: Athabasca Witschi and Nathan Heidelberger
Cover Illustration: Gina Redman
Interior Design: Octopod Studios
Technical Reviewer: Jeffrey Boyd
Copyeditor: Andy Carroll
Compositor: Jeff Lytle, Happenstance Type-O-Rama
Proofreader: Emelie Battaglia

The following images are reproduced with permission: Figure 96-1 by Brocken Inaglory, printed under the GNU Free Documentation License, Version 1.2.

For information on distribution, bulk sales, corporate sales, or translations, please contact No Starch Press, Inc. directly at info@nostarch.com or:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1-415-863-9900
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

```
Names: Parker, J. R. (Jim R.), 1955- author.
Title: An artist's guide to programming : a graphical introduction / Jim
   Parker.
Description: San Francisco : No Starch Press, 2022. | Includes index. |
Identifiers: LCCN 2021046087 (print) | LCCN 2021046088 (ebook) | ISBN
   9781718501645 (print) | ISBN 9781718501652 (ebook)
Subjects: LCSH: Multimedia systems. | Computer graphics. |
   Microcomputers--Programming. | Processing (Computer program language) |
   Java (Computer program language)
Classification: LCC QA76.575 .P357 2022 (print) | LCC QA76.575 (ebook) |
   DDC 006.7--dc23
LC record available at https://lccn.loc.gov/2021046087
LC ebook record available at https://lccn.loc.gov/2021046088
```

shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Author

Jim Parker is a professor, author, and artist who has published a dozen books and over 170 technical papers, in addition to writing short stories. He has degrees in mathematics and computer science, and a PhD from the State University of Ghent, Belgium. His areas of expertise include computer simulation, image processing, artificial intelligence, game design, and generative art. He has exhibited generative art, and even sent art into space. He lives on a small ranch in the foothills of the Rocky Mountains, where he helps raise small animals and Tennessee walking horses.

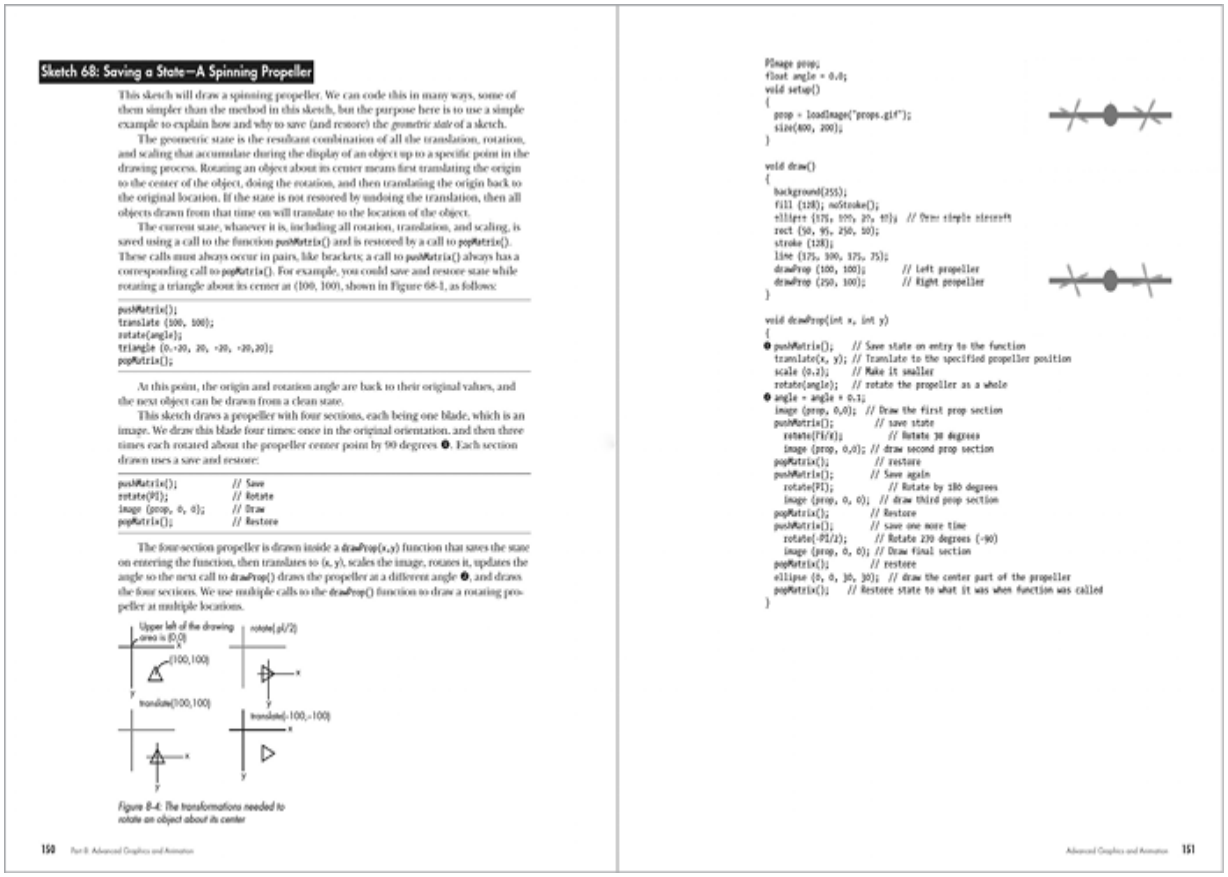
About the Technical Reviewer

Jeffrey Boyd received his PhD in computer science from the University of British Columbia and is currently an associate professor at the University of Calgary. His research focuses on sensing and computer vision, with applications in analyzing human motion, interactive art and music, sonification, and computational musicology. Dr. Boyd and his students do diverse work that includes wearable systems that provide real-time, sonic feedback to train and rehabilitate speed skaters; numerous art and sound installations; and the musicological study of contemporary composers through the analysis of ambisonic recordings of their work.

AUTHOR'S NOTE

Processing is a programming language designed by Casey Reas and Ben Fry to be used by artists creating generative art. It is based on Java, and it extends Java in many useful ways: it allows the programmer to easily read, display, and write image files; it has functions for drawing elementary shapes and curves; it makes manipulating colors simple; and so on. Most importantly, it opens a window into which the programmer/artist will draw. All Processing programs are intended to create an image, a visual output.

This book presents the Processing language and its many applications in a set of graduated examples. The details of the syntax are not the focus, although they are explained briefly. The idea is to present a collection of programs that the reader can experiment with. When the book is open to any sketch, the left side of the page will have descriptive text, while the right will show the program and the result, as an image.



Sketch 68: Saving a State—A Spinning Propeller

This sketch will draw a spinning propeller. We can code this in many ways, some of them simpler than the method in this sketch, but the purpose here is to use a simple example to explain how and why to save (and restore) the *geometric state* of a sketch.

The geometric state is the resultant combination of all the translation, rotation, and scaling that accumulate during the display of an object up to a specific point in the drawing process. Rotating an object about its center means first translating the origin to the center of the object, doing the rotation, and then translating the origin back to the original location. If the state is not restored by undoing the translation, then all objects drawn from that time on will translate to the location of the object.

The current state, whatever it is, including all rotation, translation, and scaling, is saved using a call to the function `pushMatrix()` and is restored by a call to `popMatrix()`. These calls must always occur in pairs, like brackets; a call to `pushMatrix()` always has a corresponding call to `popMatrix()`. For example, you could save and restore state while rotating a triangle about its center at (100, 100), shown in Figure 68-1, as follows:

```
pushMatrix();
translate (100, 100);
rotate(angle);
triangle (0,-20, 20, -20, -20,20);
popMatrix();
```

At this point, the origin and rotation angle are back to their original values, and the next object can be drawn from a clean state.

This sketch draws a propeller with four sections, each being one blade, which is an image. We draw this blade four times: once in the original orientation, and then three times each rotated about the propeller center point by 90 degrees ❶. Each section drawn uses a save and restore:

```
pushMatrix();          // Save
rotate(PI);            // Rotate
image (prop, 0, 0);    // Draw
popMatrix();           // Restore
```

The four-section propeller is drawn inside a `drawProp(x,y)` function that saves the state on entering the function, then translates to (x, y), scales the image, rotates it, updates the angle so the next call to `drawProp()` draws the propeller at a different angle ❷, and draws the four sections. We use multiple calls to the `drawProp()` function to draw a rotating propeller at multiple locations.

Figure 8-4: The transformations needed to rotate an object about its center


```
PImage prop;
float angle = 0.0;
void setup()
{
  prop = loadImage("props.gif");
  size(400, 200);
}

void draw()
{
  background(255);
  fill (128); noStroke();
  ellipse (175, 100, 37, 37);  // Draw simple aircraft
  rect (50, 95, 250, 10);
  stroke (128);
  line (175, 100, 175, 75);
  drawProp (100, 100);        // Left propeller
  drawProp (250, 100);        // Right propeller
}

void drawProp(int x, int y)
{
❶ pushMatrix();    // Save state on entry to the function
  translate(x, y); // Translate to the specified propeller position
  scale (0.2);     // Make it smaller
  rotate(angle);   // rotate the propeller as a whole
❷ angle = angle + 0.1;
  image (prop, 0,0);  // Draw the first prop section
  pushMatrix();         // save state
    rotate(PI/2);       // Rotate 90 degrees
    image (prop, 0,0); // draw second prop section
  popMatrix();          // restore
  pushMatrix();         // Save again
    rotate(PI);         // Rotate by 180 degrees
    image (prop, 0, 0);  // draw third prop section
  popMatrix();          // Restore
  pushMatrix();         // save one more time
    rotate(-PI/2);      // Rotate 270 degrees (-90)
    image (prop, 0, 0); // Draw final section
  popMatrix();          // restore
  ellipse (0, 0, 30, 30);  // draw the center part of the propeller
  popMatrix();    // Restore state to what it was when function was called
}
```


The code is available to you. Use it, change it, share it. You can download the code and all the necessary supporting files (images, sound files, and so on) at *https://nostarch.com/artists-guide-programming/*.

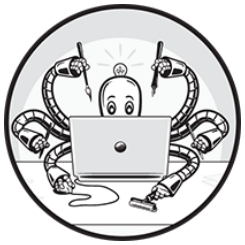
Some concepts will be more complex than others, of course. There is plenty of documentation for Processing on the internet, beginning with the *https://processing.org/* site.

Processing can be used with Arduino computers too. It has modules for sound, video, and scientific calculations, and it can be used to present images in a browser. It is my hope that this book will allow you to start experimenting with programming and generative art.

Why are there few comments in the code? To save space on the right page! The entire left page is a description of the program and method, so it is in effect a large comment.

—Jim Parker

INTRODUCTION



The Basics of a Programming Language: Processing

When someone programs a computer, they are really communicating with it. It is an imperative and precise communication. Imperative because the computer has no choice; it is being told what to do, and it will do exactly that. Precise because a computer does not apply any interpretation to what it is being told. Computers do not think and so can't evaluate a command that would amount to "expose the patient to a fatal dose of radiation" with any skepticism. So we, as programmers, must be careful and precise in what we instruct the machine to do.

When humans communicate with each other, we use a language. Similarly, humans use languages to communicate with computers, but these languages are artificial (humans invented them for this purpose), terse (there are few if any modifiers—no way to express emotions or gradations of any feeling), precise (each item in the language means one thing), and written (we do not yet speak to computers in a programming language).

The process of programming begins with a problem to be solved, and the first step is to state the problem as clearly as possible. Then we analyze the problem and determine methods by which it may be solved. Computers can only directly manipulate numbers, so it is common for solutions discussed at this stage to be numerical or mathematical. A sketch of the solution, perhaps on paper in a human language and math, is created. This is then translated into computer language and typed into the computer using a keyboard. The resulting text file is called a *script, source code*, or more commonly the *computer program*. Next, another program called a *compiler* takes the program and converts it into a form that can be executed on the computer. Basically, all programs are converted into *machine code,* which consists of numbers, and which the computer can execute.

You are going to learn a language called *Processing*. It was developed for use by artists, but it's pretty good for lots of things, and it's good for teaching because it makes a lot of things easy and it always has graphical visual output. It is much like a lot of other languages in use these days in terms of structure (syntax). It is, in fact, the language Java enclosed in some

special easy-to-use packaging. A Processing program is called a *sketch* in honor of its artistic origins.

In order to use a programming language, you need to understand some basic concepts and structures, at least at a basic level. These concepts will be introduced in this introduction. The rest of the book will teach you to program by example: when you open the book to a random location, the left page will almost always outline a problem or Processing language concept, and the right page will almost always show code that illustrates that concept, along with a screen image of the output from that program. The idea is to introduce only one or two new things on any page. The code will execute on a computer running any major operating system, once the free Processing language download has been installed. Go to *https://processing.org/download* and download the latest stable version for your OS.

To begin programming, you need to appreciate that a language has a syntax or structure, and for computer languages this structure cannot be varied. The computer will always be the arbiter of what is correct, and if any program has a syntax error or produces erroneous results, it is the program and not the computer that is at fault.

Next you need to appreciate that the syntax is arbitrary. It was designed by a human with attitudes and biases and new ideas, and while the syntax might be ugly or hard to recall, it is what it is. You might not understand parts of it at first, but after a while and after reading and executing the first 50 or 60 sketches in this book, most of it will make sense.

A program consists of symbols, and their order matters. Some symbols are special characters with a defined meaning. For example, `+` usually means *add*, and `-` usually means *subtract*. Some symbols are words, and words defined by the language, like `if`, `while`, and `true`, cannot be also defined by a programmer—they mean what the language says they mean, and they are called *reserved words*. Some names have a definition given by the system but can be reused by a programmer if needed. These are called *predefined names* or *system variables*. However, some words can be defined by the programmer and are names for things the programmer wants to use in the program: *variables* and *functions* are examples.

The Beginning

All sketches have the same basic structure. There is something called `setup()` (a predefined name) that gets executed just once, when the program begins. This is where we will do initializations, such as defining the size of the output window. If we need to read a bunch of images or sounds from files, this is where we might do it.

NOTE

This discussion presents a template that can be used to start coding any program. Detailed explanation of the syntax will come later.

The syntax of `setup()` is as follows:

```
void setup ()
{
    your code goes here
}
```

This is something we call a *function* in Processing (see Sketch 24). It is a bunch of code that is enclosed in braces (the `{` and `}`) and is given a name. It gets executed (*called*, we say) when we use the name in code later on. In this case the function is named `setup()`, and it is invoked automatically by Processing just once, when the program starts executing. The word `void` (a reserved word) is not important just now, but it means the function does not return a value.

After `setup()` has finished, a window will open on the screen where the program will draw. This is called the *sketch window*, and its size is one of the things initialized within `setup()`.

The Middle

The second part of a sketch is another function, one named `draw()`. This function is called many times each second (the default is 60 times, but this can be changed), and its purpose is to update the drawing being made by the

program. Processing assumes that the programmer is writing a program to draw a picture of some kind.

Every 1/60 of a second, the Processing system will call the `draw()` function. Whatever code appears there will be executed each time, and the idea is that the programmer can update the picture being created there as a user watches. For example, if a set of images of a moving animal is displayed one at a time, the result will be an animated image of the animal. The programmer can draw shapes, display text and images, change colors, and move shapes about the screen as the user watches.

The syntax of `draw` (a predefined name) is as follows:

```
void draw ()
{
  your code goes here
}
```

The Rest

The programmer writes code that is inside either of the functions `setup()` and `draw()` or that is executed by those functions. Any part of the program that cannot be reached from `setup()` or `draw()` will never be executed (except for some of the mouse and keyboard functions).

The programmer can name and provide code for other functions, and these can be executed by (called from) `draw()` or `setup()`. These functions are usually placed after the `draw()` function in the program. For example, if the programmer wanted to define a function named `doSomething()`, it might look like this:

```
void doSomething ()
{
  your code goes here
}
```

This would be executed when its name was used in a call:

```
void draw ()
{
```

```
  doSomething();
}
```

The semicolon is used to end a statement so that Processing knows when the programmer thinks a statement ends. It is used to detect errors: if a programmer thinks the statement is complete and the Processing compiler does not, the compiler issues an error message. The compiler is, after all, always right.

Variables

The concept of a variable is one that most beginners find difficult. Essentially, a variable is a place to put a result, usually a number. In a programming language, a variable is represented by a name, and the connection between the name and the value is established by a statement in the language called an *assignment statement*: it assigns a value to a variable. Here's an example:

```
count = 0;
```

This establishes that the value of a variable named `count` is `0`. How do we know that the name `count` is a variable? It must appear in a *declaration*: we "declare" that `count` is a variable, and we specify a type. The type defines the set of values that can be assigned to the variable. For a numerical variable, common types are `integer` and `float` (a decimal fraction). If `count` is to be an integer, then this would be the declaration:

```
int count;
```

The predefined name `int` means integer, and this declaration states that the name `count` will hold an integer. If it were supposed to be a number with a fraction (a *real* or *floating-point* number) the declaration would be as follows:

```
float count;
```

A variable can only be used after it has been declared. It is an error to attempt to use a variable that has not been in a declaration, partly because its type would not be known.

Now that you can define variables, you can do complex computations. For arithmetic the usual operations are possible: `+` (add), `-` (subtract), `*` (multiply), and `/` (divide). Both variables and constants can be used in mathematical expressions, just as in algebra. The following would be a legal assignment statement (assuming that the name `radius` was declared):

```
count = 2 * 3.1419926 * radius;
```

It would calculate the circumference of a circle with the given radius.

How to Write a Program

When Processing is started, either by clicking *processing.exe* or by clicking a Processing source file, the integrated development environment (IDE) will open a window on the screen. It will look something like *Figure 1*, though it may look a little different depending on your operating system and the version of Processing you use.

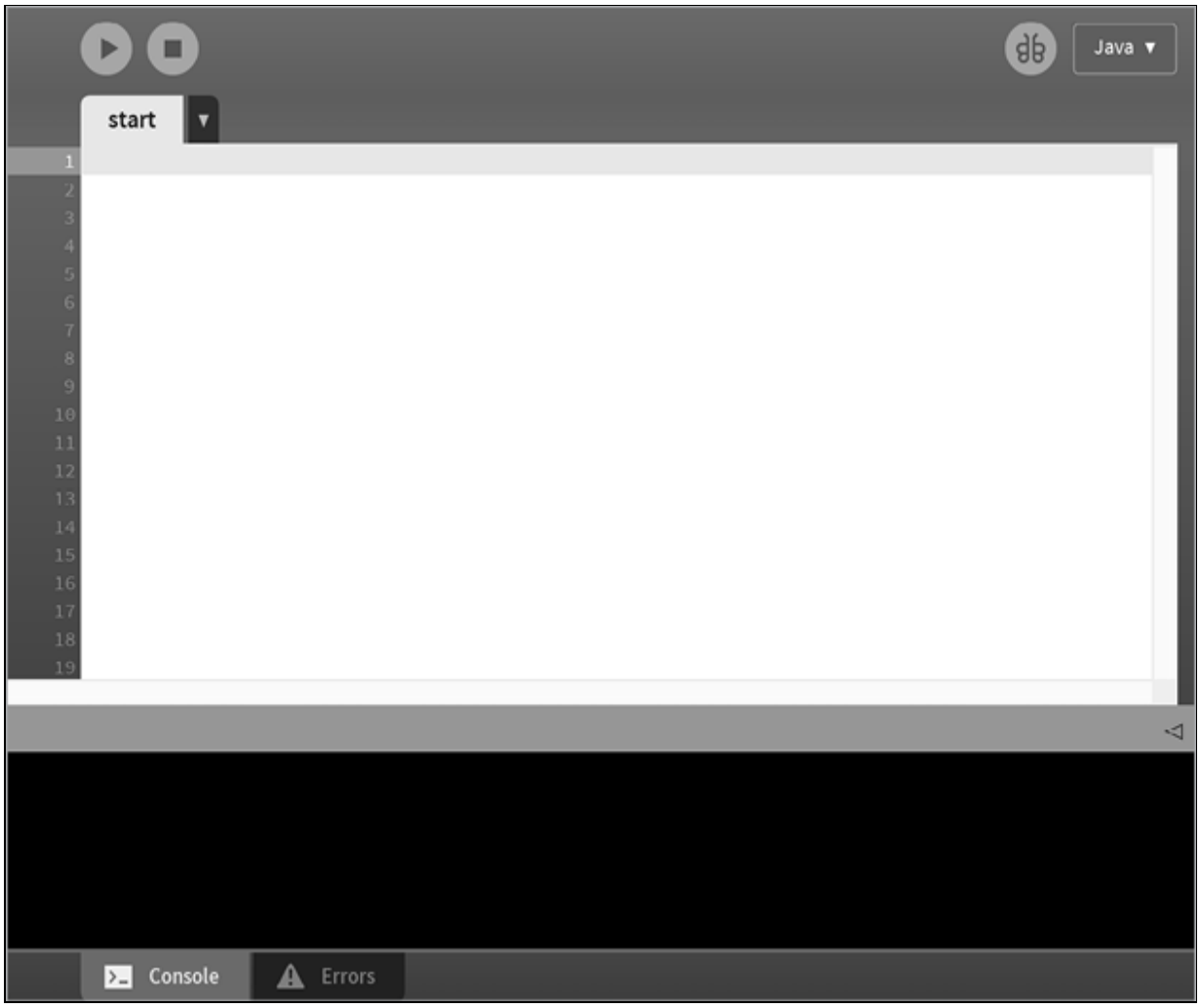


Figure 1: A new window in the Processing integrated development environment

This particular sketch is called *start*, and it resides in a file named *start.pde* (*pde* stands for *Processing Development Environment*). The *start.pde* file must also be located within a directory named *start*. That's just the rule.

Now you can start typing code, and it will appear inside the white rectangle in the window. This code will execute when the start icon is clicked, and running code will halt when the stop icon is clicked.

Let's try a simple program: one that draws a circle. First, enter the basic empty program just described, as shown in *Figure 2*.

Figure 2: The basic structure of a Processing program

Now we can write our code. We wish to draw a circle, and Processing will open a drawing window for us. We should specify its size so it's not too small. In `setup()` we can use the predefined `size()` function to specify a sketch window with a size of 400 pixels horizontally and 300 pixels vertically.

We want the `draw()` function to draw a circle every time it is called, 60 times per second by default. In Processing, a circle is a special case of an ellipse, having equal width and height. The `ellipse()` function draws an ellipse with its center at specified coordinates (the first two values in the parentheses after the function name) and having a width and height specified by the second pair of values. These values in parentheses after a function name are called *parameters.* The background color is set by default to a medium grey, and the color that fills the circle is white. The circle is outlined by a black line.

The call `ellipse (200, 100, 50, 50)` will draw an ellipse centered at (200, 100) that is 50 pixels wide and 50 pixels high. Once this code is entered, the window will look like *Figure 3*.

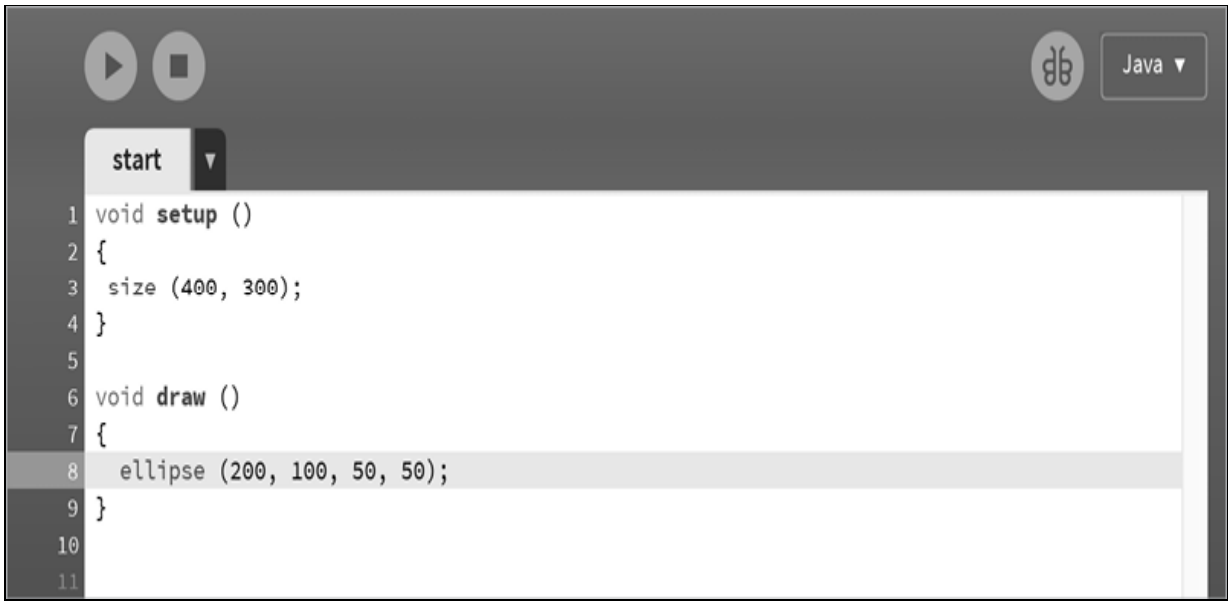


Figure 3: The code for drawing a circle

Now click the start icon. A new window opens with our drawing, as shown in *Figure 4*.

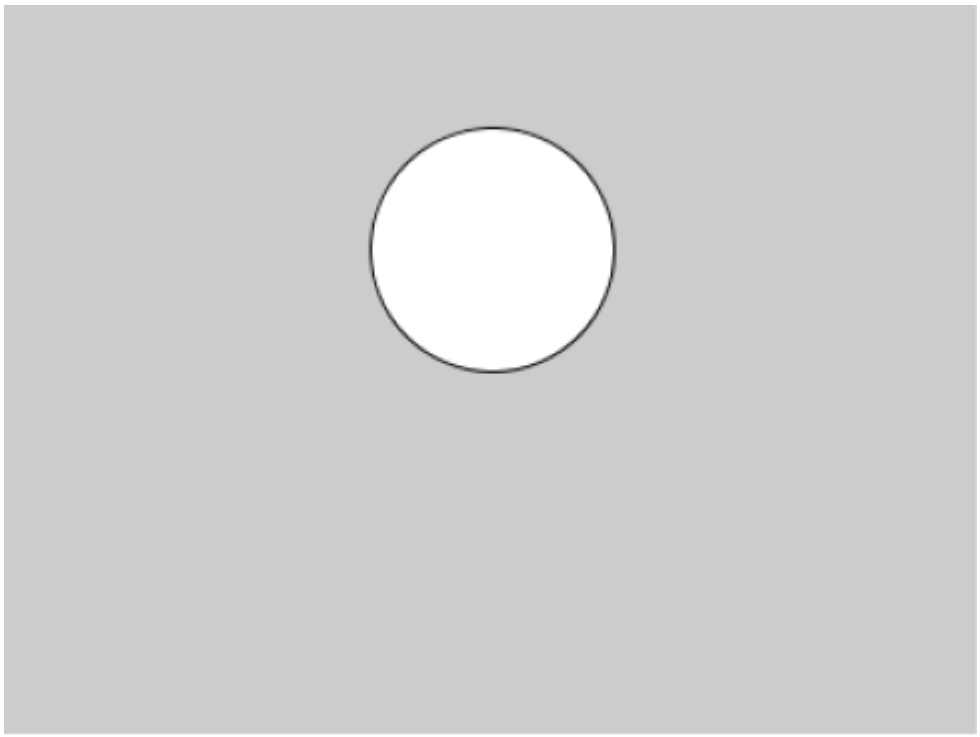
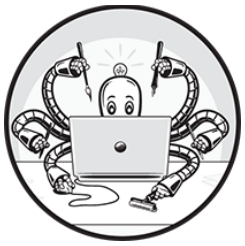


Figure 4: The drawing window

You have learned a few things. The value 200 in the ellipse is the x or horizontal position, and 100 is the y or vertical position. The value 50 is the size of ellipse, which in this case is a circle because the horizontal and vertical sizes are the same. The circle is filled with a color, in this case white, and it has a black line around it.

The remainder of this book essentially involves learning by doing. There's a lot of code and relatively little explanation. You can experiment with the code, change the parameters, and see what happens. That's the whole point. You'll learn the syntax by example and by trying things out.

1
THE FUNDAMENTALS OF DRAWING



Sketch 1: A Circle

Drawing a circle requires quite a bit of code in C or Java, but it's one of the simplest programs in Processing. There isn't a circle function in Processing, so to draw a circle we draw an ellipse that has equal width and height, which is the same thing as a circle.

Example A

The `setup()` function calls the predefined `size()` function to open a sketch window with a width of 400 pixels and a height of 300 pixels 1.

The `draw()` function draws a circle every time it's called (60 times per second by default) with the `ellipse()` function, which has four parameters. The first and second parameters specify the pixel coordinates of the ellipse's center. The third and fourth parameters specify the ellipse's width and height. The call `ellipse (200, 150, 50, 50)` 2 draws an ellipse centered at (200, 100) that is 50 pixels wide and 50 pixels high, which is essentially a circle with a diameter of 50 pixels.

By default, the background color is set to a medium grey, and the color that fills the circle is white. The circle is outlined by a black line.

Example B

This example is much like Example A, but now the background color is set to white, and the color that fills the circle (and any other basic closed shape) is set to black.

The `background()` function 1 specifies the background color with a single number parameter ranging from 0 to 255 that indicates levels of gray, where 0 is black and 255 is white. Numerical values outside of this range are illegal. In this case the color is set to white (`255`). The `background()` function is specified in the `draw()` function so that the background is redrawn each time. If `background()` was called in `setup()`, the background would only be drawn once, at the beginning of execution.

The `fill()` function 2 specifies the fill color of basic closed shapes with the same single number parameter as the `background()` function. In this case the fill color is set to black (`0`), and it remains so until changed by another call to `fill()`. Thus, `fill()` could have been called just once within `setup()` and the effect would have been the same.

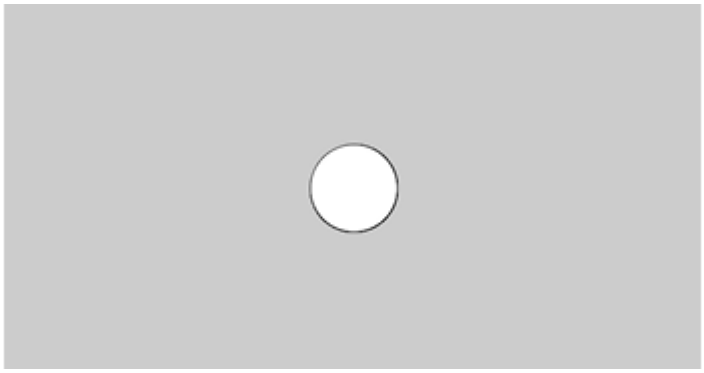
Example C

In this case the background color (white = `255`) 1 and fill color (black = `0`) 2 are specified in `setup()`. This sketch draws two ellipses, not circles, in the `draw()` function to show how the width and height parameters are used. The first call to `ellipse()` 3 draws the leftmost ellipse, which is 100 pixels wide and 50 pixels high. The second call to `ellipse()` 4 draws the rightmost ellipse, which is 50 pixels wide and 100 pixels high.

The `noFill()` function causes ellipses and other objects to be drawn without any fill color so that the background color shows inside the object.

Example A

```
void setup ()
{
1 size (400, 300);
}

void draw ()
{
2 ellipse (200, 150, 50, 50);
}
```



Example B

```
void setup ()
{
  size (400, 300);
}

void draw ()
{
1 background (255);
2 fill (0);
  ellipse (200, 150, 50, 50);
}
```

Example C

```
void setup ()
{
  size (400, 300);
1 background (255);
2 fill (0);
}

void draw ()
{
3 ellipse (100, 150, 100, 50);
4 ellipse (300, 150, 50, 100);
}
```

Sketch 2: Colors

We can specify a shade of grey with a single numerical component, but we can also specify a color by providing three numerical components to the same function. These components are given in the traditional order: red, then green, then blue. Each component fits in a single byte (8 bits), and it is represented by a number ranging from 0 to 255 that determines the shade of the component. Smaller values yield a darker color.

The numbers (255, 0, 0) specify the brightest shade of red, while the numbers (254, 0, 0) specify a slightly darker shade of red. Green would be (0, 255, 0) and blue would be (0, 0, 255). Yellow is red and green, so a set of RGB coordinates for yellow would be (255, 255, 0). Magenta is red and blue, so it would be written as (255, 0, 255). Grey values have the three components nearly equal.

This is the *RGB representation of color*. There are other representations.

Example A

This sketch draws circles of various colors. In the `draw()` function, the first three calls to `fill()` and `ellipse()` draw the first row of circles: red 1, green 2, and blue 3. The fill color changes prior to drawing each circle.

The second row of circles is filled with yellow 4, magenta 5, and cyan 6. Each color here has two nonzero color values.

The final row contains circles filled with increasingly brighter grey values 7. Each color here has three equal color values.

Example B

We can also use a fourth color component that represents *transparency*, sometimes referred to as the *alpha* channel. The components (255, 0, 0, 128) indicate that red is 255, green and blue are 0, and transparency is 128, or 50 percent. Higher numerical values indicate lower transparency. We can give any color any legal transparency value in addition to R, G, and B values.

This sketch draws sets of overlapping red, green, and blue circles to show transparency.

In the `draw()` function, the first three calls to `fill()` and `ellipse()` 1 draw the upper-left set of circles with a fill color transparency value of 20.

The second three calls 2 draw the upper-right set of circles with a transparency value of 100.

The third three calls 3 draw the lower-left set with a transparency value of 180.

The final three calls 4 draw the lower-right set of circles with a transparency value of 255, which means the color is completely opaque.

NOTE

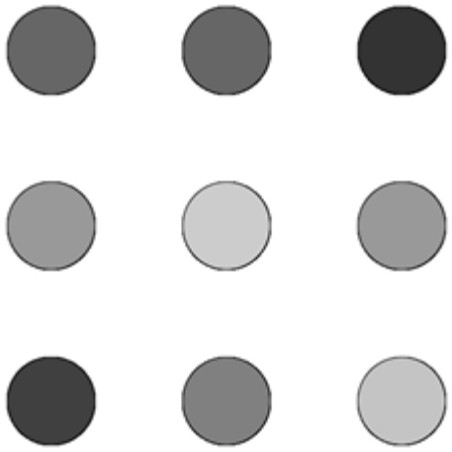
Processing defines a `color` type for specifying colors. It contains values for red, green, blue, and transparency. The function `color(r, g, b, a)` takes numerical values and returns a value of the `color` type. Variables can also hold values of the `color` type.

We could declare a variable and initialize it as follows:

```
color r;
r = color(255, 0, 0);   // The variable r now holds a
red color.
```

Example A

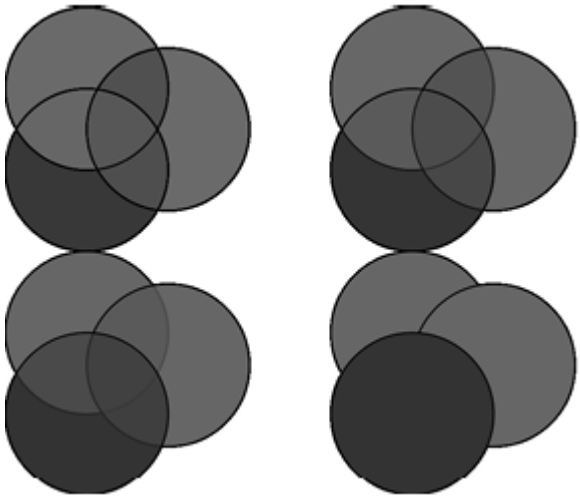
```
void setup ()
{
  size (400, 300);
  background (255);
}

void draw ()
{
1 fill (255, 0, 0); ellipse (100, 50, 50, 50);
2 fill (0, 255, 0); ellipse (200, 50, 50, 50);
3 fill (0, 0, 255); ellipse (300, 50, 50, 50);
4 fill (255, 255, 0); ellipse (100, 150, 50, 50);
5 fill (255, 0, 255); ellipse (200, 150, 50, 50);
6 fill (0, 255, 255); ellipse (300, 150, 50, 50);
7 fill (64, 64, 64); ellipse (100, 250, 50, 50);
  fill (128, 128, 128); ellipse (200, 250, 50, 50);
  fill (196, 196, 196); ellipse (300, 250, 50, 50);
}
```



Example B

```
void setup ()
{
  size (400, 300);
  background (255);
}
```

```
void draw ()
{
1 fill (255, 0, 0, 20); ellipse (50,  50, 100, 100);
   fill (0, 255, 0, 20); ellipse (100, 75, 100, 100);
   fill (0, 0, 255, 20); ellipse (50, 100, 100, 100);
2 fill (255, 0, 0, 100); ellipse (250, 50, 100, 100);
   fill (0, 255, 0, 100); ellipse (300, 75, 100, 100);
   fill (0, 0, 255, 100); ellipse (250, 100, 100, 100);
3 fill (255, 0, 0, 180); ellipse (50,  200, 100, 100);
   fill (0, 255, 0, 180); ellipse (100, 220, 100, 100);
   fill (0, 0, 255, 180); ellipse (50,  250, 100, 100);
4 fill (255, 0, 0, 255); ellipse (250, 200, 100, 100);
   fill (0, 255, 0, 255); ellipse (300, 220, 100, 100);
   fill (0, 0, 255, 255); ellipse (250, 250, 100, 100);
}
```



Sketch 3: if Statements—Changing Colors Conditionally

In daily life, people often deal with conditional actions, although little if any thought is given to the idea. We express the conditions in human language, of course:

“If it is raining, we’ll watch TV, but if it is sunny, we’ll go skiing.”

“If the light is red, then stop, but if it is green, just drive on through.”

We can also use conditional actions when programming a computer. If some situation is true, we execute a certain section of code. The condition or situation has to be expressed in numerical terms, and the result is a `true` or `false` result. Such conditions are frequently the result of comparisons between numbers, such as “is *i* equal to 10” or “is the x-coordinate less than the width.”

Conditional code is dealt with using an `if` statement, which has the following syntax:

```
if ( condition ) code ;
```

Conditions can be comparisons between numbers, so the following are all conditions:

```
(x > 2)       (P < q+1)      (width == 640)      (width != height)
```

There are some special symbols in use here. The = symbol means assignment, so to compare for equality a different symbol must be used: Processing uses ==. To compare for inequality, the symbol != is used, meaning “not equal.”

Example A

This sketch uses an `if` statement to increase an integer variable, `count`, every time `draw()` is called, and it changes the background color from red

to green when the count reaches 100 1.

Example B

The previous English condition examples illustrate a normal use of another idea: *otherwise*. One example was “If it is raining, we’ll watch TV, but if it is sunny, we’ll go skiing.” That example could also be phrased as “If it is raining, we’ll watch TV; otherwise we’ll go skiing,” meaning that if it’s not raining, we’ll go skiing. In most computer languages this is written as an `else` part to an `if` statement with the following syntax:

```
if ( condition ) code ;
else code;
```

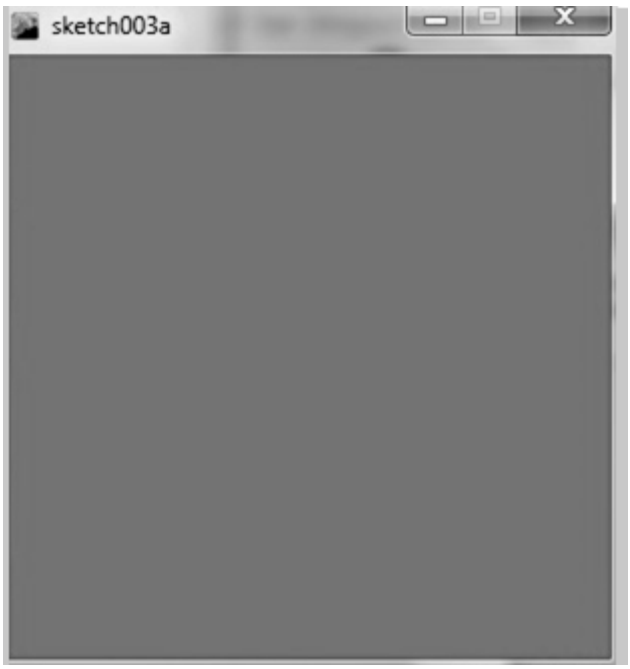
Example B uses an `else` to accomplish the same task as Example A 1.

Example C

The third code example alternates between red and green each time `draw()` is called, creating a colored flashing effect.

Example A

```
int count = 0;
void setup()
{
  size (300, 300);
}

void draw ()
{
  background (0, 255, 0);
1 if (count<100)
    background(255, 0, 0);
  count = count + 1;
}
```



Example B

```
int count = 0;
void setup()
{
  size (300, 300);
}

void draw ()
{
```

```
1 if (count<100)
    background(255, 0, 0);
  else background (0, 255, 0);
  count = count + 1;
}
```

Example C

```
int count = 0;
void setup()
{
  size (300, 300);
}

void draw ()
{
  if (count == 0)
  {
    background(255, 0, 0);   count = 1;
  } else
  {
    background(0, 255, 0);   count = 0;
  }
}
```

Sketch 4: Loops—Drawing 20 Circles

Programmers often need to execute the same code over and over again, sometimes with small variations. A program that draws 50 ellipses within the sketch window could be written using fifty calls to the `ellipse()` function, one for each ellipse drawn. Another way is to have one statement with a call to `ellipse()`, and execute it 50 times in a loop.

A *loop* in a program is a collection of statements that executes repeatedly from the first statement to the last, in the same order. You must specify the condition upon which the loop will exit. It's pretty common to know in advance how many times the loop should execute, as in the example of drawing 50 ellipses. Sometimes you won't know the number ahead of time, but you can calculate it, so the loop will execute *N* times, and *N* depends on some other thing. In either case, a counting loop is called a `for` loop in Processing because the reserved word `for` is used to begin the loop. For example,

```
for (i=0; i<10; i=i+1)  statementA ;
```

This loop executes 10 times: once when the variable `i=0`, again when `i=1`, again when `i=2`, and so on until `i=9`. When `i` is 10, the condition (`i<10`) becomes false and the loop ends. As a result, `statementA` executes 10 times, once for each value of `i` from 0 to 9.

The `for` loop has four parts:

`i=0` The *initialization* is executed the first time through the loop.

`i<10` The loop will continue to execute so long as the *continuation condition* is true.

`i=i+1` At the end of each iteration, after the statement is executed, the *increment* will be executed.

`statementA` This is the code that gets executed repeatedly.

If the expression is false at the beginning, the loop does not execute even once.

The statement that executes can be a *compound statement*, which is a collection of statements enclosed in braces. In fact, any time I refer to a *statement*, it can mean a compound statement.

Example A

A simple loop in Processing can draw 20 ellipses 1 starting at (20, 40) and ending at (210, 40). The ellipses are circles and are drawn next to each other. The `draw()` function exists but does not do anything 2.

Example B

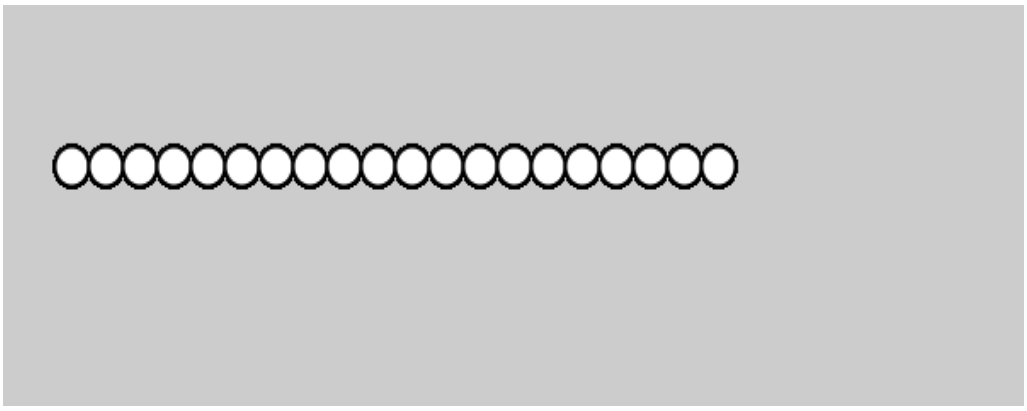
We can make the color change for each circle by using a compound statement. Let's change the green value by 10 each time a circle is drawn, starting with `green = 10` 1. Red and blue both stay at their maximum of 255. The code within the loop needs to set the fill color, draw the circle, and adjust the fill color for the next iteration 2.

The loop executes for 20 values of `i`: 0 to 19 inclusive. If we were to expand the code to show what was being executed, it would look like this:

```
(i=0);  fill (255, 10, 255);  ellipse (20, 40, 10, 10);  g =
20;
(i=1);  fill (255, 20, 255);  ellipse (30, 40, 10, 10);  g =
30;
(i=2);  fill (255, 30, 255);  ellipse (40, 40, 10, 10);  g =
40;
--snip--
(i=19);  fill (255, 190, 255); ellipse (210, 40, 10, 10); g =
200;
```

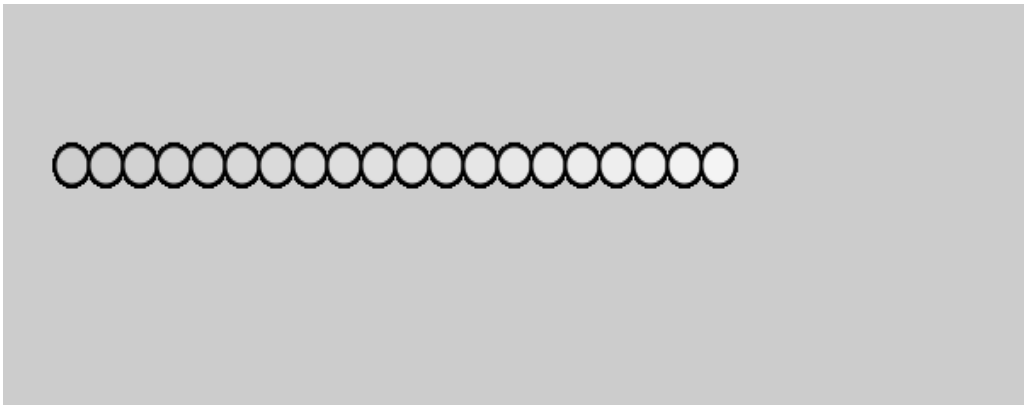
Example A

```
void setup ()
{
  size (500, 300);
1 for (int i=0; i<20; i++)
    ellipse (i*10+20,40,10,10);
}

2 void draw () { }
```



Example B

```
void setup ()
{
  size (500, 300);
1 int green = 10;
  for (int i=0; i<20; i++)
  {
    fill (255, green, 255);
    ellipse (i*10+20, 40, 10, 10);
 2 green = green+10;
  }
}
void draw (){ }
```



Sketch 5: Lines

Drawing lines is a basic thing to do in graphics. A *line* in Processing is really a *line segment*, and it is specified by identifying two endpoints that are to be connected by the line. The function that draws a line is named `line()`, and it takes the coordinates of the endpoints as parameters (for a total of four parameters). The call `line (10,10, 20,20)` will draw a line in the window between coordinates (10, 10) and (20, 20).

Example A

Let's draw some note paper. We can draw a horizontal line that runs the full width of the sketch window using this call, for some vertical position `y`:

```
line (0, y, width, y);
```

The width of the window is given by the variable `width`, and the height of the window is given by the variable `height`. The start of the line is `(0, y)` at the left of the image window `y` pixels down from the top; the end of the line is at `(width, y)` at the right of the window and the same *y* value.

The color for drawing lines can be specified using a call to `stroke()` with a color as the parameter. For example, `stroke (255,0,0)` 1 specifies that red lines will be drawn.

Example B

Processing will tell you where the mouse cursor is within the window using the built-in variables `mouseX` and `mouseY` 1. Whenever a mouse button is pressed, Processing calls a function called `mousePressed()`, if it exists. You have to write it if you want to use the mouse. When a mouse button is released, Processing calls the `mouseReleased()` function 2. You have to write that one too. The `mousePressed()` and `mouseReleased()` functions are referred to as *callbacks*, and they offer a very simple way to access button presses. Additionally, press and release amount to touches on a touch screen device, so the program will work on touch screen devices as well.

This example uses *clicks* (presses and releases) to draw lines. The first mouse click defines the starting point for the line (`x0`, `y0`) 3. The second click (when `x1 < 0`) 4 defines the endpoint of the line. A third click (when `x1 >= 0`) 5 clears the endpoints and starts again.

NOTE

The thickness of a line can be specified by calling the function `strokeWeight(n)`. The parameter `n` is the number of pixels thick the line will be.

The symbol `&&` means and*. It can be applied to any pair of variables or expressions that have the value `true` or `false`; in other words, things of Boolean type. It is used in `if` statements, and the effect is that the result is `true` if both sides of the expression are `true`. So the expression `(a && b)` is only true if both `a` and `b` are true. For example, this statement tests whether the value of `x` is horizontally within the window boundaries:*

```
if ( (x>=0) && (x<width) ) statementA;
```

That is the same as this:

```
if (x >= 0)
  if (x<width)
    statementA;
```

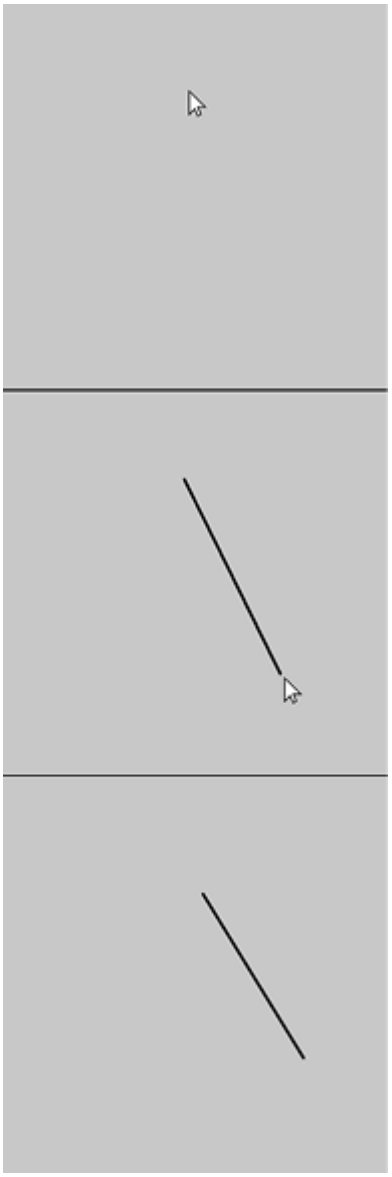
Example A

```
void setup ()
{
  size(500, 500);
}
void draw ()
{
  background (220, 220, 220);
1 stroke (255, 0, 0);    // Red margin line
  line (20, 0, 20, height);
  stroke (100, 0, 250);  // Blue horizontal lines
  for (int i=4; i<50; i++)
    line (0, i*10, width, i*10);
}
```

Example B

```
int x0=-1, y0=-1;
int x1=-1, y1=-1;

void setup ()
{
  size(300, 300);
}

void draw ()
{
  background (200, 200, 200);
```

```
  if (x1 >= 0)    line (x0, y0, x1, y1);
  else if (x0 >= 0)  line (x0, y0, 1mouseX, mouseY);
}

2 void mouseReleased ()
{
  if (x0 < 0)
    {
 3 x0 = mouseX;    y0 = mouseY;
    }
  else if 4(x1 < 0)
    {
      x1 = mouseX;    y1 = mouseY;
    }
5 else  {  x0 = y0 = -1;    x1 = y1 = -1;  }
}
```



Sketch 6: Arrays—Drawing Many Circles

A variable can hold a value, such as a single number. If we want more values, we can use more variables. For example, to draw two circles, we could have two sets of coordinate variables, say `x0`, `y0` and `x1`, `y1`, and we could draw the two circles with two calls:

```
ellipse (x0, y0, 10, 10);
ellipse (x1, y1, 10, 10);
```

But what if we wanted to draw a circle every time the mouse button was clicked, and to draw it where the cursor is on the screen? We don't know how many circles to draw in advance, so we don't know how many variables to declare. Instead, we can keep track of *x* and *y* using what Processing calls an *array*. An array is a collection of values all having the same type. The syntax for declaring an array is

```
int [] x = new int[100];
```

This declaration defines an array named `x` that can hold 100 integers. The phrase `int [] x` means "define a new array named x," and the phrase `new int[100]` defines the size, where 100 could be replaced by any constant. The preceding declaration could also be done in two parts:

```
int [] x;
x = new int[100];
```

You access the values in the array using an *index*, a number that specifies which of the values you want, starting from 0: `x[0]` is the first element (value) in the array with an index of `0`, `x[1]` is the second with an index of `1`, and so on to the last one, `x[99]`.

The example sketch uses two arrays, one for `x` and one for `y`, and it draws a circle at the coordinates where the mouse button is clicked (pressed and released). Initially each element in the `x` and `y` array is given the value −1 1 in `setup()`. This is called a *sentinel* value, and it indicates that there is no circle defined at that index. The `ncircles` variable indicates how many

circles have been defined, which is how many mouse clicks have been recorded; it starts at 0 and is incremented up to the maximum number of circles (`MAXCIRCLES`, a constant defined to be 100). When the mouse button is released, the system calls the `mouseReleased()` callback function 3, which saves the current value of the mouse coordinates in the arrays `x` and `y` at the current position (`ncircles`) and increases `ncircles` by 1. If `ncircles` becomes equal to `MAXCIRCLES`, it is reset to 0 4, which means that new circles will be saved over the earliest ones drawn. The old ones will, of course, be lost.

The `draw()` function first sets the background and then draws a circle at the mouse coordinates. Then all the elements of the `x` array are examined, and if the value of element `i` is greater than 0, a circle is drawn at `x[i], y[i]` 2 using this call:

```
ellipse (x[i], y[i], 18, 18);
```

The constant value `MAXCIRCLES` is defined using a special `final` property in its declaration:

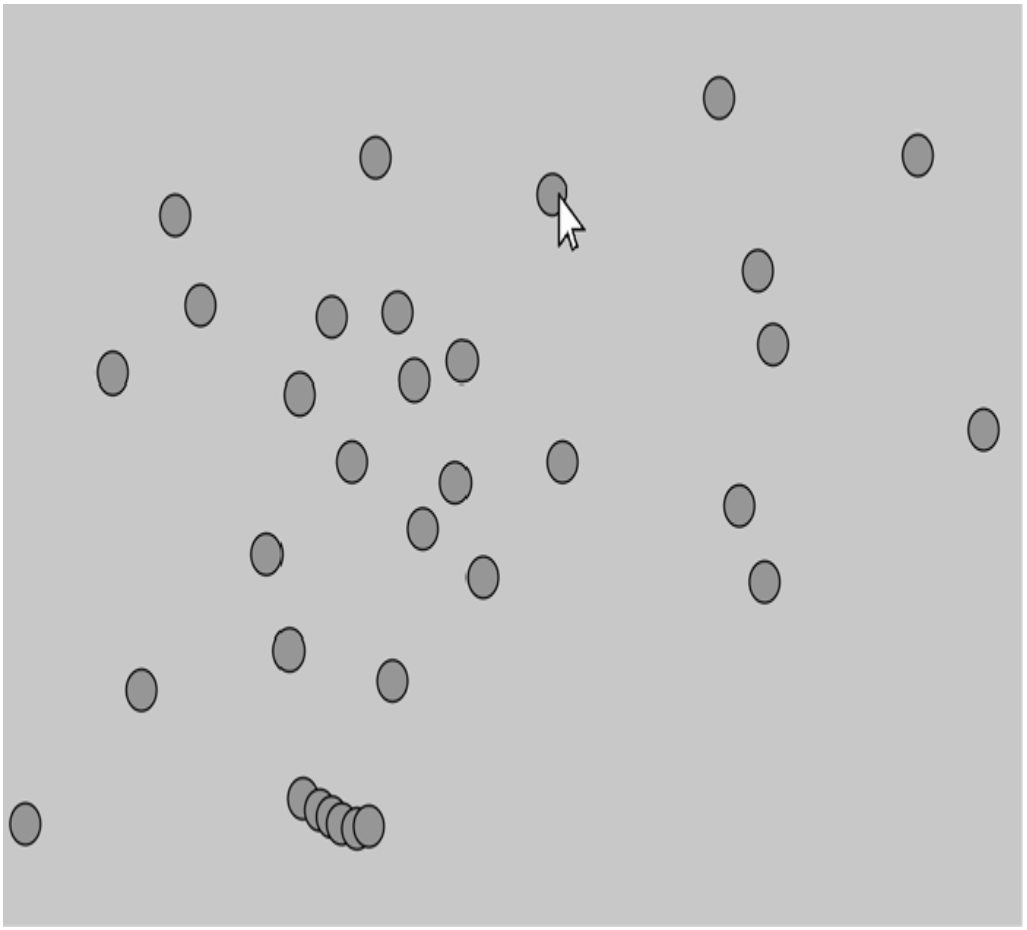
```
final int MAXCIRCLES = 100;
```

The value of `MAXCIRCLES` can't be changed anywhere in the program because it is `final`. It can (and should) be used to define the size of the two arrays:

```
int x[] = new int[MAXCIRCLES];
```

Defining the size of the arrays using a constant means that to increase the number of circles allowed, you only need to change the value of `MAXCIRCLES`.

```
final int MAXCIRCLES = 100;
int x[] = new int[MAXCIRCLES];
int y[] = new int[MAXCIRCLES];
int ncircles  = 0;

void setup ()
{
  size (600, 400);
  fill (200, 150, 100);
  for (int i=0; i<MAXCIRCLES; i++)
  1 x[i] = -1;
}

void draw ()
{
  background (200);
  ellipse (mouseX, mouseY, 20, 20);

  for (int i=0; i<MAXCIRCLES; i++)
    if (x[i] >= 0)
    2 ellipse (x[i], y[i], 18, 18);
    else break;
}

3 void mouseReleased ()
{
  x[ncircles] = mouseX;
  y[ncircles] = mouseY;
  ncircles = (ncircles+1);
  if (ncircles>=MAXCIRCLES)
  4 ncircles = 0;
}
```



Sketch 7: Lines with Rubber Banding

We are going to use the mouse to draw lines again. A line consists of a starting point and an endpoint, each having an x and a y component. We previously drew a line when the mouse was clicked on start and end points on the screen, but it only drew *one* line. What if we wanted to be able to draw many lines like this?

We can define a starting point when the mouse button is pressed 3 and the endpoint when the button is released 4, as we did before. But now we can store these points in arrays and draw them all during each screen update. The array `x0` saves the starting x-coordinate of a line, and `y0` has the corresponding y-coordinate. The arrays `x1` and `y1` will store the end coordinates. When the mouse button is pressed, we save the starting point (`x0[n]`, `y0[n]`), and when the mouse button is released, we save the endpoint as `x1[n]` and `y1[n]` and increment the value of `n`. This program will allow us to draw 256 lines because of the fixed size of the arrays.

When the starting point has been selected, we draw a line from that point to the current mouse coordinates to show how the line *would* look 2. This is called *rubber banding* because the line appears to stretch and contract as the mouse moves. When the mouse button is released, we finalize the end coordinates and draw the final line.

During each frame (the default is 30 frames per second) we draw all of the saved lines by calling `line (x0[i], y0[i], x1[i], y1[i])` for all `i` from `0` to `n-1` 1. We then draw the rubber band line if the mouse button is currently depressed (when `down` is set to `true`). Setting `down` to `true` happens in `mousePressed`, and it is set to `false` when the button is released, within `mouseReleased`. If `down` is `true`, a line is drawn from the last selected point to the mouse coordinates:

```
line (x0[n], y0[n], mouseX, mouseY);
```

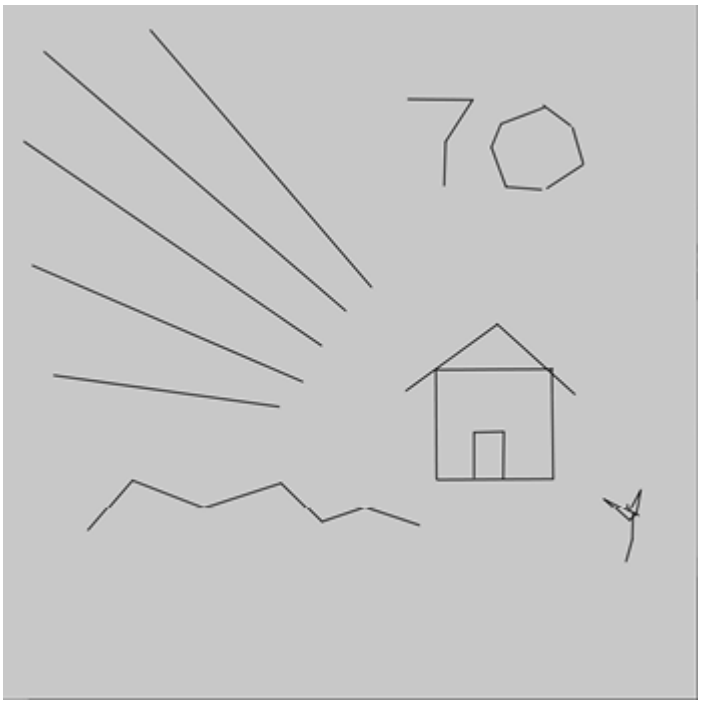
This implements the rubber banding.

As a new idea, the sketch implements an *erase* feature. If the user types the BACKSPACE key, the most recent line is deleted. When the system

detects a key press, Processing calls a user-defined function named `keyPressed()` 5. A variable named `key` provides the value of the key that was pressed, so inside `keyPressed()` we check if the key is BACKSPACE, and if so we decrease the value of `n` (the number of lines so far) by 1. As a result, the last line will not be drawn, and the next line will be saved over the erased line in the coordinate arrays.

```
int N = 10;
int x0[] = new int[N];
int y0[] = new int[N];
int x1[] = new int[N];
int y1[] = new int[N];
int n = 0;
boolean down = false; // A boolean variable can only be
true or false

void setup ()
{
  size(500, 500);
}

void draw ()
{
  background (200, 200, 200);
1 for (int i=0; i<n; i++)
   {
     line (x0[i], y0[i],
        x1[i], y1[i]);
   }
2 if (down) line (x0[n], y0[n], mouseX, mouseY);
}

void mousePressed()
{
  down = true;
  if (n<N)
  {
  3 x0[n] = mouseX;
    y0[n] = mouseY;
  }
}

void mouseReleased ()
{
  if (n<N-1)
  {
 4 x1[n] = mouseX;
    y1[n] = mouseY;
    n = n + 1;
  }
  down = false;
}
```

5
```
void keyPressed ()
  {
    if (key==BACKSPACE && n>0)
       n = n - 1;
  }
```



Sketch 8: Random Circles

This sketch draws circles at random places on the screen, with random colors. Randomness refers to unpredictability, and it is a complex concept. If you try to draw straight lines with a pencil, it is impossible that any two of them will be identical. There are variations that creep in and cause minor changes in each line. The same is true of brush strokes when painting. No two human activities will be exactly the same, and the differences will be unpredictable but apparent.

When using a computer, a *random number generator* creates numbers that are random with respect to each other. Random numbers can be used to simulate random events in games like dice or poker, to do things that a user would find unpredictable, or to simulate complex real-world situations. For example, things like the spacing between cars on a road and the appearance of raindrops on a window appear random because we do not understand all of the complex factors that went into the situation.

The random number generator in Processing is named `random`. The call `random (100)` will generate a real number between 0 and 100, not including 100. The call `random (10, 20)` will return a real number between 10 and 20, but less than 20. The call `random (0, width)` generates a random *x* position within the sketch window, and `random (0, height)` generates a random *y* position.

Like Sketch 6, this sketch stores coordinates in arrays and uses them to draw circles with calls to `ellipse()`, but instead of drawing circles when the mouse is clicked, a new circle is created automatically every second. To do this, we set the rate at which `draw()` is called (the *frame rate*) to 1 using the call `frameRate(1)` 1 in `setup()`. Each time `draw()` is called, we generate a new x- and y-coordinate using `random()` and save it in the `x` and `y` arrays 2:

```
x[ncircles] = (int)random(0, width);
y[ncircles] = (int)random(0, height);
```

The `(int)` in front of the calls to random converts the result, a `float`, into a new type, `int`. This is called a *cast*, and we are changing the *floating-point* value into an integer because we can't use values with decimal points as coordinates. This can also be done using a call to the function `int()`:

```
x[ncircles] = int(random(0, width));
y[ncircles] = int(random(0, height));
```

NOTE

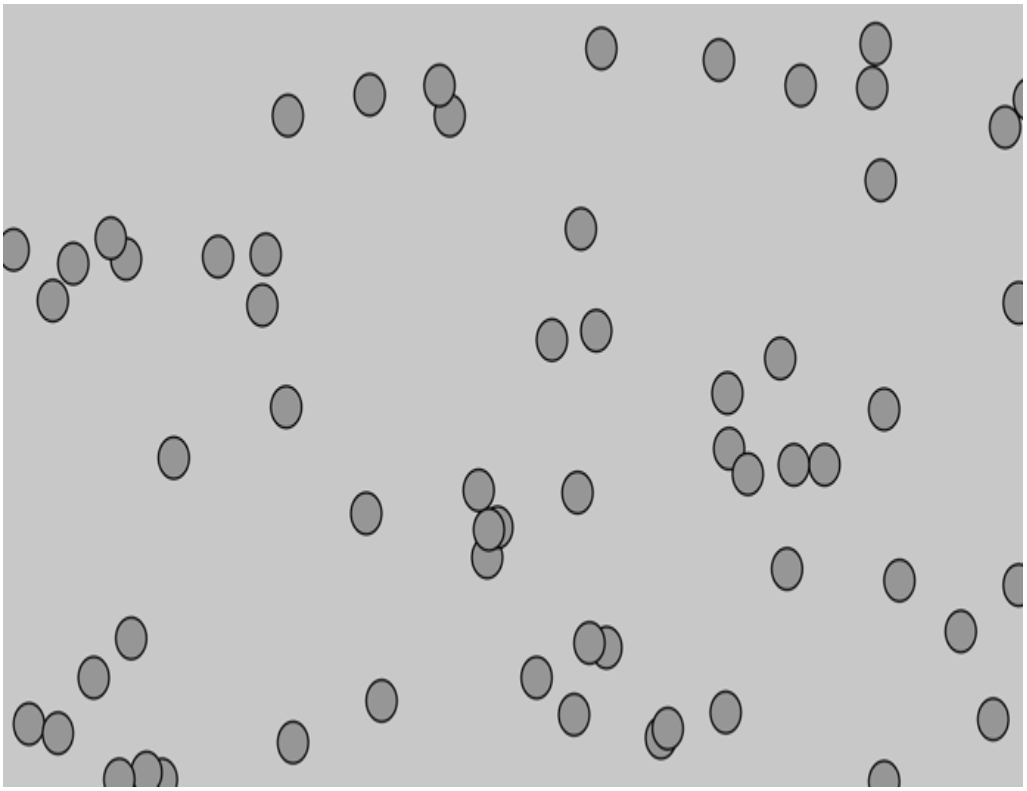
The random number generator can be used to generate random die rolls using this call:

```
roll = int(random(1, 7));
```

This call can never return the value 7, and it truncates the float values between 1 and 2 to be 1, 2 and 3 to be 2, and so on. A random coin toss (1=heads, 2=tails) would be generated as follows:

```
toss = int(random(1, 2) + 0.5);
```

```
final int MAXCIRCLES = 256;
int x[] = new int[MAXCIRCLES];
int y[] = new int[MAXCIRCLES];
int ncircles  = 0;

void setup ()
{
  size (600, 400);
  fill (200, 150, 100);
  for (int i=0; i<MAXCIRCLES; i++)
    x[i] = -1;
1 frameRate(1);
}

void draw ()
{
  background (200);

  for (int i=0; i<MAXCIRCLES; i++)
    if (x[i] >= 0)
      ellipse (x[i], y[i], 18, 18);
    else break;

2 x[ncircles] = int(random(0, width));
  y[ncircles] = int(random(0, height));
  ncircles = (ncircles+1);
  if (ncircles>=MAXCIRCLES)
    ncircles = 0;
}
```



Sketch 9: A Rectangle

You could draw a rectangle by drawing four lines that represent the edges, but the Processing system would not consider this to be a rectangle; it has no way to know that the four lines are a single object. Instead, Processing has a function for drawing rectangles, called `rect()`. Rectangles will be filled using the current fill color, just as circles were.

The default way to specify a rectangle is `CORNER` mode, where the first two parameters you supply are the coordinates of the upper-left corner of the rectangle, followed by the width and the height, in pixels. If you specify `CENTER` mode, the first two parameters are the coordinates of the center of the rectangle. `CORNERS` mode specifies the coordinates of the first corner, then the coordinates of the diagonally opposite corner. You can change the mode using one of the following calls:

```
rectMode(CORNER);
rectMode(CENTER);
rectMode(CORNERS);
```

In this sketch we'll use `CORNERS` mode 1, as specified in the `setup` function, and fill the rectangle with a shade of purple: `(200, 0, 160)`. As in the previous sketches, the `mousePressed()` function sets a Boolean `flag` variable to `true` when the mouse button is pressed 3, and `mouseReleased()` clears the variable (sets it to `false`) 5.

The global variables `x` and `y` represent the first corner of the rectangle and are initialized to −1. When the mouse button is pressed, we set `x` and `y` to the current value of `mouseX` and `mouseY` 4, and the `flag` variable is set to indicate that `x` has been set. Then the `draw()` function will draw a rectangle with `(x, y)` as one corner and the current mouse position `(mouseX, mouseY)` as the other 2. This implements the rubber band effect.

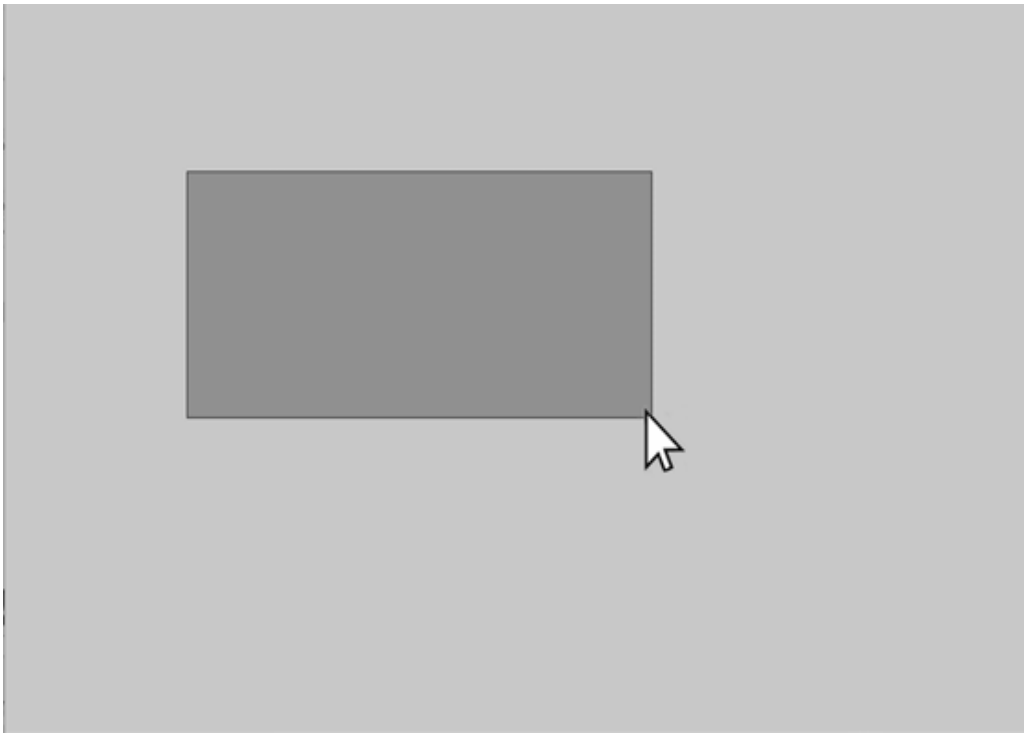
Global variables `x1` and `y1` are the coordinates of the second corner of the rectangle. When the mouse is released, we see the values of `x1` and `y1` to the current mouse coordinates 6, and this completes the rectangle. The `draw()`

function will draw the rectangle with the value of `x1` and `y1` as the opposite corner because `flag` is now false.

NOTE

Ellipses have drawing modes too, as set by the function `ellipseMode()`. `ellipseMode(CENTER)` *is the default mode, and the one with which you are familiar.* `ellipseMode(CORNER)` *treats the first two parameters of ellipse as the coordinates of the upper-left corner and the remaining two as the width and height.* `ellipseMode(CORNERS)` *treats the first two parameters as the coordinates of one corner of a box enclosing the ellipse and the second two parameters as the coordinates of the second corner.*

```
int x = -1, y = -1;
int x1 = -1, y1 = -1;
boolean flag = false;

void setup ()
{
  size (600, 400);
  fill (200, 0, 160);
1 rectMode (CORNERS);
}

void draw ()
{
  background (200);
  if (flag)
  2 rect (x, y, mouseX, mouseY);
  else
    rect (x, y, x1, y1);
}

void mousePressed ()
{
3 flag = true;
4 x = mouseX; y = mouseY;
}

void mouseReleased ()
{
5 flag = false;
6 x1 = mouseX; y1 = mouseY;
}
```



Sketch 10: Triangles and Motion

Just as rectangles are drawn using the built-in `rect()` function, triangles are drawn using the built-in `triangle()` function. Triangles can't be drawn using a height and width; their shape is determined by their three angles. As a result, the `triangle()` function has six arguments: the x, y coordinates of the three *vertices* (corners).

This sketch draws triangles using the mouse. Like the previous sketches that draw rectangles and lines, this sketch uses `mouseReleased()` 3 to determine when a point has been selected. After three clicks, a triangle will be drawn using the three selected points as the vertices.

After the triangle is drawn, it begins to move downward, as if it had been pushed slightly. It continues to move downward until it hits the bottom border of the sketch window, where it disappears.

We accomplish the motion by adding a small value 1 (`delta` = 1) to the y-coordinates of the triangle after each time it is drawn. This draws the triangle at successively lower locations in the window until it appears to pass beyond the bottom edge of the window. In fact, the triangle still exists to the Processing system, and its coordinates continue to update even though it can't be seen.

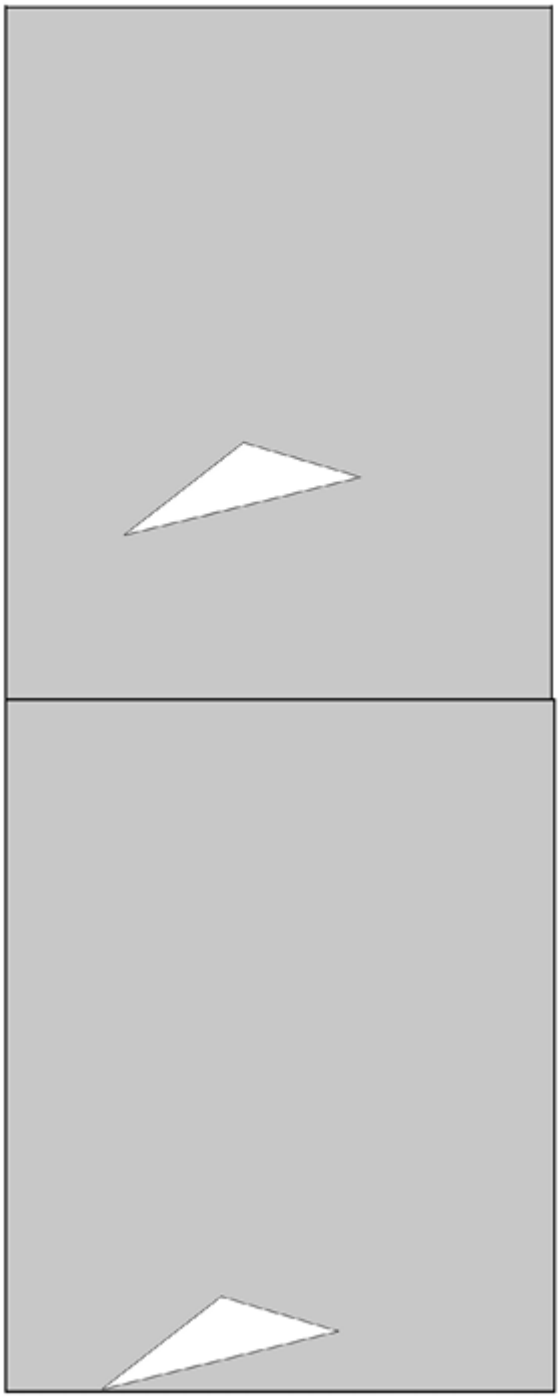
If the user of this program clicks the mouse after the triangle is drawn, the triangle disappears and the drawing process begins again. We restart the drawing process by re-initializing all of the vertices to −1 4, which indicates that they have not been defined yet.

The following line of code is commented out inside of `draw()`:

```
2 // delta = delta + 1;
```

If you remove the `//` at the beginning of the line, the line will execute and the triangle will fall faster and faster, as if being pulled by a force (for example, gravity). Remove the // from the line near the end of `mouseReleased()` as well and the initial speed will reset to 1 with each new triangle drawn.

```
int x0=-1, y0=-1;
int x1=-1, y1=-1;
int x2=-1, y2=-1;
int delta = 1;

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

void draw ()
{
  background (200);

  if (x2 >= 0)
  {
    triangle (x0, y0, x1, y1, x2, y2);
1 y0 = y0 + delta; y1 = y1 + delta; y2 = y2 + delta;
2 // delta = delta + 1;
  }
  else if (x1 >= 0)
  {
    line (x0, y0, x1, y1);
    line (x1, y1, mouseX, mouseY);
  } else if (x0 > 0)  line (x0, y0, mouseX, mouseY);
}

3 void mouseReleased ()
  {
    if (x0 < 0)  {  x0 = mouseX; y0 = mouseY; }
    else if (x1 < 0) {  x1 = mouseX; y1 = mouseY; }
    else if (x2 < 0)
    {
      x2 = mouseX; y2 = mouseY;
    } else
    {
4 x0 = y0 = -1;
      x1 = y1 = -1;
      x2 = y2 = -1;
      // delta = 1;
    }
  }
```



Sketch 11: Displaying Text

Text is essential in nearly all practical computational programs and in many generative art and net art programs as well. Text is a primary way that humans communicate, and while we say that "a picture is worth a thousand words," it is frequently true that a few carefully chosen words can make an otherwise incomprehensible image into a valuable communications tool. Think of the labels along the axis of a graph, for instance.

We draw text in the sketch window in the same way that we draw lines and ellipses, using a simple function. The first thing you need to know is that text is drawn starting at a particular (*x*, *y*) location, where *x* and *y* represent the coordinates of the lower-left corner of the box that encloses the text *when not considering descenders*. Characters such as *y* and *j* extend below this box, and they so have *y* values greater than the value specified.

We will draw text using a call to the function `text()` 2:

```
text ("This is a string to be drawn", 100, 20);
```

In this case, the coordinates of the lower left of the string are `(100, 20)`. The initial font and size are defaults, and these defaults are system dependent. Size is easy to specify using the `textSize(n)` function 1, passing the desired size of the characters *in pixels* (not points). The color used to draw the text is the current fill color, not the stroke color.

The alignment of the text can be specified using calls to the `textAlign()` function 3. Horizontal alignment can be `LEFT`, `CENTER`, or `RIGHT` with respect to the x- and y-coordinates specified in the `text()` function call; the default is `LEFT`. Vertical alignment can be `TOP`, `CENTER`, `BOTTOM`, or `BASELINE` with respect to the x- and y-coordinates specified in the `text()` function call; the default is `BASELINE`. `BOTTOM` is the line that defines the lowest *y* value for any character, such as the bottom of a descender. `BASELINE` defines the lowest point of a typical character with no descender. So, the call

```
textAlign (CENTER, BOTTOM);
```

will center the current text from left to right (the specified *x* value is the center of the string) and aligned so that the specified *y* value is the bottom of the string.

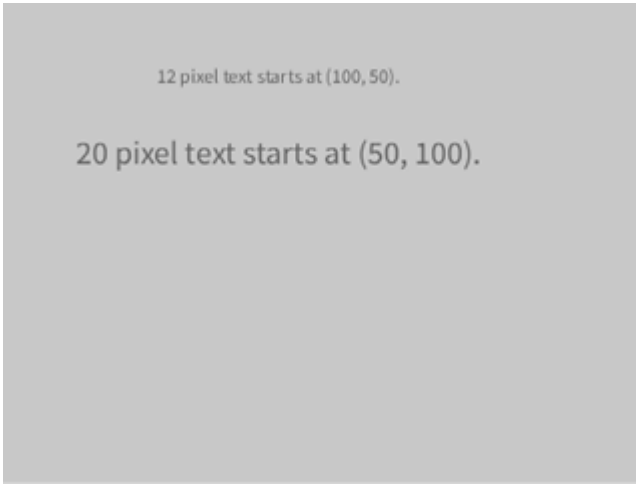
Example A illustrates how to display text in two different sizes. Example B shows a line drawn horizontally on each x-coordinate and vertically down each y-coordinate specified in the `text()` call. It shows the alignment of the text with respect to the specified coordinates.

NOTE

The names `CENTER`, `LEFT`, `RIGHT`, and so on are just constants declared by the Processing system. Their specific values are not important, but the system must know what that value is so that the system can make the proper alignments.

Example A

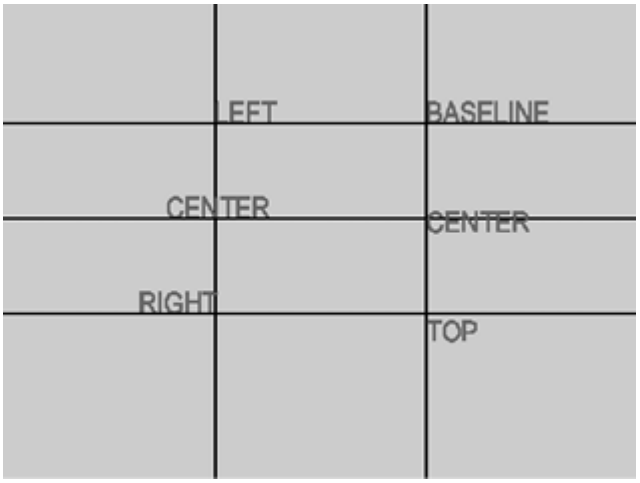
```
void setup ()
{
  size (400, 300);
  fill (255, 0, 0);
}

void draw ()
{
  background(200);
1 textSize(12);
2 text ("12 pixel text starts at (100, 50).", 100, 50);
  textSize (20);
  text ("20 pixel text starts at (50, 100).", 50, 100);
}
```



Example B

```
void setup ()
{
  size (300, 200);
  fill (255, 0, 0);
}

void draw ()
{
  line (100, 0, 100, height);
3 textAlign(LEFT); text ("LEFT", 100, 50);
  line (0, 50, width, 50);
```

```
  textAlign(CENTER); text ("CENTER", 100, 90);
  line (0, 90, width, 90);
  textAlign(RIGHT);  text ("RIGHT", 100, 130);
  line (0, 130, width, 130);
  textAlign (LEFT, BASELINE); text ("BASELINE", 200,
50);
  line (200, 0, 200, height);
  textAlign (LEFT, CENTER); text ("CENTER", 200, 90);
  textAlign (LEFT, TOP); text ("TOP", 200, 130);
}
```



Sketch 12: Manipulating Text Strings

The previous sketch is really an introduction to character strings, which are a natural way for a human to communicate with a computer. A *string* is a sequence of characters; so is a word, a sentence, or a paragraph. At a high level, a string consists of a collection of characters in a specific order. There is a first character, a second, and so on until the final one is reached. The number of characters in this sequence is the *length* of the string.

String constants are character sequences enclosed in double quotes like this: `"To be or not to be"`. We can use string constants to declare variables that are `String`s and assign values to them. At 1, for example, we declare two string variables and assign string constants to them:

```
String s1 = "To be or not to be"
String s2 = "that is the question."
```

Strings can be constructed by sticking other strings together. The `+` operator, when applied to strings, means *concatenate* or append, so the quote can be completed by concatenating these two strings:

```
s1 = s1 + s2;
```

This makes `s1` become `"To be or not to bethat is the question."` Unfortunately, this is not quite right, because we need a comma and a space between the two strings. This would be better:

```
2 s1 = s1 + ", " + s2;
```

The first character, “T”, in this new string has an *index* of 0, meaning it is in the 0 position in the string. The character “o” is in position 1, and so on.

A *substring* is a sequence of characters within the string specified by indices. The substring of `s1` from index 6 to 11 is the string `"or not"`, and it is found in Processing as follows:

```
s1.substring (6,12)
```

The length of this string is six characters, and that length is returned by the function `length()`:

```
s1.substring(6,12).length()
```

The character at a specific location can be found with the `charAt()` function. For example, `s1.charAt(3)` is “b” and `s1.charAt(18)` is “,”.

Strings cannot be compared using the standard relational operators (because they are really *class instances*, which will be discussed later). Instead, there are functions for comparison. Comparing `s1` and `s2` could be accomplished like this:

```
if (s1.equals(s2)) ...;
```

This sketch shows some string operations and their results, drawn using the `text()` function discussed in Sketch 11. The sketch includes examples of `length()` 3, `charAt()` 4, and `substring()` 5.

```
1 String s1 = "To be or not to be";
  String s2 = "that is the question.";

  void setup ()
  {
    size (500, 400); fill(0);
    text ("s1 = '"+s1+"'", 10, 20);
    text ("s2 = '"+s2+"'", 10, 35);
    text ("s1+s2 is '"+s1+s2+"'", 160, 35);
    text ("s1+\", \"+s2 is '"+ s1+", "+s2+"'", 160, 50);
    text ("Let s1 = \"To be or not to be, that is the
  question.\"", 10, 75);
2 s1 = s1 + ", " + s2;
3 text ("Then s1.length() is "+s1.length(), 25, 90);
4 text ("s1.charAt(0) is '"+s1.charAt(0)+"'", 160, 90);
    text ("s1.charAt(6) is '"+s1.charAt(6)+"'", 160,
  105);
    text ("s1.charAt(13) is '"+s1.charAt(13)+"'", 160,
  120);
    text ("s1.charAt(41) is '"+s1.charAt(40)+"'", 160,
  135);
    text ("The length() function returns the number of
  characters in the string.", 10, 150);
    text ("The index of the final character is
  length()-1. It's an error to index past the end.", 10,
  165);
    text ("Putting a \" into a string  is done by using a
  backslash: \\\" does it.", 10, 180);

5 text ("s1.substring (0, 10) is
  \""+s1.substring(0,10)+"\"", 15, 200);
    text ("s1.substring (10, 20) is
  \""+s1.substring(10,20)+"\"", 15, 215);
    text ("s1.substring (12) is
  \""+s1.substring(12)+"\"", 15, 230);
    text ("s1.substring (20, s1.length()-1) is
  \""+s1.substring(20,s1.length()-1)+"\"", 15, 245);
    noLoop();
  }
  void draw () {    }
```

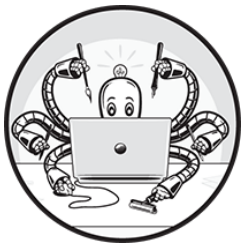
s1 = 'To be or not to be'
s2 = 'that is the question.' s1+s2 is 'To be or not to bethat is the question.'
s1+", "+s2 is 'To be or not to be, that is the question.'

Let s1 = "To be or not to be, that is the question."
Then s1.length() is 41 s1.charAt(0) is 'T'
s1.charAt(6) is 'o'
s1.charAt(13) is 't'
s1.charAt(41) is '.'

The length() function returns the number of characters in the string.
The index of the final character is length()-1. It's an error to index past the end.
Putting a " into a string is done by using a backslash: \" does it.

s1.substring (0, 10) is "To be or n"
s1.substring (10, 20) is "ot to be, "
s1.substring (12) is " to be, that is the question."
s1.substring (20, s1.length()-1) is "that is the question"

2
WORKING WITH PREEXISTING IMAGES



Sketch 13: Loading and Displaying an Image

Images are everywhere in cyberspace, and even people without explicit computer skills know the names of the image formats, or at least the file suffixes: GIF, JPEG, BMP, PNG, and so on. Each of these sequences of letters is symbolic of a different way of storing images in computer form, and each has specific pros and cons for different purposes. GIF images were developed for use on the early internet, and they can have transparent color as well as the ability to store animations. The JPEG (or JPG) format is used by almost all digital cameras, and it compresses pictures into relatively few bytes.

You should recognize the importance of images and how complex a task it is to read data from one of these file formats. A program to read most GIF files would require more than a thousand lines of code. The fact that Processing provides an easy-to-use facility to read, display, and write images is one of its many advantages over other programming languages.

Processing has a type that represents an image, much as an integer is represented by the `int` type, and the system can read an image file into a variable with one call to a function. The type is `PImage` (short for *Processing Image*), and the function is `loadImage()`. For the image to load, it should be saved in the same folder as the sketch file, or in a subfolder called *data*.

Example A

Let's assume that an image file named *image.jpg* exists and that we want to read this image and display it in the sketch window. The first thing to do is declare a `PImage` variable, `im`, into which we'll place the image 1. Inside of `setup()`, we will create a sketch window (using `size()`) and read the image. The following statement reads the image and assigns it to the variable `im`:

```
2 im = loadImage ("image.jpg");
```

Now the image data is stored in some internal form in the variable `im`.

Displaying the image is done from the `draw()` function, although it could be done in this instance from `setup()` as well. The Processing system gives

us a function named `image()` that will draw a `PImage` into the sketch window at a particular (*x*, *y*) location (specifying the location of the upper-left corner of the image). The following call draws the image so that its upper-left corner corresponds to the window's upper-left corner:

```
3 image (im, 0, 0);
```

Example B

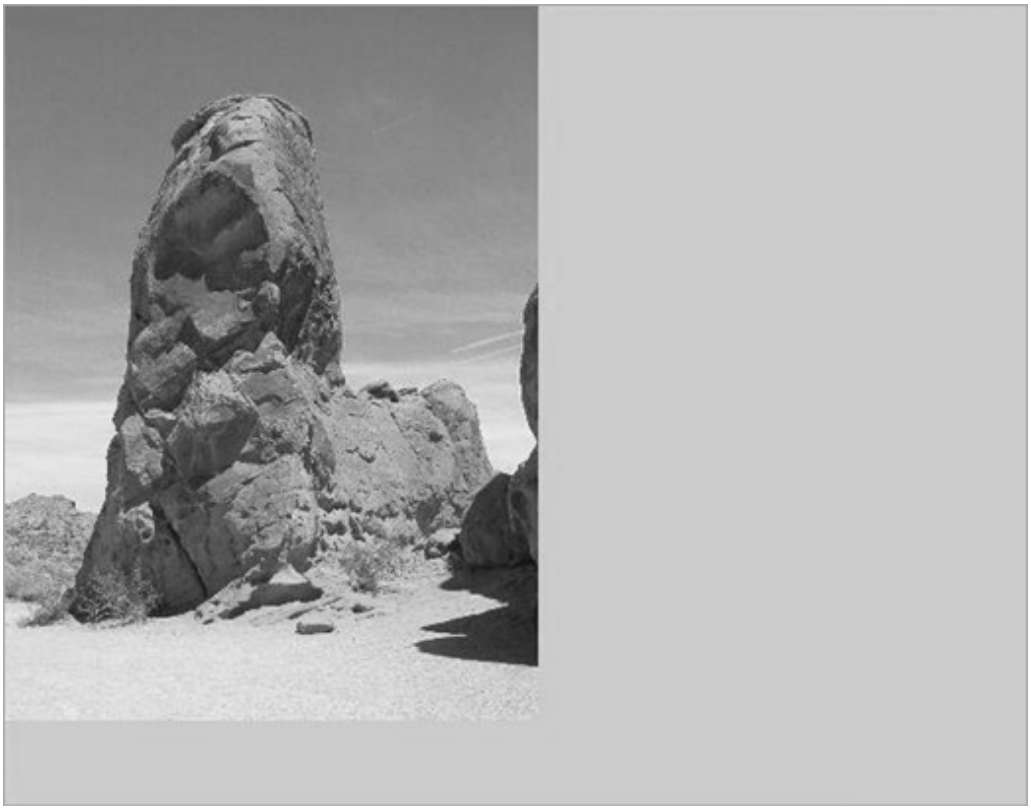
This program is the same as Example A, but it draws the image at location `(150, 30)` 1. Now the image is more neatly displayed in the available space.

NOTE

The type `PImage` is defined by Processing but is a little different in detail from types like `int` and `float`. In fact, `PImage` is something called a class, and we can make our own classes.

Example A

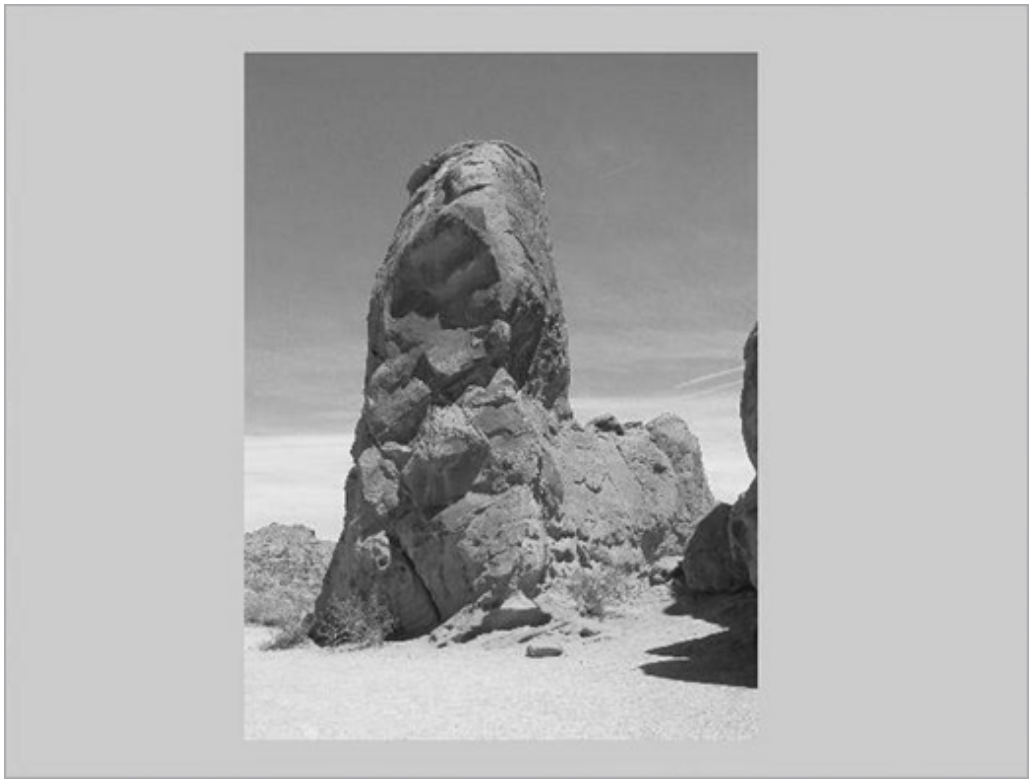
```
1 PImage im;

  void setup ()
  {
    size (640, 480);
2 im = loadImage ("image.jpg");
  }

  void draw ()
  {
3 image (im, 0, 0);
  }
```



Example B

```
PImage im;

void setup ()
{
  size (640, 480);
```

```
  im = loadImage ("image.jpg");
}

void draw ()
{
1 image (im, 150, 30);
}
```



Sketch 14: Images—Theory and Practice

Images are used often in the visual arts, and Processing was designed for artists, so it's no surprise that images are pretty easy to use in the program. There are some basic things you need to know, though.

One is that for an image to be used on a computer, it must be *digitized*; that is, it must be turned into numbers. If an image was not created by a computer in the first place, then it has to be scanned or photographed, and each location on the original image must be given a number indicating the color seen there. The result is a two-dimensional array of numbers, each of which represents the color at a specific location. Each small area of an image is considered to be uniform in color, even if it's not, so the most prominent color is selected to represent the entire area. This color is stored at the corresponding (*x*, *y*) location in the internal representation, and it's called a *picture element*, or *pixel* for short. The complete collection of these pixels is an approximation of the original image. *Drawing* an image on screen means setting the pixels on a portion of the computer screen to match those in the image. This is what the `image()` function does in Processing.

Images are most often thought of as being $N \times M$ pixels in size, where N is the number of rows and M is the number of columns. The total number of pixels in such an image is $N \times M$.

The `PImage` data type offers programmers a variety of ways to access the pixels in an image and manipulate them. Properties of an image can be accessed using "." (dot) notation. For a `PImage` variable named `myImage`, for example, we have the following properties:

```
myImage.width   // Width of the image, in pixels
myImage.height  // Height of the image, in pixels
```

We often wish to create a graphics area that is the same size as a particular image, but the `size()` function in `setup()` can only use constants to set the window size. To get around this, we can add `surface.setResizable(true)` to `setup()`. It lets us resize the graphics area while the sketch is running with a call to `surface.setSize()`, which can use non-constants like `myImage.width`.

Individual pixel values can be accessed using functions that return or set colors based on (*x, y*) coordinates:

```
myImage.get(x,y);                    // Returns the color of the
pixel at column x and row y
myImage.set(x,y, color(255,0,0))     // Sets the pixel at (x,y)
to red
```

If we simply call `get()` or `set()` with no image specified, Processing assumes that the image being referenced is whatever is being displayed in the sketch window.

Example A

This sketch reads an image file and checks to see if it was read in successfully 1; if not, the program is ended by calling the function named `exit()`. The `loadImage()` function returns a special value named `null` if the image could not be read, so that can be used as an indicator that the image file, for example, was not found. If the image is okay, the program sets the size of the sketch window to be the same size as that image, using the image's `width` and `height` properties 2. When the `setup()` function displays the image, it will fill the entire window.

Example B

The second sketch does not call `exit()` if the image file can't be opened. Instead, it displays an error message in the graphics window 1.

Example A

```
PImage img;                   // The image to be loaded
String name = "image.jpg";    // Name of the image file

void setup ()
{
  size (100,100);
  surface.setResizable(true);
  img = loadImage ("image.jpg");
1 if (img == null)
  {
    println ("File image.jpg is missing.");
    exit();
  }
2 surface.setSize(img.width, img.height);
}


void draw()
{
  image (img, 0, 0);  // Display the image
}
```

Example B

```
// Display error message if the image is not read in.

PImage img;                          // The image to be
loaded

void setup ()
{
  fill(0);
  size (400,200);
  surface.setResizable(true);
  img = loadImage ("image.jpg"); // Load the original
image
  if (img == null)
  {
    background(255,0,0);
1 text ("Error: Image file not found", 100, 100);
  }
  else surface.setSize(img.width, img.height);
}

void draw()
{
  if (img !=null) image (img, 0, 0); // Display the
image
}
```

Sketch 15: Manipulating Images I—Aspect Ratio

In the previous sketch we used the size of an image to define the size of the sketch window. It's also possible to change the size of an image so that it fits into an existing space. The `resize()` function, part of the `PImage` data type, can be used to specify a new size for an image. It does not make a copy but resizes the `PImage` itself. Here's an example call to this function:

```
1 img.resize (w, h);
```

This call will cause the image stored in the `img` variable to be expanded or contracted to be `w` pixels wide by `h` pixels high.

Example A

In the first example, we scale the image to be the size of the window, which is 240×480. Note that the image has been distorted, squashed from the sides and made taller. Also note that all of the work is done in `setup()`, and `draw()` has no code.

Any image has an *aspect ratio*, which is the relationship between the width and height of the image. It is often expressed as *w*:*h*. For example, 16:9 would be the aspect ratio of an image that had 16 pixels in the x-direction (width) for every 9 pixels in the y-direction (height). The aspect ratio is sometimes expressed as a fraction, dividing the height into the width, so the ratio of 16:9 would be written as 1.8 in this way. The reason that the image in Example A looks odd is that the aspect ratio has been changed by forcing the image to fit into an arbitrary rectangle.

Example B

This sketch draws an image into a window, scaling it so that the aspect ratio remains intact. The first thing to be done is to compute the aspect ratio of the original image:

```
1 aspect = (float)w/(float)h;
```

We use `float` variables here because the aspect ratio will be a fraction. When we place an image into a fixed space, its largest dimension (height or width) determines the overall size of the image within the window. We'll adjust the largest side of the image to exactly fit the corresponding side of the window 2, whether that means making the image larger or smaller. The other dimension of the image is kept proportional to this new scaled value. So if the image is taller than it is wide, we will map the height of the image to the height of the window:

```
h = height;
```

and the width will be in proportion to the original aspect ratio (converted to an integer):

```
w = (int)(h*aspect);
```

Now the image can be resized for display:

```
img.resize (w, h);
```

Example A

```
PImage img;

void setup ()
{
  img = loadImage ("image.jpg");
  size (240, 480);     // Fixed size window
1 img.resize (240, 480);
  image (img, 0, 0);
}

void draw ()
{ }
```

Example B

```
PImage img;
int w, h;
float aspect = 1.0;

void setup ()
{
  img = loadImage ("image.jpg");
  size (540, 480);
  w = img.width; h = img.height;
1 aspect = (float)w/(float)h;

2 if (w > h)
  {
    w = width;
    h = (int)((float)w/aspect);
  } else
  {
    h = height;
    w = (int)(h*aspect);
  }
  img.resize (w, h);
  image (img, (width - w)/2, (height-h)/2);
}

void draw () { }
```

Sketch 16: Manipulating Images II—Cropping

Cropping an image refers to the removal of some outer parts. You could think of it more generally as the selection of an arbitrary rectangular sub-image. We crop images to make a more appealing image or to remove extraneous material. In Paint or Photoshop we use the mouse, clicking first on the desired upper-left corner of the cropped image, then dragging the mouse to the desired new lower-right corner, and releasing the button. All parts of the image outside of the selected rectangle will be discarded. This sketch will crop an image and optionally expand the cropped region to fill the entire image window.

First the image is read in and the sketch window is resized to fit the image. The `draw()` function displays the image (named `img`) centered in the window using the following code 2:

```
image (img, (width-img.width)/2, (height-img.height)/2);
```

If the image has not been cropped, `width-img.width` will be 0, and the call will be `image(img, 0, 0)`. Otherwise the image will be smaller than the window, and (`width-img.width)/2` will be the number of pixels needed on the left to center the cropped image. The same is done for height, which places the image in the center of the window.

When the mouse button is pressed (`mousePressed()`), the cropping process starts, using the point where the cursor is, which is saved as `x0` and `y0`. Then a rectangle is drawn from this location to the current mouse coordinates, implementing a rubber band rectangle 3.

When the mouse button is released, the mouse coordinates are evaluated to ensure that the current `mouseX` and `mouseY` represent the lower-right corner of the crop box; in other words, make sure that `mouseX` is bigger than it was when the mouse button was pressed, and the same for `mouseY`. If not, the values of `x0` and `y0` are swapped with the values of `mouseX` and `mouseY`. Then we create a cropped image with the `get()` function, using the upper-left and lower-right coordinates 4:

```
sub = get(x0, y0, (mouseX-x0), (mouseY-y0));
```

The `get()` function returns a rectangular region of an image specified by a coordinate pair, a width, and a height. In the preceding call (`x0`, `y0`) are the upper-left coordinates, the width is the distance between the `mouseX` value and the upper-left *x* value, and the height is the distance between `mouseY` and the upper-left *y* value. In this case, `get()` is using the image displayed in the sketch window as the original.

The sub-image returned by the `get()` function becomes the current image to be displayed in `draw()` (the variable `img`) centered in the window 5.

A new idea in this sketch is the test to see which mouse button was pressed. In the `mouseReleased()` function, this statement tests for the right mouse button:

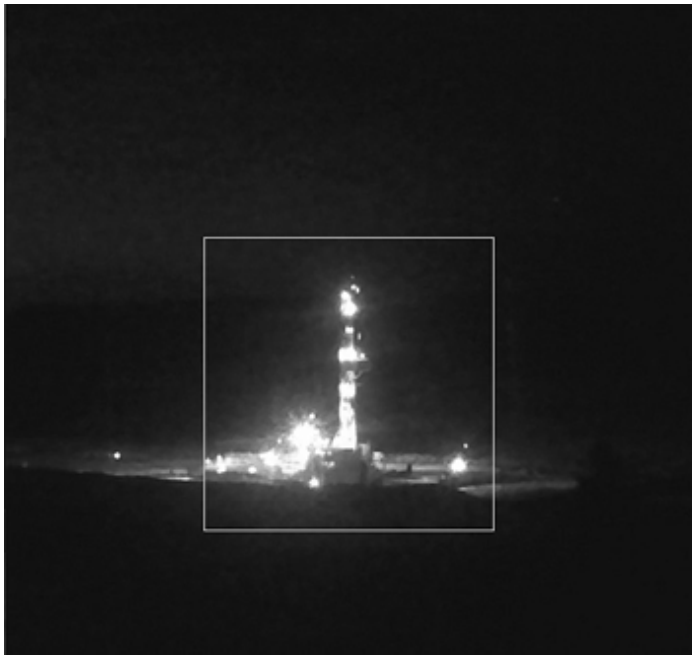
```
if (mouseButton == RIGHT) sub.resize (width, height);
```

If that was the one released, the sub-image is rescaled to fit the window.

At 1 we resize the graphics window to be the size of the image, as we've done before.

```
PImage img;
boolean flag = false;
PImage sub;
int x0=0, y0=0;


void setup ()
{
1 size (100, 100);
  surface.setResizable(true);
  img = loadImage ("image.jpg");
  surface.setSize(img.width, img.height);}

void draw ()
{
  background (200, 200, 200);   // White background
2 image (img, (width-img.width)/2, (height-
img.height)/2);

3 if (flag)                        // If a mouse button is
down then x0,y0 are defined
  {                                 // Draw a rectangle from
(x0,y0) to the mouse cursor
    noFill(); stroke(200);
    rect (x0, y0, (mouseX-x0), (mouseY-y0));  // Draw
rectangle
  }
}
void mousePressed ()
{
  flag = true;
  x0 = mouseX; y0 = mouseY;
}

// Mouse button released. Select the sub-image that lies
in the rectangle
// and rescale it; replace current display image with
the new cropped one.
void mouseReleased ()
{
  int t;
  flag = false;
  if (x0 > mouseX) { t = mouseX; mouseX = x0; x0 = t; }
  if (y0 > mouseY) { t = mouseY; mouseY = y0; y0 = t; }
4 sub = get(x0, y0, (mouseX-x0), (mouseY-y0));
```

```
  if ((mouseX-x0 > 0) && (mouseY-y0 > 0))
  {
    if (mouseButton == RIGHT) sub.resize (width,
height);
 5 img = sub;
  }
}
```



Sketch 17: Manipulating Images III—Magnifier

Some computers have a “magnifying glass” object that is controlled by the mouse and displays a close-up (magnified) view of a part of the screen. It allows people with a minor visual impairment to see things more clearly, and it allows everyone to get a better look at menus and other screen-based objects.

Magnification is done by increasing the size of each pixel in the original image. If each pixel in the original becomes four pixels (in a square) in the new image, then the size of the new image will be double that of the original, giving the appearance of a magnified version, as shown in *Figure 17-1*. The image will contain no more detail than the original; it will just be easier to see.

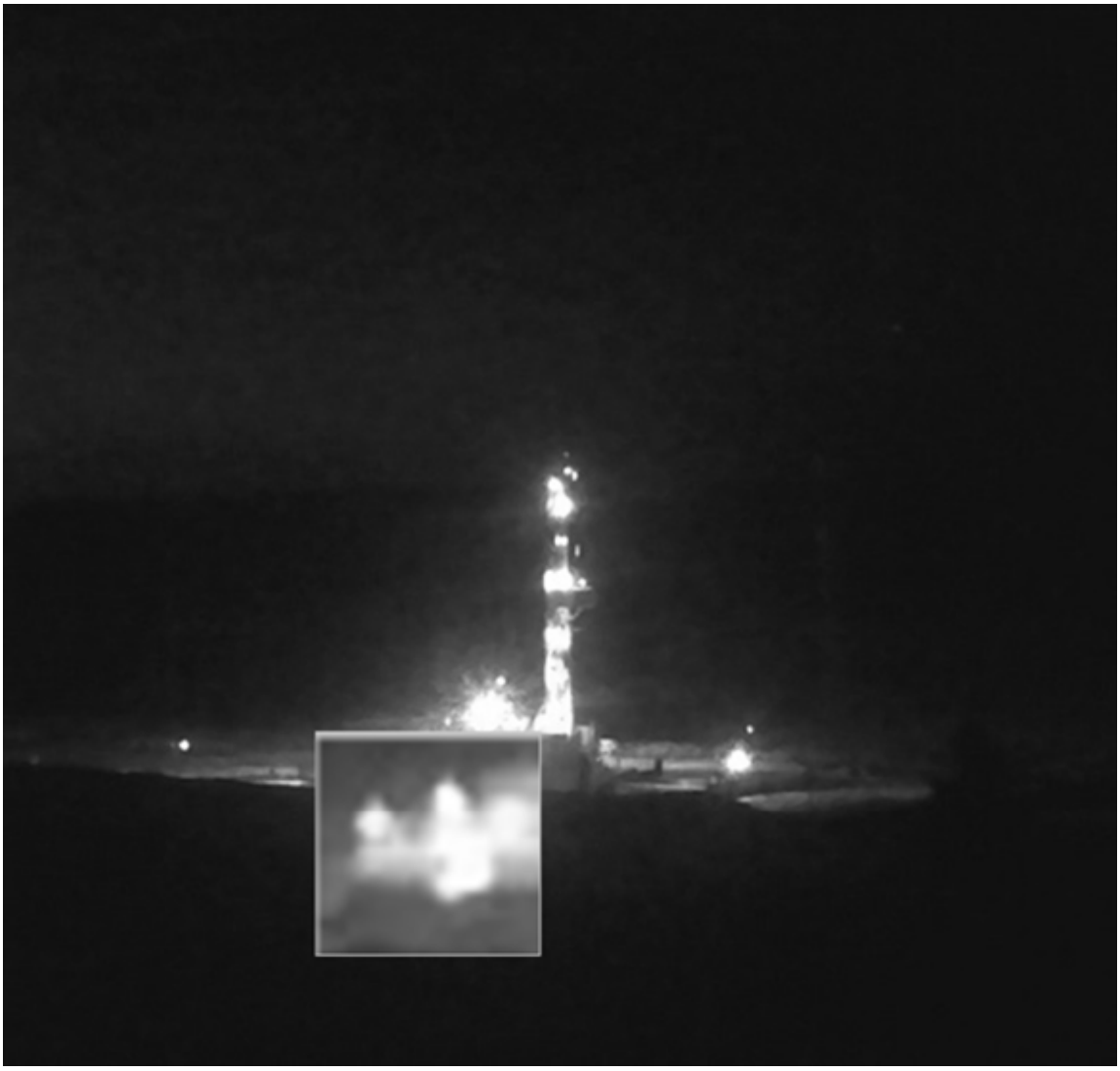


Figure 17-1: Magnifying an image

Implementing a magnifying glass is a simple matter using the functions that Processing provides. First we display the target image and use the techniques discussed in previous sketches to select a rectangular region in the sketch window to be magnified. By pressing the mouse button, the user selects a square beginning at the mouse coordinates with a size of 50×50 1. The Processing functions `mousePressed()` and `mouseReleased()` are called when the button is pressed and released 3, and we use these functions to set a flag variable named `mag`. If `mag` is set, we copy the selected part of the original image into another `PImage` named `sub` using the `get()` function. The copied image is then resized to be 100×100 pixels using the `resize()` function 2:

```
sub.resize (100,100);
```

Taking a 50×50 image and making it 100×100 effectively doubles its size. Now the resized image is drawn on the screen at the location from which it was copied, more or less. The new image is larger than the extracted one, so the new position is approximate, and some pixels from the original will be hidden behind the new, larger copy.

```
PImage img;             // Original image
boolean mag = false;    // Has the mouse been pressed?
PImage sub;             // Smaller, magnified image


void setup ()
{

  size (100,100);
  surface.setResizable(true);
  img = loadImage ("image.jpg");
  surface.setSize(img.width, img.height);
}

void draw ()
{
  background (200, 200, 200);
  image (img, 0, 0);  // Display the image

  if (mag)            // If the mouse is being pressed,
                      // compute and display
  {                   // a rectangular and magnified
region
                      // with mouse at UL
    stroke(200);
    noFill();
    rect (mouseX, mouseY, 100, 100);     // Outline the
magnified region
1 sub = get(mouseX, mouseY, 50, 50);   // Get the sub-
image
2 sub.resize (100,100);                // Double its
size
    image (sub, mouseX, mouseY);         // Draw the
sub-image
  }
}

// Set the flag 'mag' when the mouse button is pressed.
3 void mousePressed ()
{
  mag = true;
}

// Clear the flag 'mag' when the mouse button is
released.
void mouseReleased ()
```

```
{
  mag = false;
}
```

Sketch 18: Rotation

When rotating something, we always need to specify an *axis of rotation*, which in two dimensions is a *point* and an *angle*. A rotation is specified using a call to the function `rotate()`,

```
rotate(angle);
```

where `angle` is specified in *radians*. A circle contains 2π radians and also 360 degrees, so to convert from degrees to radians means multiplying the degrees by 3.14159/180.0, or by using the Processing function `radians(x)`. The point (axis) about which the rotation will take place is the origin of the window's coordinate system, (0, 0), by default. It is the upper-left corner of the window, and the rotation will be *clockwise* about this point.

When a rotation is specified, all things drawn from that point on will be rotated. Calling `rotate()` again rotates by a further angle. Turning off the rotation is not possible as such, but a call to `rotate (-angle)` will undo the call to `rotate(angle)`.

Example A

The first example draws a figure illustrating rotations. A horizontal line is drawn and labeled 1, followed by a line that is rotated by 10 degrees 2 and then by one rotated by 20 degrees 3. To avoid having the text rotated, the rotation is "undone" (rotated by the negative angle) before drawing the text labels.

Example B

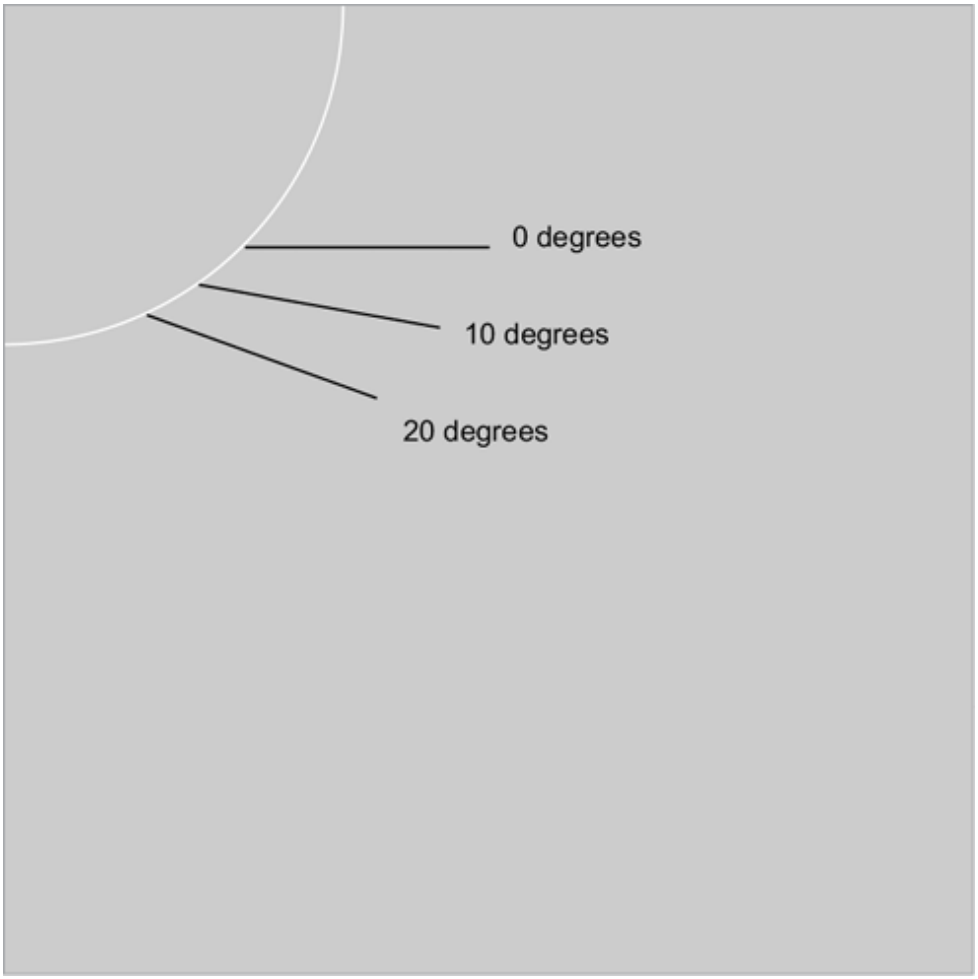
A line is drawn from the origin. It has a small ball on the end. This line is rotated from 0 to 90 degrees in small steps, with each step displayed within `draw()` because the line and ball are drawn there. When the line is rotating clockwise, the angle is incremented by 0.01 radians each time `draw()` is called 1. When the line becomes vertical and further rotation would take it out of the field of view, the change in angle for each frame (variable `d`) is changed to `-d` 2. Now the line rotates back to its original position, and at that

point (0 degrees) the value of `d` is changed to become positive again. The object, which looks like a pendulum, bounces between 90 degrees and 0 degrees.

The rotation angle is reset to 0 each time `draw()` is called.

Example A

```
float d2r = 3.14159/180.0;
void setup ()
{
  size (400, 400);
  noFill();
  stroke(255);
  ellipse (0,0,280,280);
  stroke(0);
  fill(0);

1  line (100, 100, 200, 100);
   text ("0 degrees", 210, 100);

2  rotate (radians(10));
   line (100, 100, 200, 100);
   rotate (-10*d2r);
   text ("10 degrees", 190, 140);

3  rotate (20*d2r);
   line (100, 100, 200, 100);
   rotate (-20*d2r);
   text ("20 degrees", 165, 180);
}
```



Example B

```
float angle = 0.0,  d = 0.01;

void setup ()
{
  size (150, 130);
  stroke(0);
}

void draw ()
{
  background (200);
  rotate (angle);
  line (0, 0, 50, 0);
  ellipse (50, 0, 3, 3);
1 angle += d;
2 if (angle > 1.6) d = -d;
```

```
  else if (angle < 0.0) d = -d;
}
```

Sketch 19: Rotating About Any Point—Translation

Being able to rotate objects is essential, but only being able to rotate about the upper-left corner of the screen is inconvenient. Rotation about an object's center is what we usually want, but it requires knowledge of the object. Objects can be complex things in graphics; an object might be just a circle or square, or it might be a building or a car. Processing cannot be expected to know what an object is or where the center might be. However, Processing makes it possible to move the center of rotation to any coordinate we choose, using the `translate()` function.

`translate()` takes an x- and a y-coordinate and changes the origin to that location for all future drawing. The following example moves the origin to the location (100, 200) in the window, which now becomes the coordinate (0, 0):

```
translate (100, 200);
```

The word *translate* means, in mathematical terms, to *reposition*, so a translation involves changing the position of an object. If we translate the origin to (50, 50) and then draw a circle at (0, 0), the circle will appear at window coordinates (50, 50) on the screen. Further circles will be drawn relative to window coordinates (50, 50).

Because rotations always use (0, 0) as the axis, this means we can set the axis to any coordinates we like and rotate an object about any point.

Example A

As a basic example, we'll draw a circle at (0, 0) using the `ellipse()` function 1 and then call `translate()` to change the origin to (50, 50). A second call to `ellipse()` that is exactly the same as the first draws the circle at screen coordinates (50, 50). A further `ellipse()` call drawing a circle at (30, 40) draws the circle at screen coordinates (80, 90); that is, (30, 40) relative to the new origin at (50, 50).

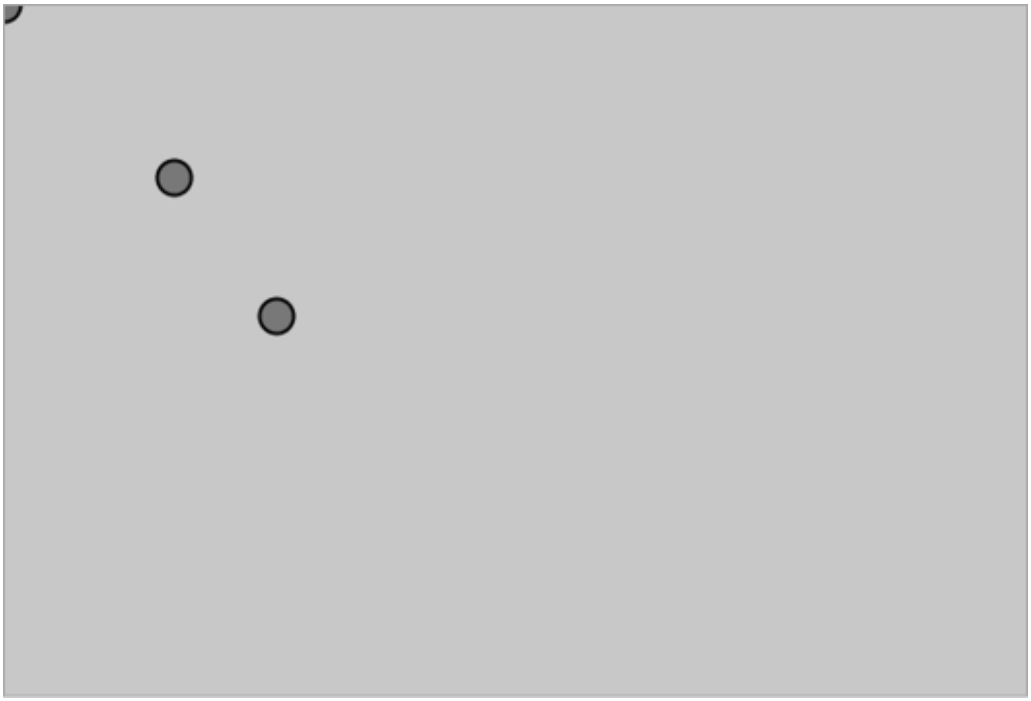
Example B

In this sketch we rotate a line about its center. The process is to `translate()` to the center of the line, in this case (150, 100) 1; rotate by the current angle 2; and then draw the line. The coordinates of the line must reflect the fact that the origin is the line's center, not one end. Because the center of the line is at (150, 100), the line should be drawn from −50 to +50 in the x-direction so as to be 100 pixels long. The translated coordinates of the start would be (150 − 50, 200 − 100 − 0), or (100, 100). The coordinates of the endpoint will simply be 100 pixels further in x, or (200, 100). A small circle is drawn at the midpoint (origin) so that it can be seen.

The rotation angle increases each time `draw()` is called 3. Since the line is drawn each time `draw()` is called, the image shows a slowly rotating line.

Example A

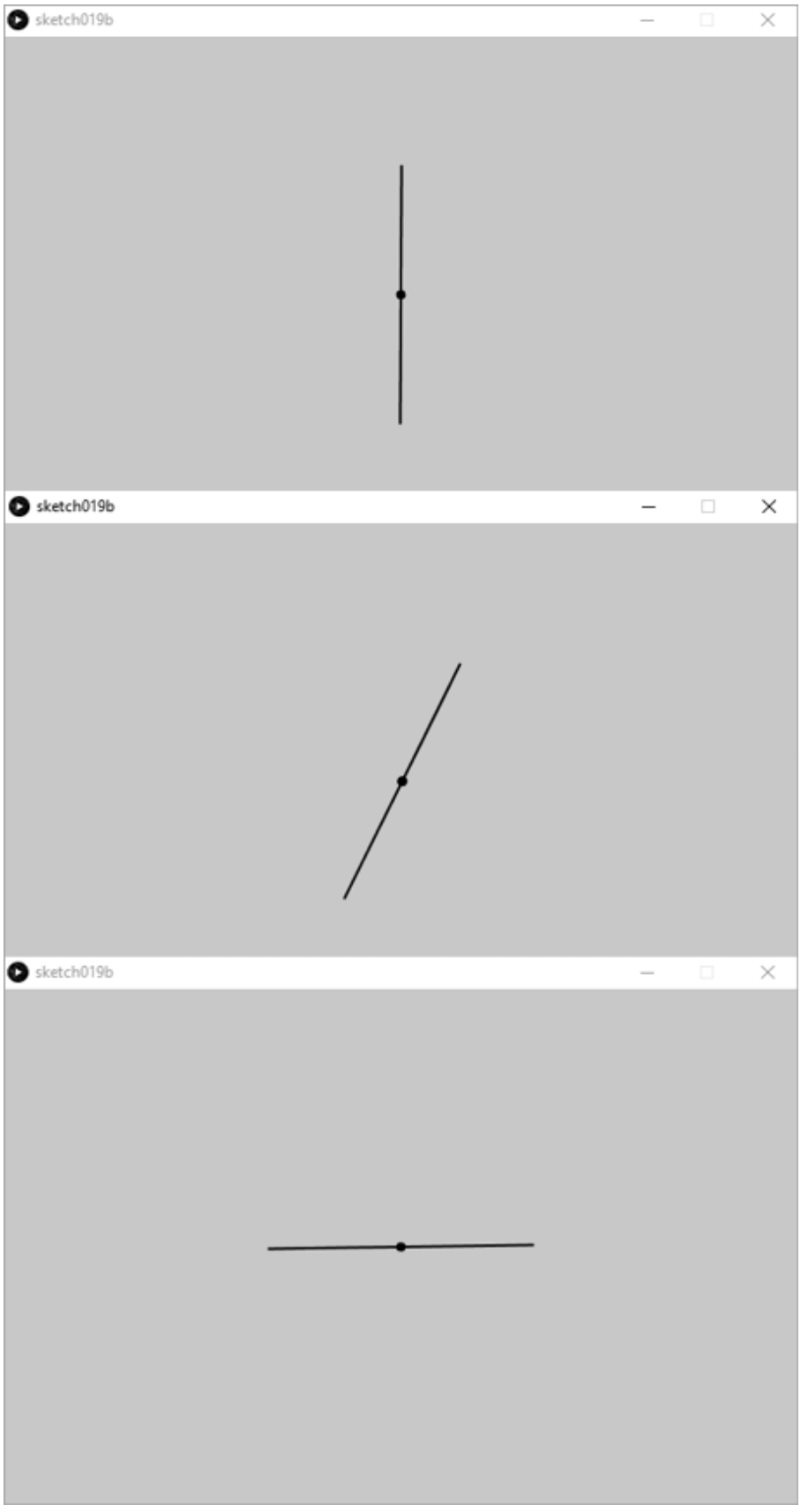
```
// Translate a circle
void setup ()
{
  size (300, 200);
  stroke(0);
  fill(200, 100, 50);
}

void draw ()
{
  background (200);
 ellipse (0, 0, 10, 10);
  translate (50, 50);
  ellipse (0, 0, 10, 10);
  ellipse (30, 40, 10, 10);
}
```



Example B

```
// Rotate a line about its origin
float angle = 0.0;

void setup ()
{
```

```
  size (300, 200);
  stroke(0);
  fill(0);
}

void draw ()
{
  background (200);
1 translate (150, 100);
2 rotate (angle);
  ellipse (0, 0, 3, 3);
  line (-50, 0, 50, 0);
  translate (-150, -100);
3 angle += .01;
}
```


sketch019b
sketch019b
sketch019b

Sketch 20: Rotating an Image

Rotation and translation can be applied to complex objects as well as simple lines and circles. In particular, we can rotate images about arbitrary points by any angle.

There can be a problem in determining how to place the image so that it lies entirely on the screen. Images are rectangular, and rotating them increases their width or height. If we don't place the image properly within the window, one or more corners could rotate out of the window's boundaries, as shown in *Figure 20-1*.

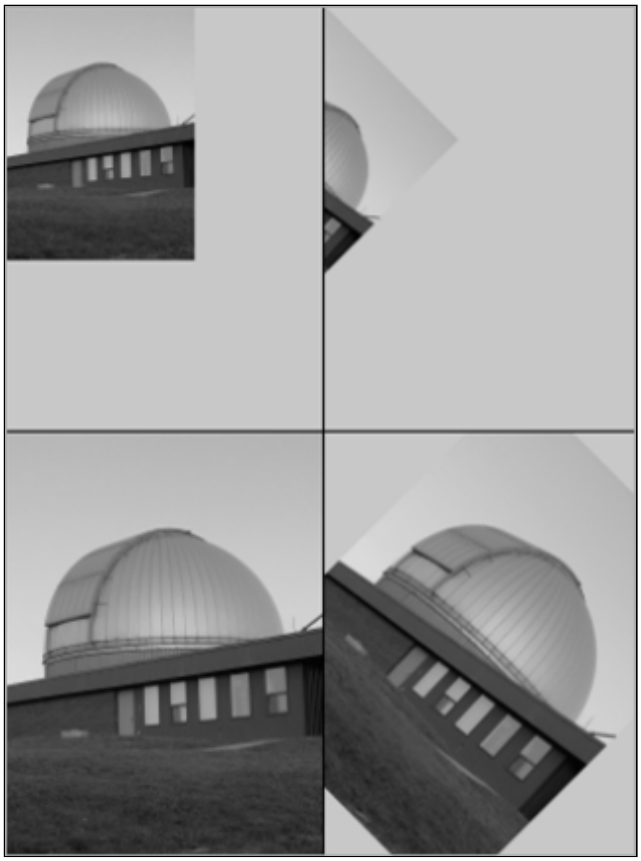


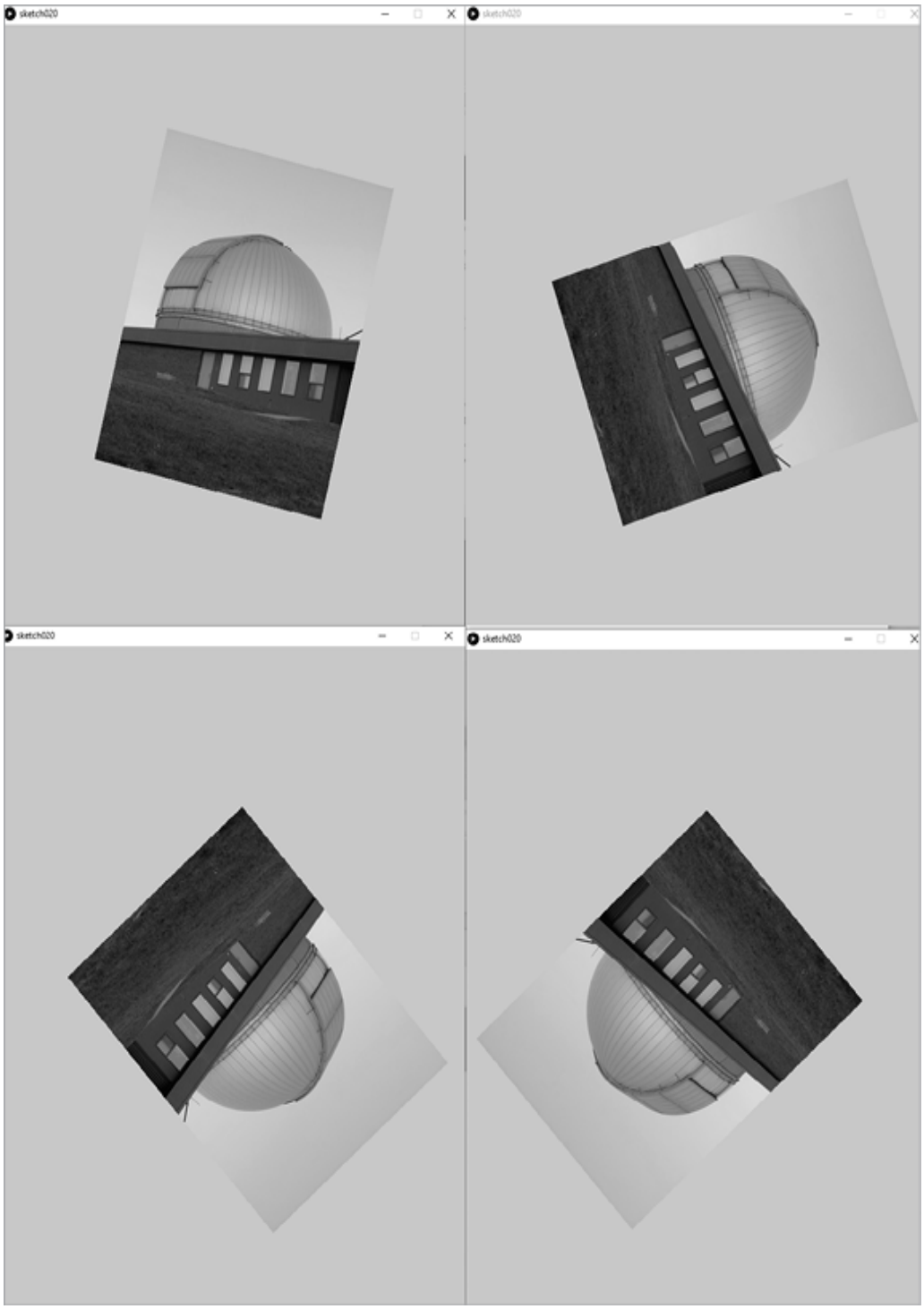
Figure 20-1: Rotating an image out of the window's boundaries

The top pair of images shows the result of rotating an image that was displayed in the upper-left corner of the window. Rotating by 45 degrees moves half of the image off of the screen. The bottom pair of images shows

what can happen when an image is rotated about its center without a large enough window: the corners of the image are cut off.

This sketch displays a continuously rotating image. The image is read in, and the window size is set to *double* the image size in each dimension 1. The `draw()` function translates the origin to the center of the image and then rotates the image by `angle` and displays it, thus rotating the image about its own center 2. The value of `angle` is then increased by a tiny amount for the next time `draw()` is called 3. The image appears in the center of the window and appears to rotate.

```
PImage img;          // Image to rotate
float angle = 0.0;  // Angle of rotation

void setup ()
{
  size (100,100);
  surface.setResizable(true);
  img = loadImage ("image.jpg");
1 surface.setSize(img.width*2, img.height*2);
}

void draw()
{
  background(200);                          // White
background
  translate(img.width, img.height);         // Move image
to origin
2 rotate(angle);                           // Rotate
  translate (-img.width, -img.height);      // Move back
  image (img, img.width/2, img.height/2);  // Draw
3 angle = angle + 0.01;                    // Increment
angle
}
```


sketch020
sketch020
sketch020
sketch020

Sketch 21: Getting the Value of a Pixel

While developing Sketch 14, we discussed how to get a pixel value from a `PImage` using the `get()` function. We can get the color value of the pixel at (`x`, `y`) in a `PImage` named `im` as follows:

```
color c;
PImage im;
c = im.get (x, y);
```

The picture currently being displayed in the sketch window has a privileged position because it can be accessed without using a variable. A pixel value on the screen can be obtained by simply calling `get()`:

```
c = get (x, y);
```

We can therefore get the color of the pixel at the current mouse position with

```
c = get (mouseX, mouseY);
```

This sketch loads an image and allows the user to click any pixel to see its color, which will be displayed on the right side of the screen as a colored bar and as RGB values in text form. First the sketch loads an image and sizes the window to fit it, with an extra region on the right side. In `draw()` it displays the image with a background color of (200, 200, 200) 1; when the mouse button is pressed, it assigns the pixel value (color) at the `mouseX`, `mouseY` coordinates to the color variable `c` 3; then it displays the color on the right side of the image and the RGB values as text at the upper-right corner of the screen 2. Whenever the mouse button is clicked, the color value displayed will change.

NOTE

Another way to describe a color is with the HSB system, in which the three color values represent the hue, saturation, and brightness. Hue is the actual color, such as red or blue. Saturation indicates the intensity of the color. Brightness indicates a value of light. Changing to HSB mode in Processing is done by calling `colorMode (HSB, 256)`*, for example, in which each of* `H`*,* `S`*, and* `B` *have values between 0 and 255. The call* `colorMode (RGB, 256)` *would return to the default RGB coordinates.*

No matter what the color mode is, the function `hue(c)`*, when* `c` *is a color, will return the hue value. The functions* `saturation(c)` *and* `brightness(c)` *yield the other two HSB components.*

```
PImage img;
color c;

void setup ()
{
  size(200, 200);
  surface.setResizable(true);
  img = loadImage ("image.jpg");
  surface.setSize(img.width+55, img.height);
  c =  color (200, 200, 200);     // Default background
}

void draw()
{
1 background (c);
  image (img, 0, 0);
  if (red(c) != 200)
  {
 2 text ("R="+red(c), img.width+2, 20);
    text ("G="+green(c), img.width+2, 35);
    text ("B="+blue(c), img.width+2, 50);
  }
}

void mousePressed ()
{
  if (mouseX > img.width)
    c = color(200, 200, 200);
  else
  3 c = get (mouseX, mouseY);
}
```


R=144.0
G=82.0
B=95.0

Sketch 22: Setting and Changing the Values of Pixels

Pixel values in an image, including the drawing area, can be changed using the `set()` function. We specify a pixel location using coordinates and identify the color to draw at that point. For example,

```
set (i,j, color(255, 255, 0));
```

This sets the pixel in the graphics area at location (`i`,`j`) to yellow, or RGB (255, 255, 0). If the coordinates lie outside of the window, the pixel will be drawn but will not be visible.

We can set a color for *all* pixels in the window using the `background()` function:

```
background (255, 100, 40)
```

This call fills the sketch window with orange.

Example A

Setting all pixels in the window without using the `background()` function requires a *loop*—two nested loops, in fact. The first loop examines all pixels in the horizontal direction; that is, all pixels in a specified row. The second loop looks at all possible values of `i`, which is to say all rows. The first loop is nested within the second so that all pixels in all rows are modified:

```
for (i=0; i<width; i++)
   for (j=0; j<height; j++)
     set(i,j, color(255, 100, 40));
```

This sets all pixels in the sketch window to orange.

Example B

The pixel values of an image can be modified before the image is displayed in the sketch window. Not only can the color be replaced, but a pixel value can be changed more subtly to a variation of what is already there. This example first loads and displays an image. When the program detects a button press with `mousePressed()`, it sets the flag `grey` 2, which indicates that the image on the screen is to be modified, pixel by pixel, in a loop like that of the previous example. In this case, we replace the RGB value of each pixel on the screen with its brightness value 1, and the result is a grey image showing no color. When the mouse button is released (`mouseReleased()`), the program clears the flag (sets it to `false`), and the image is displayed in color again 3.

Example A

```
void setup ()
{
  size (400, 300);
}

void draw ()
{
  int i,j;

1 for (i=0; i<width; i++)
    for (j=0; j<height; j++)
      set (i,j, color(255, 100, 40));
}
```

Example B

```
PImage img;
color c1, c2, c;
boolean grey = false;

void setup ()
{
  size(200, 200);
  surface.setResizable(true);
  img = loadImage ("image.jpg");
```

```
  surface.setSize(img.width, img.height);
  c =  color (200, 200, 200);     // Default background
}
void draw ()
{
  int i,j;
  color c1, c2;
  background (200);
  image (img, 0, 0);
  if (mousePressed)
  {
    for (i=0; i<width; i++)
      for (j=0; j<height; j++)
      {
        c1 = get (i,j);
1       c2 = (int)brightness(c1);
        set (i,j,color(c2,c2,c2));
      }
  }
}
2   void mousePressed () {  grey = true; }
3   void mouseReleased () {  grey = false; }
```

Sketch 23: Changing the Values of Pixels—Thresholding

The act of *thresholding* an image changes the color value of each pixel to either black or white, depending on the original color or brightness. Thresholding creates a *binary* image: each pixel can be thought of as being either *on* or *off*. Why do this? Some images have content that is fundamentally binary: a scan of a page of text has black characters on a white background. In other cases, it is a way to simplify an image so that we can perform other operations, such as detecting edges or faces. Thresholding an image of red blood cells might facilitate counting them, for example.

Thresholding an image is a two-step process. First we determine a threshold value—one that retains the required features of the image. We usually do this by examining all of the pixels in the image and computing a value using some statistical formula. A threshold value is a number between 0 and 255; all pixel brightness values smaller than the threshold will be set to black (0), and those greater will be set to white (255). The second step is looking at all of the pixels and actually applying the threshold.

To address the second step first, applying the threshold is a simple matter of looking at each pixel and deciding whether it is less than or greater than `threshold`. We assign the pixel value to the variable `g`, extract the brightness, and then test it:

```
2 if (g<threshold) g = black;
    else g = white;
```

The value `black` is the color (0, 0, 0), and `white` is (255, 255, 255).

In this sketch, we will determine the threshold manually, using the mouse position. This is the horizontal position of the mouse as a percentage of the total window width:

```
mouseX/width
```

If we multiply this fraction by 255, we get a value between 0 and 255 1 that is in proportion to how far to the right the mouse is. We'll use this value as a

threshold. When the mouse is on the left side of the window, the threshold will be small and most of the image will be white; when the mouse is on the right, the threshold will be large and the image will be largely black.

If the `draw()` function calculates and applies the threshold, it will be dynamic, and we can watch the image change as the mouse moves.

NOTE

When calculating a threshold from pixel values, we could select the average pixel level as the threshold; the implication here is that half of the pixels would be black and half white. However, this is not a good threshold for text, because many more pixels in a text image will be white than will be black. There are many algorithms for thresholding images, but there is no best one in general.

We could also threshold each color value separately to create interesting images, with each color either full on or full off.

```
PImage img;
int threshold;
color black = color(0, 0, 0);
color white = color (255, 255, 255);

void setup ()
{
  size(200, 200);
  surface.setResizable(true);
  img = loadImage ("image.jpg");
  surface.setSize(img.width, img.height);
  threshold = 128;
}

void draw ()
{
  color c;
  int i,j,g;

  background(200);
  image (img, 0, 0);

  threshold = (int)((1(float)mouseX/(float)width) *
255);
  for (i=0; i<width; i++)
    for (j=0; j<height; j++)
    {
      c = get (i,j);
      g = (int)brightness(c);
   2 if (g<threshold) g = black;
      else g = white;
      set (i,j,g);
    }
}
```

Sketch 24: User-Defined Functions

Up to this point, we have been using drawing functions provided by the Processing system: `ellipse()`, `line()`, `mousePressed()`, and so on. We have not analyzed the concept of functions much, partly because it appears fairly obvious what is going on. However, if we wish to create our own functions, there are some things we need to understand.

A *function* is a name given to a collection of code. When the name of the function is used in a statement, that function is said to be *invoked* or *called*, and the code within the function is executed. This means that functions can be executed from many different places without repeating the code itself, merely by calling it.

A function can return a value: we've used such functions before. For example, `color()` and `get()` return colors and pixel values. Functions that do not return a value are said to return `void`, and that's the reason for the word `void` in front of `setup()` and `draw()`. They are functions that do not return a value.

To create a new function, we must follow the same syntax as `setup()` and `draw()`: we write the return type, the function name, parentheses, and then the function body in curly brackets. For example:

```
void newFunctionA ()
{
  code

}
```

```
int newFunctionB ()
{
  code
  return value;
}
```

Above, `newFunctionA()` does not return a value, and it is called or invoked using the function call `newFunctionA();`. `newFunctionB()` returns an integer value and must have a `return` statement indicating the value to be returned. This type of function is called as if it were part of an expression:

```
a = newFunctionB();
x = newFunctionB()*2 + 1;
```

There can be more than one `return` statement in a function, but only one will be executed, because once a function returns, no other code in the function can execute.

Functions may have *parameters* or *arguments*, values that are given to the function when it is called. When calling the function `color()`, we list three values in the parentheses: red, green, and blue. These variables are specifically *passed* for use by the function, and their values are available to do calculations within the function.

Let's make a function that calculates the distance between two points, (x0, y0) and (x1, y1):

```
float distance (int x0, int y0, int x1, int y1)
```

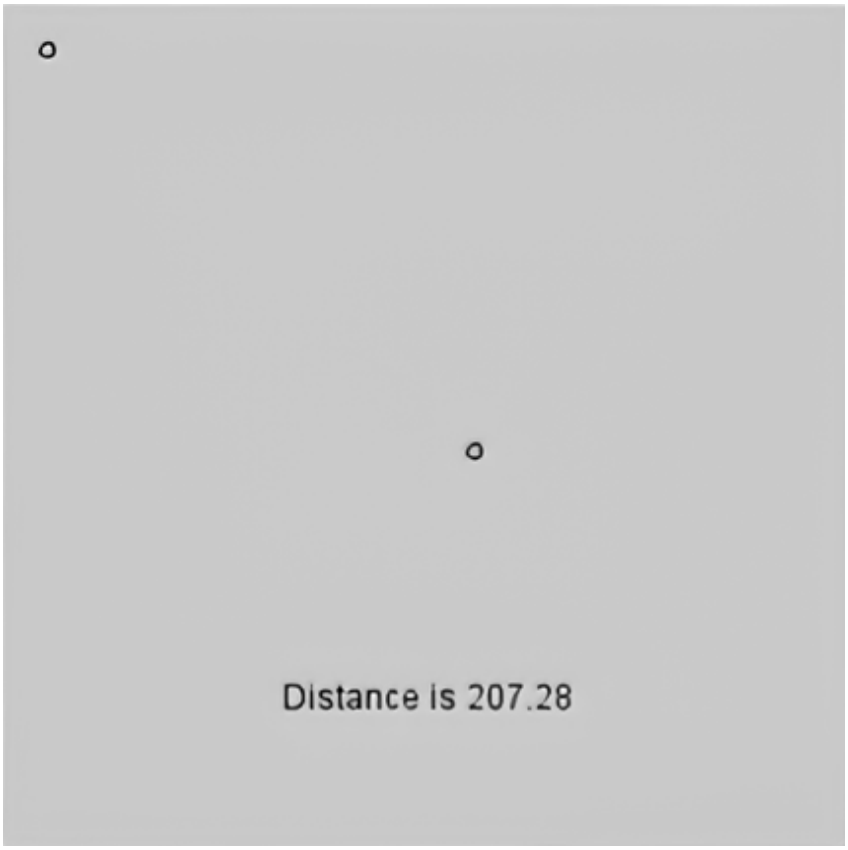
The arguments to the function are named `x0`, `y0`, `x1`, and `y1`, and they have types, in this case integer (`int`). The arguments are used to calculate the distance between the points (x0, y0) and (x1, y1) as follows:

$$\sqrt{(x0 - x1)^2 + (y0 - y1)^2}$$

This sketch uses two mouse clicks to determine the points: (x0, y0) 2 and (x1, y1) 3. It draws a marker at each point and displays the distance between them as a text message at the bottom of the window 1.

```
int x0 = -1, y0 = -1;
int x1 = -1, y1 = -1;
boolean ok = false;

void setup ()
{
  size (300, 300);
}

void draw ()
{
  background(200);
  fill (100, 255, 40);
  if (x0 >= 0)
    ellipse (x0, y0, 5, 5);
  if (x1 >= 0)
    ellipse (x1, y1, 5, 5);
  if (ok)
  {
    fill (0);
1 text ("Distance is "+distance(x0,y0,x1,y1), 100,
250);
  }
}

float distance (int x0, int y0, int x1, int y1)
{
  float d;

  d = (x0-x1)*(x0-x1) + (y0-y1)*(y0-y1);
  d = sqrt(d);
  return d;
}

void mousePressed ()
{
  if (x0 < 0)
  {
2 x0 = mouseX; y0 = mouseY;
  } else if (x1 < 0)
  {
3 x1 = mouseX; y1 = mouseY;
    ok = true;
  }
}
```


Distance is 255.78311


Distance is 207.28

Sketch 25: Elements of Programming Style

Style in a program refers to aspects of the code that don't usually impact the execution but that do have an effect on how other people read, modify, or repair it.

For example, there is a way to place human-readable text within a program for other programmers to read. Any text that follows a pair of slashes is called a *comment*, and it is ignored by Processing, as is any text in between the symbols `/*` and `*/`, which delineate comments that can span many lines. Programs should have relevant comments embedded within the code to explain what is going on to any human beings looking at it. Comments should be clear, offer an explanation, and never simply repeat the code itself. The nature of the comments in a program is one aspect of what we call *programming style*.

Another aspect of style is the use of *indentation* to convey structure. There is no single correct way to indent, but the standard shown in the sketches in this book has certain consistent features. For example, the "{" and "}" characters used to enclose blocks of code always line up with each other vertically so that the blocks are easy to identify. The only exception is when they are on the same line:

```
if (x0 < 0)
{
  x0 = mouseX;
  y0 = mouseY;
}
```

In other books, you might see another style:

```
if (x0 < 0)  {
  x0 = mouseX;
  y0 = mouseY;
}
```

The location of brackets doesn't matter to the programming language's compiler, but a programmer should be consistent.

Variables should have meaningful names. The variables `x0` and `y0` above represent x- and y-coordinates, so the names make sense. A variable named `pixelCount` should contain a count of pixels. It is pretty easy to give variables good names, and doing so does not impact how fast the code is or how much memory it requires in order to execute.

Numeric constants should be named like variables so that the purpose of the constant can be inferred from the name. A perfect example is `PI` instead of 3.1415. A program should contain very few if any numerical constants, and names should be used instead. Consider the following code:

```
r = d*0.01745;
```

The number 0.01745 is meaningless to most people. Now consider this:

```
r = d *2*PI/360.0;
```

This is better. Two times PI/360 is the conversion between radians and degrees. Best would be

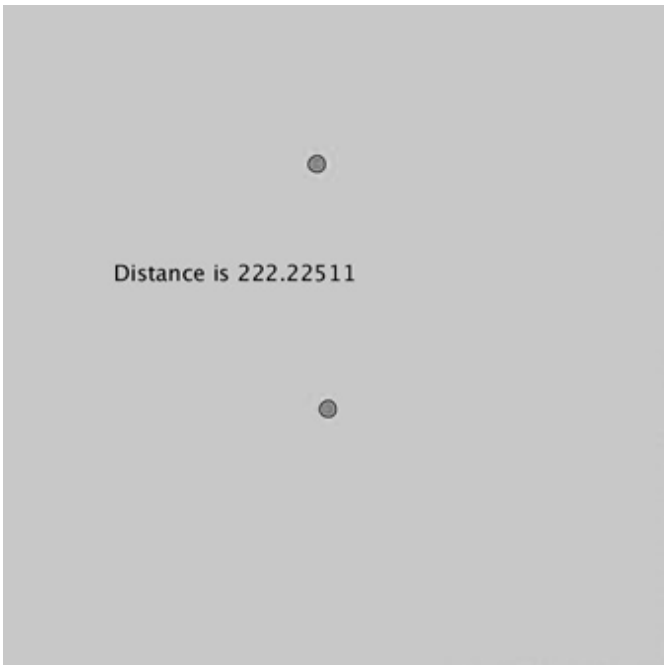
```
radians = degrees * degrees_to_radians;
```

where `degrees_to_radians` equals `0.01745`. Now anyone reading the code can easily see what is happening.

The code in this sketch does the same thing as the previous sketch, but it shows better style. Note, though, that it takes more space on the screen—this is typical, and it's why these rules are not followed all of the time, even in this book (where it's important for the text to fit on a single page).

```
int x0 = -1, y0 = -1;               / The first pixel
clicked on
int x1 = -1, y1 = -1;               // The second pixel
clicked on
color beige = color(200,200,200);  // Background
color black = color(0,0,0);         // The color black
(text)
color green = color (100,255,40);  // Circle color
boolean bothPixelsSet = false;      // Mouse clicked
twice?
int textX=100, textY=250;           // Where to draw the
distance
int circleSize = 5;                 // Size of the
circles at the click points

void setup ()                       // Set up a 300x300
window
{  size (300, 300);   }
void draw()
{
  background(beige);                // Fill the window
with beige
  fill (green);                     // Objects will be
filled with green
  if (x0 >= 0)                      // x0 >= 0 means it
has been set
     ellipse (x0, y0, circleSize, circleSize);
  if (x1 >= 0)                      // x1 >= 0 means it
has been set
     ellipse (x1, y1, circleSize, circleSize);
  if (bothPixelsSet)
  {
    fill (black);                   // Set text color
    text ("Distance is "+distance(x0,y0,x1,y1), textX,
textY);
  }
}
// Compute the distance between two points
float distance (int x0, int y0, int x1, int y1)
{
  float d;
  d =sqrt( (x0-x1)*(x0-x1) + (y0-y1)*(y0-y1));
  return d;
}
// Mouse is released when a point is defined
void mouseReleased ()
{
```

```
  if (x0 < 0)                          // First click has
not been made
  {  x0 = mouseX; y0 = mouseY;
  } else if (x1 < 0)                   // This is the second
click
  {   x1 = mouseX; y1 = mouseY;
      bothPixelsSet = true;
  }
}
```



Sketch 26: Duplicating Images—More Functions

The purpose of this sketch is to give you some ideas about how to organize code into functions properly. This program will read an image, make a copy, and increase the brightness of the copy. The brighter version will be displayed when a mouse button is pressed.

The first function is named `brighten()`. It is passed an image (named `img`) and an integer value (named `val`) as parameters. Its purpose is to increase the brightness value in an image by a specified amount. It does this by extracting the HSB value from each pixel in turn in a nested loop 2, adding the amount `val` to the brightness portion, and saving the pixel back in the image. This is the essential code 3:

```
// Extract the HSB values from the pixel at (i,j)
c = img.get(i,j);
// Add val to the brightness and save again.
img.set (i,j, color(hue(c), saturation(c),
brightness(c)+val));
```

We will use a new feature in `draw()`. Processing provides us with a variable named `mousePressed` 1 that is `true` if a mouse button is depressed and `false` otherwise, and this can be used in place of the `mousePressed()` callback function in very simple cases. In this instance, we display the brightness-enhanced image when the mouse button is pressed, and the original otherwise.

```
if (mousePressed) image (img2, 0, 0);
  else image (img1, 0, 0);
```

The second function in this sketch makes a copy of the original image. We define the duplicate function as follows:

```
PImage duplicate (PImage from)
```

According to the definition of this function, the function takes an image as an argument and returns an image. In fact, it returns a new image that is a

copy of the one passed in. The Processing-supplied function that creates a new image is named `createImage()` 4, and it has this form:

```
createImage (width, height, RGB);
```

The width and height should be self-explanatory; the constant `RGB` specifies the form of the image, which in this case is RGB color. The image returned by `CreateImage` is *uninitialized*, having pixels with unknown values, so after creating an image the same size as the one passed in, our `duplicate()` function sets each pixel in the new image to the value of the corresponding pixel in the original, with a standard nested loop.

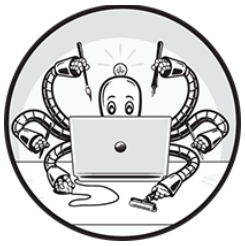
NOTE

Alternatives to the `RGB` parameter for `createImage()` are `ARGB` and `ALPHA`. Look them up. All of the constants refer to color formats.

```
PImage img1, img2;

void setup ()
{
  size(100,100);
  surface.setResizable(true);
  img1 = loadImage ("image.jpg");
  surface.setSize(img1.width, img1.height);
  img2 = duplicate (img1);
  colorMode (HSB);
  brighten(img2, 60);
}

void draw()
{
  background (128); // Grey background
1 if (mousePressed) image (img2, 0, 0);
  else image (img1, 0, 0);
}

void brighten (PImage img, int val)
{
  color c;
2 for (int i=0; i<img.width; i++)
    for (int j=0; j<img.height; j++)
    {
   3 c = img.get(i,j);
      img.set (i, j, color(hue(c), saturation(c),
brightness(c)+val));
    }
}

PImage duplicate (PImage from)
{
  PImage newImage;
  color pixel;
  if (from == null) return from;
4 newImage = createImage (from.width, from.height, RGB);
  for (int i=0; i<from.width; i++)
    for (int j=0; j<from.height; j++)
    {
      pixel = from.get (i,j);
      newImage.set(i,j,pixel);
    }
```

```
  return newImage;
}
```

3
2D GRAPHICS AND ANIMATION



Sketch 27: Saving an Image and Adjusting Transparency

We are going to write a sketch that will allow the user to select a color in an image that will become transparent, and then save the image as a GIF. We can save any `PImage` in a file, just as most image files can be read into a `PImage`. If `img` is a `PImage` variable, we can save it as a file using this function call:

```
img.save ("image.jpg");
```

The parameter is the name of the file to be created. In the situation above, it will create a file named *image.jpg* and save the pixels of the `PImage` in JPEG format. The format is conveniently determined by the last three letters of the filename: *.jpg* for a JPEG file, *.gif* for a GIF file, *.png* for a PNG file, and so on. If no `PImage` variable is given, Processing saves the image that appears in the sketch window.

For this sketch, the first step is to read and display the image. Next, we position the mouse over a pixel with the color we want to make transparent, and click the button. Finally, we save the image in a format that allows transparency (GIF).

In Sketch 2 I mentioned *transparent* colors. We can set a fourth color component, referred to as *alpha*, to a value between 0 (completely transparent) and 255 (completely opaque), as long as the `PImage` color format allows transparency; the format that does this is `ARGB`. In this sketch, when the image is read in, we make a copy as in the previous sketch, but using `ARGB` as the color format. When we click the mouse button, the program looks at the pixel at the cursor's coordinates and adds an alpha value of 0 to the color coordinates. Then the color in the `PImage` is updated with the new alpha value.

The original image that we read from the file is a variable named `img1`; the copy that includes alpha values is `img2`. Processing makes a copy of the image using the following statement, as we do at 2:

```
img2 = createImage (img1.width, img1.height, ARGB);
```

This creates an empty image of the correct size, and now we must copy all of the pixels from `img1` into `img2`. When we do so, the pixels in `img2` have the alpha component, because it was specified in the `createImage()` call. When a mouse click specifies a background color, all pixels of that color are given an alpha value of 0 1. Then `img2` is saved in a file named *out.gif*.

The program ends with a call to `exit()`, because otherwise it would continue to save the same file again and again.

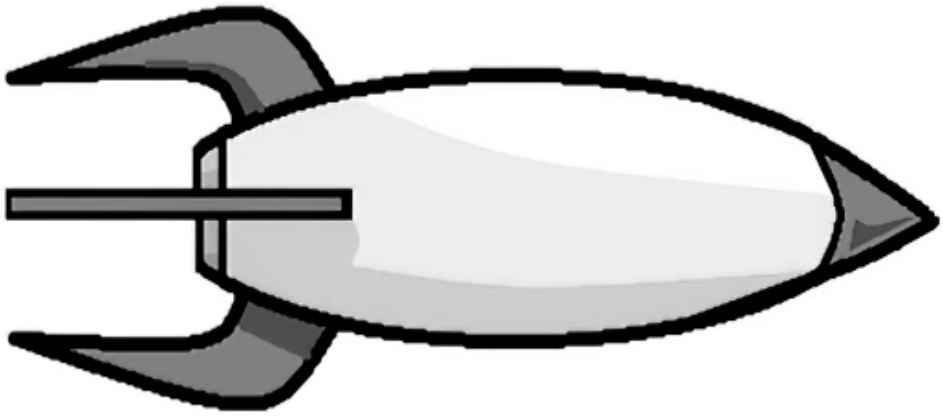
Why is it important to set a transparent background for an image? Computer games!

NOTE

The string parameter in `img.save ("image.jpg");` can include a full path name, so the file can be saved in any directory on your PC.

```
PImage img1, img2;
color c=color(0,0,0);
void setup ()
{
  size(100,100);
  surface.setResizable(true);
  img1 = loadImage ("image.bmp");
  surface.setSize (img1.width, img1.height);
  img2 = duplicate (img1);
}
void draw ()
{
  color c1;
  background (255);
  image (img1, 0, 0);
  if (mousePressed)
  {
    c = get(mouseX, mouseY);
    for (int i=0; i<width; i++)
      for (int j=0; j<height; j++)
      {
        c1 = img1.get(i,j);
        if (c1 == c)
        {
1       c1 = color(red(c1), green(c1), blue(c1), 0);
          img2.set (i,j,c1);
        }
      }
    img2.save ("out.gif");
    exit();
  }
}
PImage duplicate (PImage from)
{
  PImage newImage;
  color pixel;
  if (from == null) return from;
2 newImage = createImage (from.width, from.height,
ARGB);
  for (int i=0; i<from.width; i++)
    for (int j=0; j<from.height; j++)
    {
      pixel = from.get (i,j);
      newImage.set(i,j,pixel);
    }
```

```
  return newImage;
}
```



Sketch 28: Bouncing an Object in a Window

This sketch illustrates a good way to check whether an object is within a sketch window (though it is only completely accurate when the object is circular). The object here is a circle, or a ball if you prefer. The program moves the ball, and when the ball reaches the window boundary (the "wall"), it bounces, or reverses direction.

A simple test establishes whether the ball has exceeded the boundary. In the case of the right boundary wall, for example, it's whether `x + radius > width` 2, where `x` is the ball's center position, `radius` is the ball's radius, and `width` is the width of the window. If the ball is moving slowly enough, we can simply reverse the direction of motion when the ball passes this test by changing `dx` (the amount the ball moves horizontally between each frame) to `-dx`. However, this approach isn't completely accurate, and it gets worse when the ball moves at high speeds. Why? Because the ball will move past the boundary before the program determines that it has reached the boundary. Consider the situation in *Figure 28-1*.

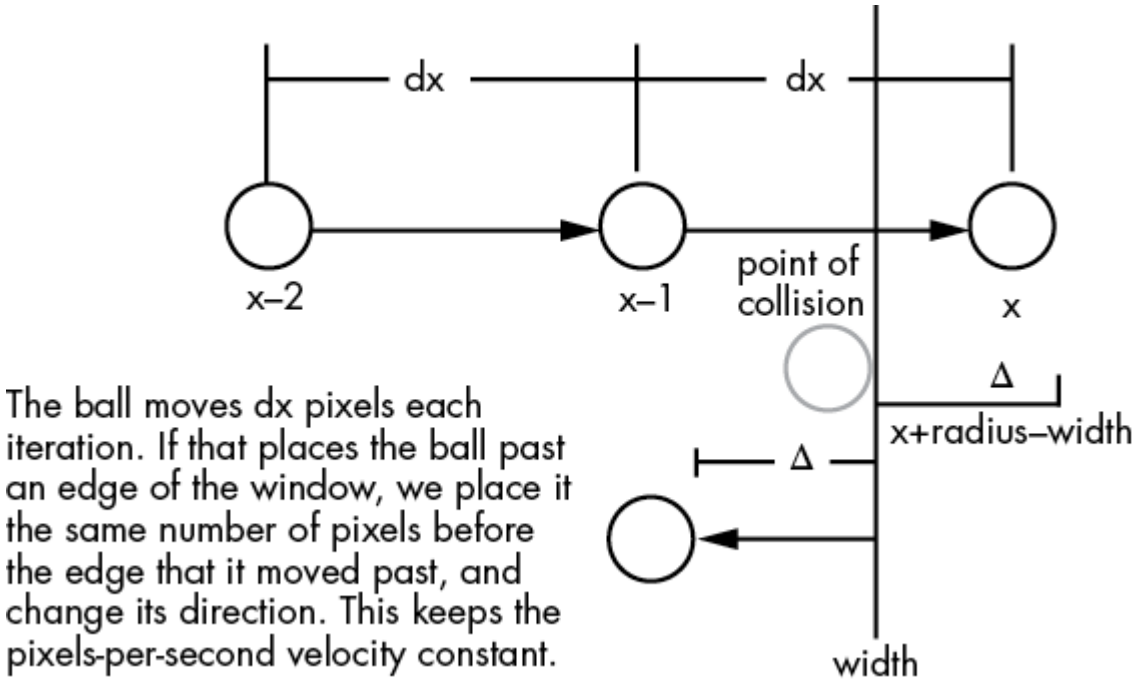


Figure 28-1: A fast-moving ball might overshoot a boundary before you can tell it to bounce back.

If the chosen `dx` value has the ball moving several diameters per frame, it can easily be on the left of the wall in one frame and on the right of the wall in the next. At some time in between, it must have collided with the wall. In that case, the amount the ball has overshot the wall should be found, and the ball should be placed an equivalent distance to the left of the wall, to simulate a bounce. We calculate that distance as `delta` (Δ), and it equals `(x + radius) - width` 1 for a circle. Given this distance, the ball's new, post-bounce x position is `width - delta - radius` 3, as shown at the bottom of *Figure 28-1*.

At the left side of the window, we know the ball has overshot the boundary when `x < radius` 4. In this case, we reposition the ball by setting `x` to `(2 * radius) - x` 5, and we reverse the ball's direction of motion.

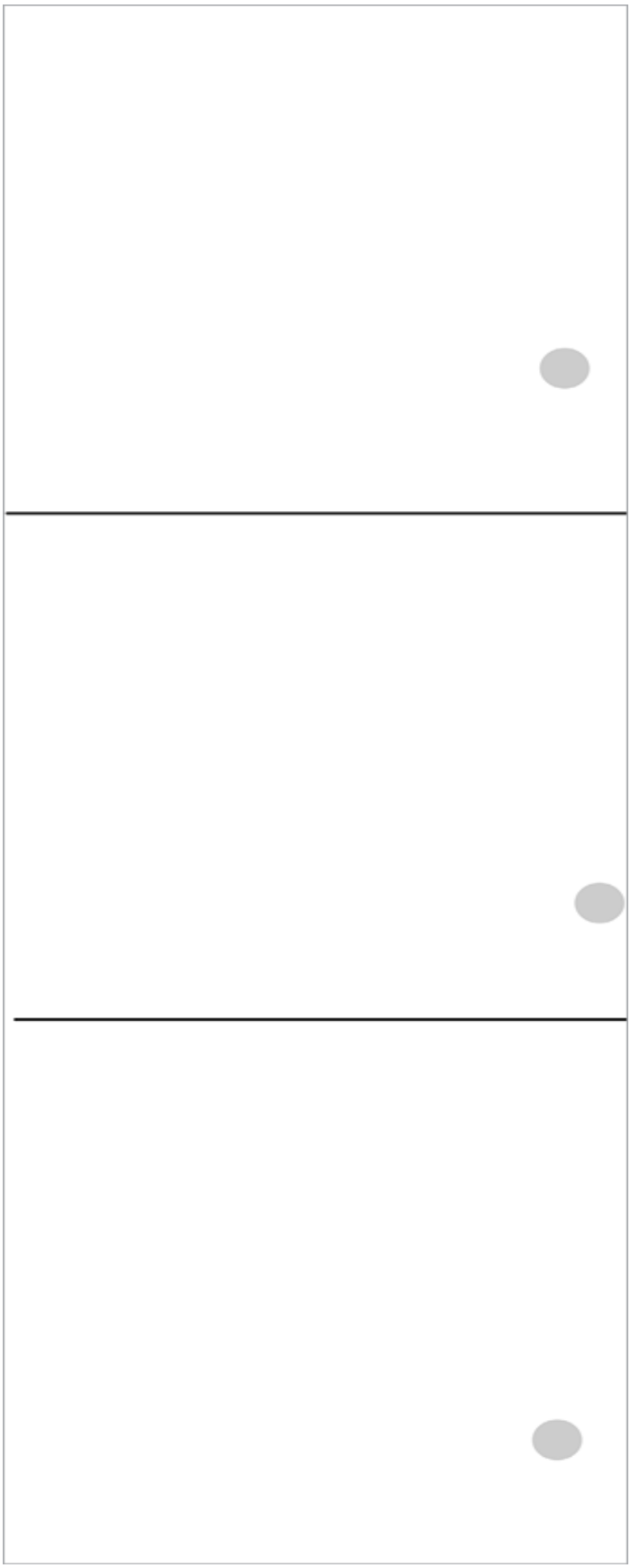
The vertical (*y*) situation is symmetrical 6.

NOTE

Most objects are not circular but can have a (virtual, invisible) circle drawn around them, and we can use this circle to detect collisions against the boundary.

```
int x=320, y=240;   // Coordinates of the circle (ball)
int radius=20;      // Size of the circle (ball)
int dx=42, dy=22;   // Speed of the circle (ball)

void setup ()
{
  size (640, 480);        // Typical window size
  fill (255, 0, 255);     // Magenta fill
  noStroke();             // Don't draw outlines
}

void draw ()
{
  background (255);                     // White
background
  ellipse (x, y, radius*2, radius*2);  // Draw the ball
  x = x + dx;  y = y + dy;              // Move
  xbounce();
  ybounce();
}

void xbounce ()
{
  int delta = 0;
1 delta = (x+radius) - width;
2 if (x+radius > width)  // right side
  {
  3 x = width-delta-radius;
    dx = -dx;
4 } else if (x < radius) // left side
  {
  5 x = (2*radius)-x;
    dx = -dx;             // Reverse x-direction
  }
}
6 void ybounce ()
{
  int delta = 0;
  delta = (y+radius) - height;
  if (y < radius)               // top side
  {
    y = (2*radius)-y;
    dy = -dy;
  } else if (y+radius > height) // bottom side
```

```
  {
    y = height-delta-radius;
    dy = -dy;                    // Reverse y-direction
  }
}
```



Sketch 29: Basic Sprite Graphics

We can combine the previous two sketches to show how programmers move sprites about in computer games. A *sprite* is a relatively low-resolution graphic that represents an object in a game. Sprites are usually primitive shapes or imported images. If the latter, the sprite image must have a transparent color so that we can see the background behind the sprite; otherwise the sprite would look like a rectangle of solid color with an image within it.

This sketch uses the rocket of Sketch 27 as the sprite and the code of Sketch 28 to move it about in the window. The rocket will move over a background image of stars to complete the game-like appearance.

The test to see whether the rocket has reached a side differs from the circle example because the sprite is a rectangular image drawn from the upper-left corner, and the distance to the boundary differs between left/right and up/down. The test against the left edge is nearly the same as before, but the offset by the radius is missing because the x-coordinate is on the left side of the sprite and not at its center 2:

```
if (px < 0) // left side
{
  px = -px;
  dx = -dx;          // Reverse x-direction
}
```

The test on the right is different because the entire width of the sprite is also to the right of the coordinate `px` 1:

```
delta = (px+sprite.width) - width;
if (delta > 0) // right side
{
  px = width-delta-sprite.width;
  dx = -dx;
}
```

So `px+sprite.width` is the coordinate for the right side of the sprite.

The checks are symmetrical for the y-coordinate 3.

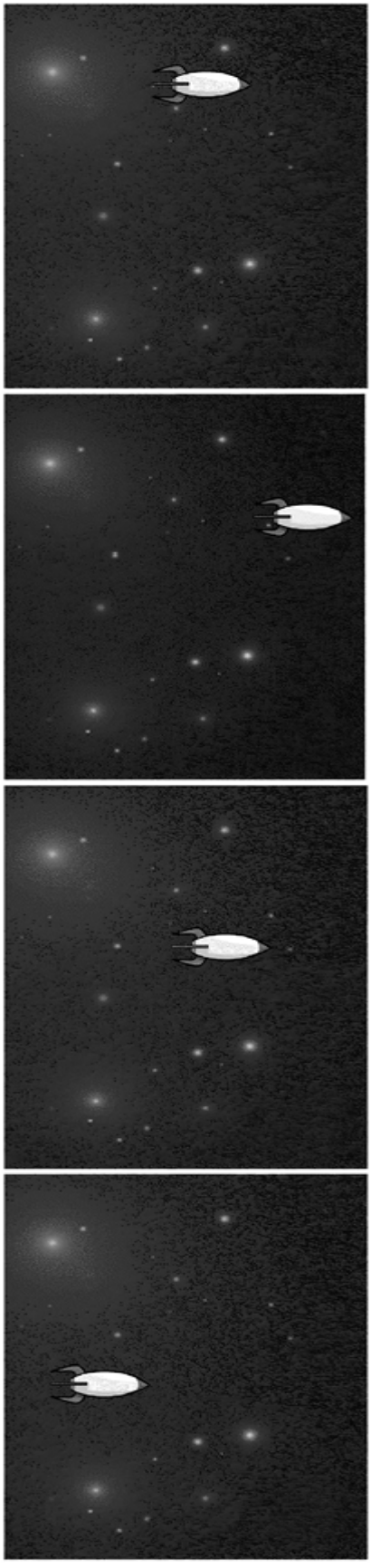
NOTE

Most games allow the player to move one or more of the sprites. The convention is to do this using key presses: W for up, A for left, D for right, and S for down. You'd put the code to move the sprite in the function `keyPressed()`*:*

```
void keyPressed()
{
  if (key == 'w') py = py - 1;
  if (key == 's') py = py + 1;
  if (key == 'd') px = px + 1;
  if (key == 'a') px = px - 1;
}
```

```
PImage img1, sprite;
color c=color(0,0,0);
int px=100, py=100, dx=2, dy=1;

void setup ()
{
  size(100,100);
  surface.setResizable(true);
  img1 = loadImage ("background.bmp");
  surface.setSize (img1.width, img1.height);
  sprite = loadImage("image.gif");
  sprite.resize (90, 50);
}

void draw ()
{
  background (255);
  image (img1, 0, 0);
  image (sprite, px, py);
  px = px + dx; py = py + dy;
  xbounce(); ybounce();
}

void xbounce ()
{
  int delta;
  delta = (px+sprite.width) - width;
1 if (delta > 0)     // right side
  {
    px = width-delta-sprite.width;
    dx = -dx;
2 } else if (px < 0) // left side
  {
    px = -px;
    dx = -dx;        // Reverse x-direction
  }
}
void ybounce ()
{
  int delta;
  delta = (py+sprite.height) - height;
3 if (py < 0)           // top side
  {
    py = -py;
    dy = -dy;
```

```
  } else if (delta > 0) // bottom side
  {
    py = height-delta-sprite.height;
    dy = -dy;           // Reverse y-direction
  }
}
```



Sketch 30: Detecting Sprite-Sprite Collisions

It is a relatively simple matter to decide whether a sprite is still within a window, because the size of the window remains fixed and the window doesn't move. But what if there were many sprites moving at the same time? How would we determine if any two had collided when both were moving? The situation of circular objects is the simplest and is a general solution, so this sketch will handle an arbitrary number of circular objects (balls) that will bounce off the boundaries and each other.

The coordinates of each ball will be stored in the `xpos[]` and `ypos[]` arrays 1. Drawing object `i` is simple 2:

```
ellipse (xpos[i], ypos[i], 10, 10);
```

Any two objects collide if they get nearer to each other than twice the radius, or in this case 10 pixels. These are the steps in the sketch:

- Define positions and speeds (`dx`, `dy`) for each of `nballs` objects.
- Each step (frame) is defined by a call to `draw()`. First, draw a circle at each location `xpos[i]`, `ypos[i]` 2.
- Change the position: `xpos[i] = xpos[i] + dx[i]`, and the same for `y` 3.
- Check for a collision with the boundary (bounce), and if there is one, implement the reaction to the collision. A bounce? An explosion? 4.

For each ball, check the distance between it and every other ball. If the distance is less than twice the radius, then change the direction of both balls (implementing a collision as a bounce) 5.

And that's it. The `bounce()` function 6 is a little different from the previous one, but it effectively does the same thing. The `distance()` function calculates the Euclidean distance between the two balls, as you saw in Sketch 24. If two balls overlap after bouncing, they could stick together until they collide with another ball.

NOTE

A rectangular object N×M *pixels in size (*N > M*) has a circle that surrounds it that can be used to check collisions. The center is (*N*/2,* M*/2) and the width is* N*. Using a bounding circle is not precise, but it is quick. The enclosing circle for the spaceship in Sketch 29 is shown in Figure 30-1.*

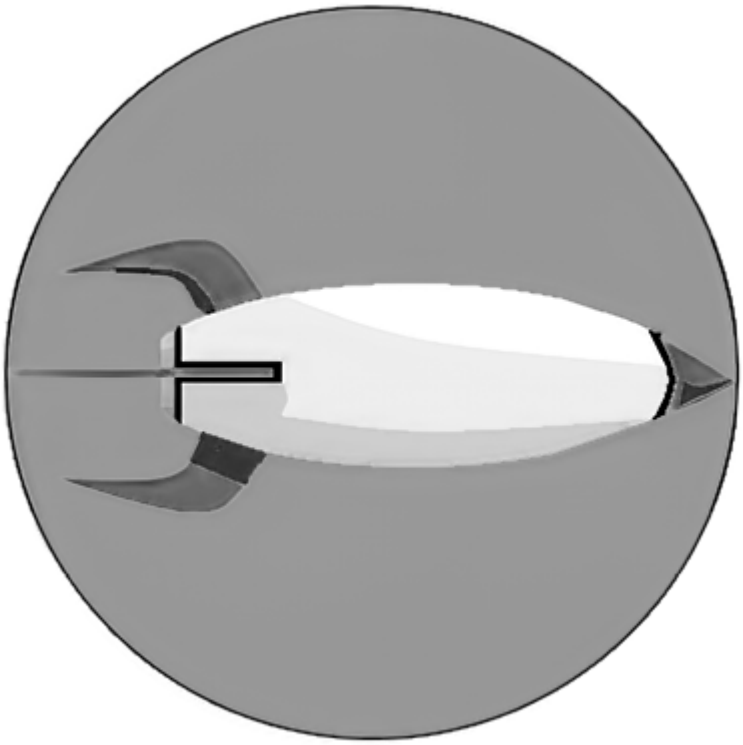
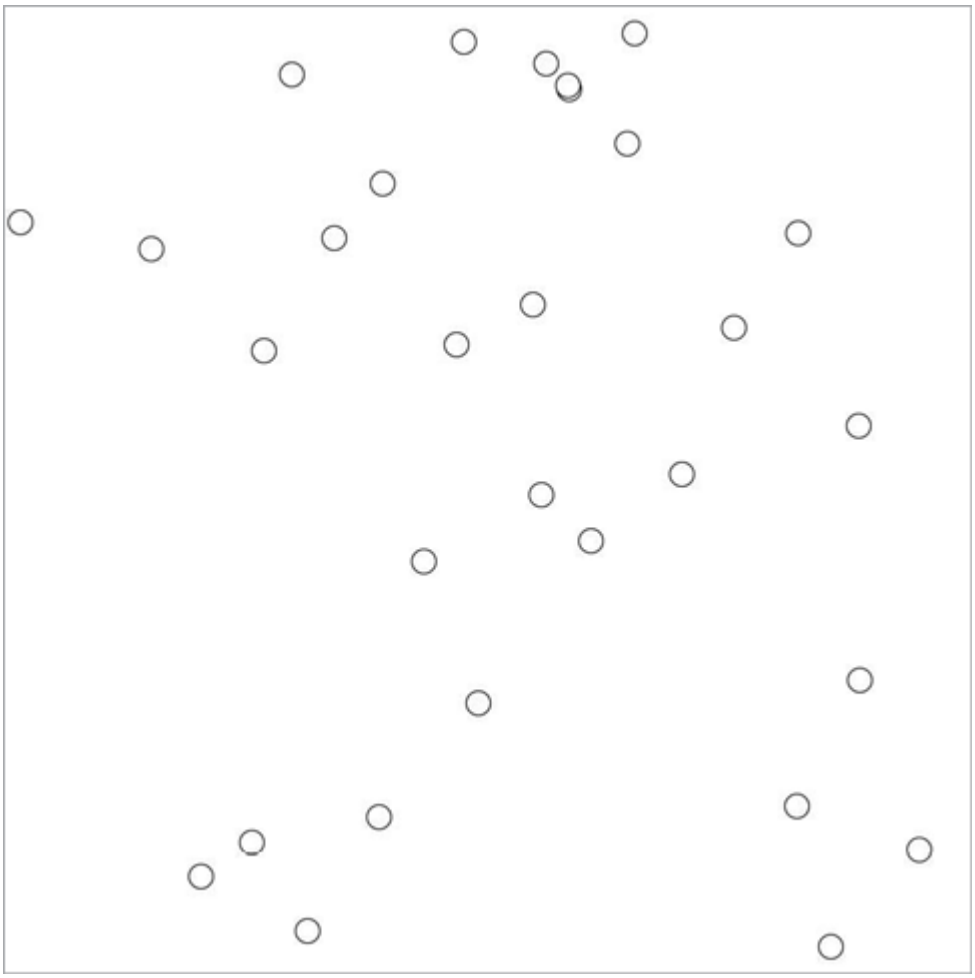


Figure 30-1: The enclosing circle for a rectangular object

```
int MAXBALLS = 100;
1 int []xpos = new int[MAXBALLS];
int []ypos = new int[MAXBALLS];
int nballs = 30;
int []dx= new int[MAXBALLS];
int []dy = new int[MAXBALLS];
void setup ()
{
  size (400, 400);
  for (int i=0; i<nballs; i=i+1)
  {
    xpos[i] = (int)random(width-10)+5;
    ypos[i] = (int)random(height-10)+5;
    dx[i] = (int)random(10)-5;
    dy[i] = (int)random(10)-5;
  }
}

void draw ()
{
  background (255);
  for (int i = 0; i<nballs; i++)
  {
 2 ellipse (xpos[i], ypos[i], 10, 10);    xpos[i] =
xpos[i] + dx[i];
 3 ypos[i] = ypos[i] + dy[i];
 4 bounce(i);
  }
  for (int i=0; i<nballs; i++)
    for (int j=i+1; j<nballs; j++)
   5 if (distance (xpos[i], ypos[i], xpos[j], ypos[j])
< 10)
      {
        dx[i] = -dx[i]; dy[i] = -dy[i];
        dx[j] = -dx[j]; dy[j] = -dy[j];
      }
}

float distance (int x0, int y0, int x1, int y1)
{ return sqrt ( (x0-x1)*(x0-x1) + (y0-y1)*(y0-y1) ); }

6 void bounce (int i)
{
  if (xpos[i] < 10) dx[i] = -dx[i];
```

```
  if (xpos[i] > width-10) dx[i] = -dx[i];
  if (ypos[i] < 10) dy[i] = -dy[i];
  if (ypos[i] > height-10) dy[i] = -dy[i];
  xpos[i] = xpos[i] + dx[i]; ypos[i] = ypos[i] + dy[i];
}
```



Sketch 31: Animation—Generating TV Static

We have used random numbers before, in Sketches 8 and 30. Random numbers serve a few important functions in games, simulations, and other software:

Nature uses unpredictable forms and shapes. Placing trees in a forest in a two-dimensional grid is a giveaway that there was a mind at work in the planting. This does not happen in nature. Instead, trees in a forest have an average distance from each other and seem otherwise to form a random collection.

Intelligent creatures do not behave predictably. Cars on a freeway that all behave in the same manner look very odd. Cars have random distances from each other, random speeds, and random behaviors within a possible range.

When playing poker or craps, the cards and dice ought to display random values, or the game is simply no fun.

This sketch draws a television set that looks as if it were tuned to a vacant channel. What is seen on the screen used to be called *snow*, and it is really pixels created by random voltages from signals received from space and various local electronic and electrical devices. We cannot predict what the TV will receive at any particular moment, so we draw a 2D set of random grey pixel values. This set of values changes every time the screen updates. There is an impression of random motion, rapid flashing of spots on the screen, but no organized images.

First, we display a background image of a TV set 1 and then set the pixels within the screen section to random black/white values each time `draw()` is called 3:

```
if (random(3)<1) set (i, j, BLACK);
  else set (i, j, WHITE);
```

To make it appear as though a channel were poorly tuned in, we could display an image faintly over the static by setting the alpha for the image to a low value, perhaps 30 or so. The static would be visible through the

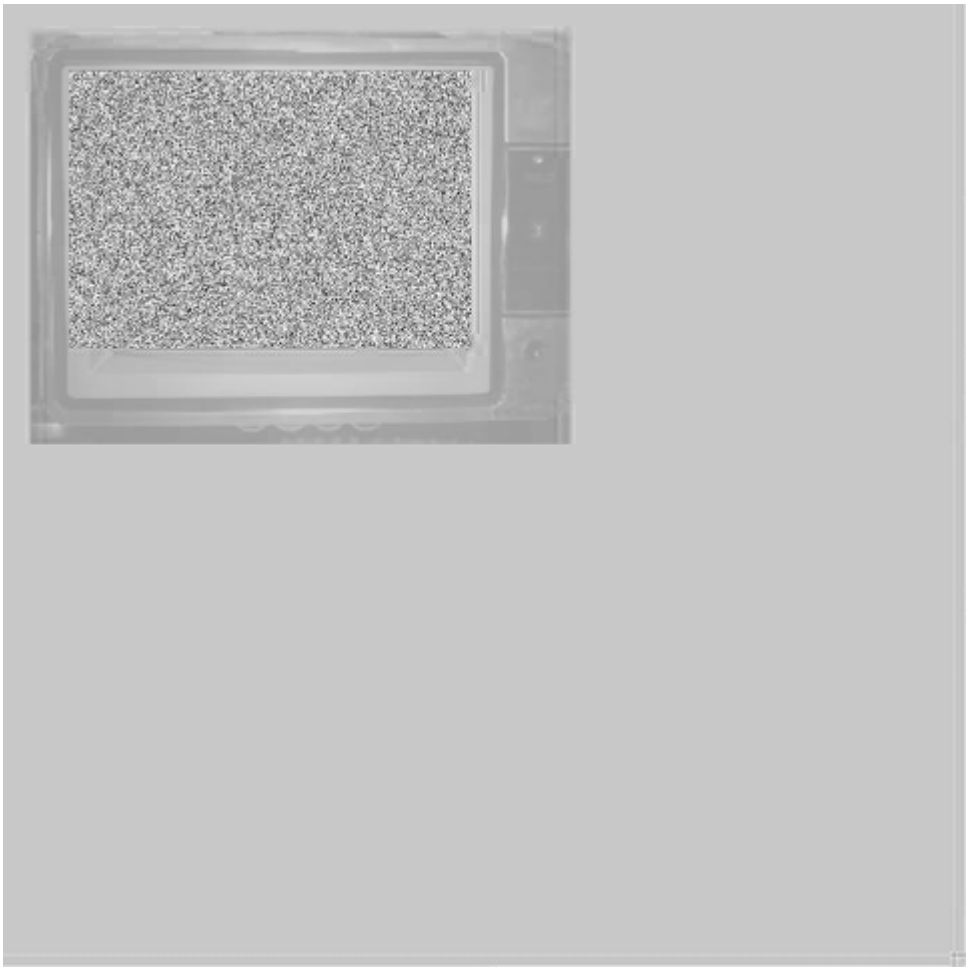
image. The `tint()` function changes the color and transparency of whatever is drawn from then on, so we could use it to change the transparency of the channel image, as follows:

```
tint (255, 255, 255, 127);
image (back, 49, 49);
```

The parameters to `tint()` are color coordinates, the first three being RGB and the fourth transparency (alpha). In the preceding example, the color is white (no actual tint) but the transparency is 127, which is half transparent.

In the code for this sketch, the tint and TV image are commented out. To see the image, remove the comment characters from those two lines 2.

```
PImage tv;
PImage back;
int x0=250, y0=445;
color WHITE = color (255, 255, 255, 90);
color BLACK = color (0,0,0, 90);

void setup ()
{
  size(350, 250);
  tv = loadImage("tv.jpg");  // Load TV set image
  back = loadImage ("screen.jpg");
}

void draw ()
{
  background (90, 90, 200);   // Blue background
1 image (tv, 20, 20);         // Display the TV
  snow (20, 20);               // Display random pixels
on the screen
2 //  tint (255, 60);
//  image (back, 49, 49);
}

// Display random black/white pixels
void snow(int x, int y)
{
  for (int i=x+29; i<x+160; i++)     // TV screen
coordinate offsets fixed
    for (int j=y+29; j<y+115; j++)   // at UL = 29,29
and LR = 152,115
   3 if (random(3)<1) set (i, j, color(0,0,0,4));
       else set (i,j, WHITE);
}
```



Sketch 32: Frame Animation

Animation involves displaying a sequence of still images on the screen at such a rate that the human visual system interpolates changes in position in the images and perceives motion. It is an illusion, in much the same way that any motion picture is an illusion. The previous sketch animated a display in a very basic manner, creating the illusion of random TV images by generating them with code. Most animations require that an image sequence be created by an artist and then displayed as a sequence.

For a Processing sketch to display an animation, the program has to read in the images (*frames*) to be displayed and then display them one after the other. The set of frames can be stored in an array of `PImage` values, one per frame.

The two examples in this sketch use an image sequence that represents the gait of a human; the 11 images compose one entire cycle of a single step, and repeating them makes it appear as if the character is walking.

Example A

Eleven images, named *a000.bmp* through *a010.bmp*, represent the animation. The program reads the images into consecutive elements of the `frames` array 1. The `draw()` function displays the next image in sequence each time it's called, increasing an index variable `n` from 0 to 10 and decreasing it to 0 again repeatedly 2.

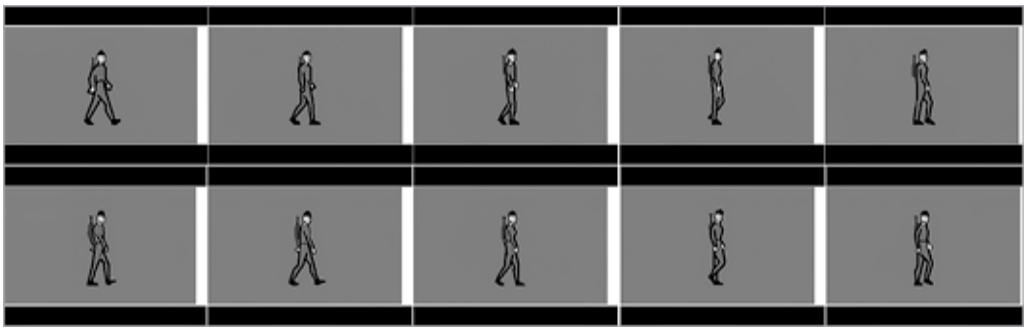
Example B

In Example A we needed to know in advance how many images belonged to the animation. In Example B we only require that the names of the files begin with *a000.bmp* and that the number increases by one for consecutive images. When the program fails to read an image file, as indicated by the fact that `loadImage()` returns `null`, the program presumes that all of the images have been loaded 1. The program counts the images as they are read and then displays them as before.

The loop within which the images are loaded has a `break 2` statement in it to escape the loop when `null` is detected.

Example A

```
PImage []frames = new PImage[12];
int nFrames = 11, n=0;
void setup ()
{
  size(100,100);
  surface.setResizable(true);
1 frames[0] = loadImage("a000.bmp");
  frames[1] = loadImage("a001.bmp");
  frames[2] = loadImage("a002.bmp");
  frames[3] = loadImage("a003.bmp");
  frames[4] = loadImage("a004.bmp");
  frames[5] = loadImage("a005.bmp");
  frames[6] = loadImage("a006.bmp");
  frames[7] = loadImage("a007.bmp");
  frames[8] = loadImage("a008.bmp");
  frames[9] = loadImage("a009.bmp");
  frames[10] = loadImage("a010.bmp");
  surface.setSize(frames[0].width, frames[0].height);
}
void draw ()
{
  frameRate (10);
2 image (frames[n], 0, 0);      // Display the Frame
  n = (n + 1)%nFrames;
}
```



Example B

```
int MAXFRAMES = 100;
PImage []frames = new PImage[MAXFRAMES];
int nFrames = 0, n=0;
void setup ()
{
```

```
  for (int i=0; i<MAXFRAMES; i++)
  {
    if (i<10)
      frames[i] = loadImage("a00"+i+".bmp");
    else
      frames[i] = loadImage("a0"+i+".bmp");
1 if (frames[i] == null)
    {
      nFrames = i;
2 break;
    }
  }
  size(100,100);
  surface.setResizable(true);
  surface.setSize(frames[0].width, frames[0].height);
}
void draw ()
{
  frameRate (10);
  image (frames[n], 0, 0); // Display the Frame
  n = (n + 1)%nFrames;
}
```

Sketch 33: Flood Fill—Filling in Complex Shapes

Drawing a rectangle or ellipse that is filled with a particular color is easy to do in Processing. You simply specify a fill color using the `fill()` function and then draw the shape. However, there's no function for filling an arbitrary shape or region, so let's make one. It has the advantage of showing you how filling is done in general.

This sketch reads an image with a white background that contains regions outlined with black (though you can use other colors). The regions do not have to be regular polygons, but they should be *closed*, in that there is an inside and an outside, with no gaps in the edges. When the user clicks on a pixel, the region surrounding that pixel will be filled with a random color.

The pixel that is clicked on has a color, the background color (`bgcolor` in the sketch). A random color will be selected for the fill color (variable `fillColor`). The goal is to set all of the pixels within the region that currently have the background color value to the fill color. The first step is to set the selected pixel to the fill color, followed by setting all neighboring pixels repeatedly, until no more candidates remain.

After the first pixel is changed, every background-colored pixel that is a neighbor of it is also set to the fill color 1. A *neighbor* is defined as a pixel that is immediately adjacent either vertically or horizontally. Then all of the pixels are scanned again, and any background pixel that is a neighbor of a fill-colored pixel is set to the fill color. The process is shown in *Figure 33-1*.

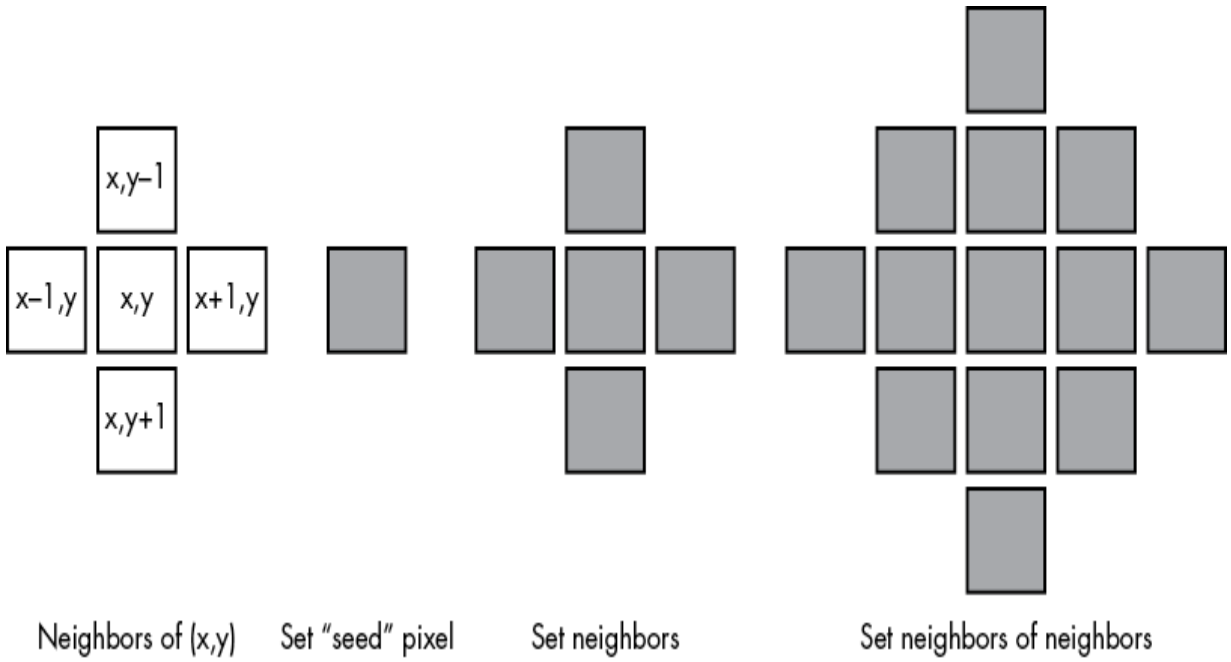


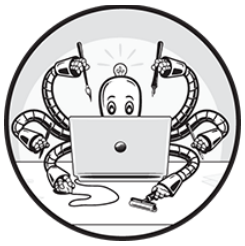
Figure 33-1: Filling in neighboring pixels

The process is repeated until no change is made. The process stops at the boundary because boundary pixels do not have the background color and are not changed. This is not the only method for implementing a fill, nor is it the fastest, but it is probably the easiest to comprehend.

The `mouseReleased()` function sets the values of the `bgColor` and `fillColor` variables and sets the first (*seed*) pixel to the fill color 3. The `nay()` function returns true if the pixel indicated by the parameters is a neighbor to a fill-colored pixel 2. Each time `draw()` is called (once per frame), it displays one iteration of the filling process, so the process appears animated.

```
PImage inputImage;
color bgColor, fillColor;

void setup ()
{
  size(100,100);
  surface.setResizable(true);
  inputImage = loadImage ("image.bmp");
  surface.setSize (inputImage.width,
inputImage.height);
  bgColor = inputImage.get(0,0);
  fillColor = color (40, 200, 30);
}

void draw ()
{
  image (inputImage, 0, 0);

  for (int i=0; i<inputImage.width; i++)
    for (int j=0; j<inputImage.height; j++)
      if ((inputImage.get(i,j)==bgColor) &&
nay(i,j,fillColor))
      {
1      inputImage.set(i,j,fillColor);
      }
}

2 boolean nay (int x, int y, int c)
  {
    if (get(x-1, y) == c) return true;
    if (get(x+1, y) == c) return true;
    if (get(x, y-1) == c) return true;
    if (get(x, y+1) == c) return true;
    return false;
  }

  void mouseReleased ()
  {
3   bgColor = get(mouseX, mouseY);
    fillColor = color
  (random(128,255),random(128,255),random(128,255));
    inputImage.set (mouseX, mouseY, fillColor);


  }
```

4
WORKING WITH TEXT AND FILES



Sketch 34: Fonts, Sizes, Character Properties

When text is drawn on the screen, there are many ways to draw each character. The size, weight, orientation, and style can vary widely. A *font* specifies a particular size, weight, and style of a typeface. Fonts are saved as files that contain the instructions for drawing each character. Bold, italic, normal, and each important size are individual files. The font name, a style, and a size are frequently part of the filename.

Processing allows many fonts, but each one must be set up in advance as a file using the Tools menu. Select **Tools▸Create Font** to open a font-creation window, within which you can choose the font name, style, and size, as shown in *Figure 34-1*.

Figure 34-1: Setting up a font

Select CourierNewPS-BoldMT with size 48 and click **OK** to create a file named *CourierNewPS-BoldMT-48.vlw* inside a local directory named *data*. You can repeat this process as often as needed, creating many font files. You need font files in order to load and use fonts in Processing.

Using a font is a somewhat involved process. You need to first create a variable of type `PFont` (Processing font) for each font desired, and then load the font using the `loadFont()` function 1:

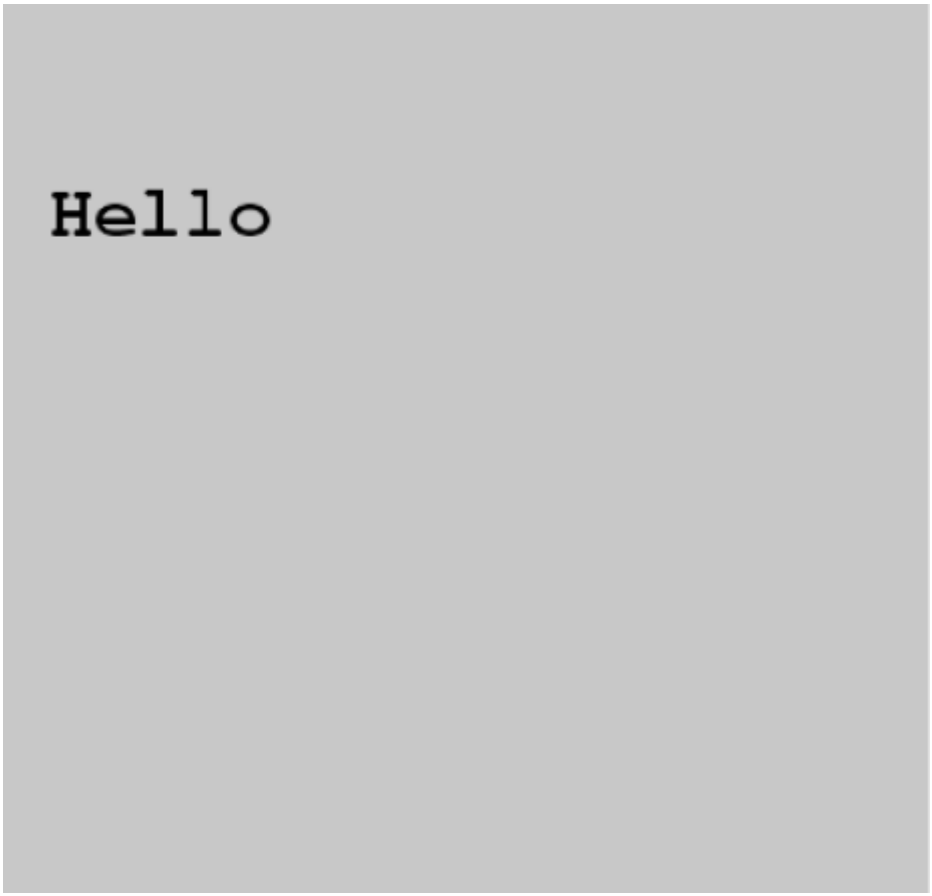
```
PFont font1;
font1 = loadFont ("CourierNewPS-BoldMT-48.vlw");
```

To establish a font as the one to use, call `textFont()` with the font variable and desired size: `textFont(font1, 48)` 2. The size is specified in pixels, not the standard for a font, which is *points*. Finally, you can always change the font size by calling `textSize(size)` 3.

This sketch loads the Courier Bold 48 font and establishes it. Then it draws the string “Hello” in sizes varying from 2 pixels to 55 pixels, changing by one pixel size each time `draw()` is called.

```
PFont font1;
int x=20, y=100;
int size = 55, ds=-1;

void setup ()
{
  size (200, 200);
1 font1 = loadFont ("CourierNewPS-BoldMT-48.vlw");
2 textFont (font1, 48);
  fill (0);
}

void draw ()
{
  background(200);
3 textSize (size);
  text ("Hello", x, y);
  size = size + ds;
  if (size < 2)  ds = -ds;
  if (size > 55) ds = -ds;
}
```


Hello


Hello

Sketch 35: Scrolling Text

A news *scroll* is a common feature of television news and weather stations. It is a summary of stories that scrolls from right to left across the bottom of the screen as other things are happening on the rest of the screen. It's common to see stock prices displayed in this way as well. How could we do this in a Processing sketch window?

First, the text for a particular item has an x-coordinate where it is drawn, and it will be drawn using the `text()` function. The y-coordinate is constant and will be somewhere near the bottom of the screen. In this sketch the screen is 400×200 and the y-coordinate for the text is 190. The x-coordinate changes.

The text to be displayed should start near the right side of the screen; for example, at `width-10` pixels 2. Each frame displayed should move the text to the left, so `draw()` will subtract one from `x` each time it is called:

```
text (s1, x, y);
x = x - 1;
```

There will usually be more than one message in the scroll. The first message could disappear before the second one is displayed, but this is unusual for a text scroll. Another idea is to have multiple scroll strings being drawn next to each other, moving in lockstep. So the strings themselves are in an array called `headlines` 1.

Suppose we have just two strings. Each one has an index into the array that accesses the strings (`i1` and `i2`) and x position (`x1`, `x2`). If the first string, `headlines[i1]`, is drawn at location `x1`, the second string should be drawn at location `x1` plus the number of pixels in the string `i1` plus a small space. In Processing terms, it looks like this:

```
x2 = x1 + (int)textWidth(headlines[i1])+ 10;
```

`textWidth()` is a function that takes a string as a parameter and, using the current font size, returns the width in pixels of that string when drawn.

The value 10 is the small space. When the first string disappears on the left of the screen, its plotted position plus its length will be less than 0 3:

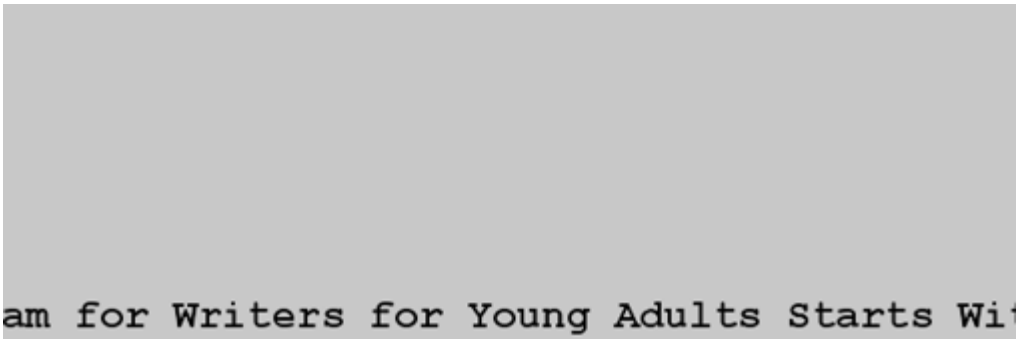
```
x1+textWidth(headlines[i1]) < 0
```

At this point, a new string (that is, the next index) should be obtained and positioned to the right of the second string:

```
i1 = (i2+1)%5;
x1 = x2 + (int)textWidth(headlines[i2])+ 10;
```

The same happens when the second string disappears on the left.

```
PFont font1;
int x1, y=190, x2;
int size = 55;
int i1, i2;
String []headlines = new String[5];

void setup ()
{
  size (400, 200);
  font1 = loadFont ("CourierNewPS-BoldMT-48.vlw");
  textFont (font1, 12);

  headlines[0] = "2 Die, 8 Hurt in Pasadena as Vehicle
Hits Crowd * ";
  headlines[1] = "L.A.'s Open Enrollment Plan Shrinks
for 5th Year * ";
  headlines[2] = "Program for Writers for Young Adults
Starts With Duo Behind 'Buffy' Books * ";
  headlines[3] = "Pickets Want Laguna Festival to Stay
Put *         ";
  headlines[4] = "3rd Whale in a Month Washes Up on
Coast *          ";
  fill (0);
  i1 = 0; i2 = 1;
  x1 = width-10;
  x2 = x1 + (int)textWidth(headlines[i1])+ 10;
}

void draw ()
{
  background(200);
  text (headlines[i1], x1, y);  text (headlines[i2],
x2, y);  x1 = x1 - 1; x2 = x2 - 1;
  if (x1+textWidth(headlines[i1]) < 0)
  {
    i1 = (i2+1)%5;
    x1 = x2 + (int)textWidth(headlines [i2])+ 10;
  }

  if (x2+textWidth(headlines[i2]) < 0)
  {
    i2 = (i1+1)%5;
    x2 = x1 + (int)textWidth(headlines [i1])+ 10;
  }
}
```


am for Writers for Young Adults Starts Wi

Sketch 36: Text Animation

Animating text can create an interesting effect. It has been used in commercials and by artists in the past, but it has never been as easy to do as it is now. A string can be drawn along a curved path, even a moving curved path; characters in the string can change in orientation, size, color, or even font. Motion can even vary according to user input, either by following the mouse or moving as a result of audio or video input.

A key to animating text is to access each character in the string using the `charAt()` function. The first character in the string `str` is returned by `str.charAt(0)`, the second character is `str.charAt(1)`, and so on. In this way, each character can be accessed individually and be made to behave in a different way from other characters.

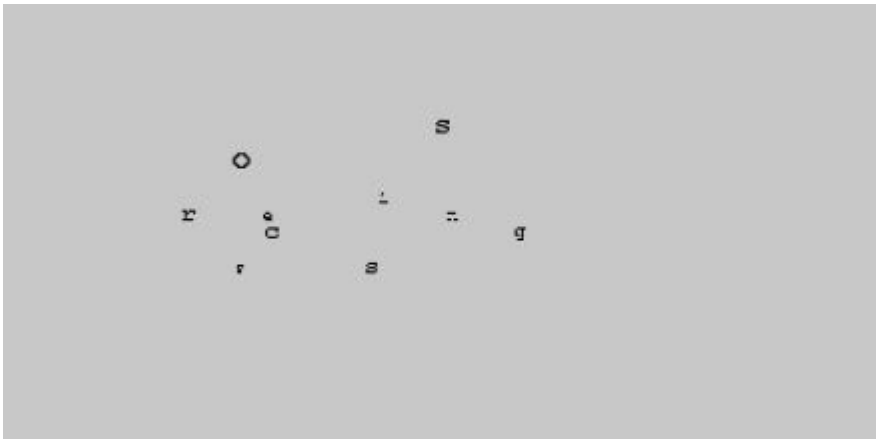
This sketch causes the word *Processing* to explode, the component letters flying in all directions at different speeds; character sizes change too. Each character has a distinct position (arrays `x` and `y`), velocity (arrays `dx` and `dy`), and size (array `size`) 1.

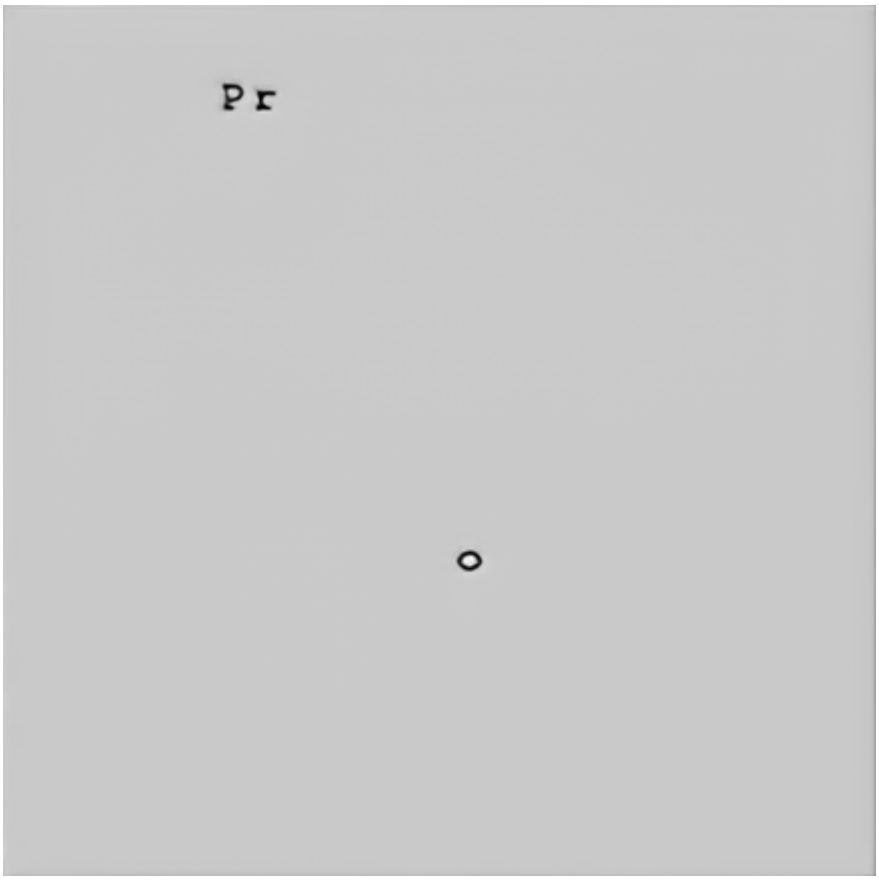
Initially, we draw the word *Processing* neatly in the center of the screen as a set of individual characters 2:

```
for (int i = 0; i<10; i++)
  text (s1.charAt(i), x[i], y[i]);
```

After a few seconds (60 frames) 3, we change the position of each character every frame 4, thus moving them, and we adjust individual sizes too. The characters move off in random directions, eventually disappearing from the screen.

```
PFont font1;
int count = 0;
int x[] = new int [10];
int y[] = new int [10];
int size [] = new int [10];
int dx[] = new int [10];
int dy[] = new int[10];
String s1 = "Processing";

void setup ()
{
  size (400, 200);
  font1 = loadFont ("CourierNewPS-BoldMT-48.vlw");
  textFont (font1, 12);
  for (int i=0; i<10; i++)
  {
    x[i] = 100+15*i; y[i] = 100;
    size[i] = 12;
    dx[i] = (int)(random(11)-6);
    dy[i] = (int)(random(11)-6);
  }
  fill (0);
}

void draw ()
{
  background(200);
  for (int i = 0; i<10; i++)
    if (size[i] > 0)
    {
     textSize(size[i]);
     text (s1.charAt(i), x[i], y[i]);
    }
  count = count + 1;
  if (count > 60)
    for (int i=0; i<10; i++)
    {
      x[i] = x[i] + dx[i];
       y[i] = y[i] + dy[i];
       size[i] = size[i] + (int)(random (5)-3);
    }
}
```





Sketch 37: Inputting a Filename

All of the sketches developed in this book so far use filename constants when reading an image. To be more flexible, most programs allow the user to enter a command or filename, even a number, from the keyboard, and that user input directs the code to use specific data. This is our next task—to ask the user to enter an image filename from the keyboard and display that image in the sketch window.

We already know that the `keyPressed()` function is called whenever the user presses a key, and the variable `key` contains the character that represents the key that was pressed, at least for letters and numbers. Other keys, like arrow keys, use a keycode value, like `ENTER` or `BACKSPACE`, to tell us what the key is. Given these facts, one way to read a user-given filename would be to append the characters typed by the user to a string and, when we see the ENTER value, to use the preceding string as a filename. This should work fine, but we need to handle some conventions.

First, the user needs to see what they are typing. The string that the user has entered so far must appear somewhere on the screen so that the user can see what has actually been typed.

Next, corrections must be possible. Traditionally one presses the BACKSPACE key to move backward over the string and delete characters so that new, correct ones can be entered, so we'll implement corrections using BACKSPACE. Finally, if an incorrect name is entered, a corresponding image file might not exist, and the user needs to be informed.

When the user types a letter or number, indicated by the variable `key`, we add that character to a string named `s` using the concatenation operation 3:

```
s = s + key;
```

If that character is a backspace and the string has characters in it, we remove the last character entered 1:

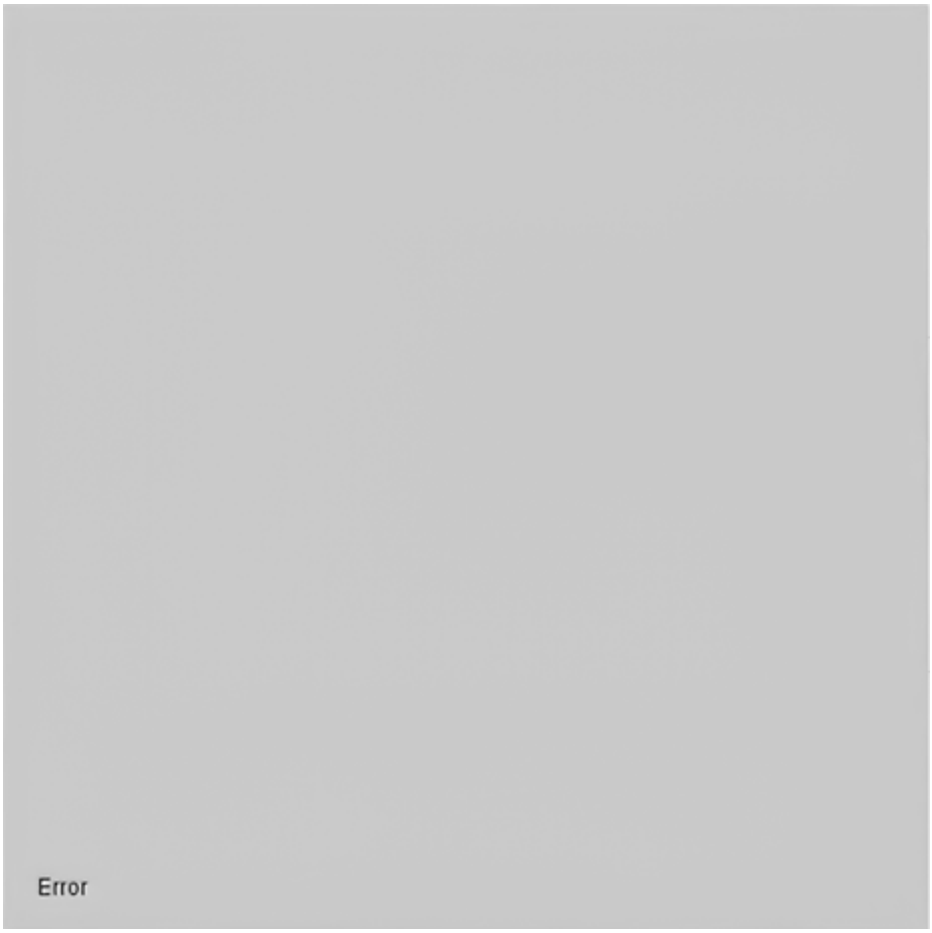
```
if (s.length()>0 && key==BACKSPACE)
s = s.substring (0, s.length()-1);
```

The `draw()` function will display this string each time the screen is updated, allowing the user to see the current string. Finally, if the key pressed was ENTER, then the string is complete and we should open and display the file. If `loadImage()` returns `null`, there is no such image, and the word `Error` is displayed in place of the filename 2.

```
if(key==ENTER || key==RETURN)
{
  img = loadImage (s);
  if (img == null) s = "Error";
}
```

```
PImage img;
String s = "";

void setup ()
{
  size(500, 500);
}

void draw ()
{
  background (200, 200, 200);
  if (img != null) image (img, 0, 0);
  fill(0);
  text (s, 20, height-20);
}

void keyPressed()
{
  fill(0);
1 if (s.length()>0 && key==BACKSPACE)
  {
    s = s.substring (0, s.length()-1);
  }
 2 else if(key==ENTER || key==RETURN)
    {
      img = loadImage (s);
      if (img == null) s = "Error";
    } else
     3 s = s + key;
}
```

stars2.jpg


Error

Sketch 38: Inputting an Integer

In the previous sketch, we had the user enter a string from the keyboard, and we used the string as a filename. This is a basic use of a string—using a sequence of characters to communicate data to the computer and back. What if, instead of entering a filename, we wanted to specify some number of things to input? This would mean entering an integer. However, when a number is entered at the keyboard, the string is *not* the number but is a text representation of the number. To get the actual number, the characters that compose it have to be converted into numeric form.

The string "184" is an integer in string form, obviously representing the number one hundred eighty-four (184). This is one hundred plus eight tens plus four, or $10^2 + 8\times10^1 + 4\times10^0$. To convert from string form into numeric form, we need to peel off the digits one at a time and multiply by the correct power of 10.

We can take the first digit, 1, and add it to a sum. Then we take the next digit and add to the sum *multiplied by 10*; and repeat again and again until the incoming character is not a digit. The powers of 10 accumulate with the first digit representing the highest power and the final digit representing 10^0, or one.

This is the essential piece of code 1:

```
val = val * 10 + (key-'0');
```

The expression `key-'0'`, where `key` is a digit, represents the numeric value of a digit character (that is, from 0 to 9). Assuming that `val` is initially 0, we get this after the user types `'1'`:

```
val = 0*10 + ('1'-'0') = 0 + 1 = 1
```

Now the user types `'8'`, and we get this:

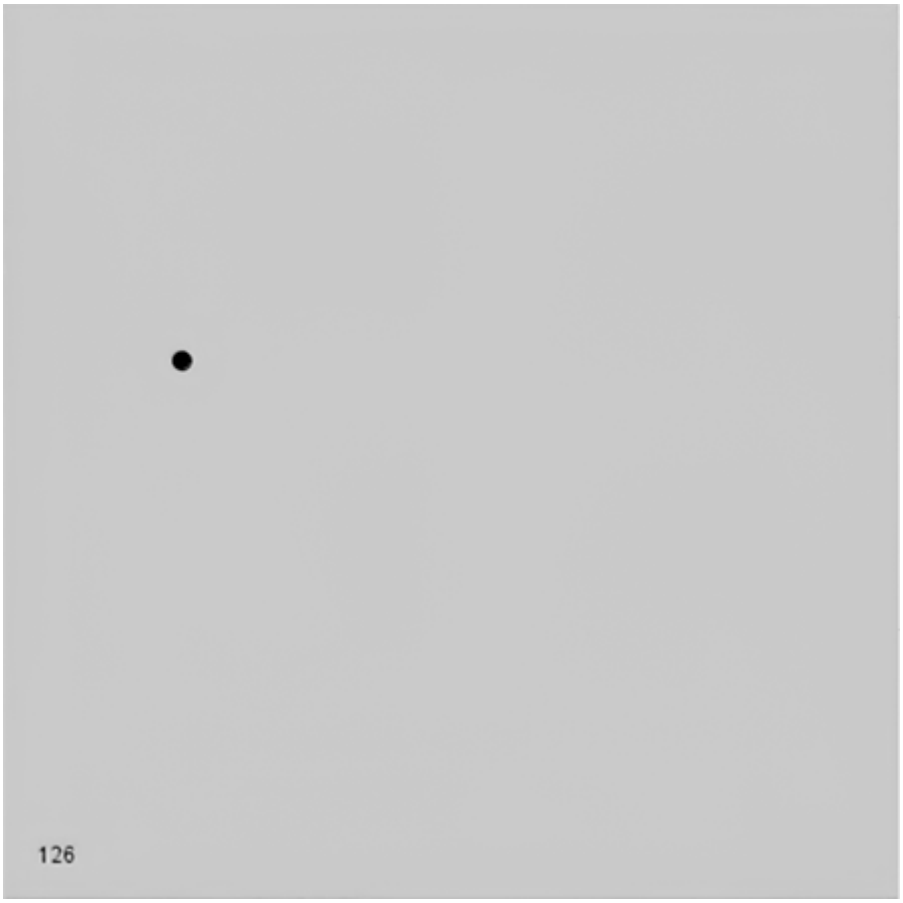
```
val = 1 * 10 + ('8'-'0') = 10 + 8 = 18
```

Finally the user types `'4'`:

```
val = 18*10 + ('4'-'0') = 180 + 4 = 184
```

To make this sketch marginally useful, it allows us to enter two values, an *x* and a *y* value, and draws a circle at these coordinates. An error on entry sets the coordinate to 0.

```
String s = "";
int val = 0;
int x=-1, y=-1;

void setup ()
{
  size(500, 500);
}

void draw ()
{
  background (200, 200, 200);
  fill(0);
  text (s, 20, height-20);
  if (y>=0) ellipse (x, y, 10, 10);
}

void keyPressed()
{
  fill(0);
  if (s.length()>0 && key==BACKSPACE)
  {
    s = s.substring (0, s.length()-1);
    val = val/10;
  } else if(key==ENTER || key==RETURN)
    {
      if (x<0) x = val;
      else if (y<0) y = val;
      s = ""; val = 0;
    } else if ( (key>='0') && (key<='9'))
      {
        s = s + key;
     1  val = val * 10 + (key-'0');
      } else
        {
          s = "Error"; val = 0;
        }
}
```


126

Sketch 39: Reading Parameters from a File

Many computer programs save values in files for use when the program starts, or restarts. Initial values, locations for buttons and other interface objects, high scores for a game: all can be read from files when a program begins. Most people have had the experience of playing a computer game and saving the state so that they can resume playing at a later time; this also involves saving data in a file and then retrieving it later. This sketch retrieves the state of a game, albeit a simple one—checkers—from a text file that contains the positions of all of the checkers in a game.

Checkers uses an 8×8 grid of squares on which disks of two colors, usually referred to as black and white, are placed. Only half of the squares are really used, and these squares also have two colors. Checkers can only sit on one of those colors, so the easy part of this sketch is to draw the squares and place checkers on those squares when it is known what the locations are. The new part is reading the data and interpreting that data as checker positions.

As a scheme for representing a checker board, imagine a set of squares with eight rows of eight columns each. A square can be indexed as (*i*, *j*), where *i* is the row and *j* is the column. The color of the checker on the square can be 0 for one color and 1 for another—the actual colors do not matter, only that checkers of color 0 belong to one player and the color 1 checkers belong to the other. The squares have fixed positions, but the checker locations are read from the file, which contains a row for the position and color of each checker, like this:

```
row col color   (e.g. 1 2 1)
row col color   (e.g. 1 4 1)
...
```

The file contains one-digit integers separated by single spaces, three per line. A structured format is easy to read and is, in fact, typical of data that has been created by a computer.

To read a file in Processing, we'll use the built-in function `loadStrings()`, which reads a set of strings from a file (given as a string

parameter), with one string being one line in the file. `loadStrings()` returns an array of strings that we'll assign to the variable `dlines` 2. To find the number of items in the array (the same as the number of lines of data in the file), we use the `length` property in `dlines`: `dlines.length`.

When a line is read in, we use it to place a checker on a square, and when all checkers are read in, we draw them on the screen. To place the checker, we extract the three integers from each string in `dlines` and then place the correct piece in the correct place using the row and column integers.

We convert the string data into numbers as follows 3:

```
y[i] = dlines[i].charAt(1) - '0';
```

Each piece is one of two colors, indicated by the variable `k[i]`. A checker is 20 pixels wide, so we draw one at location (`x[i], y[i]`) with these lines:

```
if (k[i]==0) fill (200,0,0); else fill (200,2000,0);  //
Color?
  ellipse (x[i]*40+20, y[i]*40+20, 20, 20);
```

The horizontal position is offset from the left by 20 pixels, and each successive position is 40 pixels further right. The expression `x[i]*40+20` gives the *x* location at which to draw checker number `i`. It is symmetrical for the vertical *y* position.

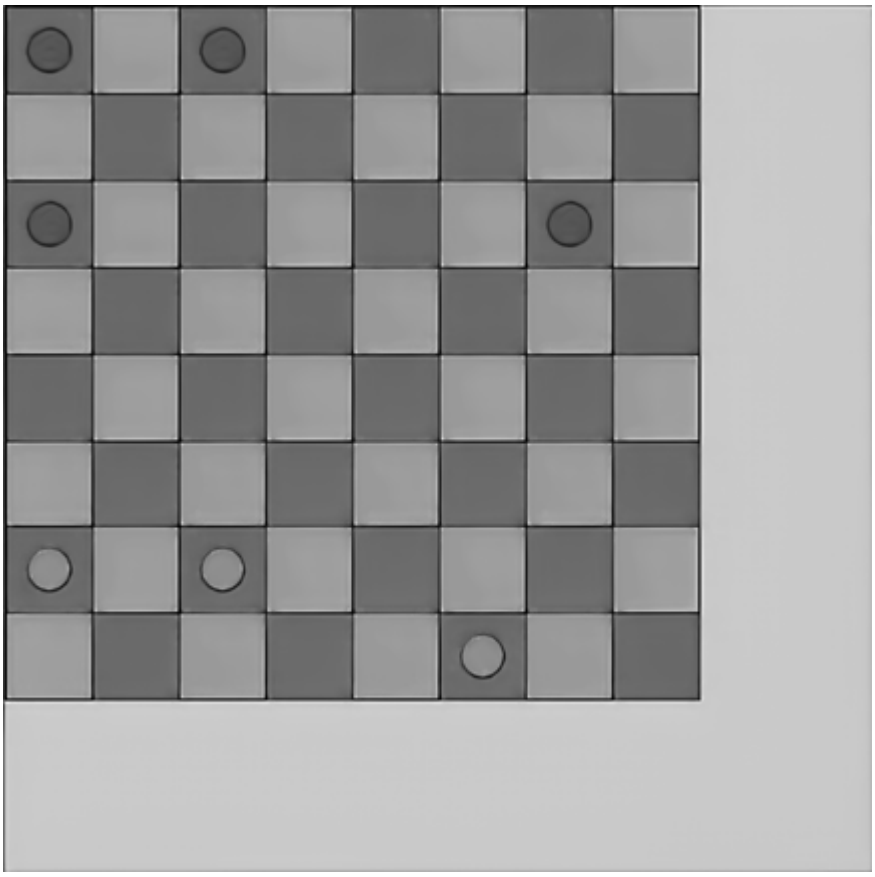
Squares are 40×40 pixels and alternate in color, so when we draw a red one, we toggle the fill color to that of the next square. After 8 squares, an extra toggle is done so that the colors alternate vertically as well. If `i` and `j` are the coordinates of a square, we draw it this way:

```
1 rect (i*40, j*40, 40, 40)
```

In the sketch, the checkers are red or green, and the squares are red or yellow.

```
String dlines[];
boolean errorFlag = false;
int []x = new int[12];    // Column for checker
int []y = new int[12];    // Row for checker
int []k = new int[12];    // Color of checker
int n = 0;

void setup ()
{
  size (400, 400);
  readFile ("save.txt");  // Read data
}

void draw ()
{
  background (200);
  board ();               // Draw the board
}

void board ()
{  // Draw the squares
  for (int i=0; i<8; i++)    // Columns
    for (int j=0; j<8; j++)  // Rows
    {  // Alternate the color for the squares
      if ((i+j)%2 == 0) fill (255, 0, 0);
        else fill (255, 255, 0);
   1 rect (i*40, j*40, 40, 40);
    }
    for (int i=0; i<n; i++)  // Draw the checkers
    {
      if (k[i]==0) fill (200, 0, 0); else fill (100,
200, 0);  // Color?
      ellipse (x[i]*40+20, y[i]*40+20, 20, 20);
// Location.
    }
}

void readFile (String fileName)
{
2 dlines = loadStrings(fileName);       // Read the
names as strings
  for (int i=0; i<dlines.length; i++)   //
dlines.length is how many items in the array
    {
   3 y[i] = dlines[i].charAt(1) - '0';
```

```
      x[i] = dlines[i].charAt(3) - '0';
      k[i] = dlines[i].charAt(5) - '0';
    }
  n=dlines.length;
}
```



Sketch 40: Writing Text to a File

Computer programs use text to tell their users what is going on. Sometimes, as in the previous sketch, they use text to save the state of the program, often a game; sometimes the program writes numerical results or records the progress of a program. Text is a typical and natural way for computers to communicate with humans.

Here's the problem to be solved: we want to simulate a ball on the screen, moving at a constant speed, as was done in Sketch 28; write the position of the ball to a file during each frame; and record when the ball collides with the edge of the screen.

The output method that corresponds to `loadStrings()` is the function `saveStrings()`. We'll declare an array of strings, where each string will be written as a line of text to the file. When a ball position is to be saved, a string is created that represents the position, and it is stored in one of the array locations. Then the array index is incremented so the next string goes in the next location 2.

```
data[index] = "(X0,Y0)= ("+x0+","+y0+")";
index = index+1;
```

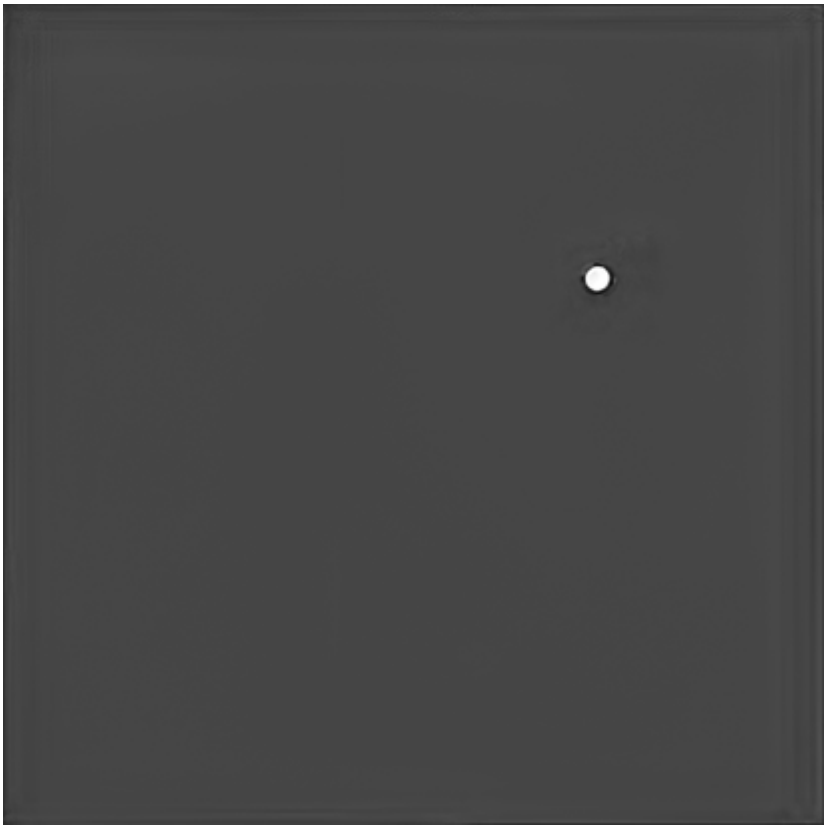
When the ball collides with a side of the screen, we put a message like "Collision left" in the array and then increment the index 1.

When the array is full, which happens when the index is greater than 499, `saveStrings()` writes all of the strings to a file and ends the program 3:

```
saveStrings("save.txt", data);
```

It is not possible to add more to a file using `saveStrings()` after the file has been closed; if you call it again with the same filename, it will overwrite the file. So you must save everything first, and then write it all out at once. With 500 strings, you can record about 7 seconds.

```
String []data = new String[501];
int x0=40, y0=90, index, dx0=3, dy0=2;
void setup ()
{  size (300, 300); }

void draw ()
{
  background (40, 40, 190);
  ellipse (x0, y0, 10, 10);
  x0 = x0 + dx0; y0 = y0 + dy0;
  if (x0<10)
  {
    dx0 = -dx0;
1 data[index] = "Collision left"; index= index+1;
  }
  if (x0>width-10)
  {
    dx0 = -dx0;
    data[index] = "Collision right"; index= index+1;
  }
  if (y0<0)
  {
    dy0 = -dy0;
    data[index] = "Collision top"; index= index+1;
  }
  if (y0>width-10)
  {
    dy0 = -dy0;
    data[index] = "Collision bottom"; index= index+1;
  }
2 data[index] = "(X0,Y0)= ("+x0+","+y0+")";
  index = index+1;
  if (index > 499)
  {
3 saveStrings("save.txt", data);
    exit();
  }
}
```



File Edit Format View Help

```
(X0,Y0)= (10,138)
Collision left
(X0,Y0)= (7,136)
(X0,Y0)= (10,134)
(X0,Y0)= (13,132)
(X0,Y0)= (16,130)
(X0,Y0)= (19,128)
(X0,Y0)= (22,126)
(X0,Y0)= (25,124)
(X0,Y0)= (28,122)
(X0,Y0)= (31,120)
(X0,Y0)= (34,118)
(X0,Y0)= (37,116)
(X0,Y0)= (40,114)
```

Ln 1,

Sketch 41: Simulating Text on a Computer Screen

Imagine working on a made-for-TV movie. It's about computers and hackers and programmers, and the actors playing the roles of the hackers are, well, *actors*. They don't know anything about programming. They can't type, and they certainly can't enter code. So, in the scenes where the camera is looking over the main character's shoulder at the screen while she types, we need a special effect—something that makes it appear as if she's coding. Do we use computer animation? That can be expensive. No, the usual trick is to use a simple program that displays text, specific text, no matter what keys are struck. That way the actors don't have to know anything except how to press a key.

Making this program in Processing is straightforward, given what we know so far. The program opens a window and initializes a string, `message`, to the text to be typed onto the screen 1, which could be read from a file. A variable `N` starts as `0` and indexes the string: every character up to character `N` has been typed and should appear on the screen. The `draw()` function draws all of the characters up to `N` each time it is called, one character at a time, spacing them (in the example) nine pixels apart horizontally.

To organize the text into lines, we use the "!" character to indicate where lines end. When the program sees that character in the string, it doesn't display it, but instead resets the *x* position to the starting value and increases the *y* position by 15 pixels (one line).

The `draw()` function outputs the text, starting at the statement 2:

```
for (int i=0; i<N; i++) // display the next character
```

Either it displays one of the characters in the string 4:

```
text (""+message.charAt(i), x, y);
x = x + 10;
```

or the character in the string is "!" and it begins a new line 3:

```
if (message.charAt(i) == '!')
{
  y = y + 15;     // Move vertically down to next line
  x = 15;         // and start over at pixel 15.
}
```

Finally, when a key is pressed, as indicated by Processing calling the `keyPressed()` function, the count value `N` increases by one so that one more character appears on the screen 6. Regardless of what character was typed, the predefined character in the `message` string will be displayed. If `N` exceeds the string length, the program can set `N` to `0`, which starts over again with a fresh screen, or further key presses could just be ignored.

NOTE

Another possibility is to have the text display automatically, one character at a time, without anything being typed. The code to do so appears in the Processing program at the end of the `draw()` *function but has been commented out* 5*:*

```
//  N = N + 1
//  if(N >= message.length()) N = 0;
```

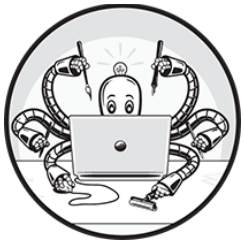
Removing the comment characters will make the text appear magically without a typist.

Note also that the background color is green because in older programming days, like the 1960s and 1970s, screens tended to be green. This is easy to change, and for a real movie application, the designers would specify the color they wanted.

```
int count = 0;
int N = 0;
int increment = 2;
String message;

void setup ()
{
    size (450, 500);
    background (0, 80, 0);
1 message = "Processing 3.5.4 September 2021. !// J
Parker - Sketch 041!";
    message = message + "void draw()!{!  boolean more =
true;!  int x, y;!"+
          "!  x = 15; y = 50;!  background (0, 80,
0);!"+
          "!    for (int i=0; i<N; i++)!    {!      if
(message.charAt(i) == '-')"+
          "!      {!        y = y + 15; !        x =
15;!      }!      else!"+
          "      {!        text (message.charAt(i), x,
y);!        x = x + 10;"+
          "!      }!    }!  !  count = count + 1;!  if
(count > increment) !  "+
          "{ count = 0; N++; }!"+
          "  if(N > message.length()) N = 0;!}!";
    message = message + "-- Abort at line 201 --!'   c
= chr(128)'!        ^!!!";
}

void draw()
{
  boolean more = true;
  int x, y;

  x = 15; y = 50;
  background (0, 80, 0);
2 for (int i=0; i<N; i++)
  {
3 if (message.charAt(i) == '!')
    {
      y = y + 15;
      x = 15;
    }
      else
      {
```

```
      4 text (""+message.charAt(i), x, y);
          x = x + 10;
        }
      }
5 // N=N+1;  if(N >= message.length()) N = 0;
}

void keyPressed()
{
6 N=(N+1)%message.length();
}
```

5
CREATING USER INTERFACES AND WIDGETS



Sketch 42: A Button

After text from the console or a file and basic mouse gestures, the simple button is the third most popular user input method. It is ubiquitous on web pages, game screens, and any system that requires on/off or yes/no choices from a user. It is, of course, based on the old-fashioned push button that has existed for a long time as an electrical device, and it works in a natural way: push the button and something happens.

Graphically, a button is really just a rectangle. It is usually filled with a color and has a text label or image to indicate its function. When the user clicks the mouse button while the cursor is within the button, the task assigned to the button is executed, usually by calling some function. Properties that a button has include its *position* (the x- and y-coordinates of the upper left corner of the button), *size* (the width and height of the button), *label* (the string that is written in the button), and a *color* or *image* that will appear in the button.

A button is said to be *armed* when the mouse cursor lies within it. When armed, a mouse click will execute the function of the button. Sometimes the button is drawn with a different color or font when it is armed to indicate the activation to the user.

The button implemented in this sketch causes the background color of the sketch window to change. It is armed when the mouse enters the rectangle 3:

```
if ( (mouseX>=bx) && (mouseX<bx+bw) && (mouseY>=by) &&
(mouseY<by+bh) )
```

where (`bx`, `by`) is the position and (`bw`, `bh`) is the size of the button.

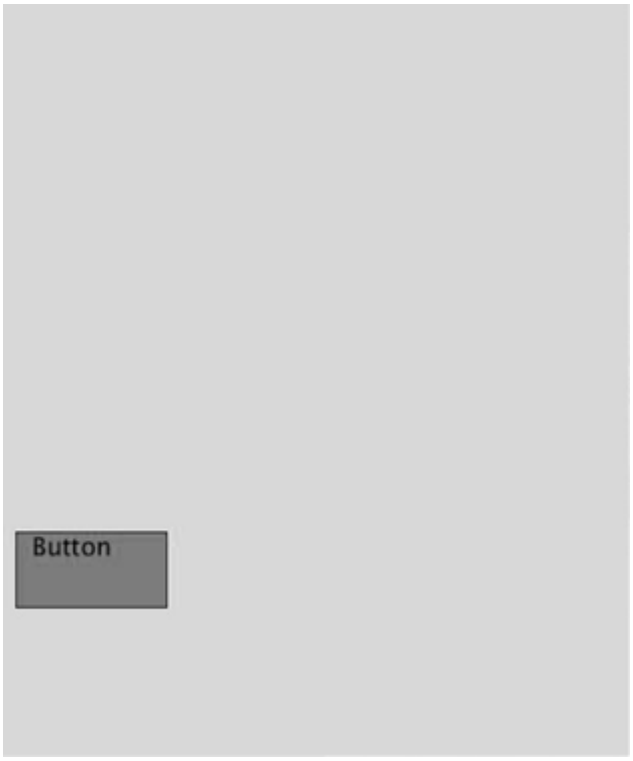
The `buttonArmed()` function returns true when this `if` condition is true. The `drawButton()` function draws and fills the rectangle and draws the text 1. When the button is armed, `drawButton()` also changes the fill color to green from red. And, of course, the `mousePressed()` function determines whether the button was armed when the mouse button was pressed and changes the background color if so 4.

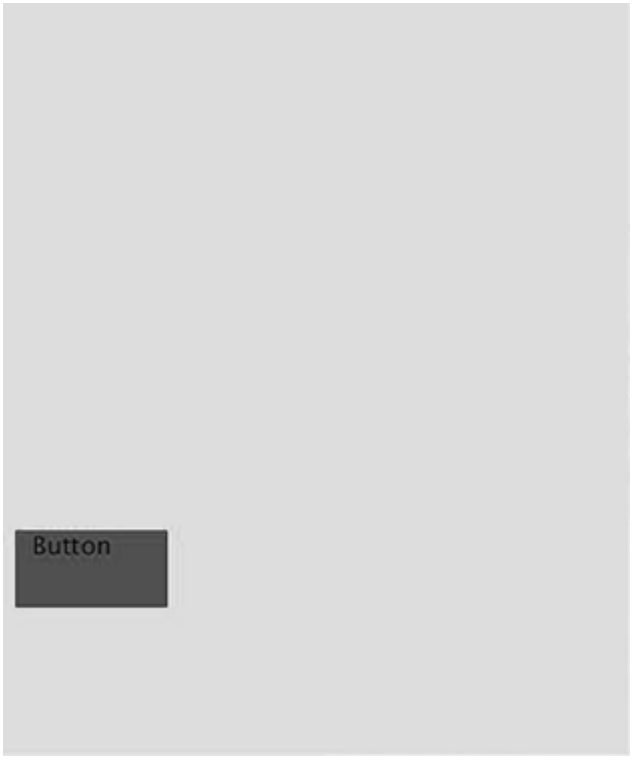
Because this sketch only implements a single button, it doesn't use much code. It is common for an application to have many buttons, as you'll see in the next sketch.

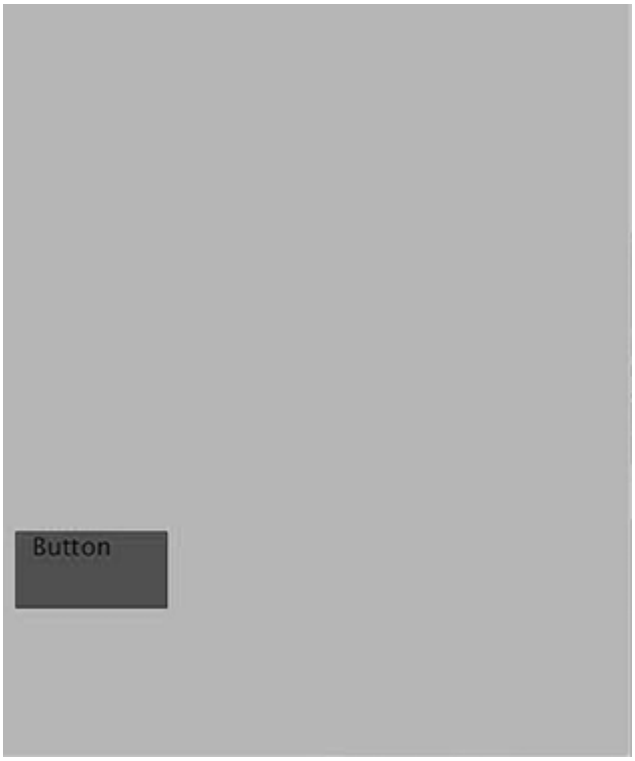
NOTE

Instead of drawing and filling a rectangle, you can draw an image to represent the button, using the `image()` *function instead of* `rect()` 2. *In this case, the unarmed and armed images would also be properties of the button. The test of the mouse coordinates is against the numeric values of the rectangle or image size and is independent of whether anything is actually drawn. The graphical rendition of the button is for the convenience of the user.*

```
color bgcolor = color (200, 200, 200);
int bx=10, by=260, bw=60, bh=30;

void setup ()
{
  size (250, 300);
}

void draw ()
{
  background (bgcolor);
  drawButton ();
}

1 void drawButton ()
{
  if (buttonArmed()) fill (20, 200, 40);
    else fill (200, 60, 80);
2 rect (bx, by, bw, bh);
  fill (0);
  text ("Button", bx+13, by+19);
}

boolean buttonArmed ()
{
3 if ( (mouseX>=bx) && (mouseX<bx+bw) &&
       (mouseY>=by) && (mouseY<by+bh) ) return true;
  return false;
}

void mousePressed ()
{
4 if (buttonArmed())
    bgcolor = color(random(128,255), random(128,255),
random(128,255));
}
```


Button


Button


Button

Sketch 43: The Class Object—Multiple Buttons

This sketch will create and display three buttons, one for each color component: red, green, and blue. When a button is clicked, the corresponding components of the background color will change randomly.

If an application needs many buttons, the scheme presented in Sketch 42 becomes awkward. What we want is a type, like `PImage` or `PFont`, that represents a button, so we can declare button variables or an array of buttons. The new `button` type should contain within it all of the properties of a button along with all of the code, written as functions, that performs the legal button operations.

Making a custom type with associated functions is done using a feature called a class. A *class* is a way to enclose some variables and functions and give them a name. The `button` class would look like this:

```
class button
{
  your code here
}
```

Inside the braces, we declare the variables used by the button: `x`, `y`, `width`, `height`, `label`, and so on. The functions `drawButton()` and `buttonArmed()` go inside the class too, along with something called a *constructor*: a function that is called automatically each time a new button (or, in general, a class object) is created. The `class` statement and what follows inside the braces declares the class as a custom type, and when you declare a variable of that class, you create an *instance*, one specific object that has the class variables and functions within it.

A variable of class `button` is declared just like a `PImage` variable:

```
button b1, b2, b3;
```

The next step, as with a `PImage` or `PFont`, is to create an instance of the `button` class using `new` and assign it to a variable:

```
b1 = new  button (100, 150, 90, 30, "Button");
```

When you use `new`, Processing calls the constructor for the class. The constructor accepts parameters, such as position or size, and saves them for later use in drawing the button. The constructor function has the same name as does the class 2 (in this case, `button`), and it has no function type—it is not preceded by `void` or a type name. The constructor itself has no return value, but the `new` operator will return a new instance of the class. If you define more than one constructor, Processing calls the one that matches the type and number of parameters given in the `new` statement. The constructor then returns a new instance of the class. You can create as many instances as your computer memory allows.

You access variables and functions in a class variable using *dot notation*. For the `button` class instance `bred` 1, the *x* position is `bred.bx`, and to draw it, you'd call `bred.draw()`. The main draw function must call `draw()` for each of the buttons, or they won't be displayed, and the `mousePressed()` function in the main program must check each button to see if it was clicked (that is, if the mouse cursor is inside the button) using the `armed()` function in each button.

```
color bgcolor = color (200, 200, 200);
button bred, bgreen, bblue;
void setup()
{
  size (450, 300);
1 bred = new button (10, 200, 50, 30, "Red");
  bgreen = new button (220, 200, 50, 30, "Green");
  bblue = new button (300, 200, 50, 30, "Blue");
}

void draw ()
{
  background (bgcolor);
  bred.draw();
  bgreen.draw();
  bblue.draw();
}

void mousePressed ()
{
  if (bred.armed()) bgcolor = color(random(128,255),
          green(bgcolor), blue(bgcolor));
  if (bgreen.armed()) bgcolor = color(red(bgcolor),
          random(128,255), blue(bgcolor));
  if (bblue.armed()) bgcolor = color(red(bgcolor),
          green(bgcolor), random(128,255));
}
class button
{
  int bx, by, bw, bh;
  color armedColor= color(20,200,20);
  color unarmedColor = color (200,200,40);
  String label;

2 button (int x, int y, int w, int h, String s)
  {
    bx = x;  by = y; bw = w;  bh = h;
    label = s;
  }

  void draw ()
  {
    if (armed()) fill (20, 200, 40);
      else fill (200, 60, 80);
    rect (bx, by, bw, bh);
```

```
    fill (0); text (label, bx+13, by+19);
  }

  boolean armed ()
  {
    if ( (mouseX>=bx) && (mouseX<bx+bw) &&
         (mouseY>=by) && (mouseY<by+bh) ) return true;
    return false;
  }
}
```


Red
Green
Blue

Sketch 44: A Slider

A *slider* is a user interface widget that allows the user to move a small object (a *cursor*) along a linear path, either horizontally or vertically. The relative position of the cursor along the path represents a number. The cursor in one extreme position corresponds to the minimum value, and the cursor in the other extreme position represents the maximum. If the cursor is halfway between the min and max positions, the value associated with the slider is halfway between the min and max values.

This widget can be used to position a large image in a small window or a lot of text within a smaller area, and we call it a scroll bar in those cases. The purpose of a slider is, more generally, to allow the user to select a number geometrically by sliding a cursor between two limits, rather than typing it. It is a natural idea to choose a number as a fraction of a total, or as a part of a range of values. If we define `sliderPos` as the position of the cursor in pixels from the start of the slider, `sliderWidth` as the width of the slider in pixels, and `sliderMax` and `sliderMin` as the numerical values associated with the min and max cursor positions, this is the selected value 3:

```
value = (int)(((float)sliderPos/sliderWidth)*sliderMax +
sliderMin);
```

This expression is based on the fact that the slider position is a fraction of the total possible set of positions, and this represents the same fraction of the range between the `sliderMin` and `sliderMax` values (see *Figure 44-1*).

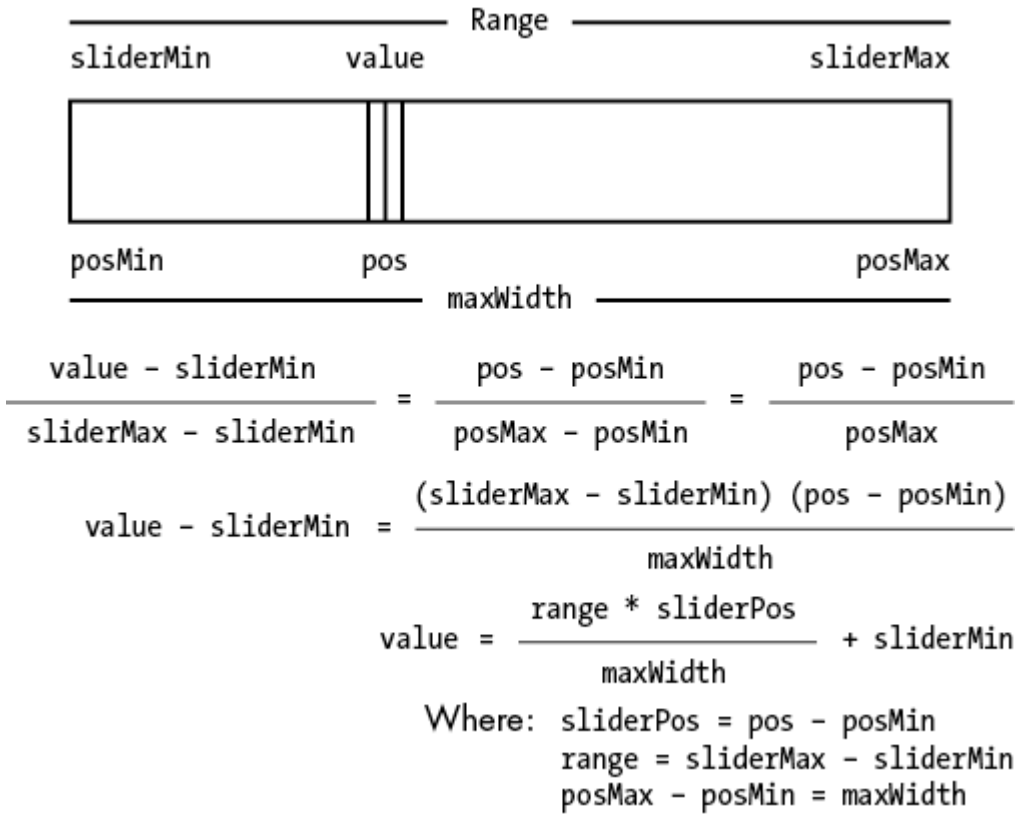


Figure 44-1: A slider

A slider can be represented graphically in many different ways. In this sketch, the widget is a horizontal rectangle with a circular cursor, and the current numerical value is drawn to the right. However, the cursor can be rectangular, elliptical, triangular, a pointer, or other shapes.

The `drawSlider()` function 1 draws the rectangle and positions the cursor using the `sliderPos` variable, which is set when the user selects the cursor with the mouse and then moves (slides) it between the ends of the rectangle. To build a slider class, you would make class variables for the position, size, current cursor position and value, and class functions to draw the slider and position the cursor (which you'd then call as, for example, `slider.drawSlider()` or `slider.draw()`).

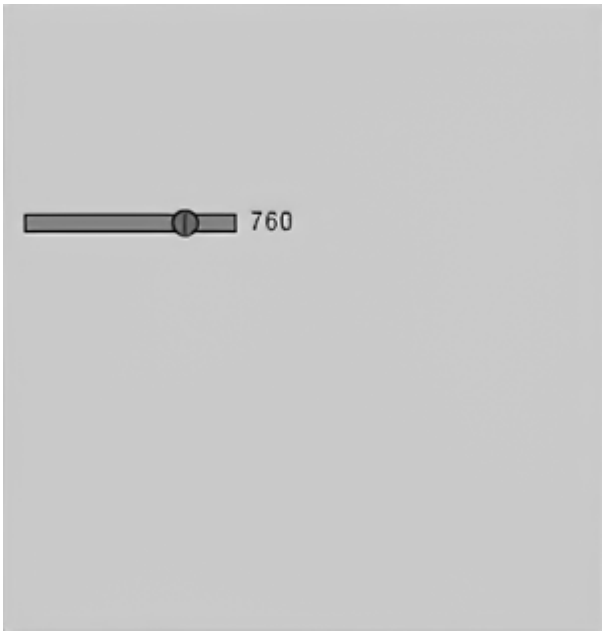
A common use for sliders is as a way to display an image. Often an image will not fit into a particular window, or into any window; some images are very large. Rather than resize the image, it is common to have a slider at the bottom and the right side of the window, and to use the cursor to position the window over the image so that various parts can be seen.

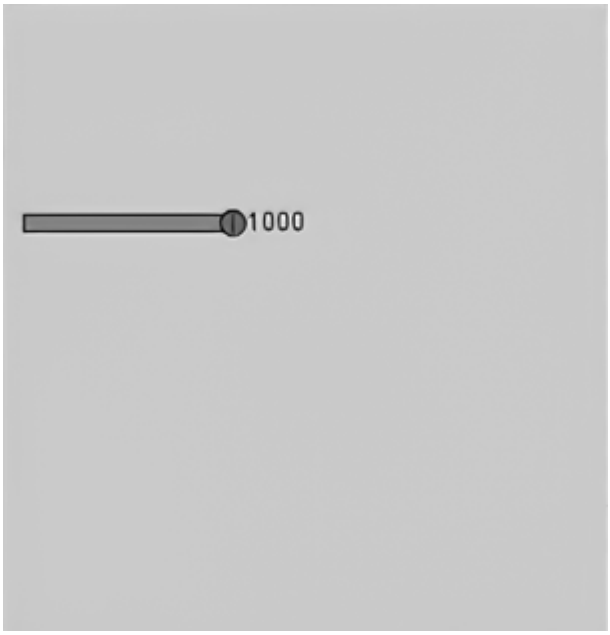
The values selected with the sliders represent the (x, y) location of the window over top of the larger image.

NOTE

The `mouseDragged()` *function* 2 *is called once every time the mouse moves while a mouse button is pressed. Also, note that mouse and keyboard events only work when a program has a* `draw()` *function, even if that function does not do anything.*

```
int sliderX=10, sliderY=100, sliderWidth=100,
sliderHeight=8;
color sliderColor = color(128,128,128);
int sliderPos=0, sliderMin=0, sliderMax=1000;
int value=0;

void setup ()
{
  size(600,300);
}

void draw ()
{
  background (200);
  drawSlider ();
}

1 void drawSlider ()
{
  fill (sliderColor);    // The bar part
  rect (sliderX, sliderY, sliderWidth, sliderHeight);
  fill (200,40,40);      // Slider partl
  ellipse (sliderX+sliderPos,
sliderY+sliderHeight/2,12,12);
  fill (0);
  line (sliderX+sliderPos, sliderY, sliderX+sliderPos,
sliderY+sliderHeight);
  text (value, sliderX+sliderWidth+7,
sliderY+sliderHeight);
}

2 void mouseDragged()
{
  if ((mouseY<sliderY) ||
(mouseY>sliderY+sliderHeight))
    return;
  if ((mouseX>=sliderX) &&
(mouseX<=sliderX+sliderWidth))
    sliderPos = mouseX - sliderX;
3 value = (int)(((float)sliderPos/sliderWidth)*sliderMax
+ sliderMin);
}
```


760


1000

Sketch 45: A Gauge Display

The obvious way for a computer to display a numeric result is to simply display the number, but sometimes a more analog approach is easier for people to deal with. Some people like digital clocks, and some prefer the old kind with hands. The analog display can be faster for a human to process. A common kind of display is a *gauge*, where a pointer of some kind rotates and points to a number. Most older speedometers are displays of this type, for example. *Figure 45-1* illustrates a gauge as a graphic and shows a simple abstraction of the situation.

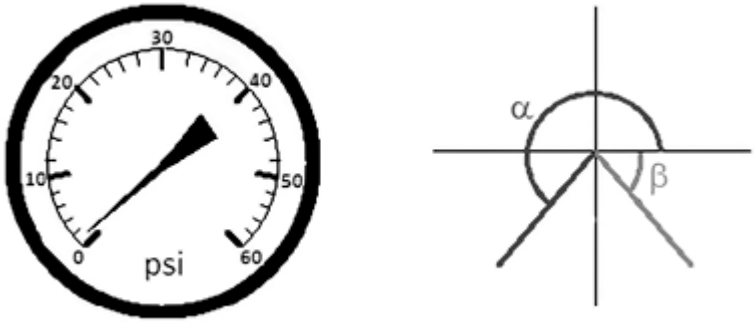


Figure 45-1: A gauge showing a value near 0 (left), and the angles that are involved in the display (right)

A gauge can display values between a minimum and a maximum numeric value. The minimum value corresponds to the minimum angle the pointer can have (labeled α in the figure), and the maximum value corresponds to the maximum angle the pointer can have (labeled β). In this sketch, angles map directly onto values so that a difference of one degree always represents the same amount of change. To display a value, we calculate the angle that corresponds to that value, named `theta` 1 in the sketch, and draw the pointer at that angle.

One way to look at this is as a *slider* that is shaped like a curve. Although the *gauge* is only a display, the mathematics of where to place the pointer is the same as for a slider, except we use angles instead of straight-line distances, and it is reorganized to provide a value for the position. *Figure 45-2* shows how the slider situation converts into what we need for a gauge,

and shows the formula for finding where to draw the pointer. This formula is really the same as the one used for the slider.

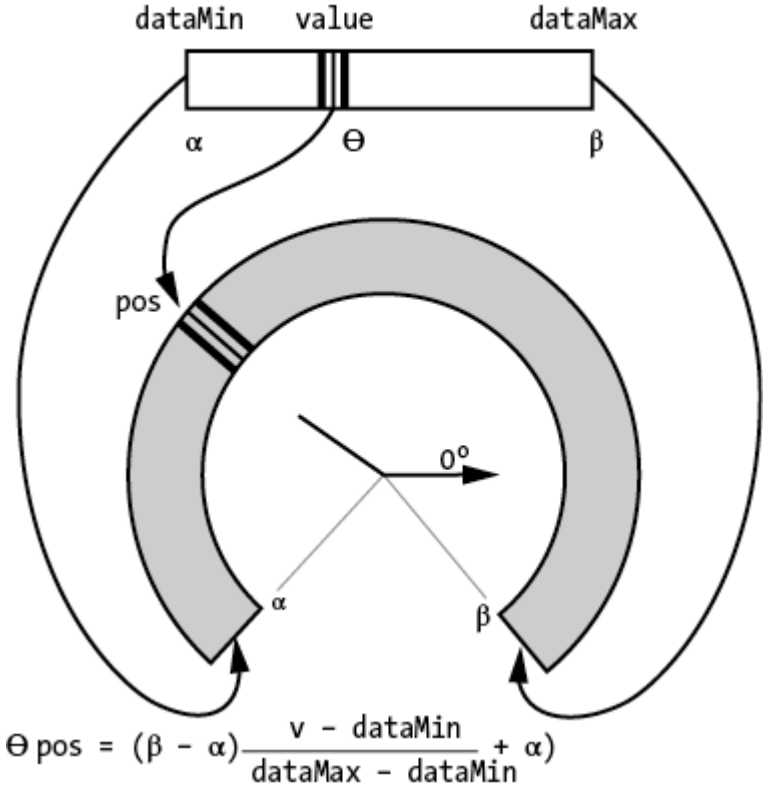
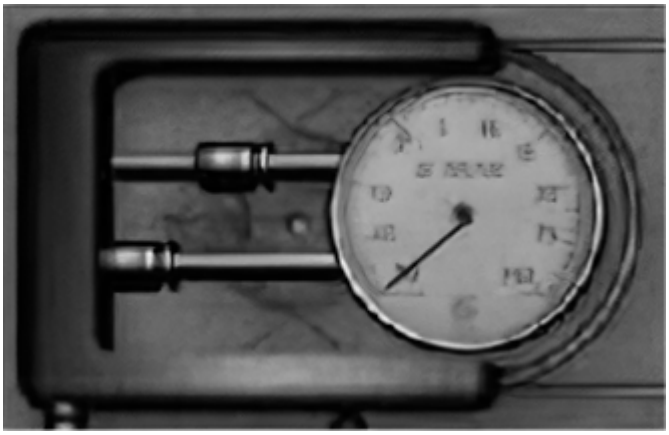


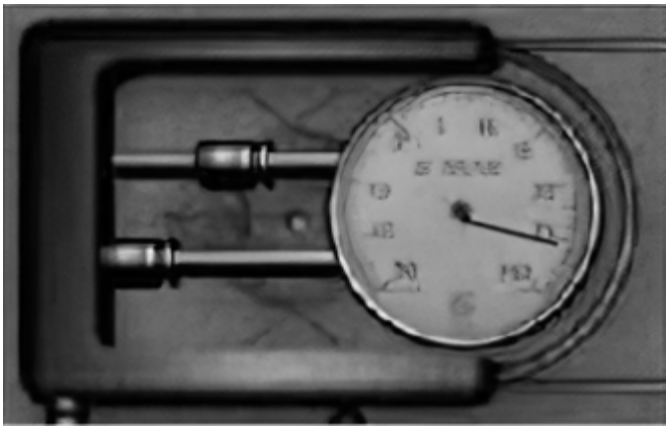
Figure 45-2: The gauge is like a bent slider. The equation shown here determines a position value (angle) given a numerical value, but it is otherwise the same as the one we used for the slider.

We do need to understand that 0 degrees is horizontal, and we convert the starting (α) and ending (β) angles so they are relative to 0. Starting at α, we decrease the angle of the pointer as the value increases toward the maximum. If α is 140, then β should be −45 rather than the equivalent angle, 315, so that β < α.

The `gauge()` function draws the pointer at the angle specified by the equation in *Figure 45-2* given a data value, `v`. Don't forget that angles in Processing need to be given as radians, so `pos` has to be converted from degrees.

```
int val = 0;
int dv  = -1;
int count = 0;
float dtor = 180.0/3.14159;
PImage background_image;   // Rendered gauge
int dataMin=0, dataMax=100;
float alpha=230, beta=-40;

void setup ()
{
  size (170, 108);
  background_image = loadImage
                ("data/gauge.png");
  frameRate( 30 );
  background (255);
}

/* Test main for a gauge widget */
void draw ()
{
  image (background_image, 0, 0);
  gauge (119, 55, val, 25);
  val = val + dv;
  if (val > dataMax)
  { val = dataMax; dv = -dv; }
  else if (val < dataMin)
  { val = dataMin; dv = -dv; }
}

void gauge (int x, int y, int v, int dial_length)
{
  float theta;
  int xx, yy;
// Calculate rotation angle of pointer
1 theta = radians (((v-dataMin)*(beta-alpha))/(dataMax-
dataMin) + alpha);
  stroke (0, 0, 0);
  yy = int(dial_length * sin(theta));  // x-coordinate
of rotated pointer end
  xx = int(dial_length * cos(theta));  // y-coordinate
of rotated pointer end
  yy = y-yy; xx = xx + x;
  line (x, y, xx, yy);
}
```





Sketch 46: A Likert Scale

A *Likert scale* is a rating scale for answering questions, commonly used in questionnaires. The person being asked the question selects one of the answers from a set of choices (often five) ranging from "Strongly Disagree" to "Strongly Agree." The idea is to collect standard answers upon which statistics can be computed.

This sketch poses a question by drawing it near the top of the screen 2. The possible answers are numbered from 1 (Strongly Disagree) to 5 (Strongly Agree), and each answer corresponds to a circle. To select an answer, the user clicks on a circle, and the circle gets filled in 3. When the user has answered to their satisfaction, then they type any key and the sketch asks another question.

The questions reside in a text file named *questions.txt* that is opened within `setup()`. We assume that there are multiple questions, and each is one line of text in the file. The `loadStrings()` 1 function reads them all into an array named `question`, the length of which is the number of questions. Each question is asked (displayed) according to its index variable, `questionNo`, which iterates from 0 to the number of questions. The user selects an answer, one of five possible, by clicking the mouse within one of the five circles. That answer is chosen as the current selection (using a variable named `select`) in the `mouseReleased()` function.

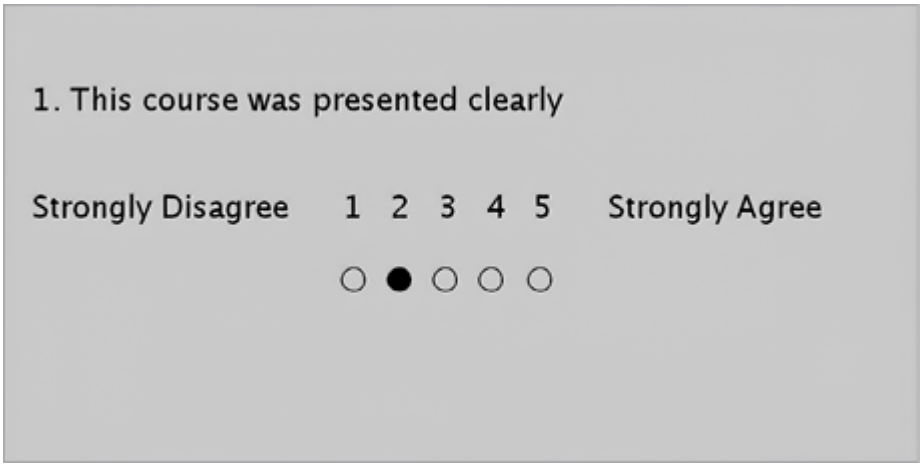
When the user types a key, `keyPressed()` is called, and the selection will be written to a file named *save.txt* 4. Then the `questionNo` variable will be incremented, resulting in the next question being displayed. When all questions have been asked (that is, when `questionNo > question.length`), the file is closed and the program ends. The answers chosen by the user to all questions are now stored in the *save.txt* file.

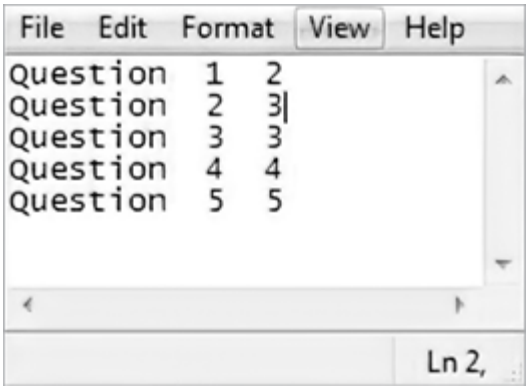
NOTE

There is only one output file, but there may be many users. We can prompt users for their name or an ID number, and that could be used as the basis for a filename, so that many distinct sets of answers could be saved. Another option would be to create filenames that end in consecutive numbers. When the program begins, it could try to open files until it arrived at one that did not exist; that would be the next usable filename.

```
int questionNo = 0;
String [] question;
int select = 0;
String list[];

void setup ()
{
1 question = loadStrings("questions.txt");
  list = new String[question.length];
  size (600, 300);
}

void draw ()
{
  background(200);
  drawGraphic();
  fill (0);
  textSize (20);
2 text ((questionNo+1)+". "+question[questionNo], 20,
70);
}

void drawGraphic ()
{
  text ("Strongly Disagree      1   2   3   4   5
Strongly Agree",
          20, 140);
  noFill();
  if (select==1) fill(0); else noFill();
  ellipse (230, 180, 15, 15);
  if (select==2) fill(0); else noFill();
  ellipse (260, 180, 15, 15);
  if (select==3) fill(0); else noFill();
  ellipse (290, 180, 15, 15);
  if (select==4) fill(0); else noFill();
  ellipse (320, 180, 15, 15);
  if (select==5) fill(0); else noFill();
  ellipse (352, 180, 15, 15);
}

void mouseReleased ()
{
3 if (mouseY<173 || mouseY > 188) return;
  if (mouseX>=223&&mouseX<=238) select = 1;
  if (mouseX>=253&&mouseX<=268) select = 2;
```

```
  if (mouseX>=283&&mouseX<=298) select = 3;
  if (mouseX>=313&&mouseX<=328) select = 4;
  if (mouseX>=345&&mouseX<=360) select = 5;
}
void keyPressed ()
{
  list[questionNo] = "Question  "+str(questionNo+1)+"
"+str(select);
  questionNo = questionNo + 1;
  if (questionNo >= question.length)
  {
4 saveStrings("save.txt", list);
    exit();
  }
}
```





Sketch 47: A Thermometer

The original thermometer, made of glass with a colored fluid inside, had a design imposed by its function, but it was also an excellent way to display numeric data. It represents a number as the height of a colored line or rectangle. It is easy to see how tall a rectangle is and easy to compare it to others. This idea has been used in many places, most noticeably on sound equipment to show volume.

The representation on a computer is straightforward. A colored rectangle grows and shrinks as a function of how large a numeric variable is. Such a variable has a minimum and maximum value, and the rectangle has a minimum (usually 0) and maximum height. The mapping between the number and the height can be done as it was for the slider (Sketch 44) and the dial gauge (Sketch 45). In this sketch, it is implemented a bit differently, but it is computed in the same way.

This sketch computes how much taller the rectangle gets for each increase in the variable 1. If the rectangle's height can go from `ystart` to `yend`, and the range of data values is from `dataMin` to `dataMax`, then the change in rectangle height for each data increment is as follows:

```
delta = (float)(ystart-yend)/(float)(dataMax-dataMin);
```

Then for any data value, `data`, the height of the rectangle relative to `ystart` is the following:

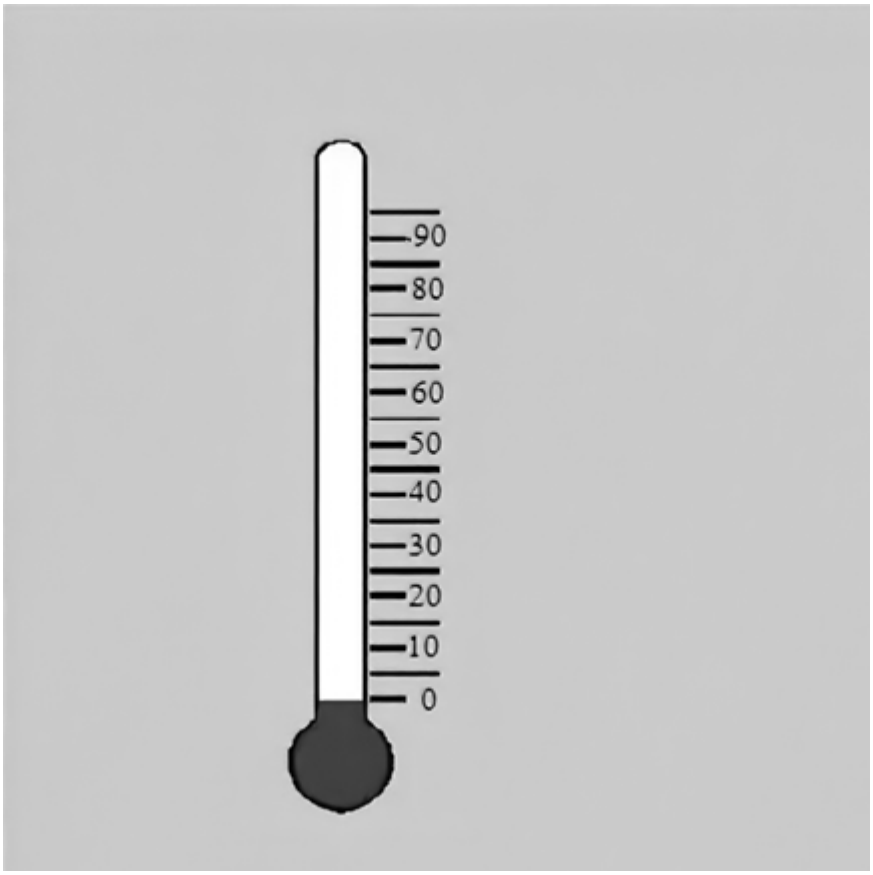
```
val = ystart-(int)(data*delta);
```

This process only draws a rectangle, which is not very exciting, so we'll add a background image (created specifically for this program) that contains an image of a glass thermometer and gradations that allow the user to interpret the height as a number. The coordinates of the rectangle have to be mapped specifically onto the image so that the rectangle aligns with the thermometer column, using a similar process as in Sketch 45.

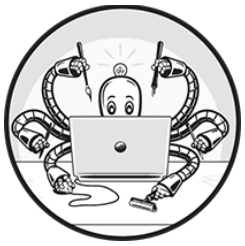
This example generates a random numeric value for display. After starting arbitrarily at `data = 15`, the value changes by a small random amount each frame.

```
PImage thermo;                  // Thermometer image
int xpos=100, ypos=50;          // Position of upper left
int ystart = 240;               // Position of Y lowest
point
int yend = 44;                  // Position of Y highest
point
int xstart = 32;                // Left of red column
int xend = 50;                  // Right of red column
int dataMin=0, dataMax=90;      // Range of data values
float delta = 1;
float data = 15.0;

void setup ()
{
  size (400, 400);                          // Window
size
  thermo = loadImage ("thermo.gif");        // Read
button images
1 delta = (float)(ystart-yend)/(float)(dataMax-dataMin);
  rectMode (CORNERS);
  noStroke();
}

void draw ()
{
  int val;

  background (200);
// White background
  image(thermo, xpos, ypos);
// Draw the basic thermometer
2 val = ystart-(int)(data*delta);
// Scale data to Y range
  fill (140, 4, 20);
// Fill with red
  rect (xstart+xpos, val+ypos, xend+xpos, ystart+ypos);
// Draw red
  text (""+(int)data, xstart+xpos, ystart+ypos+30);
// Draw data value
  data = data + random(2) - 1;
// Modify data for display purposes
  if (data > dataMax) data = dataMax;
  else if (data < dataMin) data = dataMin;
}
```


90
80
70
60
50
40
30
20
10
0


90
80
70
60
50
40
30
20
10
0

6
NETWORK COMMUNICATIONS



Sketch 48: Opening a Web Page

A web page is really just a text file containing the description of that page in enough detail to draw it on the screen. A program called a *browser* reads and renders that file into a viewable page. The file itself resides on a computer somewhere on the internet, and in order to display it, we must first upload it to the user's computer. The browser arranges for this to be done, but the file must have a unique name that identifies it—unique in the *whole world*, because the internet is a planetwide network. This unique name is called the *Uniform* (or *Universal*) *Resource Locator*, shortened to *URL*. Most people know this by the term *web address*, and an example is *https://www.microsoft.com*.

The URL contains the directions for how to find the web page, and it is the equivalent of a filename. Displaying the page is a complex operation, and browsers are very complicated software systems.

Processing opens and displays web pages using a function named `link()`, which accepts a URL as a parameter. This function passes the URL to the default browser on your computer, which opens and displays the page. So the following call will open the Microsoft page in a browser:

```
link ("https://www.microsoft.com");
```

If the browser is already open, it may open a new tab.

Example A

This sketch opens the Microsoft page as previously described 1. It does so when a mouse button is pressed while the cursor is within the display window.

Example B

This sketch is a combination of Example A and Sketch 37. The user types a URL, and the sketch builds a string from the characters being typed. When

the user types ENTER, the sketch passes the URL to `link()`, and the browser will open and display the corresponding page.

When the user types a character, it is usually placed in the variable `key`, then added to the string. However, some keys do not produce characters, such as the arrow keys, or SHIFT. In Processing, uppercase characters involve two key presses: the SHIFT key and the character. The Processing system refers to these as *coded* keys and treats them differently. If the `key` variable has the value `CODED`, then the key pressed was one of these special ones, and the `keyCode` variable indicates what key was pressed 1. The value `UP`, for example, indicates that the up-arrow key was pressed.

In this sketch, we'll ignore all coded keys, because the SHIFT key is needed to send uppercase letters and some punctuation (like the colon, ":"), but it should not be thought of as a key press. The `keyPressed()` function ignores coded keys using the following code:

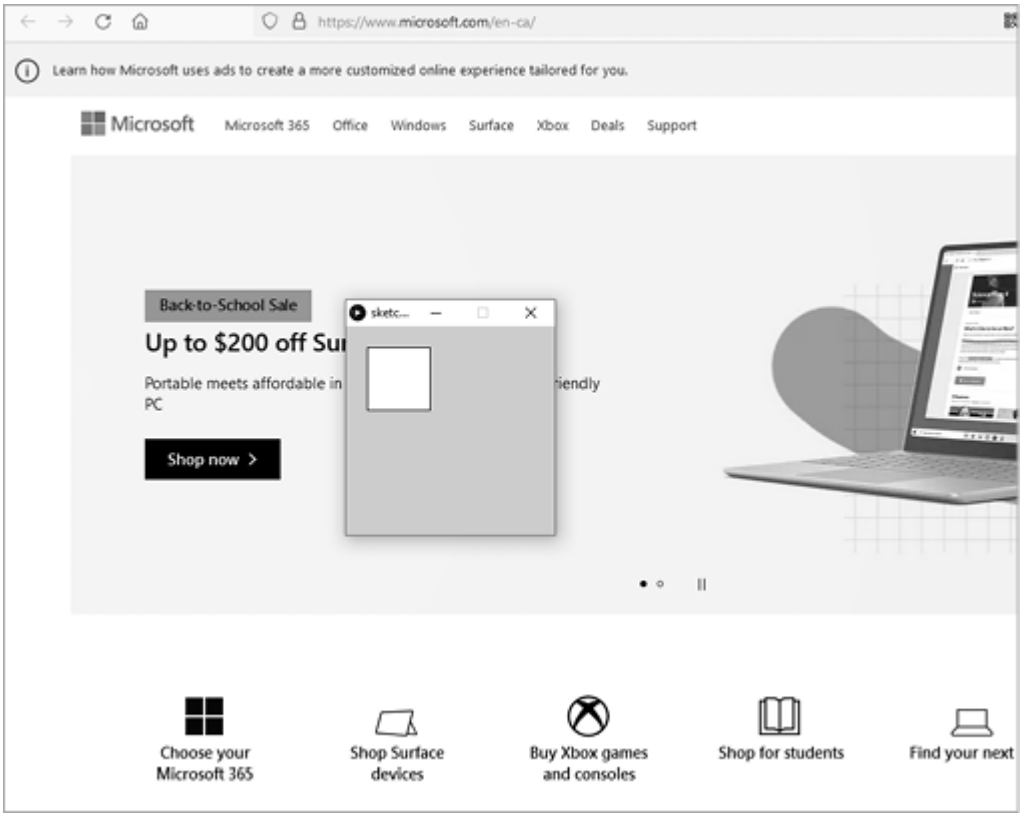
```
if (key == CODED) return;
```

Example A

```
void setup ()
{
  size (200, 200);
}

void draw ()
{
  rect(20, 20, 60, 60);
}

void mousePressed ()
{
1 link ("https://www.microsoft.com");
}
```

Example B

```
String url = "";

void setup ()
{
  size (240, 200);
}

void draw ()
{
  background (200, 200, 200);
  fill(0);
  text (url, 20, height-20);
}

void keyPressed ()
{
  fill(0);
1 if (key == CODED) return;
//  if (keyCode==SHIFT) return;
  if (url.length()>0 && key==BACKSPACE)
    url = url.substring (0, url.length()-1);
  else if(key==ENTER || key==RETURN)
    link (url);
  else
    url = url + key;
}
```


https://www.microsoft.com/en-ca/
Learn how Microsoft uses ads to create a more customized online experience tailored for you.
Microsoft
Microsoft 365
Office
Windows
Surface
Xbox
Deals
Support
Back-to-School Sale
Up to $200 off Su
Portable meets affordable in
iendly
PC
Shop now >
sketc...
Choose your Microsoft 365
Shop Surface devices
Buy Xbox games and consoles
Shop for students
Find your next

Sketch 49: Loading Images from a Web Page

Since a web page is really just a text file, as you saw in Sketch 48, it should be possible to read that file and see what is inside. For example, it should be possible to identify any sound files (for example, MP3s) accessed by the page, or which images (*.jpg*, *.gif*, *.png*, and so on) will be a part of the page. This sketch will locate image files referenced in a web page and display them in the display window.

The first thing to do is to read the page. It contains *HTML*, a language for describing the document, and reading it turns out to be easy: Processing allows URLs to be used just like filenames in the `loadStrings()` function. You can read the Mink Hollow Media web page as a text file by directly passing the URL to `loadStrings()`:

```
String webin[] = loadStrings("https://minkhollowmedia.ca");
```

Or, as is done in this sketch, `loadStrings(url+"/"+file)`, where `url` is the web address and `file` is the name of the file that we want 1. At this point, the web page is available as a collection of strings in the array `webin`, one per line in the file.

HTML uses what is called an `img` tag to display images in a page:

```
<img src="imagename.jpg">
```

The filename of an image follows the text `src="`, so the sketch should look for this sequence of characters within the strings in `webin`. If found, the following characters, up to the closing quote character (`"`), are the filename. We can locate a string within another string using the `indexOf()` function 2:

```
i = s.indexOf ("src=", j);
```

In this example, `indexOf()` searches the string `s` for the string `"src="`, starting at the character index `j`. It returns the index of the location where the string was found, or −1 if it was not found. If the string is found, we call the `getName()` function 3 to extract the filename itself from the string. The

`getName()` function reads and saves characters until it encounters the terminal double quote and returns the filename as a string 4. This string is used as a filename for `loadImage()`, and if an image with that name can be loaded, then it is displayed.

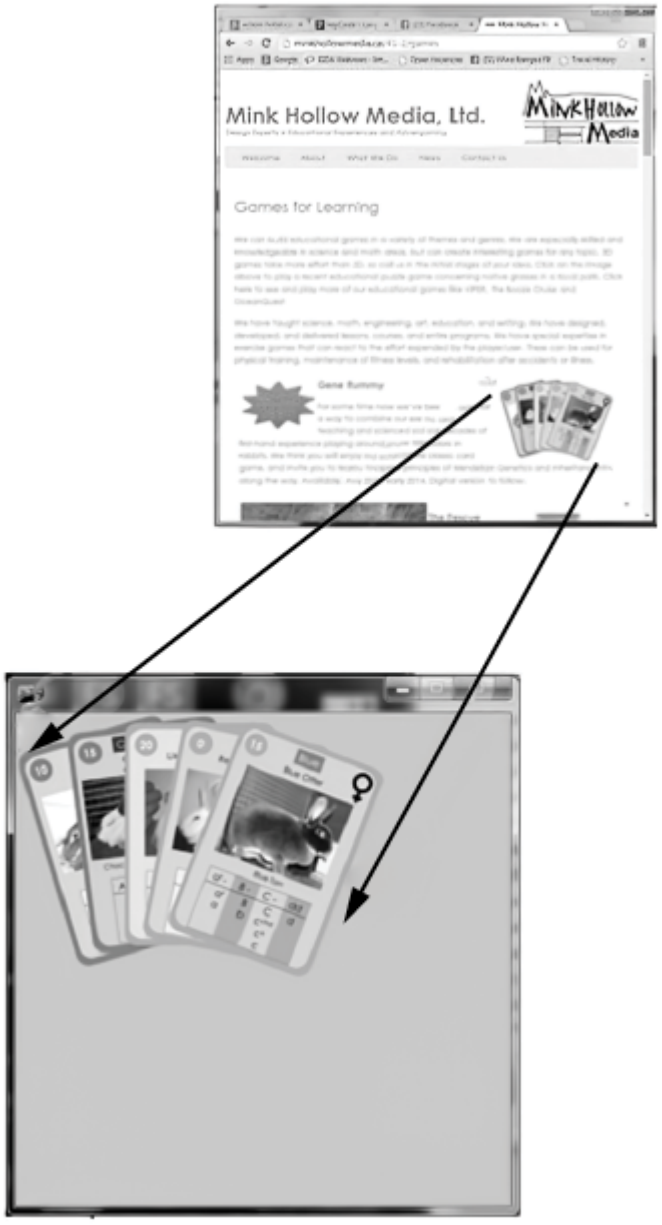
There are many legal ways to specify a filename, and the code here also tries one other: it will take the URL and add a slash (`/`) and the filename 1 to see if that works. Some images will not be located using this method, and some files that are not images (like JavaScript, video, and audio) can be extracted. They will fail to display as images, and error messages will appear in the console.

NOTE

You could save the extracted images as files by simply calling `save()` when the image is successfully loaded. The act of extracting information from a web page using a program is called scraping.

```
String [] webIn;
String url = "https://minkhollowmedia.ca/41-2/games";
String file = "",  name="",  s="";
int index = 0, i=0;
PImage next;

void setup ()
{
  size (400, 400);
1 webIn = loadStrings(url+"/"+file);
  fill (0);
}

void draw ()
{
  background (200);
  if (next != null) image (next, 0, 0);
  index = index + 1;
  if (index>=webIn.length)
  {
    text ("DONE", 10, 370);
    return;
  }
  text (webIn[index], 10, 370);
  s = webIn[index].toLowerCase();
2 i = s.indexOf ("src=", i);
  if (i<0) return;
  s = webIn[index];
3 name = getName(s.substring(i+4));
  if (name == null) return;
  if (name.charAt(0) != '/') next = loadImage
(url+"/"+name);
  if (next == null) next = loadImage (name);
}

String getName (String s)
{
  int i=1;
  if (s.charAt(0) != '"') return null;
  while (i<s.length())
  {
 4 if (s.charAt(i) == '"') return s.substring(1, i);
    i = i + 1;
  }
```

```
  return null;
}
```



Sketch 50: Client/Server Communication

A lot of computer network communication is based on what is called a *client/server* model. It could just as easily be called a *listener/speaker* or *receiver/sender* model because it amounts to having one computer or process sending information across a network (the *server*) and another computer, or many other computers, receiving that data (the *clients*).

Here's how client/server software should work. A server first announces to the world that it is active and sending data. It must have an address that can be used to identify it uniquely, and it must start sending data (bytes, for example). A client identifies a server that it wishes to collect data from by using the server's address. If the address represents an active server, the client starts to read data from the server. The server must indicate when new data is available, and if data is requested and none has yet been sent, the client waits until data is present.

This example has a server sending character data and a client receiving and displaying the data, implemented as two different sketches. The server sends the message "This is a message from J Parker" repeatedly; the client reads characters from the server, constructs a string from them, and displays this string in the display window.

Processing does not have a native ability to build client/server systems, but a library exists that enables it. Processing uses external libraries for many things, including video, audio, and various specific interfaces. For this example, we need to import the Network library at the beginning of both the client and server sketches 1, using this line:

```
import processing.net.*;
```

In the server code, the first step is to create a `Server` (part of the Network library) and assign it to the variable named `sender`, and then specify the port (in this case, port 5000), which is simply a number. A *port* is like a television channel, used to send or receive data, and all that matters here is that no other software is using this port. The server sends characters one at a

time from a string to the outside world through the port by calling the `write` function 2:

```
sender = new Server(this, 5000);
--snip--
sender.write(nextChar);
```

`nextChar` is a character from the message.

The client sketch first tries to connect to the server. The client must know the *IP address* of the server, which is its unique identifier (`***.***.***.***` in this case). The client connects to the server through the constructor using the same port 3:

```
me  = new Client(this, "***.***.***.***", 5000);
```

The client reads characters, one at a time, using the `readChar()` function 4:

```
nextChar = (char)me.readChar ();
```

In this example, you have to start the server first and find out its IP address. You can use the `ipconfig` program on the computer where you are running the server sketch to find the IP address. Then you can start the client on some other computer on your network.

NOTE

*The address ***.***.***.*** was used to avoid publishing a real address.*

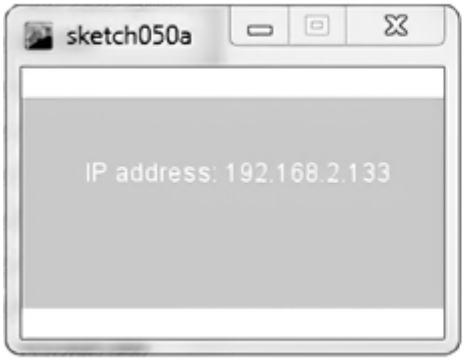
Server

```
1 import processing.net.*;
  Server sender;
  int ind = 0;
  char nextChar;
  String message = "This is a message from J Parker ";

  void setup ()
  {
    size(200, 100);
2 sender = new Server(this, 5000);   // Create server
  }

  void draw ()
  {
    background(200);
    if (ind >= message.length()) ind = 0;
    nextChar = message.charAt(ind);
    sender.write(nextChar);
    text ("IP address: "+sender.ip(), 30, 40);
    ind = ind + 1;
  }
```

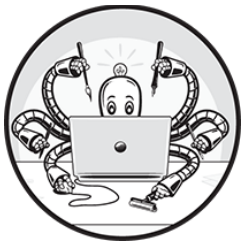
Client

```
1 import processing.net.*;
  Client me;
  char nextChar = ' ';
  String message = "";

  void setup ()
  {
    size(200, 200);
3 me = new Client(this,"***.***.***.***",
                  5000);
    fill (20);
    frameRate (10);
  }

  void draw ()
  {
    if (me.available() > 0)
    {
      background(50, 250, 50);
```

```
    text ("IP address: "+me.ip(), 30, 40);
4 nextChar = (char)me.readChar ();
    message = message + nextChar;
    if (message.length() > 20)
      message = message.substring(1,message.length());
    text (message, 10, 100);
  } else
  {
    background (200, 30, 10);
    text ("No server at port 5000.", 10, 20);
  }
}
```



7
3D GRAPHICS AND ANIMATION



Sketch 51: Basic 3D Objects

We have been drawing only two-dimensional (2D) objects so far: lines, circles, triangles, rectangles, and images. Processing can draw three-dimensional (3D) objects too, although all that we can represent on a computer screen is a *view* of these, a 2D projection onto a plane. This projection aspect is what makes 3D more difficult. The x dimension is horizontal, and the y is vertical, and displaying those coordinates on a 2D screen is obvious. The third dimension, called z, would be perpendicular to the screen's surface. In order to visualize it, the three coordinates must be reduced to two, which is what the projection does.

Processing provides a 3D box (cube) and a sphere. In this sketch, we'll draw these standard objects to show how 3D works.

To render 3D objects, Processing needs to use software that performs 3D drawing operations, called a *3D renderer*. The default renderer, called `P2D`, only handles two dimensions. To specify three dimensions, we provide the `P3D` renderer 1 as an argument to the `size` function within `setup`:

```
size (300, 400, P3D);
```

Now all 3D operations are available. Cubes and spheres are provided through functions, just as rectangles and ellipses are in 2D. The function `sphere(R)` 3 draws a sphere having radius `R` at the origin.

A sphere is drawn as a collection of triangles that have x-, y-, and z-coordinates at each vertex, oriented along the surface of the sphere and connected edge to edge. Think of it as the 3D version of drawing a circle using many short straight lines; it's not exactly smooth, but if the triangles are small enough, the illusion works. The triangles will be visible unless outlines are turned off with a call to `noStroke()`.

The `box(s)` function draws a cube where each side is `s` pixels long 4. To specify the size in each direction, we can use the second form of `box`: `box(w, h, d)`.

To draw either shape somewhere other than the origin, we must first call the `translate()` 2 function to move the origin to the location where the sphere is to be drawn. In 3D, coordinates have three values: x, y, and z, and `translate()` has three corresponding parameters.

Finally, when drawing in three dimensions, we need illumination to create depth. To enable lighting, we call `lights()` in the `draw()` function. Without the call to `lights()`, the sphere on the left of the Example A output would just look like a circle.

Example A

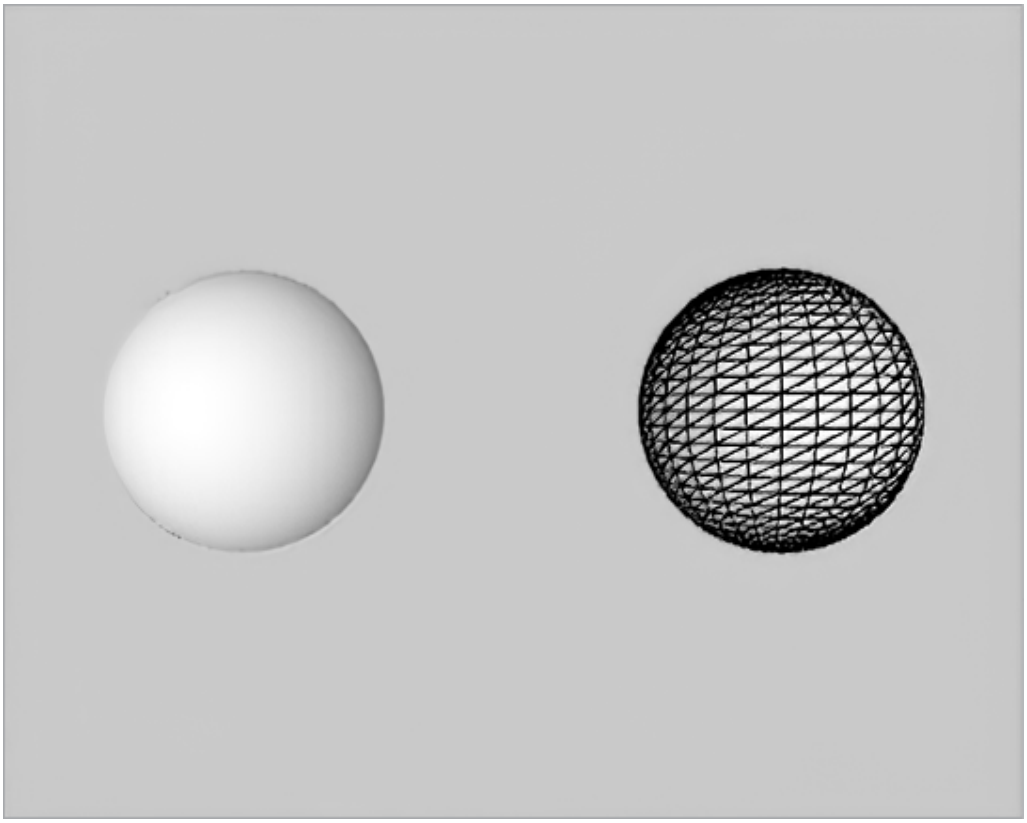
We draw two spheres: one with the triangles composing the sphere visible (right) and one with them hidden (using `noStroke()`, left). The spheres move away from the point of view and then back, showing the third dimension more clearly than if they were still.

Example B

We draw two cubes, again with the right one showing the cube outlines and with the left one not. The cubes also move away from the camera and then back again (along the z-axis).

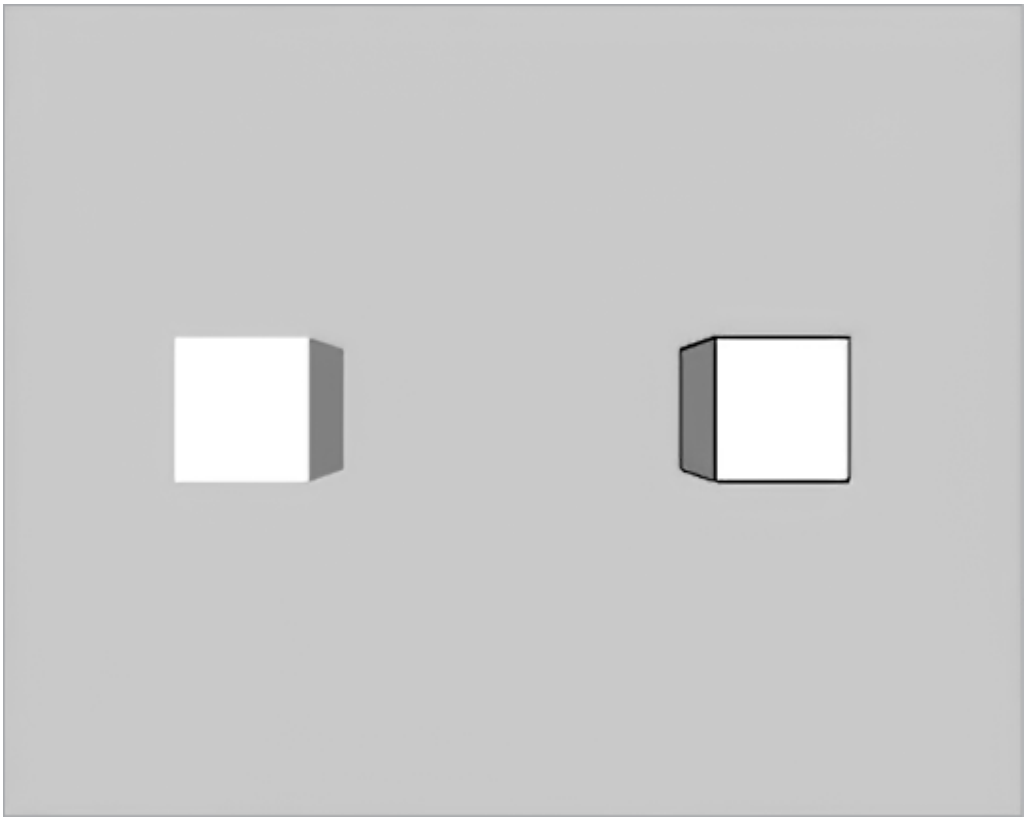
Example A

```
int z = 50, dz = 1;
void setup ()
{
  size(400, 300,1P3D);
}

void draw ()
{
  background (200);
  noStroke();
  lights();
2 translate(100, 150, z);
3 sphere(50);
  translate(200, 0, 0);
  stroke (0);
  sphere (50);
  z = z + dz;
  if (z > 50) dz = -dz;
  if (z < -350) dz = -dz;
}
```



Example B

```
int z = 50, dz = 1;

void setup ()
{
   size(400, 300, P3D);
}

void draw ()
{
  background (200);
  noStroke();
  lights();
  translate(100, 150, z);
4 box(50);
  translate(200, 0, 0);
  stroke (0);
  box(50);
  z = z + dz;
  if (z > 50) dz = -dz;
  if (z < -350) dz = -dz;
}
```



Sketch 52: 3D Geometry—Viewpoints, Projections

3D objects are really simulations in which the edges and faces have locations in a virtual space having three coordinates. Because computer screens are 2D, visualizing these objects means projecting them onto a plane so they can be drawn on a screen.

This plane lies between the object and the location from which the object is being seen, or the *viewpoint*. The viewpoint is a location in 3D space, marked by an eye in *Figure 52-1*. (2D scenes don't really have a viewpoint; the entire image is a plane in the first place.)

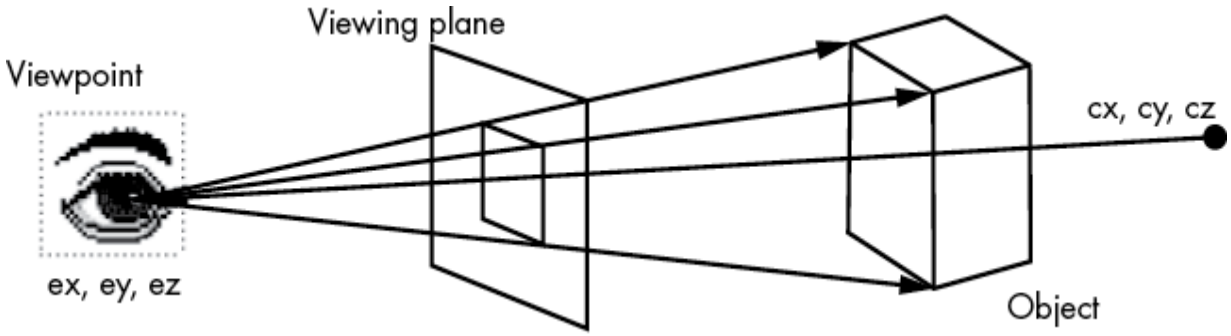


Figure 52-1: Viewing a 3D object

There is a second crucial point for defining how a 3D view appears, and that is the location where the viewer (camera) is *looking*. This is the center of the scene, denoted by `(cx, cy, cz)`. The plane on which the 3D scene is projected is perpendicular to the line between `(ex, ey, ez)` and `(cx, cy, cz)`, and precisely what can be seen depends on the *field of view*, or the angle of the visible field, which determines what can be seen without moving the camera.

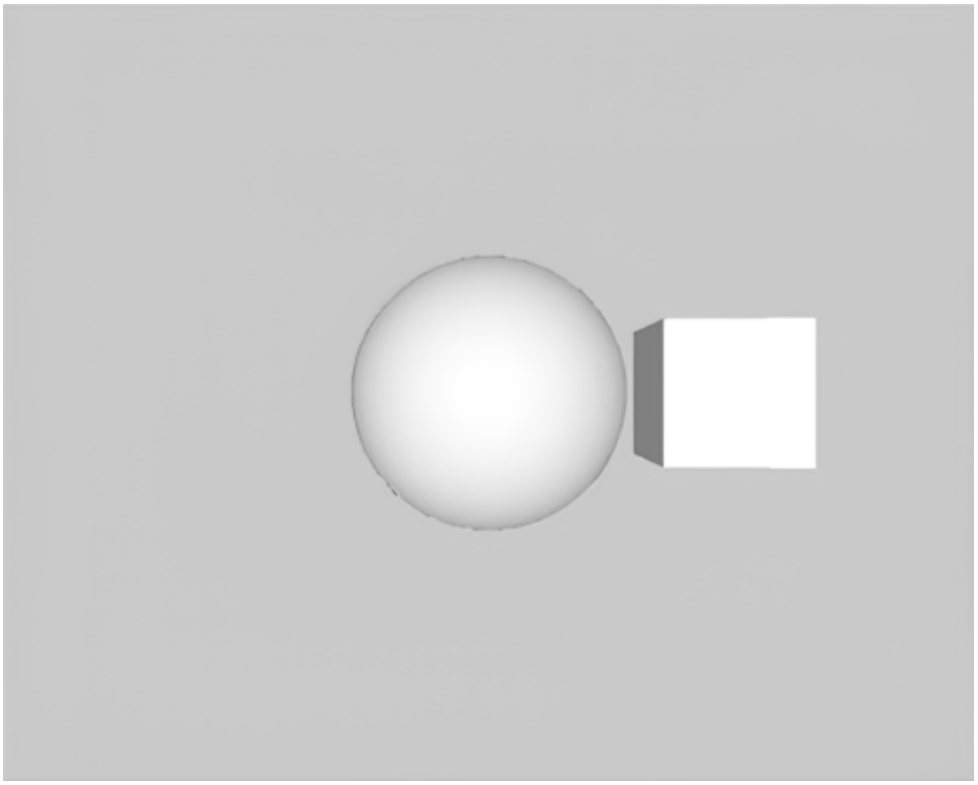
In Processing, we use a call to `camera()` 1 to set up the basic 3D configuration:

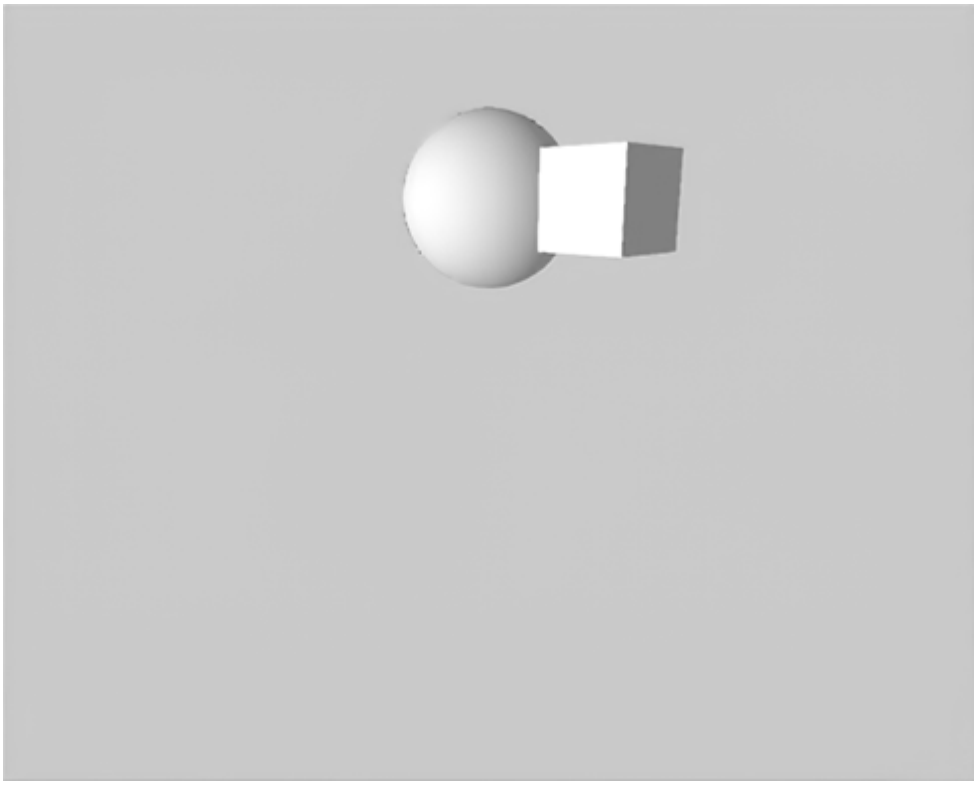
```
camera(ex, ey, ez, cx, cy, cz, 0, 1, 0);
```

The first three parameters are the viewpoint, and the next three are the center of the scene. The last three represent a vector that defines the direction *up* so that the scene is oriented correctly. In this example, *up* is the positive y-direction. It is a choice made by the programmer.

This sketch uses the `camera()` function to change the view of a pair of 3D objects according to user key presses. We move the location of the viewpoint by incrementing or decrementing the values of `ex` and `ez` inside the `keyPressed()` function 2 when the proper keys are pressed: A decreases x (moves left), D increases x (moves right), W decreases z, and S increases z (the distance to the objects). This is the equivalent of moving the player in a video game. The sphere is drawn at `(cx, cy, cz)` so that it is guaranteed to be visible at the outset. To move the center of the scene away from the sphere, we can change the value of `cy` with the up and down arrow keys. You can see the effect of changing the viewpoint and the center of the scene by experimenting with them using the keyboard.

```
int x=100, y=100, z=100;      // Sphere position
int ex=100, ey=100, ez=400;   // Viewpoint
int cx=100, cy=100, cz=100;   // Point we are looking
at

void setup ()
{
  size (500, 400, P3D);
}

void draw ()
{
  background (200, 200, 200);
1 camera(ex, ey, ez, cx, cy, cz, 0, 1, 0);
  noStroke();
  lights();
  translate (x, y, z);
  sphere (12);
  translate (20, 0, 0);
  box (12);
  translate (-x-20, -y, -z);
}

2 void keyPressed ()
{
  if (key == 'w') ez = ez - 10;
  if (key == 's') ez = ez + 10;
  if (key == 'a') ex = ex - 10;
  if (key == 'd') ex = ex + 10;
  if (keyCode == UP) cy = cy + 10;
  if (keyCode == DOWN) cy = cy - 10;
}
```





Sketch 53: 3D Illumination

Illumination can profoundly change the appearance of a scene. The location of lights will cause specific portions of objects or scenes to be visible while others are not. Colored lights can change the apparent color of objects. Directional lighting can illuminate some portions of an object and not others. Processing provides all of these options.

In this sketch, we'll draw a sphere and permit the user to select the type of lighting used by typing a number. The lighting may be ambient (1), directional (2), point (3), spot (4), or all three: directional, point, and spot (5). The default lighting is code 0. When the user changes the kind of lighting, the color changes as well: ambient is cyan, directional is violet, point is yellow, and spot is green.

The previous sketches have used a call to `lights()` to provide default illumination. Alternatively, we can use a call to the `ambientLight()` function 1 to specify a color and, optionally, a location for ambient lighting, which is illumination that permeates the scene.

```
ambientLight (r, g, b, x, y, z);
```

The first three parameters specify the RGB values for the color of the light. The next three are optional, and specify a location in three dimensions. Light spreads in all directions from this point.

The `directionalLight()` function 2 specifies light from a specific direction, so it appears brighter when striking a surface perpendicular to that direction and less bright as the angle changes.

```
directionalLight (r, g, b, dx, dy, dz);
```

Again, the first three parameters represent the color of the light. The next three specify the direction. So, for example, if `dy=1` while `dx=0` and `dz=0`, the object will be illuminated from above.

The `pointLight()` function 3 creates a single location from which illumination comes, like a lamp. This call places a light with the specified

RGB values at the given (`x`, `y`, `z`) location:

```
pointLight (r, g, b, x, y, z);
```

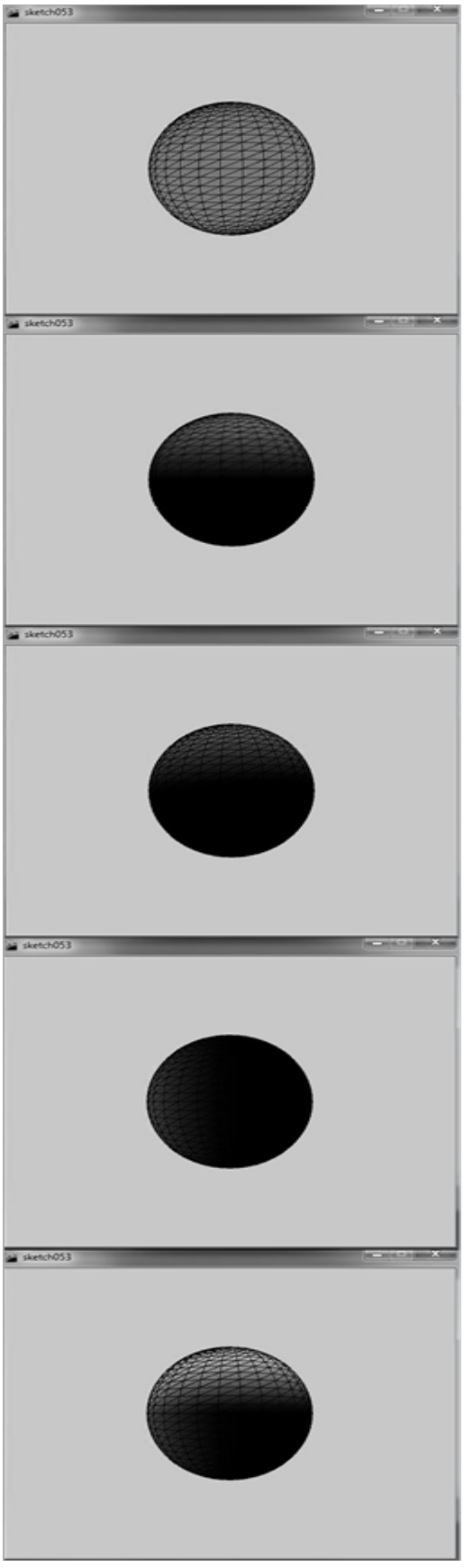
Finally, a `spotlight()` 4 is a concentrated directional light, and it is the most complex of the lighting sources. This call specifies a light of color RGB at location (`x`, `y`, `z`) pointed in direction (`dx`, `dy`, `dz`):

```
spotlight (r,g,b, x,y,z, dx,dy,dz, angle, concentration);
```

The value of `angle` is the dispersion angle of the light; the smaller the angle, the smaller the circle of light. This angle is in radians. The `concentration` specifies how the light varies across a cross section, brighter in the center and less bright at the edges. Values can vary from 1 to 10,000.

```
int x=100, y=100, z=100;          // Sphere position
int ex=100, ey=100, ez=147;       // Viewpoint
int cx=100, cy=100, cz=100;       // Point we are
looking at
int code = 0;
void setup()
{
  size (500, 400, P3D);
}

void draw ()
{
  background (200, 200, 200);
  camera(ex, ey, ez, cx, cy, cz, 0, 1, 0);
  if (code == 0)
    lights();
  else if (code == 1)
 1 ambientLight (0, 200, 200, 0, 1000, 0);
  else if (code == 2)
 2 directionalLight (200, 0, 200, 0, 1000, 0);
  else if (code == 3)
 3 pointLight (200, 200, 0, 0, -1000, 0);
  else if (code == 4)
 4 spotLight (0, 200, 0, -300, 100, 100, 100, 0, 0,
PI/16, 1000);

  else if (code == 5)  // All three!
  {
    directionalLight (200, 0, 200, 0, 1000, 0);
    pointLight (200, 200, 0, 0, -1000, 0);
    spotLight (0, 200, 0, -300, 100, 100, 100, 0, 0,
PI/16, 1000);
  }
  translate (x, y, z);
  sphere (12);
}

void keyPressed ()
{
  if (key == '0') code = 0;
  if (key == '1') code = 1;
  if (key == '2') code = 2;
  if (key == '3') code = 3;
  if (key == '4') code = 4;
```

```
  if (key == '5') code = 5;
}
```


sketch053
sketch053
sketch053
sketch053
sketch053

Sketch 54: Bouncing a Ball in 3D

Sketch 28 was a simulation of a bouncing ball. A circle (ball) moved about a window, bouncing when it struck the boundary. An obvious extension of this into three dimensions has a sphere bouncing about the inside of a cube. When the sphere (ball) strikes one of the sides of the cube, it bounces. This is conceptually the same problem as in two dimensions, but it requires quite a bit more code because there are more conditions to check and more things to draw.

The scene consists of the cube and a sphere. The cube occupies most of the field of view, bounded by the coordinate axes. We'll draw the coordinate axes in special colors to show the three primary directions: x will be green, y will be blue, and z will be red. Instead of calling `box()`, we'll draw the cube as the 12 lines that compose the edges so that we can see the ball inside.

We'll start drawing the cube from the origin in the upper-left corner, followed by the remaining nine edges, using the `mycube()` function 1. To see if the ball has collided with a side, we'll test the ball's coordinates against the x, y, and z values of the bounding planes, which are aligned with the coordinate axes.

We can still use the `sphere()` function to draw the bouncing ball at position (`x`, `y`, `z`) using a translation of the origin to that point before drawing. After each frame, we move the ball an amount (`dx`, `dy`, `dz`). If the ball coordinates are such that the ball extends past any of the cube faces, then the ball bounces—it reverses the direction of motion to move away from the face. This is implemented by the `moveSphere()` function. For example, in the x-direction, this is the specific test for a bounce 2:

```
if (x<=6 || x>=194) dx = -dx;
```

This test is specific for a sphere size of 12, because it checks against the radius of 6 pixels. A sphere of radius *r* is in contact with the cube if its center is within *r* pixels of a face, and *r* is half of the specified sphere size.

Because the cube starts at (0, 0, 0) and is 200 units in each direction, the ball collides around x-coordinates 6 and 194.

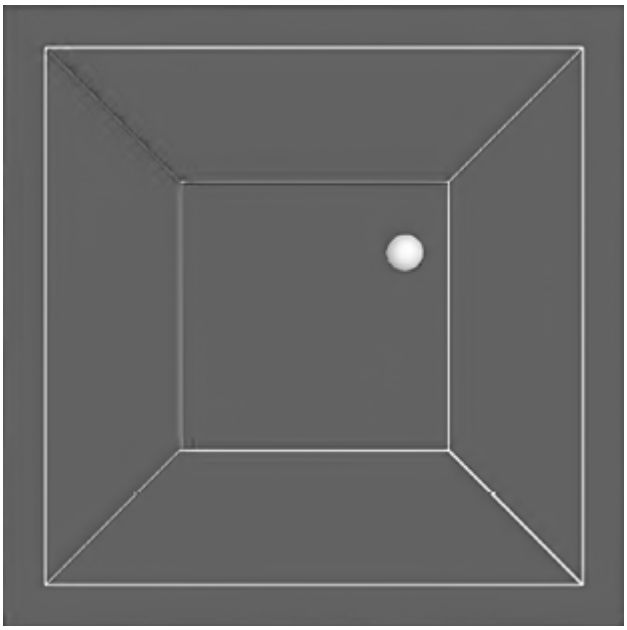
The center of the cube is at (100, 100, 100) 3. This point is the center of the scene. We stare into the cube from the viewpoint at the (`x`, `y`) center, which is (100, 100), but along the z-axis 400 units.

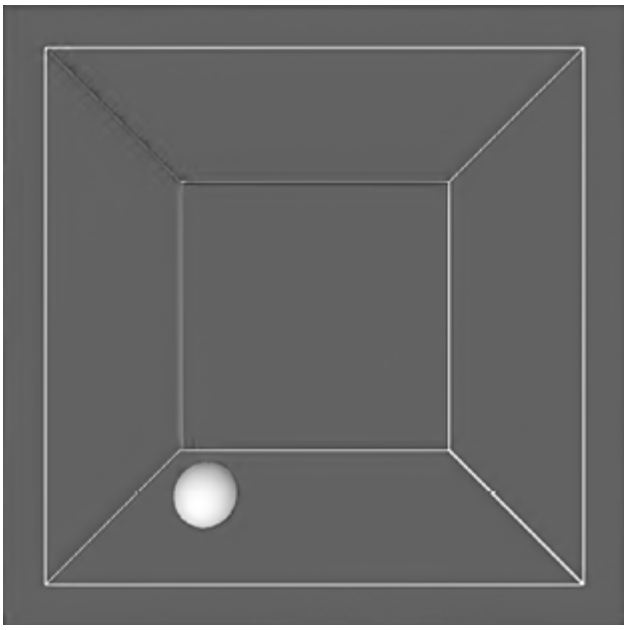
NOTE

A 3D version of the game Pong could be created from this basic program. Paddles could be small rectangles aligned to the opposite y-z axes. The left paddle could be moved with the W, A, S, and D keys, and the right paddle with the arrow keys. Bouncing off of the y-z planes would occur only if the sphere's y- and z-coordinates placed it within the rectangle defined by the paddle.

```
int x=100, y=100, z=100;                                    //
Sphere position
int dx=2, dy=3, dz=4;                                       //
Velocity of the sphere
int eyex= 100, eyey=100, eyez=400;                          //
Viewpoint
int cx=100, cy=100, cz=100;                                 //
Point we are looking at

void setup() {  size (400, 400, P3D); }

1 void mycube ()
{
  stroke (255, 0, 0);  line (0, 0, 0, 0, 0, 200);    //
Z axis is red
  stroke(0, 0, 255);  line (0, 200, 0, 0, 0, 0);     //
Y axis is blue
  stroke(0, 255, 0); line (0, 0, 0, 200, 0, 0);      //
X axis is green
  stroke (255);                                             //
All other edges are white
  line (0, 0, 200, 0, 200, 200);
  line (0, 200, 200, 0, 200, 0);
  line (0, 200, 0, 200, 200, 0);
  line (0, 200, 200, 200, 200, 200);
  line (0, 0, 200, 200, 0, 200);
  line (200, 0, 0, 200, 0, 200);
  line (200, 0, 200, 200, 200, 200);
  line (200, 200, 200, 200, 200, 0);
  line (200, 200, 0, 200, 0, 0);
  noStroke();
}

void moveSphere ()
{     //  Move the sphere position one frame
  x = x + dx;  y = y + dy;  z = z + dz;
2 if (x<=6 || x>=194) dx = -dx;
  if (y<=6 || y>=194) dy = -dy;
  if (z<=6 || z>=194) dz = -dz;
}

void draw ()
{
  background (45, 45, 120);
3 camera(eyex, eyey, eyez, cx, cy, cz, 0, 1, 0);
```

```
  mycube();
  lights();
  noStroke();
  translate (x, y, z);
  sphere (12);
  moveSphere();
}
```





Sketch 55: Constructing 3D Objects Using Planes

Processing provides only spheres and boxes as basic 3D objects, but that does not mean that we can't make more complex things. We can construct arbitrary objects from polygons. This means we need to design the objects first, either on paper or using a 3D modeling program like Blender or Maya. The design yields a set of coordinates of the *vertices* (corner points) of the polygons in three dimensions. Then we can use Processing to draw these polygons and thus display the object.

Since prisms are the easiest objects to build, this sketch will draw a prism and color the various faces differently so we can tell which are which. The point of view will move in a pattern so that the 3D nature of the object is clear.

A *rectangular prism* consists of rectangles joined along their edges. A cube is a rectangular prism, for example. The first step is to determine the values of the coordinates for each of the corners of the rectangles that will compose the prism. A piece of graph paper is useful for this: sketch the prism and define the x, y, z coordinate system (x is horizontal). Then start with the origin, the (0, 0, 0) point, and place the coordinates on the drawing where they belong, as in *Figure 55-1*. Now you can simply read off the coordinates of each rectangle in any order you like. For example, the front face of the prism in the figure is defined by the following coordinates: (0, 0, 0), (sx, 0, 0), (sx, sy, 0), and (0, sy, 0).

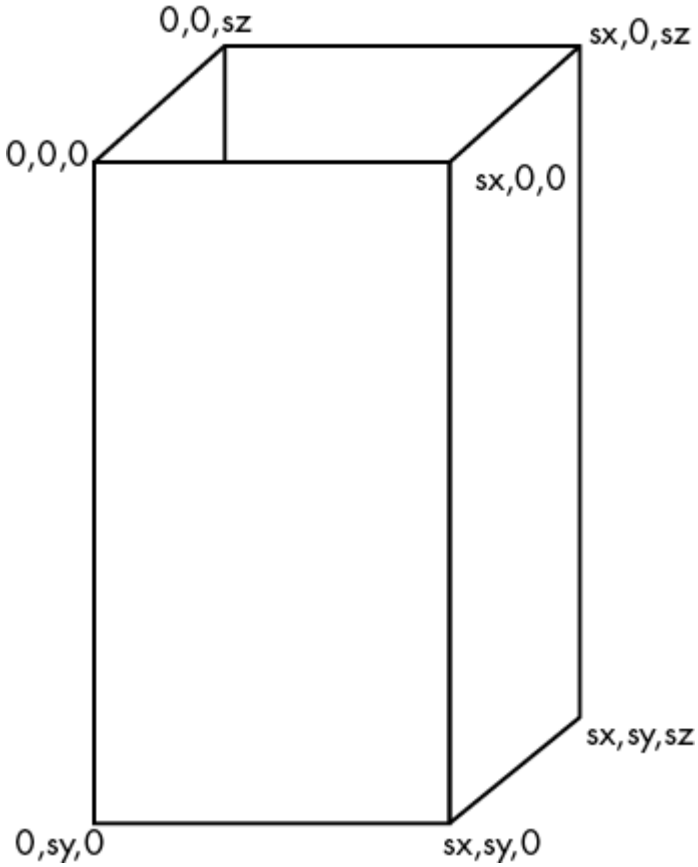


Figure 55-1: 3D coordinates of a prism

To draw polygons that are connected as an object, we bookend that drawing code between calls to the `beginShape()` and `endShape()` functions. In this case, because the polygons used are rectangles, `beginShape()` is passed the argument `QUAD` 1; another option would be `TRIANGLES`. This argument specifies to Processing the number of vertices needed for each polygon (in this case four). Between the begin and end calls, we place calls to a function named `vertex()` 2. Each such call specifies a point in 3D space that represents, in this instance, a corner of a rectangle. For example, the front face of the prism is defined by these calls:

```
vertex (0., 0., 0.);
vertex (sx, 0., 0.);
vertex (sx, sy, 0.);
vertex (0., sy, 0.);
```

The sketch draws four rectangles connected along vertical edges, creating a rectangular prism with no top or bottom. Each is filled with a different

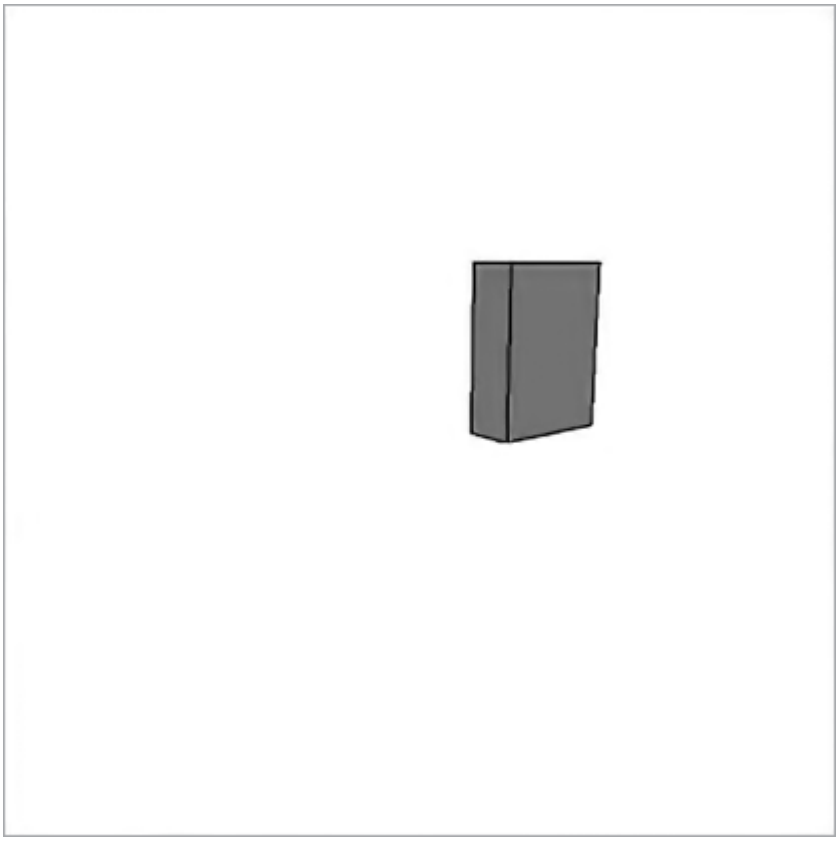
color merely by placing a call to `fill()` immediately before the four vertices for that rectangle are specified.

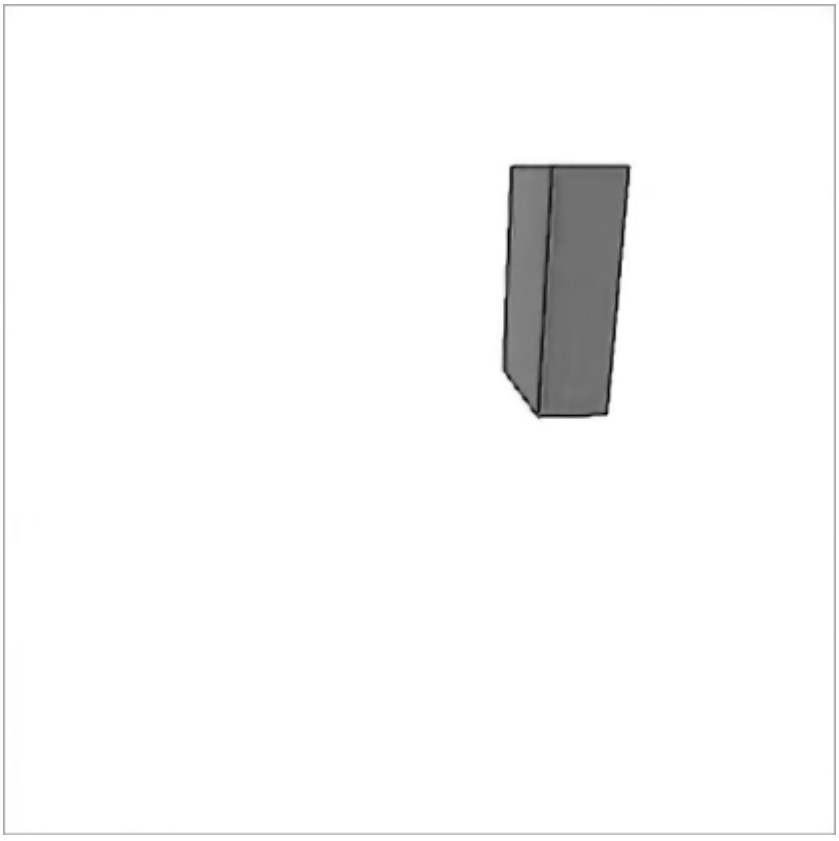
The viewpoint changes by the amount `dz` during each frame between a minimum of z = −200 and a maximum of z = 300 4 so that various views of the prism are displayed.

NOTE

After an object is complete, it can be terminated by a call to `endShape(CLOSE)` 3*; the parameter indicates the polygon should be closed. This will connect the first coordinate and the last so that there is no gap in the polygon. Floating-point approximations can cause gaps because small errors in individual calculations can accumulate and result in small errors. Adding 0.01 to a value 100 times may not be exactly the same as adding 1.*

```
float sx=30., sy=40., sz=12.;
int eyex= 144, eyey=0, eyez=245;
int cx=30, cy=40, cz=32;
int dz = -1;

void setup ()
{
  size(300, 300, P3D);
  stroke(0);
}

void draw ()
{
  background(255);
  camera(eyex, eyey, eyez, cx, cy, cz, 0, 1, 0);

1 beginShape (QUAD);
    fill (170, 120, 50);
2 vertex (0., 0., 0.);
  vertex (sx, 0., 0.);
  vertex (sx, sy, 0.);
  vertex (0., sy, 0.);
    fill (120, 170, 50);
  vertex (sx, 0., 0.);
  vertex (sx, 0., sz);
  vertex (sx, sy, sz);
  vertex (sx, sy, 0.);
    fill (170, 50, 120);
  vertex (sx, 0., sz);
  vertex (0., 0., sz);
  vertex (0., sy, sz);
  vertex (sx, sy, sz);
    fill (50, 120, 170);
  vertex (0., 0., 0.);
  vertex (0., sy, 0.);
  vertex (0., sy, sz);
  vertex (0., 0., sz);
3 endShape (CLOSE);

  eyez = eyez + dz;
4 if (eyez < -200) dz = -dz;
  if (eyez > 300) dz = -dz;
}
```





Sketch 56: Texture Mapping

In Sketch 55 we gave each side of a prism a distinct color to make it easy to identify each face. This was done as an exercise, but in most real applications, a prism would either be a single color or would have a *texture* placed on it. A texture is a pattern, often simply an image, that we apply like a decal to a polygon. In this way, we can make a simple prism look like many things: a building, a book, a chair—nearly anything with corners. This sketch applies a texture (carpet) to a polygon (a rectangle) and moves the viewpoint so that the 3D effect can be seen.

Applying an image to a polygon as a texture is a process called *texture mapping*. The details of the algorithm are complex, but the idea is simple enough, and the way it is implemented in Processing fits nicely into the scheme already explained for drawing objects. In English, the process is as follows:

- Read in an image that will serve as the texture 1. This will be a `PImage`.
- Define the coordinates of a 3D polygon, possibly part of a bigger object.
- Map each of the four corners of the texture image to a vertex of the polygon; that is, if the polygon is a rectangle, decide which corners of the texture image will be placed over which corners of the rectangle.
- Convert the coordinate mapping into calls to the `vertex()` function 4.
- Bracket the vertex calls between `beginShape()` 2 and `endShape()` 5.
- Immediately after `beginShape()`, tell Processing which texture image to use by calling the built-in `texture()` function 3.

In this example, we use an image of carpet texture. As an orientation marker, a red rectangle is placed in the upper-left corner and a green one in the upper right. The texture image is 524 by 928 pixels. This is the coordinate mapping from texture to vertices, as shown in *Figure 56-1*:

Texture (0, 0) maps to polygon (0, 0, 0).

Texture (524, 0) maps to polygon (sx, 0, 0).

Texture (524, 928) maps to polygon (sx, sy, 0).

Texture (0, 928) maps to polygon (0, sy, 0).

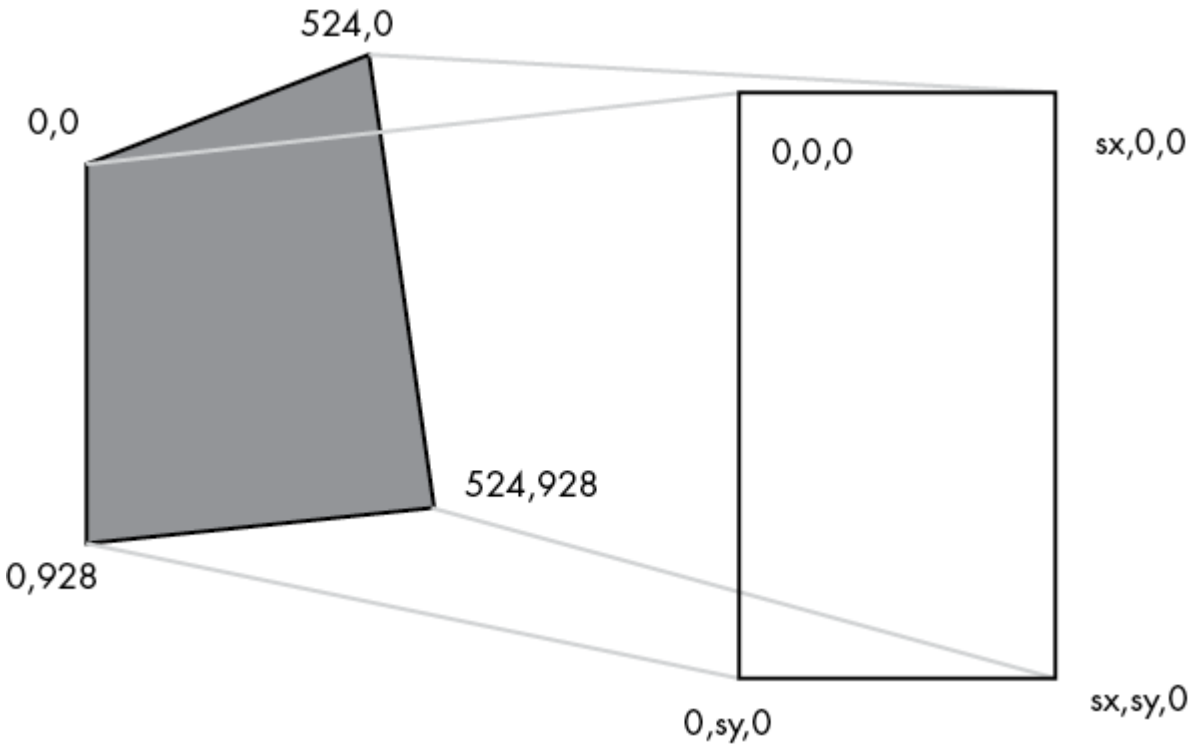


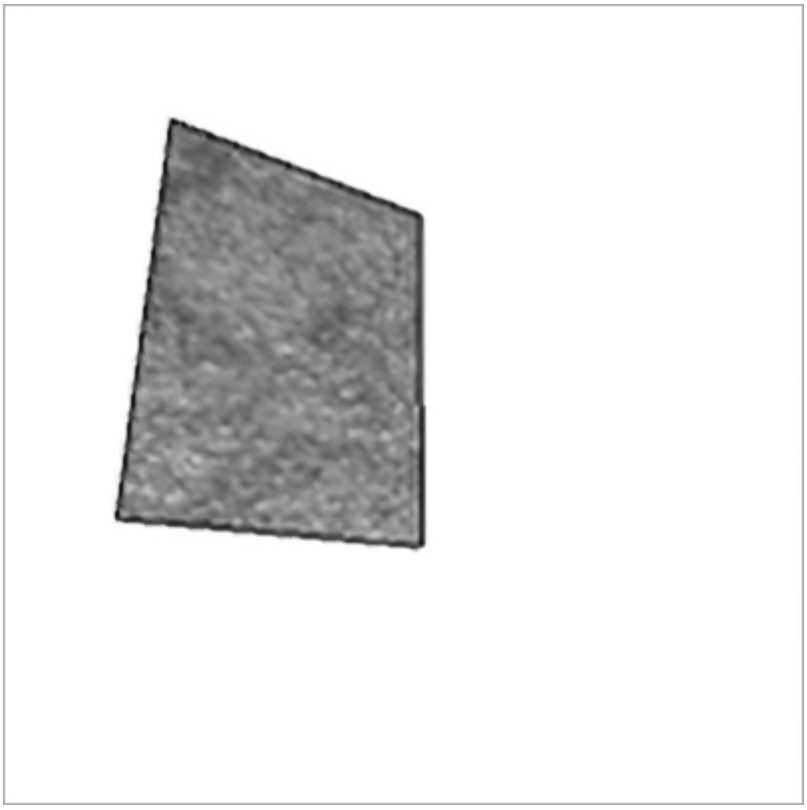
Figure 56-1: Mapping texture coordinates to a polygon

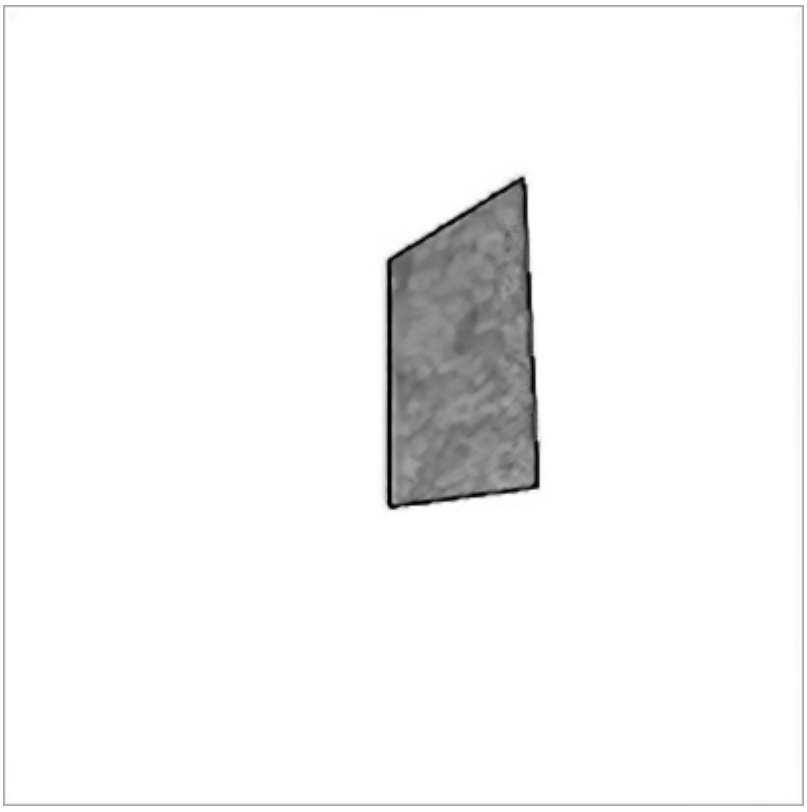
The `vertex()` function allows us to specify the mapping with two optional parameters for texture coordinates. This would be the mapping of the previous vertices:

```
vertex (0., 0., 0.,   0.,  0.);    vertex (sx, 0., 0.,   524,
0.);
vertex (sx, sy, 0.,   524, 928);   vertex (0., sy, 0.,   0.,
928);
```

Because Processing knows the size of the texture image (`timage`), the numeric constant 524 in the preceding mappings can be replaced by `timage.width` 4. Similarly, we can use `timage.height` instead of 928.

```
float sx=30., sy=40., sz=12.;
int eyex=30, eyey=50, eyez=60;
int cx=20, cy=30, cz=12;
int dx = -1;
PImage timage;

void setup ()
{
  size(200, 200, P3D);
  stroke(0);
1 timage = loadImage ("carpets.jpg");
}

void draw ()
{
  background(255);
  camera(eyex, eyey, eyez, cx, cy, cz, 0, 1, 0);

2 beginShape (QUAD);
3 texture (timage);
  vertex (0., 0., 0.,   0.,           0.);
4 vertex (sx, 0., 0.,   timage.width, 0.);
  vertex (sx, sy, 0.,   timage.width, timage.height);
  vertex (0., sy, 0.,   0.,           timage.height);
5 endShape (CLOSE);

  eyex = eyex + dx;
  if (eyex < -30) dx = -dx;
  if (eyex > 100) dx = -dx;
}
```





Sketch 57: Billboards—Simulating a Tree

Let's draw a tree in three dimensions. A prism is a simple thing, but a tree? Trees have many parts: leaves, branches, bark, and myriad details. Graphics specialists have devised very complex methods to create complex things like trees, mountains, and living things, but in most cases, it is not necessary to go to that trouble. For artwork, animations, and games, there are ways to simplify things (to "cheat") so that they look pretty good while still being easy to implement. Building a tree as a *billboard* is one of those things.

In its simplest form, a billboard is a rectangle with a texture drawn on it. It resembles the kind of billboard you can see while driving down tourist highways, and in computer graphics, it would normally occur only at a large distance from the viewer. To make a tree, we'll use two billboards at right angles to each other, joined at the vertical center of each. Each one is a rectangle with a tree image textured onto it. The idea is that from any angle one sees the entire tree, and moving the viewpoint appears to change the view of the tree. From close up it is obvious what is happening, but when seen from a medium distance or while the viewer is moving, the illusion is a good one.

Figure 57-1 shows how we arrange the two perpendicular rectangles in three dimensions. The texture placed on them needs to have a transparent background, or the white rectangles will be visible. This means using either a GIF or PNG format image file, which are the ones that support transparency.

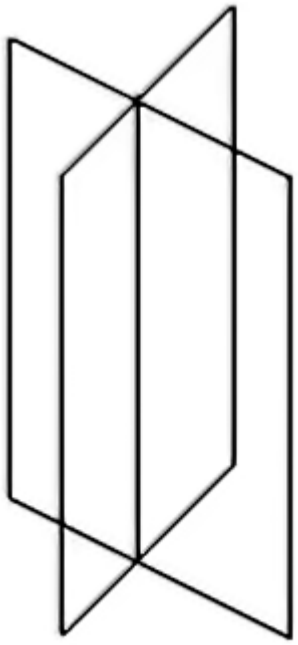


Figure 57-1: Two perpendicular rectangles

The sketch first reads in the tree image that we'll use as a texture and opens the window, as usual. The `draw()` function sets up the camera 1 and draws two rectangles at the origin, both using the tree as a texture that we map onto the rectangles 2, similar to what was done in Sketch 56. We rotate the second texture-mapped rectangle by 90 degrees 3 and translate it by 13 units in the x and z directions to align it with the center of the first rectangle. (The rectangle is 26 units wide, and 13 units is half of that.)

We also change the viewpoint slightly in each frame so that the 3D effect is obvious when the sketch is executing.

NOTE

The use of billboards can be more interesting and dynamic than this. Imagine that we need a burning torch. We could make a set of animation frames of a burning torch using Paint and then map them onto the billboard in sequence. The result is a convincing representation of a burning torch, especially if the background is otherwise dark.

```
PImage tree; // https://pngimg.com/download/204/

float sx=30., sy=40.;
int eyex=30, eyey=20, eyez=60;
int cx=20, cy=15, cz=12;
int dx = -1;

void setup ()
{
  size (500, 400, P3D);
  tree = loadImage ("tree.gif");
  noStroke();
}

void draw ()
{
  background (200, 255, 0);
1 camera(eyex, eyey, eyez, cx, cy, cz, 0, 1, 0);
  beginShape (QUAD);
2 texture (tree);
  vertex (0., 0., 0.,   0.,         0.);
  vertex (sx, 0., 0.,   tree.width, 0.);
  vertex (sx, sy, 0.,   tree.width, tree.height);
  vertex (0., sy, 0.,   0.,         tree.height);
  endShape (CLOSE);

3 rotateY (PI/2.0);
  translate (-13, 0, 13);

  beginShape (QUAD);
  texture (tree);
  vertex (0., 0., 0.,   0.,         0.);
  vertex (sx, 0., 0.,   tree.width, 0.);
  vertex (sx, sy, 0.,   tree.width, tree.height);
  vertex (0., sy, 0.,   0.,         tree.height);
  endShape (CLOSE);

  eyex = eyex + dx;
  if(eyex<-40 || eyex>60) dx = -dx;
}
```

Sketch 58: Moving the Viewpoint in 3D

In a first-person computer game, the representation of the player in the game is an *avatar*, controlled by the player. Pressing W moves the avatar forward, S moves it backward, A moves it left, and D moves it right. This scheme is easy for a player to understand but harder to implement than the scheme we have been using.

In the sketches presented so far, the movement has been automatic or based on simplistic assumptions—A and D move along the x-axis and W and S move along the z-axis—but people don't move in that way. The A and D keys should rotate the player about their own axis, and the W and S keys should move the player forward and backward along the direction defined by that angle. As a demonstration of avatar movement control, this sketch draws nine cubes and allows the user to move among them using this technique.

The avatar has a direction in which it is facing, defined by the variable `angle` (in degrees). The A and D keys allow the user to change this angle by one degree per key press 3. Changing the angle will not modify the camera position, but it does modify the center of the scene by rotating it about the avatar. Because the vertical axis is y, we can calculate this in the x-z plane as a simple trigonometric relationship 4:

```
cx = cos(radians(angle))*20000.0;
cz = sin(radians(angle))*20000.0;
```

The value 20,000 represents a large distance, effectively infinite, that provides a distant focus point.

Pressing W moves the avatar one unit along the direction it is facing, which is the variable `angle`. Each unit moved changes the `x` position by `dx` and the `z` position by `dz`, as defined in *Figure 58-1*.

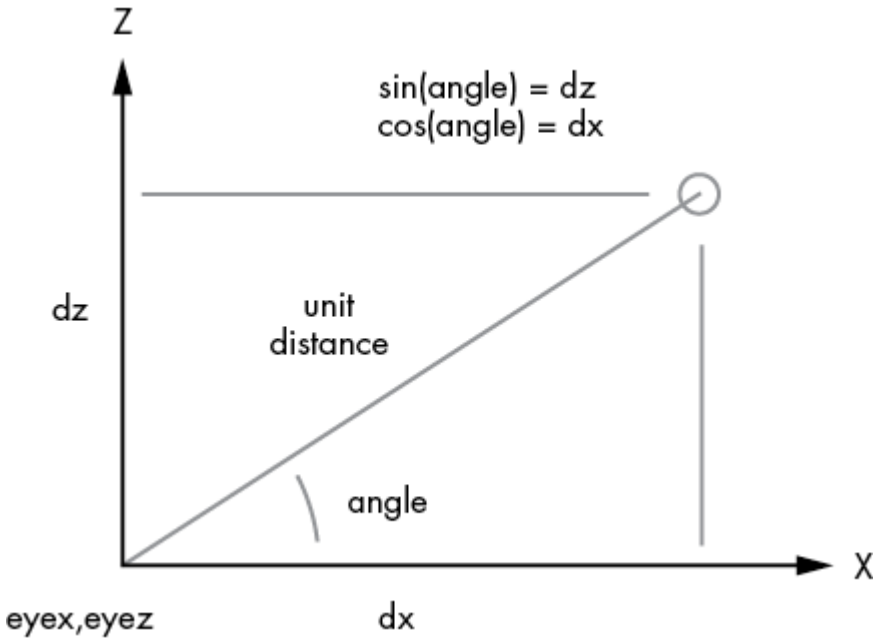


Figure 58-1: Converting (x, z) motion to (angle, distance)

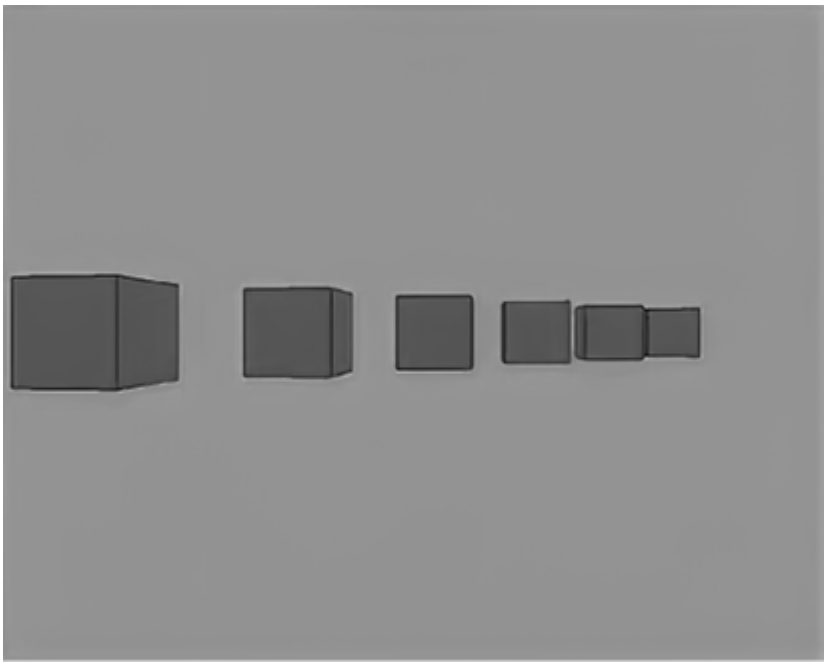
The position of the avatar is (`eyex`, `eyez`), and it is likely that for any given forward 1 or backward 2 movement, both of these values will change. One key press will move the avatar 5 units, or `dx*5`.

NOTE

The `radians()` function can be used to convert an angle in degrees into radians.

```
int eyex=30, eyey=0, eyez=60;
int cx=20000, cy=15, cz=20000;
float angle = 0.0;
float dx = 1.0, dz = 1.0, sx, sy, sz;


void setup ()
{
  size (500, 400, P3D);
  fill (200,0,0);
  keyPressed();  // Initialize the viewpoint
}

void draw ()
{
  background (200, 255, 0);
  camera(eyex, eyey, eyez, cx, cy, cz, 0, 1, 0);
  for (int i=0; i<9; i++)  // Draw nine boxes (30,0,30)
apart
  {
    translate (30, 0, 30);
    box (20);
  }
}

void keyPressed ()
{
  if (key == 'w')          // Move 'forward'
  {
1 eyex += 5*dx;    eyez += 5*dz;
  }
  else if (key == 's')     // Move 'backward'
  {
2 eyex -= 5*dx;      eyez -= 5*dz;
  }
  else if (key == 'a')     // Turn left a unit (CCW)
3 angle = angle - 1.0;
  else if (key == 'd')     // Turn right a unit (CW)
    angle = angle + 1.0;
  if (angle < 0) angle = angle + 360.0;
  else if (angle > 360.0) angle = angle - 360.0;
  dx = cos(radians(angle)); dz = sin(radians(angle));
4 cx = (int)(dx*20000.0);  // cx = x coordinate of
center point
  cz = (int)(dz*20000.0);  // cz = z coordinate of
```

```
center point
}
```



Sketch 59: Spotlights

If the ambient illumination is off and the background is dark, any objects drawn within the 3D space of the Processing graphics world will not be visible. This sketch simulates illumination in a new way—as a small spotlight source in a dark space. The spotlight shines on the center-of-scene coordinates, and the rest of the scene is unlit. The sketch places three cubes of different colors around the scene 2, and the user can explore the space by rotating and watching for the cubes to light up.

This sketch uses the same code for `keyPressed()` as does the previous sketch, so the avatar can rotate and move forward and backward 3. A Processing spotlight is placed at the camera coordinates 1:

```
spotLight(255,255,20, eyex,eyey,eyez, cx,cy,cz, PI/4, 300);
```

The first three parameters (`255`, `255`, `20`) of the spotlight represent the RGB values for the color of the light, the next three (`eyex`, `eyey`, `eyez`) are the 3D coordinates of the light, and the next three (`cx`, `cy`, `cz`) are the coordinates toward which the light is pointed. This means that wherever the camera/avatar moves, a spotlight is shining on the center of the scene. The angle for the light, `PI/4` (45 degrees) is the 10th parameter, and we can increase or decrease it to see what happens to the scene. The value `300` indicates how strongly the light concentrates near the center of the spot, with larger numbers being more focused.

We can define lights for other types of local illumination. Car headlights, for example, are simply two spotlights separated by a small distance. There is a commented-out statement that adds a second light to the one in the sketch:

```
spotLight(255,255,20, eyex+3*dz,eyey,eyez+3*dx, cx,cy,cz,
PI/4, 300);
```

Spotlights are only visible by their light reflected off of objects. They cannot be seen as glowing objects. The same is true of point lights and other sources. In that sense, lights are not objects. Surrounding a light with an

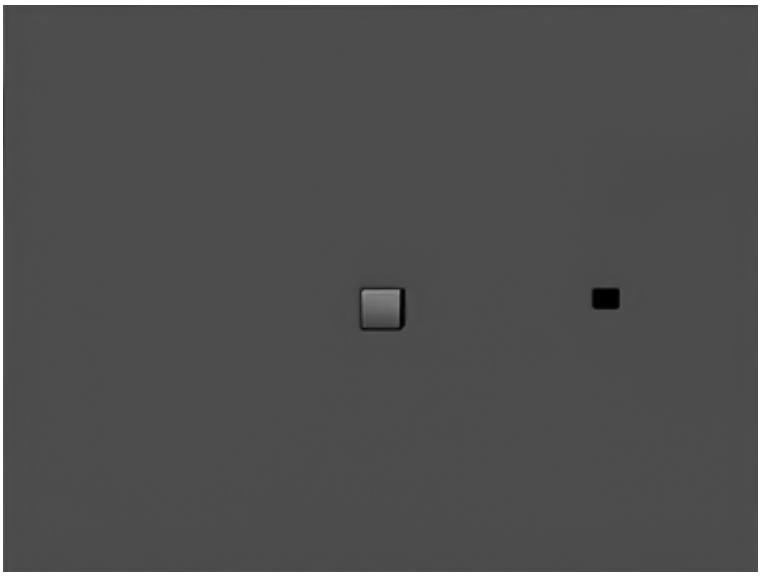
object illuminates the objects around it but does not make the light source visible.

We can make lights flash on and off or change color by alternately calling the `spotLight()` function or not, depending on a flag that is either true or false, here named `flash`. Simply change a counter after each frame, and change the flag after a fixed number of frames (20 here). The following code illuminates one of two spheres alternately with red or blue, like police car lights.

```
if (count % 20 == 0) flash = !flash;  // If flash is true,
make it false
if (flash)   spotLight(255,0,0,  0,  335,  0, 0,  -1, 0,
PI/4, 300);
else   spotLight(0,0,255,  50,  335,  0, 0,  -1, 0, PI/4,
300);
sphere(20);
translate (50, 0, 0); sphere(20); translate (50, 0, 0);
```

```
float eyex = 35, eyey = 10, eyez = -300.0;
float cx = 200.0, cy = 5.0, cz = 100.0;
float dx = 0.0, dz = 1.0;
float angle = 90.0;

void setup ()
{
  size(400, 300, P3D);
  keyPressed ();
}

void draw ()
{
  background (60);
  camera (eyex, eyey, eyez, cx, cy, cz, 0.0, -1.0,
0.0);
1 spotLight(255,255,20, eyex, eyey, eyez,cx, cy,
cz,PI/4, 300);
//  spotLight(255,255,20, eyex+3*dz, eyey, eyez+3*dx,
cx, cy, cz, PI/4, 300);
2 fill (255,255,255); box (20);
  fill (255,0,0); translate (200, 0, 300); box(20);
  fill (0,255,255); translate (-50, 0, -400); box(20);
}

3 void keyPressed ()
{
  if (key == 'w')           // Move 'forward'
  {
    eyex += 5*dx;    eyez += 5*dz;
  }
  else if (key == 's')     // Move 'backward'
  {
    eyex -= 5*dx;    eyez -= 5*dz;
  }
  else if (key == 'a')     // Turn left a unit (CCW)
    angle = angle + 1.0;
  else if (key == 'd')     // Turn right a unit (CW)
    angle = angle - 1.0;
  if (angle < 0) angle = angle + 360.0;
  else if (angle > 360.0) angle = angle - 360.0;
  dx = cos(radians(angle)); dz = sin(radians(angle));
  cx = (int)(dx*20000.0);  // cx = x coordinate of
center point = cos(angle)*20000
  cz = (int)(dz*20000.0);  // cz = z coordinate of
```

```
center point = sin(angle)*20000
}
```



Sketch 60: A Driving Simulation

Driving simulations and games have a specific, standard interface and visual presentation. Unlike previous sketches, where users can move about the space but are not themselves visible, driving simulations display the avatar as a car, and the camera (viewpoint) is usually behind and above the car so that the view of the car is always looking forward. Cars drive on roads, so a background is important; without one the user can't tell when they are on a road or have any real idea of how fast they are moving. This sketch will allow a user to drive a vehicle (a rectangular prism, actually) around a track, using the same scheme to move the avatar as before.

The first thing to do is create a track. It will simply be an image, so we can use Paint or some other drawing program. It should be large enough so that it provides some entertainment value (variety) and does not distort too badly when displayed. The example shown in *Figure 60-1* is 1,000×1,000 pixels.

Figure 60-1: A simple track for driving on

The sketch reads this image and uses it as a texture for a 1,000×1,000 square drawn on the x-z plane 1:

```
beginShape(QUADS);
  texture (track);
  vertex (0, 0, 0, 0, 0);   vertex (1000, 0, 0, 1000, 0);
  vertex (1000, 0, 1000, 1000, 1000);
  vertex (0, 0, 1000, 0, 1000);
endShape();
```

The viewpoint needs to be above and behind the car. If the variables `dx` and `dz` represent the unit change in the x and z directions for the given `angle` (the current facing angle of the car), and `carX` and `carZ` are the horizontal and vertical positions of the car, then this should be the viewpoint 3:

```
eyex = carX - dx*50;  eyez = carZ - dz*50;
```

And it should have some fixed height `eyey=20`. The value `50` is a scale factor that depends on the image size.

We draw the car at coordinates `(carX, 0, carZ)`. After each step, these coordinates change as a function of the car's speed (variable `velocity`); the `velocity` value increases or decreases as the user presses the W and S keys 4 (as opposed to previous sketches in which we moved forward and backward using those keys). The W key is the accelerator pedal, and the S is the brake. The car maintains its speed once the user presses one of those keys, and the user can focus on steering using A and D.

We calculate the motion of the car as follows:

```
carX = carX + velocity*dx
carZ = carZ + velocity*dz
```

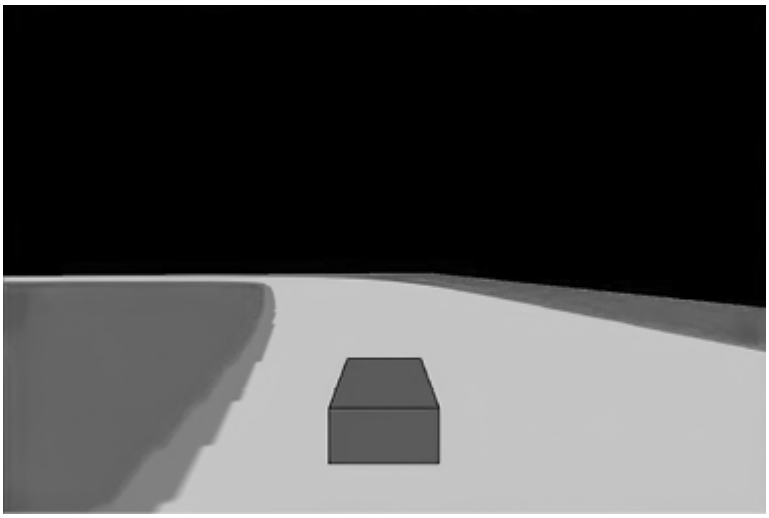
Then we use the `translate()` function to make the car face the direction of motion 2. The car should always face away from the camera, so to make the car face in the correct direction, we rotate it by `-angle`.

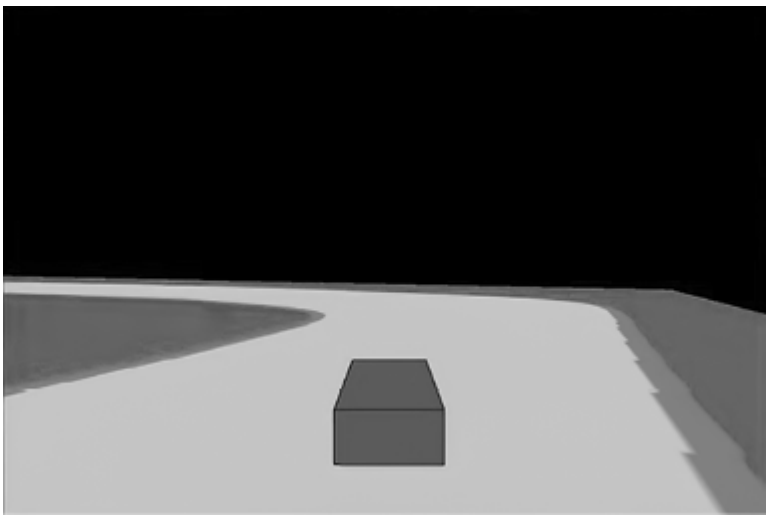
The effect is that the car (a red prism) can speed up (W) and slow down (S) and turn left (A) or right (D) so as to stay on the grey circular path, and

the camera follows the car at a discreet distance.

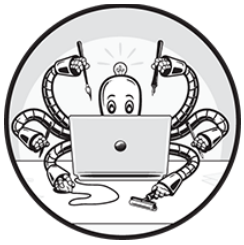
```
PImage track;
float eyey = 20,eyex = 213,eyez = 400.0;
float cx=200.0, cy=5.0, cz= 200.0,dx = 0.0, dz = 1.0;
float angle = 265.0, velocity=0.,carX=200, carY=0,
carZ=310;

void setup ()
{
  size (600, 400, P3D);
  track = loadImage ("road.png");
  keyPressed();
  fill (180, 30, 30);
}

void draw ()
{
  background(0);
1 beginShape(QUADS);
    texture (track);
    vertex (0, 0, 0, 0, 0);
    vertex (1000, 0, 0, 1000, 0);
    vertex (1000, 0, 1000, 1000, 1000);
    vertex (0, 0, 1000, 0, 1000);
  endShape();
  translate (carX, 4, carZ);
2 rotateY(-radians(angle));
  box(20, 5, 10);
  carX = carX + velocity*dx;
  carZ = carZ + velocity*dz;
3 eyex = carX - dx*50;  eyez = carZ - dz*50;
  camera (eyex, eyey, eyez, cx, cy, cz, 0.0, -1.0,
0.0);
}

void keyPressed ()
{
4 if (key == 'w') velocity += .1;  // Move 'forward'
  if (key == 's') velocity -= .1;  // Move 'backward'
  if (key == 'a') angle = angle +2.0; // Turn left a
unit (CCW)
  if (key == 'd') angle = angle -2.0; // Turn right a
unit (CW)
  if (angle < 0) angle = angle + 360.0;
  else if (angle > 360.0) angle = angle - 360.0;
  dx = cos(radians(angle)); dz = sin(radians(angle));
```

```
  cx =(int)(dx*20000.0);  // cx = x coordinate of
center point = cos(angle)*20000
  cz =(int)(dz*20000.0);  // cz = z coordinate of
center point = sin(angle)*20000
}
```





8
ADVANCED GRAPHICS AND ANIMATION



Sketch 61: Layering

One of the key features of modern graphics programs like Photoshop is the idea of *layers*. That's the creation of a set of graphical objects (images) that are placed on top of each other to achieve a complex effect. Transparency makes it possible to see objects on lower layers. This sketch uses three layers: an image of the moon, a circle around a crater, and a targeting display (*reticle*). Using the keyboard, the user can reposition the moon image. The goal of the interface is to allow the user to align the reticle with the target circle.

Drawing inside the sketch window involves using a graphics object class called `PGraphics`. The `background()`, `line()`, and `ellipse()` functions, and many others, are part of the `PGraphics` class, though we can use them without a `PGraphics` object. Alternatively, our drawing can take place in one of these objects and then be displayed on the screen later by calling `image()`. This sketch will draw the moon image with an ellipse highlighting a crater inside of a `PGraphics` instance, and we'll then display it in the sketch window.

The variable used for the `PGraphics` object is named `pg`, and the function that creates one is called (reasonably enough) `createGraphics()` 1:

```
PGraphics pg;
pg = createGraphics(moon.width, moon.height);
```

Here, `moon` is the `PImage` variable that holds the background image of the moon.

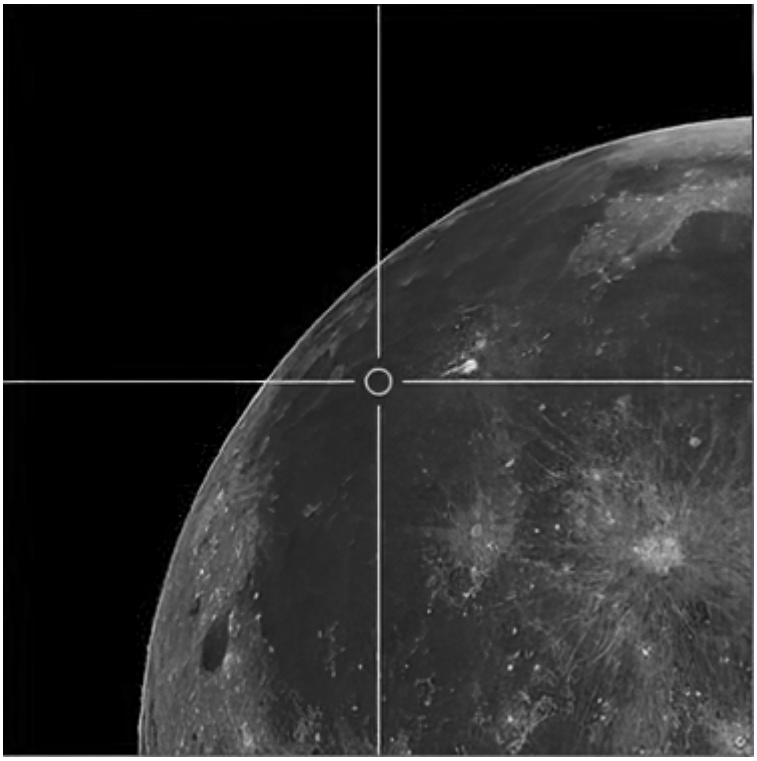
To draw in a `PGraphics` object, we use the graphics functions that we have used before, but specify the `pg` variable as the target 2:

```
pg.beginDraw();
pg.image(moon, 0, 0);
pg.stroke (0, 200, 0);
pg.noFill();
pg.ellipse (393, 233, 12, 12);
pg.endDraw();
```

Drawing is preceded by a call to `beginDraw()`, a function that is similar to a bracket; the corresponding end bracket is a call to `endDraw()`. If you don't use these calls, Processing doesn't initialize the object, and drawing will not work (even though Processing may not generate an error). The preceding code draws the moon image in the `PGraphics` objects and draws a circle around a target.

The `draw()` function displays the `PGraphics` object using the call `image(pg, xoff, yoff)`, where `xoff` and `yoff` are positional offsets that are controlled using key presses of W, A, S, and D in the traditional way 3. (A `PGraphics` object has many of the properties of a `PImage`, since `image()` can display both.) The values of `xoff` and `yoff` are generally negative so that the underlying graphic gets shifted left and up under the window, which remains stable, from its starting point in the upper-left corner. The `draw()` function also draws the reticle as a small set of lines that point to the center of the window 4.

```
PImage moon;
int xoff=0, yoff=0;
PGraphics pg;

void setup ()
{
  moon = loadImage ("moon.jpg");
  size (300, 300);
  pg =1createGraphics(moon.width, moon.height);
2 pg.beginDraw();
  pg.image(moon, 0, 0);
  pg.stroke (0, 200, 0);
  pg.noFill();
  pg.ellipse (393, 233, 12, 12);
  pg.endDraw();
  stroke (200);
  noFill();
}

void draw ()
{
  background (200);
3 image(pg, xoff, yoff);
4 line (0, height/2, width/2-10, height/2);
  line (width/2+10, height/2, width, height/2);
  line (width/2, 0, width/2, height/2-10);
  line (width/2, height/2+10, width/2, height);
  ellipse (width/2, height/2, 10, 10);
}

void keyPressed ()
{
  if (key == 'w') yoff = yoff + 1;        // Move up
  else if (key == 's') yoff = yoff -1;    // Move down
  else if (key == 'a') xoff = xoff +1;    // Move left
  else if (key == 'd') xoff = xoff -1;    // Move right
  if (xoff > 0) xoff=0;
  if (xoff < -(moon.width-width)) xoff = -(moon.width-
width);
  if (yoff > 0) yoff = 0;
  if (yoff < -(moon.height-height)) yoff = -
(moon.height-height);
}
```



Sketch 62: Seeing the World Through a Window

Many games, animations, and simulations (driving or space travel, for example) use a view through a window as a part of the interface. This sketch implements a window that looks out on a 3D scene and allows the user to move about that scene while looking through the window.

This is a more advanced application of `PGraphics`. We'll render a simple 3D scene to a `PGraphics` instance named `pg`, read a 2D image with transparent sections (the window) into a `PGraphics` instance named `g2`, and draw the two graphics objects to the screen using calls to `image()`.

The 3D primitives we'll use to draw the 3D scene are all part of `PGraphics`. We'll enable the 3D rendering engine with a parameter to `createGraphics()` 1, instead of to `size()` as in Sketch 51, and then we'll set up the 3D parameters with calls to `camera()` 2 and `ambientLight()`. The basic call to `size()` sets up the graphics window; each `PGraphics` instance is like having a distinct window to draw in, and all of the usual graphics methods can be used via the dot notation: `pg.line()`, `pg.ellipse()`, and so on. No `PGraphics` object is visible until drawn in the graphics window. Thus we can create a simulated 3D space inside the `pg` object, drawing four cubes there that provide targets to be viewed through the window.

The 2D portion involves displaying a 2D image (a `PImage` variable named `back`) that represents the window (*Figure 62-1*). The GIF image has transparent sections, created by defining a color (in this case green) as transparent using an image editor like Photoshop. We call this kind of image a *stencil*.

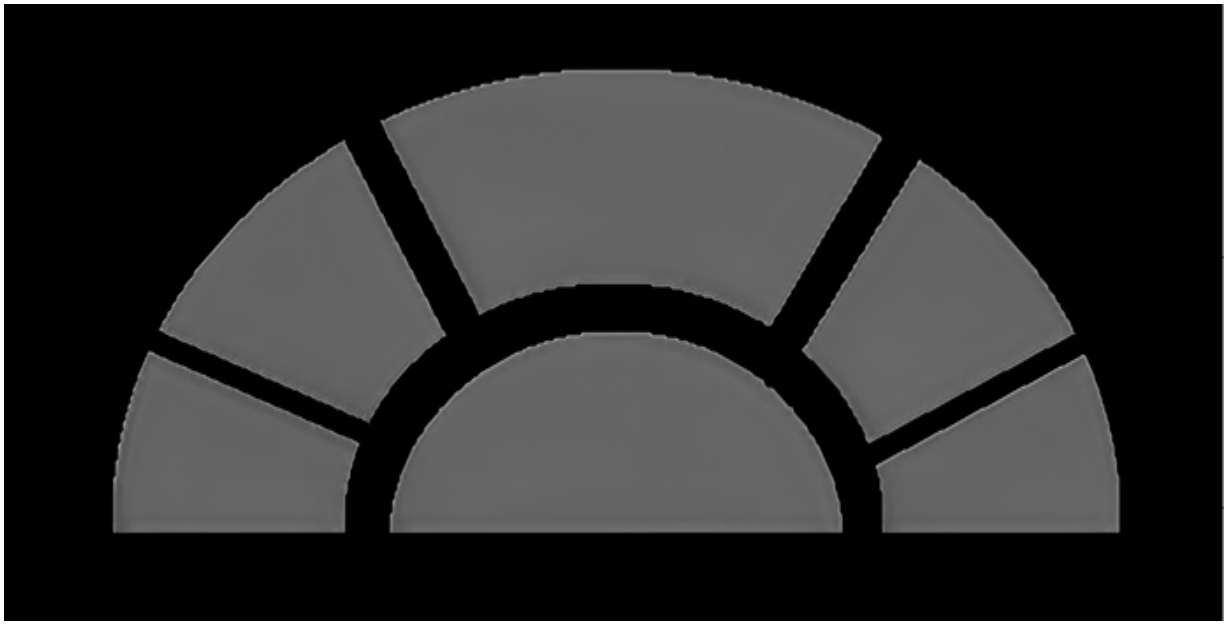


Figure 62-1: The stencil for the window

The sketch draws the two images on the window, `pg` first (the 3D rendering) 3 followed by `g2` (the stencil) 4. The transparent parts of `g2` allow the 3D scene to be seen through the window portions.

The user can control the viewpoint for the 3D scene using the keyboard in the usual way 5. The 3D scene changes as a consequence of the change in the viewpoint, but the 2D scene does not. The result is that the window stays in the same place but the view seen through it (the transparent portions) changes as a function of that viewpoint, as if the user were inside a moving vehicle looking at a scene outside.

```
int ex=-70, ey=0, ez=-225, cx=0, cy=0, cz=0;
float dx, dz, angle=74;
PGraphics pg, g2;
PImage back;

void setup ()
{
  size(200, 200, P2D);
  surface.setResizable(true);
  back = loadImage ("window.gif");
  surface.setSize(back.width, back.height);
  pg = createGraphics(back.width, back.height,1P3D);
  g2 = createGraphics (back.width, back.height);
  g2.beginDraw(); g2.image(back, 0, 0); g2.endDraw();
}
void draw ()
{
  background (200);
  pg.beginDraw();
  pg.background(0,0,200);
2 pg.camera (ex, ey, ez, cx, cy, cz, 0.0, -1.0, 0.0);
  pg.ambientLight (0, 200, 0);
  pg.translate (100, 0, 100); pg.box(20); pg.translate
(-100, 0, -100);
  pg.translate (-100, 0, 100); pg.box(20); pg.translate
(100, 0, -100);
  pg.translate (100, 0, -100); pg.box(20); pg.translate
(-100, 0, 100);
  pg.translate (-100, 0, -100); pg.box(20);
pg.translate (100, 0, 100);
  pg.endDraw();
3 image(pg, 0, 0);4image (g2, 0, 0);
}

void keyPressed ()
{
5 if (key == 'w')   // Move 'forward'
  {   ex += 5*dx;   ez += 5*dz;
  }
  else if (key == 's')  // Move 'backward'
  {   ex -= 5*dx;    ez -= 5*dz;
  }
  else if (key == 'a')     // Turn left a unit (CCW)
    angle = angle + 1.0;
  else if (key == 'd')     // Turn right a unit (CW)
```

```
    angle = angle - 1.0;
  dx = cos(radians(angle)); dz = sin(radians(angle));
  cx = (int)(dx*20000.0);  // cx = x coordinate of
center point = cos(angle)*20000
  cz = (int)(dz*20000.0);  // cz = z coordinate of
center point = sin(angle)*20000
}
```

Sketch 63: The PShape Object—A Rotating Planet

This sketch will display a rotating planet (a sphere) and allow the user to move around it in 3D. The new part in this sketch is texture-mapping the planet's surface onto the sphere, which is really a collection of polygons. One way to do this would be to build a model of a sphere out of polygons and do the texture-mapping within a `beginShape()` and `endShape()` block. An easier way is to use a `PShape` object, which is a data type for storing arbitrary shapes.

To implement the rotating planet, we'll create a `PShape` object through a call to `createShape()`, which allows us to build arbitrarily complex shapes using the large set of drawing operations provided by the `PShape` class. It is possible to create almost anything using a `PShape`, and the documentation available online is necessary for complex creations. Our case is simple, because a sphere is one of the shapes provided. This is the call that makes the planet, where `globe` is a `PShape` object 1:

```
globe = createShape(SPHERE, 100);  // 100 is the size of the
sphere
```

The texture, a map of Mars as a `PImage` variable named `timg`, is applied using `globe.setTexture(timg)` 2.

Then we display the planet in `draw()` using a call to the `shape()` function 3:

```
translate (x, y, z);
globe.rotateY(radians(0.5));  // rotateY function is a part
of PShape
shape(globe);                 // This displays the shape in
the window
```

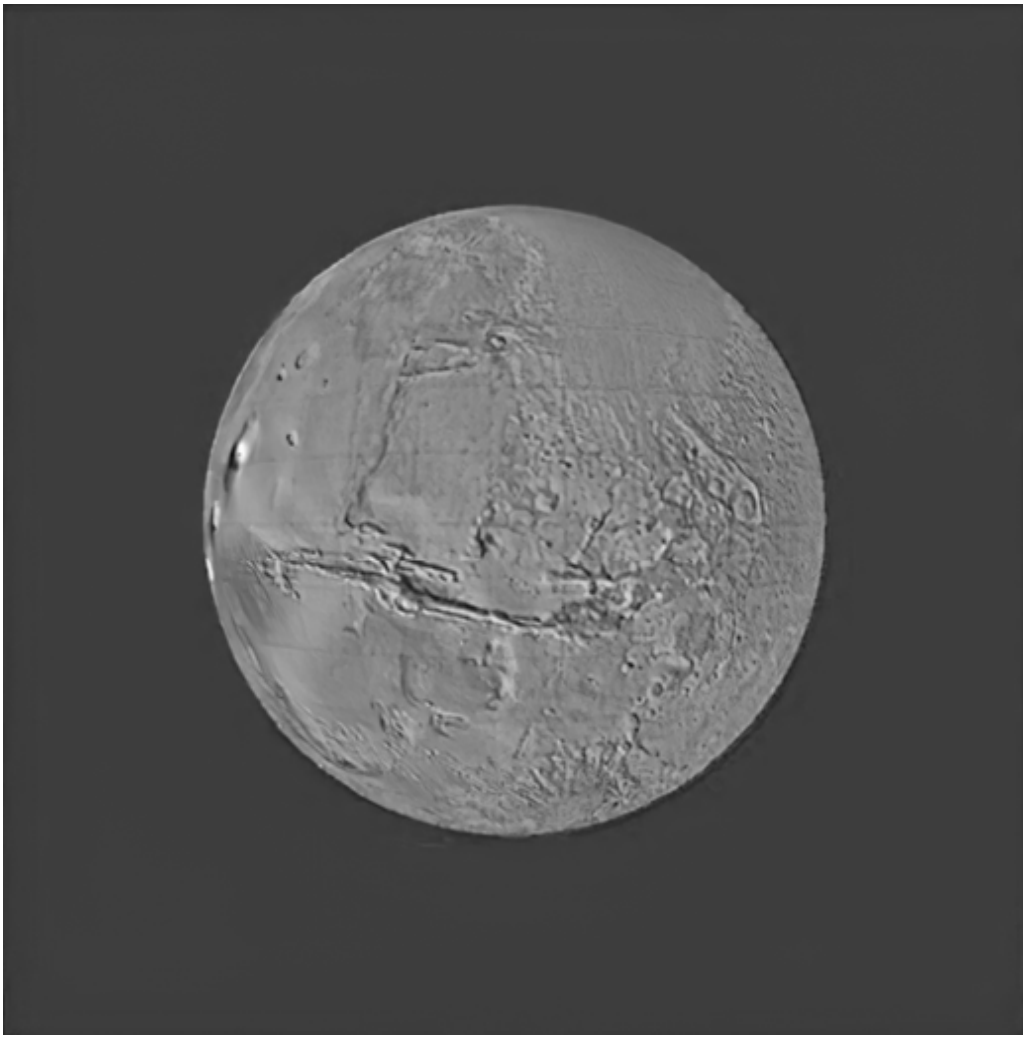
This code positions the sphere in the center of the field of view and rotates it about its own axis before displaying it. The usual keys allow the user to change the viewing position.

NOTE

A 3D modeling package such as 3D Studio Max or Maya creates shapes such as cars and chairs out of polygons and saves them to a file with the suffix .obj. *Animations and games use polygon files to create objects within their imaginary worlds by rendering and texture-mapping the polygons. Processing programs use the* `loadShape()` *function to read an* .obj *file and return a Processing* `PShape` *object. Here is an example code sequence:*

```
PShape s;
s = loadShape ("chair.obj");
shape (s);
```

```
int x=100, y=100, z=100;            // Sphere position
int eyex=100, eyey=100, eyez=400;   // Viewpoint
int cx=100, cy=100, cz=100;         // Point we are
looking at
PShape globe;
PImage timg;
float theta=270, dx=0, dz=0;

void setup ()
{
  size (400, 400, P3D);
  frameRate(10);
  timg = loadImage("globe03.jpg");
1 globe = createShape(SPHERE, 100);
  globe.setStroke(255);
2 globe.setTexture(timg);
}

void draw ()
{
  background (45, 45, 120);
  camera(eyex, eyey, eyez,
         cx, cy, cz, 0, 1, 0);
  translate (x, y, z);
  globe.rotateY(radians(0.5));
3 shape(globe);
}

void keyPressed ()
{
  if (key == 'w')           // Move 'forward'
  {  eyex += 5*dx;   eyez += 5*dz;  }
  else if (key == 's')      // Move 'backward'
  {  eyex -= 5*dx;    eyez -= 5*dz;   }
  else if (key == 'a')      // Turn left a unit (CCW)
    theta = theta + 1.0;
  else if (key == 'd')      // Turn right a unit (CW)
    theta = theta - 1.0;
  dx = cos(radians(theta)); dz = sin(radians(theta));
  cx = (int)(dx*20000.0);
  cz = (int)(dz*20000.0);
}
```



Sketch 64: Splines—Drawing Curves

So far, we've rendered simple geometric objects like lines, ellipses, rectangles, and spheres using predefined Processing functions. But many real-world objects are not linear or elliptical; they have complex shapes. Examples are legion, including cars, fan blades, jewelry, clothing, and living things—even graphs of data. In Processing, complex shapes are rendered using curves. To demonstrate, this sketch allows the user to draw curves using a series of mouse clicks and to see how the selected "control points" affect the curves.

Processing uses *splines* to render curves. In the earlier days of drafting, when people used pencils and T-squares, people used something called a *spline* to draw smooth, oddly shaped curves. It was a long, flexible metal strip that could hold a shape, align with points on paper, and allow the drafter to connect them using a pencil. Mathematically, a spline is a polynomial function that approximates a curve by using a set of points. The details can be complex, but the idea is to use many polynomials connected end to end to build the curve. Processing hides the complexity.

Processing provides a function named `curve()` that implements a type of polynomial named the *Catmull-Rom spline*. This function uses four points to define each section of the curve. The first two define the direction the curve will have at the beginning, and the second two define the direction it will have at the end. The curve itself consists of a set of points (pixels) between the middle two points. As seen in *Figure 64-1*, the angle defined by the first two points establishes the direction of the curve at point P_1, which defines the shape of the polygon between P_1 and P_2; we establish the direction of the curve at P_2 by the direction between P_2 and P_3. In the figure, the points P_1 and P_2 in the two examples are the same, but the curves have a different shape due to the different positions of P_0 and P_3.

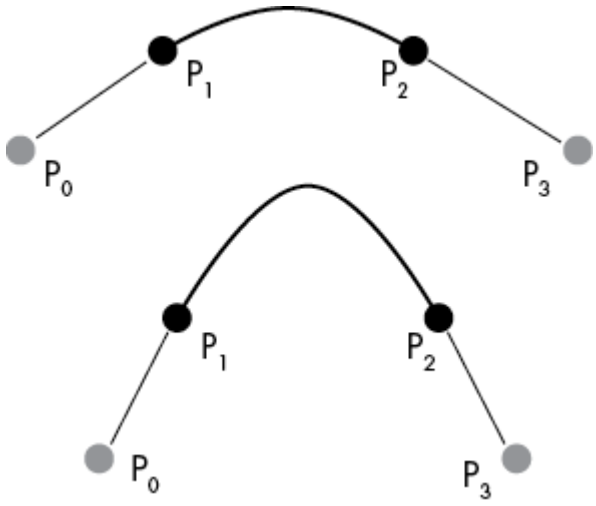


Figure 64-1: Control points of a spline curve

This is the function call used in Processing to draw a curve section between P_1=(x1,y1) and P_2=(x2,y2):

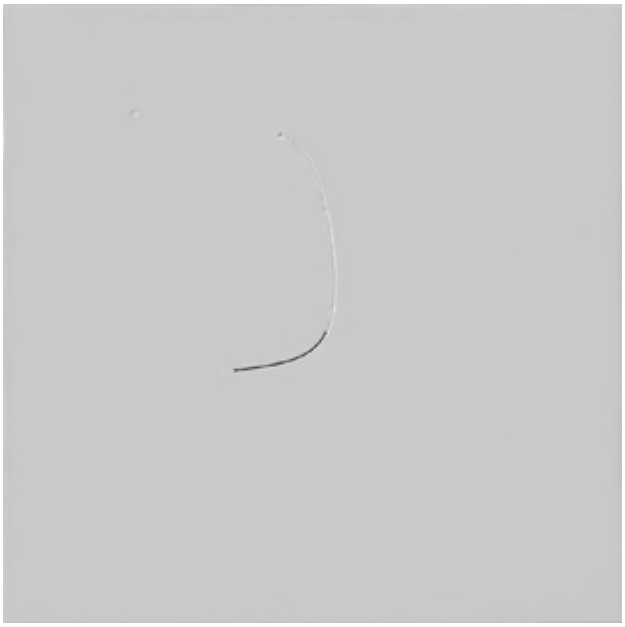
```
curve (x0,y0,  x1,y1,  x2,y2,  x3,y3);
```

The start and end points, (`x0`, `y0`) and (`x3`, `y3`), control the shape 1. To draw a longer curve, we need multiple calls to `curve()`, with the endpoints of one being the beginning of the next.

This sketch allows the user to select points, drawn as small red circles using mouse clicks 2, and to observe the shape change caused by the position of the next point as the mouse moves. Four points define a curve, so when the user selects the fourth point, a red curve is drawn using the points specified, and then a blue curve that changes as the mouse moves is drawn from the final point to the mouse position, (`mouseX`, `mouseY`). Clicking again will add a new point to the curve, extending the red portion to include the new point and showing a new blue section. Pressing the BACKSPACE key deletes the last point in the curve 3, and the spacebar turns the drawing of the final (blue) section on and off 4.

This sketch keeps the point coordinates in arrays `x` and `y` and passes successive groups of four coordinates to `curve()`.

```
final int SIZE = 200;
float x[] = new float[SIZE];
float y[] = new float[SIZE];
int N = 0;
boolean drawLast = true;

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

void draw ()
{
  background(200);
  stroke(255, 102, 0);
  if(drawLast)
    for (int i=0; i<N; i++) ellipse (x[i], y[i], 2, 2);
  if (N>=4)
  {
    for (int i=0; i<=N-4; i++)
1 curve (x[i],y[i], x[i+1],y[i+1],
           x[i+2],y[i+2], x[i+3],y[i+3]);
    stroke(0,0,200);
    if (drawLast)
      curve (x[N-3],y[N-3], x[N-2],y[N-2],
             x[N-1],y[N-1], mouseX, mouseY);
  }
}

void mousePressed ()
{
2 x[N] = mouseX; y[N] = mouseY;
}

void mouseReleased ()
{  if (N < SIZE-1) N = N + 1;  }

void keyPressed ()
{
3 if (key == BACKSPACE && N>0) N=N-1;
4 if (key == ' ') drawLast = !drawLast;  // ! means
'not'
}
```



Sketch 65: A Driving Simulation with Waypoints

Sketch 60 allowed a user to drive around a track in 3D, and Sketch 64 illustrated how to create curves, like a track that a simulated car could drive on. Computer driving games often have automated vehicles that compete with the player, giving the impression of being a real opponent. This sketch will implement a system for computer-controlled cars that is similar to the methods used in those computer games.

It's important to realize that games and simulations do not necessarily do things the way people do. A human driver would orient the car based on the next turn they could see and would steer continually to remain on the track. We could build a computer program to do this too, but it would be pretty complicated. Another option is to use predetermined knowledge about the track to steer the vehicle. In this case, the programmer has to provide more information to the program at the outset, but the resulting simplicity in the code is worth the effort.

To be specific, the programmer breaks up the track into linear pieces. The linear pieces should be as long as possible and join to each other at vertices called *waypoints*, places where the direction of the line, and hence the car, changes. (We can dissect any curve this way.) Each waypoint has a number or a label assigned by the programmer. When the car is at waypoint 1, the program will change its direction of motion to move toward waypoint 2. When it arrives at waypoint 2, it will steer to waypoint 3. Because the segments are lines, we don't need to steer between waypoints.

This sketch implements waypoints as a collection of arrays, each array containing one dimension of the waypoints. The location of waypoint `i` is in the array locations `wpx[i]` and `wpy[i]` 1. In a more accurate simulation, a waypoint would have much more information associated with it: changes in speed and acceleration, rate of change of the turn, and perhaps graphical information like brake lights turning on. In the current sketch, the only other thing needed is the angle between the current and the next waypoint so that we can rotate the car to face the new direction. We could calculate this, but it would take more code, and the positions of the waypoints and the

angles between them can be determined in advance. We declare arrays `wpx`, `wpy`, and `wpa` 2 and initialize them with the position and angle data, which implicitly defines the size of the arrays. (It is not possible to both specify the size of an array using a number and initialize it using data.)

Using the vehicle's assigned `speed` (changed using W and S), we compute its position change during each frame as `dx = speed * ( wpx(i+1) - wpx(i) )/d(i,i+1)` where `d(i,i+1)` is the distance between waypoints `i` and `i+1` 4. We say that the vehicle has arrived at waypoint `i` when it is within `speed` pixels of it, at which point it changes direction and aims for the next waypoint, `i+1` 3. The `wayPoint` variable indicates the last waypoint encountered, meaning that the vehicle is aiming for `wayPoint+1`. The waypoint count wraps around at the end, so we increment modulo-*N* where *N* is the number of waypoints: the waypoint following *N* is 0.

Pressing the spacebar allows the user to see where the waypoints and paths are.

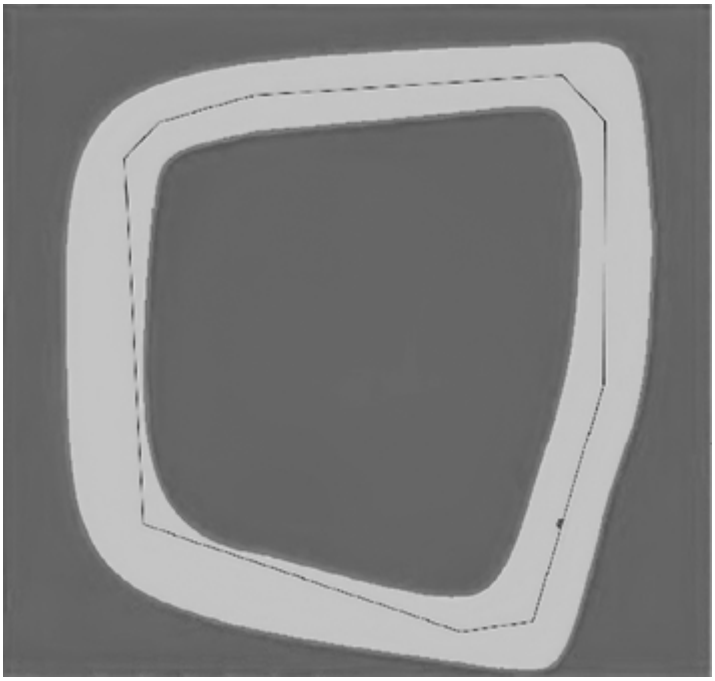
1
```
float wpx[] = { 172, 221, 354, 787, 848, 846, 747, 645,
198};
float wpy[] = { 217, 166, 129, 100, 165, 536, 869, 884,
734};
```

2
```
float wpa[] = { 39, 70, 80, 135, 175, 195, 260, 290,
354}; // Degrees
PImage track;
float x=wpx[8], y=wpy[8], dx, dy;
int wayPoint = 8, speed=2, N= wpx.length;
boolean lines = true;
void setup ()
{
  size(100,100);
  surface.setResizable(true);
  track = loadImage("road.png");
  surface.setSize(track.width, track.height);
}

void draw ()
{
  fill (255, 0, 0);
  image(track, 0, 0);
  if (lines) for (int i=0; i<N; i++)   // Draw lines?
    line(wpx[i], wpy[i], wpx[(i+1)%N], wpy[(i+1)%N]);
  translate (x, y);                    // Rotate car to
face motion
  rotate (radians(wpa[wayPoint]));
  rect (0, 0, 5, 10);                  // Arrived at
waypoint?
```

3
```
if (distance (x, y, wpx[(wayPoint+1)%N],
wpy[(wayPoint+1)%N]) < speed)
  {
    wayPoint = (wayPoint+1)%N;         // Yes. Aim at
next one
    x = wpx[wayPoint];
    y = wpy[wayPoint];
  }                      // Change x and y car position
```

4
```
dx = speed * (wpx[(wayPoint+1)%N]-wpx[wayPoint])/
        distance (wpx[wayPoint],wpy[wayPoint],
wpx[(wayPoint+1)%N],
        wpy[(wayPoint+1)%N]);
  dy = speed* (wpy[(wayPoint+1)%N]-wpy[wayPoint])/
        distance (wpx[wayPoint],wpy[wayPoint],
wpx[(wayPoint+1)%N],
        wpy[(wayPoint+1)%N]);
```

```
  x = x + dx; y = y + dy;
}

float distance (float x0, float y0, float x1, float y1)
{ return sqrt ( (x0-x1)*(x0-x1) + (y0-y1)*(y0-y1) ); }

void keyPressed ()
{
  if (key == ' ') lines = !lines;                 //
Toggle waypoint display.
  if (key == 'w') speed = speed+1;                //
Faster
  if (key == 's') if (speed>0) speed = speed-1;  //
Slower, but not backwards
}
```



Sketch 66: Many Small Objects—A Snowstorm

A Processing program redraws the screen many times each second. Visible objects must be redrawn in each frame, and to do so the program must save the graphical parameters (size, location, shape, and color) of all of them. Drawing each object takes time, so if there are many, is it still possible to redraw them all quickly enough? In many cases it is, if the objects themselves are not complex. This sketch will draw snow falling, with each snowflake being an object that moves realistically between frames.

Snowflakes are, in fact, very complex shapes, but from a distance they are just white blobs. We'll draw them as small rectangles whose width and height vary by a small random value each frame to simulate the effect of the snowflake fluttering as it falls 1. We set the dimensions with the formula `width = size + random(3)-1.5`.

`size` is a constant set to `3`, and the value of `random(3)` is a number between 0 and 3, so `random(3)-1.5` will have a value between −1.5 and +1.5, creating a change in the size between 1.5 and 4.5. Each snowflake also has a slightly different falling speed 5. This gives the illusion of depth because flakes that fall faster appear nearer to the viewer than ones that fall slower. The speed is selected at random, but it yields the desired effect.

The program creates snowflakes at the top of the screen and gives them a downward (+y) speed, which will make them appear to fall. To track the position, size, and speed in both the x- and y-directions, we use arrays: for example, the array `x` stores the x position, and `x[i]` is the x location of the i^{th} snowflake. The array size, given by the constant `SIZE`, is the maximum number of snowflakes. (The value here is 5,000, found by trial and error based on the observed number needed given the background and the maximum rate of snowfall.)

Snow does not normally fall straight down; we observe it drifting and floating with air currents. The speed at which the snowflakes fall remains constant, but the x position of each flake changes a bit at random as it falls

to try to give the illusion of real snow 2. If we set `dx` to a nonzero value, it simulates a wind, and snow will blow in the specified direction.

Each frame, we generate up to 30 new snowflakes with random horizontal positions and y-coordinates of 0 4 (at the top of the window, to maintain the illusion). The number of snowflakes created during each frame is random but is a function of the y position of the mouse 3. The nearer the mouse is to the top of the screen, the less snow will appear to fall. This is the number of flakes created:

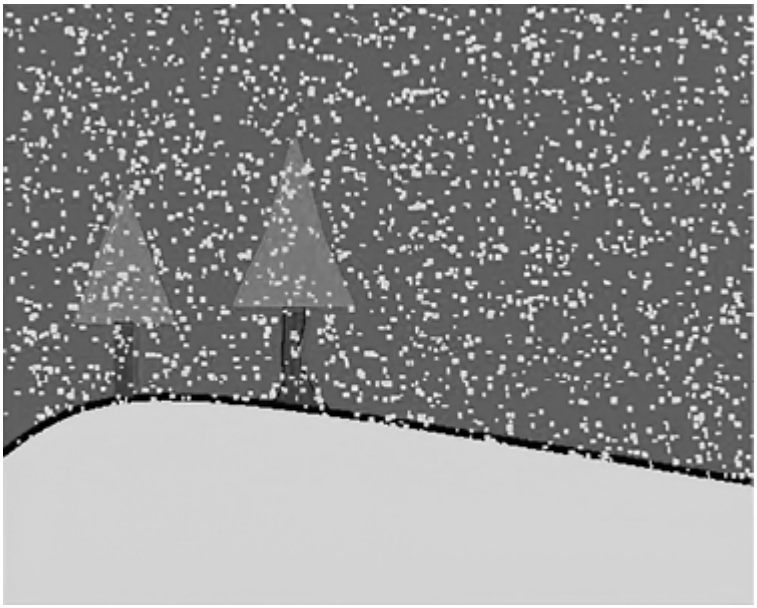
```
N = (int)random (((float)mouseY/height)*30);
```

This means that almost no flakes will fall for small values of `mouseY`, while the maximum of up to 30 new snowflakes during each frame occurs when `mouseY/height` is at its maximum of 1.0.

The global variable `SIZE` has a value of 5,000, which is the number of snowflakes that can be on the screen at any time. Initially there are only a few, but the array will fill up quickly. When all 5,000 array elements are occupied, we start over again at 0, assuming that the snowflakes at the beginning of the array have fallen past the bottom of the screen and are not visible. This technique is referred to as a *circular array*.

```
final int SIZE = 5000;
float x[] = new float[SIZE];
float y[] = new float[SIZE];
float dx[] = new float[SIZE];
float dy[] = new float[SIZE];
float size[] = new float[SIZE];
int last = 0, N=0;
PImage background;

void setup ()
{
  size(100,100);
  surface.setResizable(true);
  background = loadImage("background.png");
  surface.setSize (background.width,
background.height);
  for (int i=0; i<SIZE; i++) x[i] = -1;
}

void draw ()
{
  fill (210);
  noStroke();
  image (background, 0, 0);
  for (int i=0; i<SIZE; i++)  // Draw existing
  {
    if (x[i] >= 0)
    {
      rect(x[i], y[i],1 size[i]+random(3)-1.5,
size[i]+random(3)-1.5);
   2 x[i] = x[i] + dx[i] + random (3)-1.5;
      y[i] = y[i] + dy[i];
    } }

3 N = (int)random (((float)mouseY/height)*30);  //
Create new
  for (int i=0; i<N; i++)
  {
 4 x[last] = random(width); y[last] = 0;
 5 dx[last] = 0; dy[last] = random(2)+1;
    size[last] = 3;
    last = last + 1;
    if (last >=SIZE) last = 0;
```

```
    }
}
```



Sketch 67: Particle Graphics—Smoke

Some things are difficult to model using polygons: soft and amorphous shapes like water, fire, clouds, and smoke, for example. Such things can move in unpredictable ways and expand to fill arbitrary shapes. This sketch will draw smoke emitting from a smokestack and illustrate a key method in modern computer graphics: a *particle system*.

A particle system combines a large number of small objects to form a complex shape. The objects are usually simple, like spheres or circles, and have a set of parameters that control their display. The basic parameters of a circle are position, velocity, color, and size. Initial parameters usually have a random element: speed plus or minus a random number, for example. An *emitter* is the location where the system creates new particles (circles), usually with a small, random displacement, so the particles aren't exactly at the emitter.

The particle system in this sketch produces a large number of overlapping circles, possibly somewhat transparent, moving with slightly different speeds (not unlike the previous sketch except for density). The previous sketch drew a large number of small objects that could still be seen as individual snowflakes. In this sketch, if enough of these particles exist, we can't distinguish them individually, and they form an object in combination. As the number increases and the objects overlap, the result looks like fog or smoke.

The sketch defines a large number of circles to be created (`SIZE`) and declares arrays to hold the position, speed, and size of each one: the value of `x[121]` is the x position of the 121st circle, for example. Initially there are none, and the variable `last` holds the index of the last one defined. We increment `last` each frame as we create new circles, and we reset it to zero when the number exceeds `SIZE`.

The `draw()` function first runs through the arrays and draws each circle that exists (meaning `x[i] > 0`) 1. It changes the circle's position by a small random amount, may change the size slightly, and gives it a color that varies around RGB = (205, 205, 150). It then creates a random number of

new circles, giving them positions near the emitter, a vertical speed, and a small size 2.

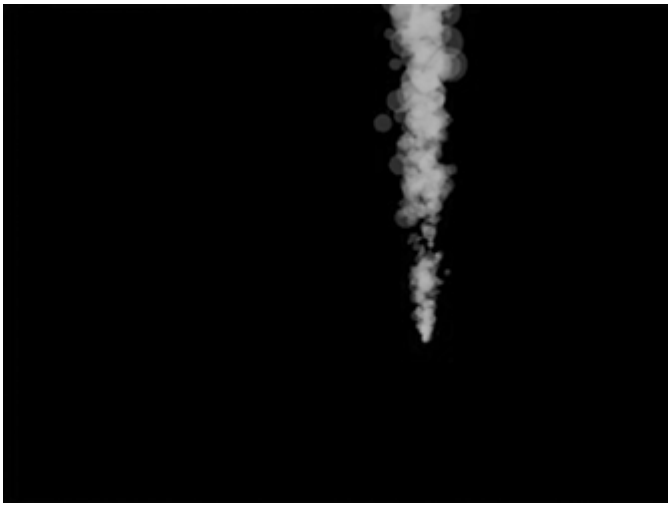
The effect is striking. With up to 800 circles, the system yields a remarkably good impression of smoke moving upward. The sketch reads and displays a background image of a smokestack for a better visual effect.

The outline around the circles has been turned off with `noStroke()`, but it is educational to delete that statement and run the program so that the particles can be seen. The way the particles move and overlap is clearer, as in *Figure 67-1*.

Figure 67-1: Particles showing the outline of the circles

```
final int SIZE = 800;
float x[] = new float[SIZE];
float y[] = new float[SIZE];
float dx[] = new float[SIZE];
float dy[] = new float[SIZE];
float size[] = new float[SIZE];
int last = 0;
PImage background;
int emitterx=252, emittery=200;

void setup ()
{
  size(100,100);
  surface.setResizable(true);
  background = loadImage ("background.png");
  surface.setSize(background.width, background.height);
  for (int i=0; i<SIZE; i++) x[i] = -1;
}

void draw ()
{
  int N=0;

  noStroke();

  image (background, 0, 0);
  for (int i=0; i<SIZE; i++)  // Draw existing
particles
1 if (x[i] > 0)
    {        // Vary the color slightly
      fill (200+random(10), 200+random(10), 150, 64);
      ellipse (x[i], y[i], size[i], size[i]);  // Draw
circle
      x[i] = x[i] + random(3)-1.5;             //
Jiggle X
      y[i] = y[i] + dy[i] + random(3)-1.5;     // Move
up
      size[i] = size[i] + random(3)-1.5;       //
Change size
    }

    N = (int)random(15);    // Create N new particles
2 for (int i=0; i<N; i++)
    {
      last = (last+1);
```

```
      if (last >= SIZE) last = 0;
      x[last] = emitterx+random(2)-1;   // X position
(emitter)
      y[last] = emittery;               // Y position
(emitter)
      dx[last] = 0; dy[last] = -2;      // Initial
speed (up)
      size[last] = 4;                   // Initial size
    }
}
```



Sketch 68: Saving a State—A Spinning Propeller

This sketch will draw a spinning propeller. We can code this in many ways, some of them simpler than the method in this sketch, but the purpose here is to use a simple example to explain how and why to save (and restore) the *geometric state* of a sketch.

The geometric state is the resultant combination of all the translation, rotation, and scaling that accumulate during the display of an object up to a specific point in the drawing process. Rotating an object about its center means first translating the origin to the center of the object, doing the rotation, and then translating the origin back to the original location. If the state is not restored by undoing the translation, then all objects drawn from that time on will translate to the location of the object.

The current state, whatever it is, including all rotation, translation, and scaling, is saved using a call to the function `pushMatrix()` and is restored by a call to `popMatrix()`. These calls must always occur in pairs, like brackets; a call to `pushMatrix()` always has a corresponding call to `popMatrix()`. For example, you could save and restore state while rotating a triangle about its center at (100, 100), shown in *Figure 68-1*, as follows:

```
pushMatrix();
translate (100, 100);
rotate(angle);
triangle (0.-20, 20, -20, -20, 20);
popMatrix();
```

At this point, the origin and rotation angle are back to their original values, and the next object can be drawn from a clean state.

This sketch draws a propeller with four sections, each being one blade, which is an image. We draw this blade four times: once in the original orientation, and then three times each rotated about the propeller center point by 90 degrees 1. Each section drawn uses a save and restore:

```
pushMatrix();              // Save
rotate(PI);                // Rotate
image (prop, 0, 0);        // Draw
popMatrix();               // Restore
```

The four-section propeller is drawn inside a `drawProp(x,y)` function that saves the state on entering the function, then translates to (`x`, `y`), scales the image, rotates it, updates the angle so the next call to `drawProp()` draws the propeller at a different angle 2, and draws the four sections. We use multiple calls to the `drawProp()` function to draw a rotating propeller at multiple locations.

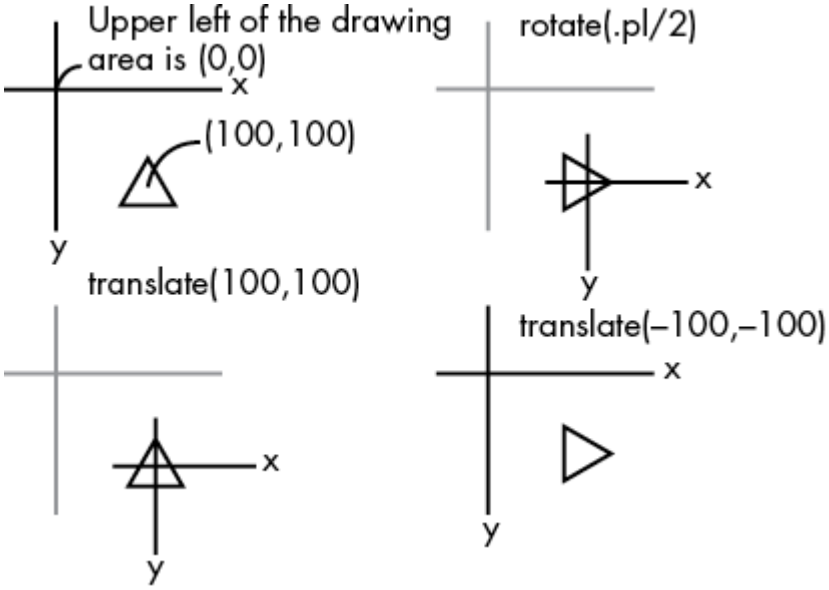
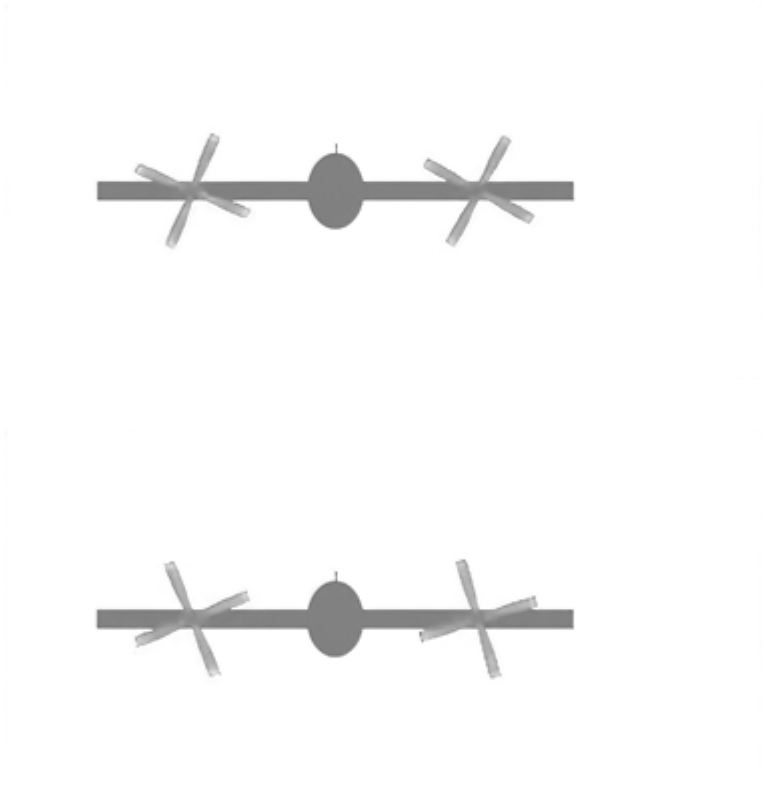


Figure 68-1: The transformations needed to rotate an object about its center

```
PImage prop;
float angle = 0.0;
void setup ()
{
  prop = loadImage("props.gif");
  size(400, 200);
}

void draw ()
{
  background(255);
  fill (128); noStroke();
  ellipse (175, 100, 30, 40);  // Draw simple aircraft
  rect (50, 95, 250, 10);
  stroke (128);
  line (175, 100, 175, 75);
  drawProp (100, 100);         // Left propeller
  drawProp (250, 100);         // Right propeller
}

void drawProp(int x, int y)
{
1 pushMatrix();    // Save state on entry to the
function
  translate(x, y); // Translate to the specified
propeller position
  scale (0.2);     // Make it smaller
  rotate(angle);   // rotate the propeller as a whole
2 angle = angle + 0.1;
  image (prop, 0, 0);      // Draw the first prop
section
  pushMatrix();            // save state
    rotate(PI/2);          // Rotate 90 degrees
    image (prop, 0, 0);    // draw second prop section
  popMatrix();             // restore
  pushMatrix();            // Save again
    rotate(PI);            // Rotate by 180 degrees
    image (prop, 0, 0);    // draw third prop section
  popMatrix();             // Restore
  pushMatrix();            // save one more time
    rotate(-PI/2);         // Rotate 270 degrees (-90)
    image (prop, 0, 0);    // Draw final section
  popMatrix();             // restore
  ellipse (0, 0, 30, 30);  // draw the center part of
the propeller
```

```
  popMatrix();              // Restore state to what it
was when function was called
}
```



Sketch 69: L-Systems—Drawing Plants

Drawing realistic-looking plants is difficult. Living things do not usually contain straight lines, which is what computers draw best. In addition, there is a random nature to life forms that humans recognize, so we are critical of renderings. In 1968 a botanist named Aristid Lindenmayer developed a scheme for describing the growth of fungi and algae, and then later expanded it to deal with more advanced plant life. This was in turn adapted by computer graphics practitioners into a scheme for drawing plants. We call this scheme an *L-system*.

An L-system is technically a *grammar*, which is a set of rules for making strings. If a grammar has two rules, X -> Xf and X -> z, then it is showing how to take a symbol, X, and transform it into a sequence of characters. For each X, we choose which replacement rule to follow, and we continue replacing capital Xs (referred to as *non-terminal symbols*) until there are no more left to replace. Here's an example expansion for this grammar: X -> Xf -> Xff -> Xfff -> zfff.

In an L-system, a grammar that can define a plant, the final string represents a recipe for drawing something. It uses the following symbols:

f Draw a straight line segment.

[Save the current state (`pushMatrix()`).

] Go back to the previous state (`popMatrix()`).

+ Rotate by a fixed positive angle.

- Rotate by a fixed negative angle.

The grammar uses two rules to produce a string of these symbols:

X -> ff

X -> f–[[X]+X]+f[+fX]–X

Unless the plant consists of only two straight lines (ff), the first step would be X –> f–[[X]+X]+f[+fX]–X. Then each X would be replaced by the right side of a production, so the second step might be f–[[X]+X]+f[+fX]–X –> f–[[f–[[X]+X]+f[+fX]–X]+ff]+f[+ff]–ff, followed

perhaps by f–[[f–[[ff]+ff]+f[+fff]–ff]+ff]+f[+ff]–ff, which can now be drawn.

The `makeString()` function 1 calls itself to expand the non-terminal X symbols into strings and append these to the string being constructed. It will only call itself to a depth specified by the first parameter, `levels`, and will then return, thus guaranteeing that the program will eventually end. The string generated by the grammar is passed to the `drawPlant()` function 2, which executes each character as a graphical operation, thus drawing the plant. In the function `void drawPlant(float length, float angle, String s, int drawLevel)`, the first parameter, `length`, is the length of the line to draw (for the symbol f); the `angle` is the rotation angle for the + and - characters; `s` is the string generated by `makeString()`; and `drawLevel` indicates a depth level for drawing lines. Essentially, `makeString()` creates a string that is a program for how to draw the plant, and `drawPlant()` executes that program.

```
public void setup ()
{
    String rules;
    size(800, 800, P2D); stroke(0);
    translate(width/2,height);
    rules = makeString(6, "X");
    drawPlant (4, 22, rules,rules.length()-1);
}
```

1
```
String makeString(int levels, String s)
{
  String next = "";
  char c;
  if (levels > 0)
  { // Check if there are any levels left to render
    for (int i=0; i<s.length(); i++)
    {
      c = s.charAt(i);
      if (c == 'X')
        next+=makeString(levels-1, "F-[[X]+X]+F[+FX]-
X");
      else if (c == 'F') next += makeString(levels-1,
"FF");
      else next = next + c;
    }
  } else next = s;
  return next;
}
```

2
```
void drawPlant(float length, float angle, String s, int
drawLevel)
{
  char c;
  int i=0;
  for (int j=0; j<s.length(); j++)
  {
    c = s.charAt(j);
    if (c == '-') rotate(radians(angle));
    else if (c == '+') rotate(-radians(angle));
    else if (c == '[') pushMatrix();
    else if (c == ']') popMatrix();
    else if (c == 'F')
    {
      if (i <= drawLevel) line(0, 0, 0, -length);
      translate(0,-length);
```

```
    }
    i++;
  }
}
// (This sketch is a reworking of the one found at
https://www.openprocessing.org/sketch/103747/)
```

Sketch 70: Warping an Image

In 1991 the general public saw *morphing* for the first time, an effect that uses a computer to smoothly convert one image into another. The computer produces a small sequence of images so that, when played back as a video sequence, an object appears to continuously change shape to become the other. The film *Terminator 2* used it and, probably most strikingly, the Michael Jackson music video for the song "Black or White" used it in a sequence where faces morphed into one another. This sketch performs a warp or bending of an image, but does not do a complete morph.

The principle underlying the morphing method is something called a *polynomial warp*. Imagine we place an image over a regular grid and then bend the grid using a mathematical function and take the image with it. The result is an image that changes in a particular way—a *warp*. Morphing between two images requires that we establish a correspondence between the image, usually by a human. A function bends (maps) one image into another (a warp) while the pixel color values change systematically from the source to the destination values.

If the image is a face and the warp is based on a sine curve, we get an effect that looks like a funhouse mirror, as shown in *Figure 70-1*. The geometry of the original face bends (*maps*) into that of the new one according to the function.

Figure 70-1: Warping a face

This sketch implements an image warp. We read an image and display the pixels according to a sine function transformation of coordinates. The original image is `source`, and this is the mapping between original and new pixel coordinates:

```
newX = (int)(x + size*sin(radians(3*y)));
newY = (int)(y + size*cos(radians(4*x)));
```

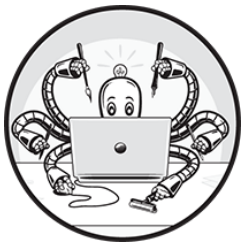
This mapping is arbitrary, chosen for an amusing effect. The loop that does the mapping 2 has to map pixel values from the destination back to the source, not the other way around. Each pixel in the source does correspond to a pixel in the destination, but there may be unmapped pixels in the result if we do the mapping the other way. So for each pixel `(x,y)` in the destination image, we transform it to `(newX, newY)` values using the desired function and then find the corresponding pixel in the source image. We then set the destination `(x,y)` to the source `(newX, newY)`.

In the sketch, the values of `ds` and `size` are parameters to the transformation function, and they change slightly each frame 1, creating a cyclical change in the image that a person will perceive as an animation of the bending or warping motion.

```
float size = 0;
float ds = .3;
PImage source, destination;

void setup ()
{
  size(100,100);
  surface.setResizable(true);
  source = loadImage("image.jpg");  // Fill in your own
image here
  surface.setSize(source.width, source.height);
  destination = new PImage(source.width,
source.height);
}

void draw ()
{
  background (200);
  warp(source);
  image(destination, 0, 0);
  size -= random(ds);
  if (abs(size) > 12)
  {
1 ds = -ds;
    size = size - ds;
  }
}

void warp(PImage source)
{
  int w = source.width, h = source.height;
  int newX, newY;
  color c;

  for(int x = 12; x < w-12; x++)
    for(int y = 12; y < h-12; y++)
    {
 2  newX = (int)(x + size*sin(radians(3*y)));
      newY = (int)(y + size*cos(radians(4*x)));

      if(newX >= w || newX < 0 || newY >= h || newY <
0)
        c = color(200);
      else
        c = source.get (newX, newY);
```

```
      destination.set (x, y, c);
    }
}
```

9
WORKING WITH SOUND



Sketch 71: Playing a Sound File

One *displays* an image but *plays* a sound; why is that? Whatever the reason, Processing has no standard facility for displaying audio. It *does* have some libraries for that purpose, however, most importantly Minim. (We used a library in Sketch 50.)

Using Minim, this sketch will play an MP3 or WAV sound file using the standard PC sound interface. Adding to this, if the user presses the A key, the sound will move toward the left speaker, and if they press the D key (which is to the right of the A key), the sound will move toward the right speaker.

The first statement in the program 1 indicates that we want to access the Minim library:

```
import ddf.minim.*;
```

Then we need to create a single instance of the Minim library. The Minim library is a class, and it contains functions that can load and play sound files. Define a variable named `minim` of type `Minim`, and initialize it in the `setup()` function 3 as follows:

```
minim = new Minim(this);
```

Now declare a sound player variable 2:

```
AudioPlayer player;
```

Assign it a sound file as read from an MP3 file using the `Minim` function `loadFile()` 4:

```
player = minim.loadFile ("song.mp3");
```

We can play this file using the PC sound hardware by using the `play()` function 5, a part of the `AudioPlayer`:

```
player.play();
```

To change the *balance* (pan) of the sound in stereo speakers, the user presses the A (left) and D (right) keys. Each key press adds a small value to or subtracts one from the `pan` variable, which is then used to set the balance 6:

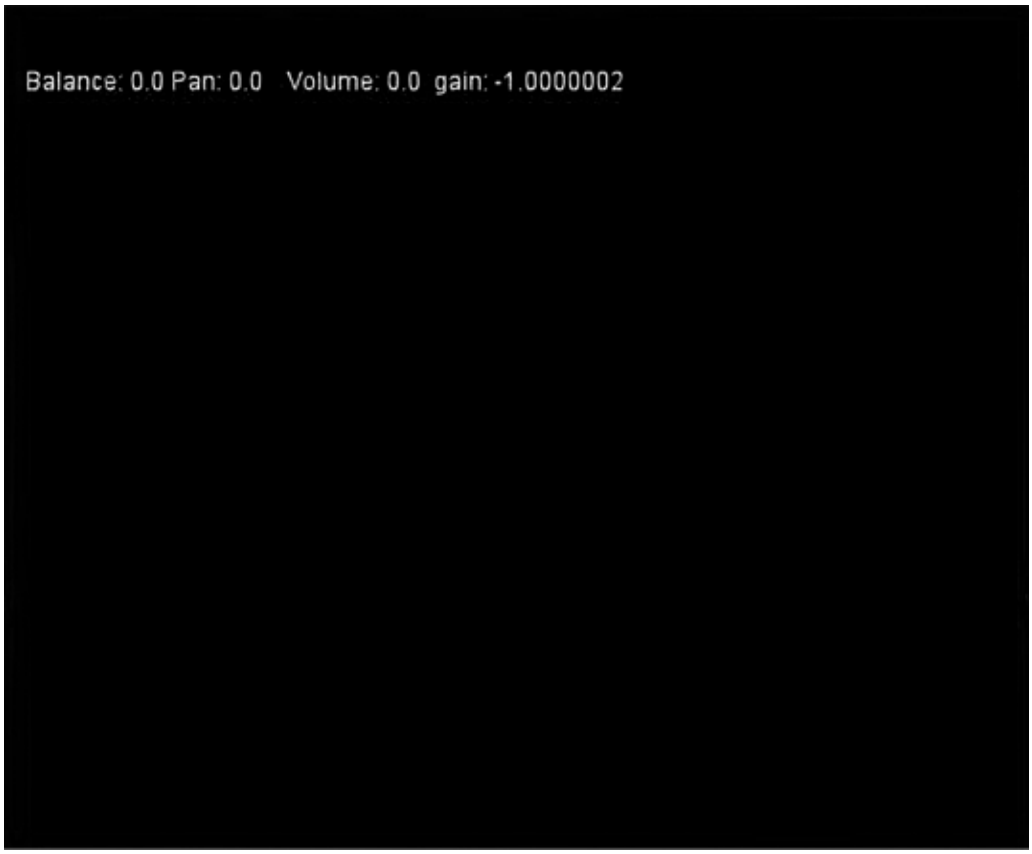
```
player.setPan (pan)
```

For other effects, there are a variety of functions that control the sound display, including the getting and setting of pan/balance, gain, and volume: `getBalance()`, `getVolume()`, `getGain()`. Documentation for Minim can move around the web, but in 2022 it's found at *http://code.compartmental.net/2007/03/27/minim-an-audio-library-for-processing/*.

NOTE

`AudioPlayer` is a class, and it's a part of the `Minim` library. Each sound file needs its own instance (variable) of the `AudioPlayer` class. Also, you can play sound files on the web by passing the `loadFile()` function a URL instead of a local file. See the commented-out statement in `setup()`.

```
1 import ddf.minim.*;
   Minim minim;
2 AudioPlayer player;
   float pan =  0;

   void setup ()
   {
     size (500, 400);
  3 minim = new Minim(this);
  4 player = minim.loadFile ("song.mp3");
   //  player = minim.loadFile ("https://file-examples-
   com.github.io/uploads/2017/11/file_
         example_MP3_700KB.mp3");
  5 player.play();
     player.printControls();
     frameRate(10.0);
   }

   void draw ()
   {
     float bal, vol=1, p, g;
     background (0);
     bal = player.getBalance();
     vol = player.getVolume();
     p = player.getPan();
     g = player.getGain();
     text ("Balance: "+bal+" Pan: "+p+"    Volume: "+vol+"
   gain: "+g, 10, 40);
     player.setGain (vol-1.0);
   }

   void keyPressed ()
   {
     if (key == 'a')
     {
       if (pan>-1) pan = pan -  .1;
    6 player.setPan (pan);
     } else if (key == 'd')
     {
       if (pan<1.0) pan = pan+.1;
       player.setPan(pan);
     }
   }
```


Balance: 0.0 Pan: 0.0 Volume: 0.0 gain: -1.0000002

Sketch 72: Displaying a Sound's Volume

Sketch 71 does not have a very visually interesting display. Its display is auditory, and while that is in keeping with its primary function, the Processing language usually creates more graphical output. One obvious way to accomplish this is to display the volume of a sound visually, as numbers on a dial or, as in this sketch, as the height of vertical bars.

To make this sketch work, we must get numerical values for the sound that we read from the file. The `AudioInput` component class of `Minim` allows a connection to the current record source device for the computer. For this sketch to function properly, the user needs to set the source device to monitor the sound as it plays. For example, if the sound input is a file, we could use this code:

```
3 player = minim.loadFile ("song.mp3");
```

Assuming this is true, the sketch uses a variable of the `AudioInput` type (named `in` 1) and initializes it using `getLineIn()` 2:

```
in = minim.getLineIn(Minim.STEREO);
```

Now the variable `in` can access the functions belonging to `AudioInput`, which include the ability to get individual data values. Sound on a computer consists of sampled voltages that have been rescaled to a convenient range. Thus, an audio value is a number, normally between −1 and +1, that represents the volume. We can access each of the stereo channels: the left channel is `in.left`, and the right is `in.right` (these are of type `AudioBuffer`, which is just an array of real numbers). The `get()` function allows access to the numerical values:

```
ly = in.left.get(128);
```

This gets the first value in the buffer, which could be positive or negative, so for display purposes it is better to use the value `abs(in.left.get(128))*2` 4, which is simply the magnitude of the value

shifted to the range 0 to 2. Now this number can represent the height of a rectangle 6, proportional to the sound volume:

```
rect (100, 200, 20, -ly*100);
```

The same process works for both the left and right channels.

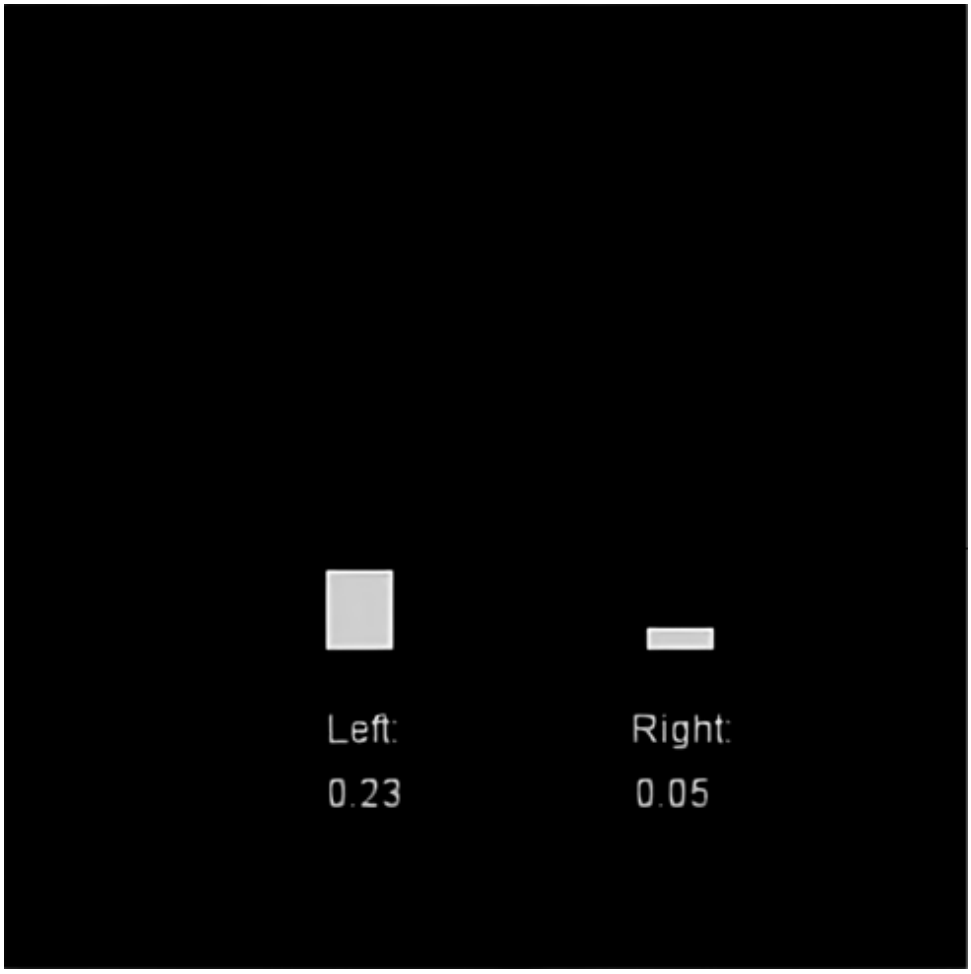
The total duration of a sound loaded into the variable player is `player.duration()`; the current position, assuming that it is playing, is `player.position()`. When the sound is over, `player.length() <= player.position()`, and the `Minim` specification says that it is important to close and stop `Minim` to ensure that resources are given back to the system (via `in.close(); minim.stop();`). In the sketch, the `stop()` function 7 does this.

The sketch also displays a numerical value for the sound data. A real number potentially has a lot of digits, most of which are not really important. To print only two decimal places, as in the sketch, multiply the value by 100 and then convert it to an integer. This removes the remaining fractional part (all other digits to the right). Then convert this back to real and divide by 100 5:

```
(int)(ly*100)/100.0
```

```
import ddf.minim.*;

Minim minim;
AudioInput in;
AudioPlayer player;

void setup ()
{
  size(300, 300);
  minim = new Minim(this);
  in = minim.getLineIn(Minim.STEREO);
  player = minim.loadFile ("song.mp3");
  player.play();
  stroke(255);
  fill (100);
  frameRate(10);
}

void draw ()
{
  float ly, ry;

  background(0);
  ly = ry = 0;
  ly = abs(in.left.get(128))*2;
  ry = abs(in.right.get(128))*2;
  fill (255, 255, 0);
  text ("Left: "+                    Right: ", 100, 230);
  text (""+(int)(ly*100)/100.0+"                      "
                  +(int)(ry*100)/100.0, 100, 250);
  rect (100, 200, 20, -ly*100);
  rect (200, 200, 20, -ry*100);
  if (player.length() <= player.position())
  {
   stop(); exit();
  }
}

void stop ()    // always close Minim audio classes when
done with them
{
  in.close();
  minim.stop();
```

```
  super.stop();
}
```



Sketch 73: Bouncing a Ball with Sound Effects

In movies, animations, theater, and computer games, a *sound effect* is (usually) a short piece of audio that indicates that something has happened. A telephone ringing, the smack of a bat hitting a baseball, and the splash of a stone falling into a lake are all examples of sound effects. This sketch will illustrate the use of a sound effect in a simple simulation.

Sketch 28 simulated a bouncing ball. It looks nice, but it would be better as an animation if a sound accompanied each bounce. Sound is an important cue to humans, and a sound effect lends realism to the graphics. It does not have to be accurate; it just has to be some click or bump noise that corresponds to the event. Beginning with the code from Sketch 28, we'll add an `AudioPlayer` object from the Minim library to play a short MP3 file when the ball strikes a side of the window.

To create the sound effect, we'll save the sound of a thump (such as a ball bouncing on the floor or a cup being set down on a table) using a PC microphone and a freely available sound editor/capture tool such as Audacity (*https://www.audacityteam.org/*) or GoldWave (*http://www.goldwave.ca/*). This sketch assumes the sound is saved as *click.mp3*.

After the initialization of `Minim` 1, an `AudioPlayer` (the variable `player`) reads the MP3 file. When the ball strikes a side of the window, as detected by the functions `xbounce()` 2 and `ybounce()` 5, the ball changes direction and we play the sound with a call to `player.play()` 3.

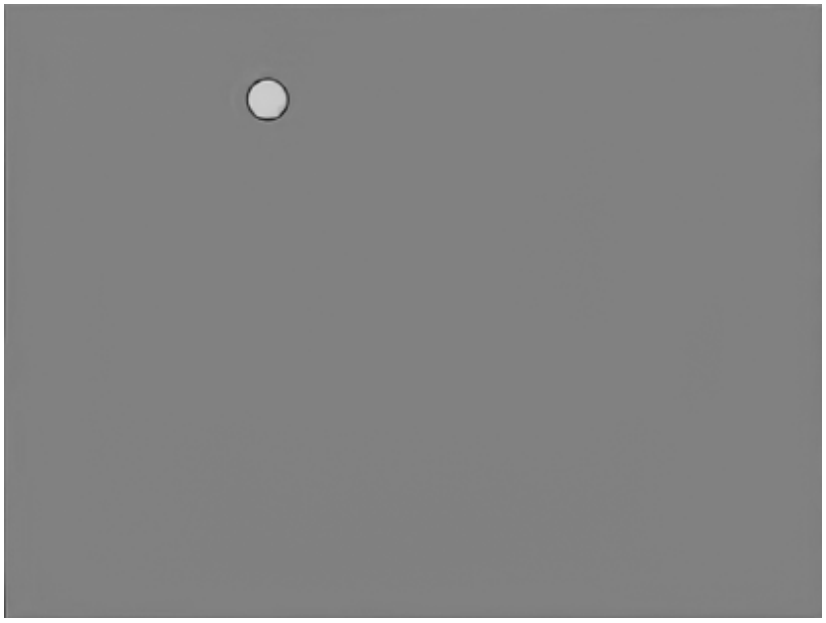
We have to rewind the sound file each time before it is played to make sure it starts from the beginning. The `rewind()` 4 function within `AudioPlayer` does this.

NOTE

Four kinds of sound are normally used in animations and computer games: sound effects, music, ambient sound, and voice. Ambient sound is continuous, sometimes random sound from the background. The sound of rain falling, traffic, crowds, and water flowing are examples. Ambient sound, like music, starts at some particular point in the animation of a graphic and has a long duration, often repeating when it has finished. Human voice usually consists of individual sentences, each saved as a separate file. These play back as a part of a narrative, sometimes in an undetermined order. The activity of a user can determine what voice snippet plays at a particular time. In that sense, it is like a sound effect, a short audio segment, but one that is not necessarily always associated with the same action.

```
import ddf.minim.*;
int x=320, y=240;                // Coordinates of the
circle (ball)
int dx=3, dy=2, radius=10;       // Size and speed of the
circle (ball)
AudioPlayer player;
Minim minim;

void setup ()
{
  size (400, 300);               // Window size
  fill (255, 0, 255);            // Magenta fill
1 minim = new Minim(this);
  player = minim.loadFile ("click.mp3");
}
void draw ()
{
  background (128);              // Grey background
  ellipse (x, y, radius+radius, radius+radius); // Draw
the ball
  x = x + dx;  y = y + dy;       // Move
  xbounce(); ybounce();
}
2 void xbounce ()
{
  if (x+radius > width)          // right side
  {   // Reverse x-direction
    x = width-((x+radius) - width);
    dx = -dx;3 player.play()4 player.rewind();
  } else if (x < radius)         // left side
  {
    x = radius-x; player.rewind();
    dx = -dx;   player.play();
  }
  x = x + dx;
}
5 void ybounce ()
{
  if (y<radius)                  // Top side
  {                              // Reverse y-direction
    y = radius+y; player.rewind();
    dy = -dy;   player.play();
  } else if (y+radius > height) // Bottom side
  {
    y = height-((y+radius)-height); dy = -dy;
```

```
    player.play();  player.rewind();
  }
  y = y + dy;
}
```



Sketch 74: Mixing Two Sounds

In the process of *sound mixing*, we assign each of a number of sound sources to different output levels or volumes. In live music concerts, this makes the sound of each instrument audible at the proper volume level. We also do this when recording multiple sources of sound, such as microphones, guitars, and other instruments, which need to have their volume levels adjusted so that no one component overwhelms the total. Mixers have been around for a long time, and most have sliding controls to adjust volume levels of multiple sound signals. This sketch will use the slider control developed in Sketch 43 to adjust the volume of two different sound files.

The sketch begins by declaring two `AudioPlayer` variables 1, one for each sound, loading the sound files 2, and starting to play them both 3. Next we create two slider controls; one is control A, having position and control variables beginning with "a" (`asliderX`, `asliderY`, `avalue`) and the other is control B (`bsliderX`, `bsliderY`, and so on). The value of slider A is used to set the volume of the first of the sound files being played (by `playera`), and slider B controls the volume of the other (`playerb`).

We set the output level by calling the `Minim` function `setGain()`. This function has a parameter that represents the value of the *gain* (proportional to volume). The units on gain are *decibels* (dB) and they begin at −80 and end at +14 for a total range of 94 dB units. The total range of the slider values is 1,000. Thus, the gain for `playera` is set using the following call 4:

```
playera.setGain(avalue/1000.0 * 94 - 80);
```

If the slider value is at the minimum of 0, the gain will be 0/1,000 * 94 − 80 = 0 − 80 = −80. If the slider value is at the maximum of 1,000, the gain will be 1,000/1,000 * 94 − 80 = 94 − 80 = 14. That the gain values have the correct output for the extreme values supports the idea that the mapping is correct. The dB scale is logarithmic, though, so this is an approximation of the truth.

When the sketch is executing, the two sound files will play. Sliding the top slider right will increase the volume of the *sounda.mp3* file, and sliding the

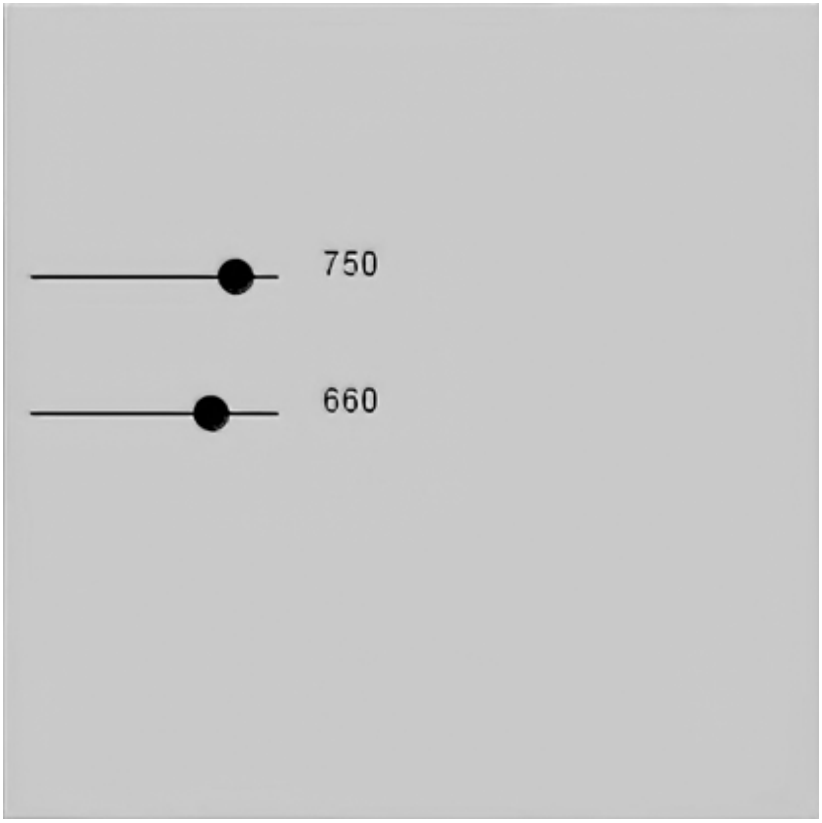
lower slider will control the volume of the *soundb.mp3* file. The idea is to find relative levels that sound right.

NOTE

It would be better to implement the slider widget that controls the volume as a class so that multiple sliders could be placed on any screen and pass multiple values to the program. Some mixers have scores of inputs, each controlled separately. Also note that in this sketch the usual call to `play()` *has been replaced by a call to* `loop()` 3*, which continually replays the sound from the beginning after it ends (for example,* `playera.loop()`*).*

```
import ddf.minim.*;
int asliderX=10, asliderY=100, avalue=0,
sliderWidth=100;
int bsliderX=10, bsliderY=150, bvalue=0;
int asliderPos=0, bsliderPos=0, sliderMin=0,
sliderMax=1000;
AudioPlayer playera, playerb;
Minim minim;
void setup ()
{
  size(300,300);
  minim = new Minim(this);
playera = minim.loadFile ("sounda.mp3");
  playerb = minim.loadFile ("soundb.mp3");
playera.loop(); playerb.loop();

  playera.setGain(-100);
  playerb.setGain(-100);
}
void draw ()
{
  background (200);  fill (0);
  drawSliders ();
}
void drawSliders ()
{
  line (asliderX, asliderY, asliderX+sliderWidth,
asliderY);
  ellipse (asliderX+asliderPos, asliderY, 12,12);
  text (avalue, asliderX+sliderWidth+7, asliderY);
  line (bsliderX, bsliderY, bsliderX+sliderWidth,
bsliderY);
  ellipse (bsliderX+bsliderPos, bsliderY, 12,12);
  text (bvalue, bsliderX+sliderWidth+7, bsliderY);
}
void mouseDragged ()
{
  if ((mouseY>=asliderY-6) && (mouseY<=asliderY+6))
  {
    if ((mouseX>=asliderX) &&
(mouseX<=asliderX+sliderWidth))
    asliderPos = mouseX - asliderX;
    avalue = (int)(((float)asliderPos/100)*sliderMax +
sliderMin);
playera.setGain(avalue/1000.0 * 94 - 80);
```

```
  }
  if ((mouseY>=bsliderY-6) && (mouseY<=bsliderY+6))
  {
    if ((mouseX>=bsliderX) &&
(mouseX<=bsliderX+sliderWidth))
    bsliderPos = mouseX - bsliderX;
    bvalue = (int)(((float)bsliderPos/100)*sliderMax +
sliderMin);
    playerb.setGain(bvalue/1000.0 * 94 - 80);
  }
}
```



Sketch 75: Displaying Audio Waveforms

Most computer-based sound editors display a graphical rendering of the audio signal and allow the user to "grab" parts of it with the mouse and move or delete them. This graphical display is actually a plot of audio volume versus time. Some music players display such a plot in real time, as the music is playing. That's exactly what this sketch will do. It draws the plot of whatever sound the computer is playing.

Drawing this requires the ability to get the sound data as numbers in real time. A bit of error does not matter, because this is not a scientific tool, so it's possible to use some of the code from Sketch 72, which also displayed an audio visualization. Here we will fill a sound buffer and then play it as sound data until the data is finished.

Audio is represented as a set of consecutive numerical values that can reasonably be stored in an array (a buffer). There are usually two channels (stereo), and any value from a buffer can be retrieved using the `in.left_get()` or `in.right_get()` functions, specifying which sample is wanted. For example, the program gets a data point from the left channel using a call to `left_get()` 3 and uses this value to represent all levels in the current buffer. This is just *one* data point from many samples, and it is possible to specify the buffer size when the `getLineIn()` call is made. The system plays sound from this buffer and refills it whenever it needs more data. We specify a size of 1,024 samples per buffer 1:

```
in = minim.getLineIn(Minim.STEREO, 1024);
```

If the window is 512 pixels wide, there is 1 pixel for every 2 samples, its height being the value retrieved using the call to `get()`. Assuming that the value of a data element is between −1 and +1, we draw the 1,024 data points as a line from (`i, datai`) to (`i+1, datai+1`) for all `i` between 0 and 1,023 by twos 2. This is illustrated in *Figure 75-1*.

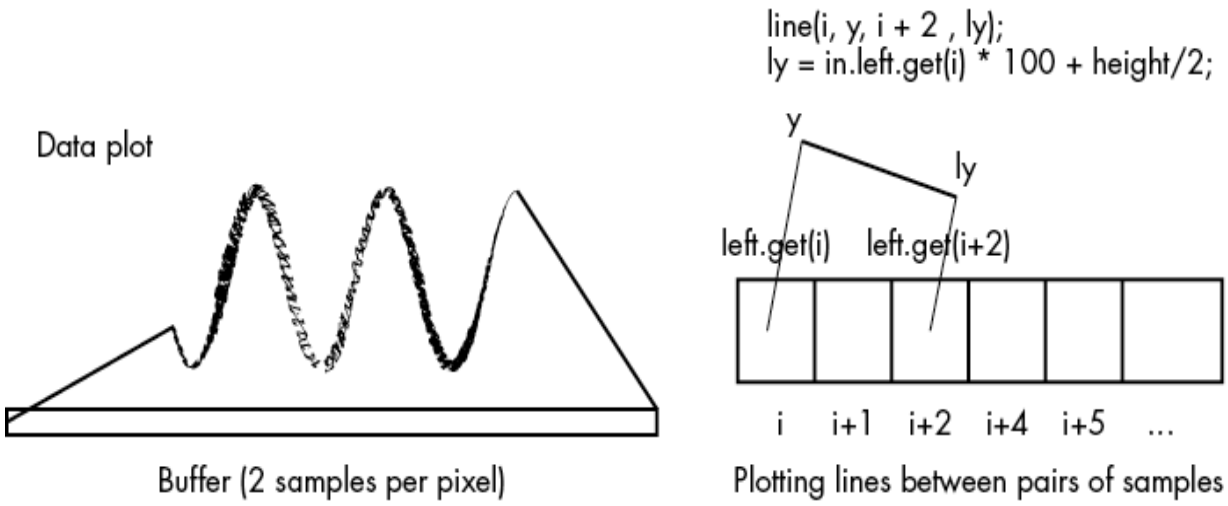


Figure 75-1: Scaling samples and plotting them as lines

In other words, we have the following:

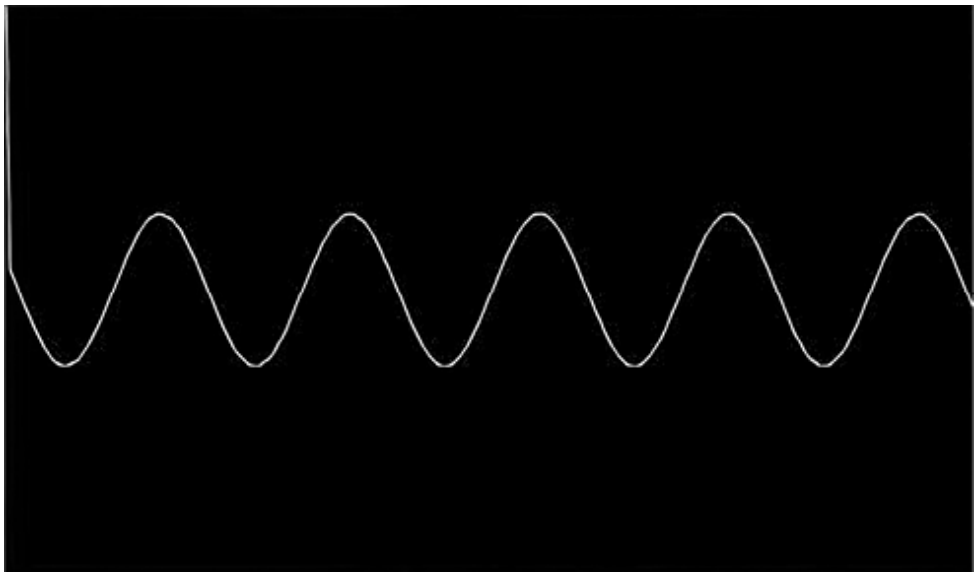
```
for (int i=0; i<1024; i=i+2)
{
  ly = in.left.get(i)*100+height/2;
  if (i!=0) line (i, y, i+2, ly);
  y = ly;
}
```

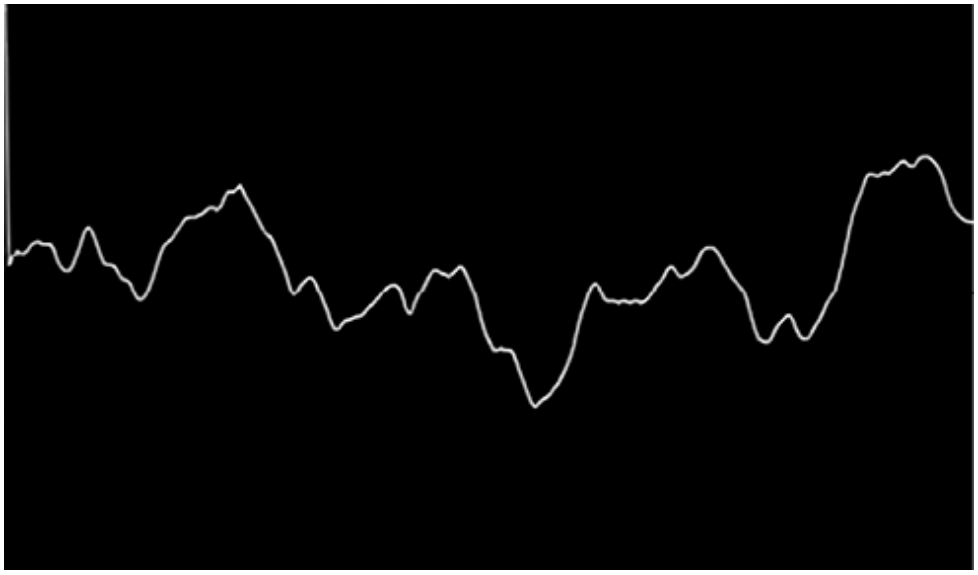
We do this in the `draw()` function so it will refresh every 10th of a second and display an animated version of the audio. We scale the data by multiplying by 100, giving a total height of 200 pixels, and then translate it to the vertical center of the window by adding this value to the data point.

NOTE

It may be better to show both stereo channels simultaneously, or to average the two as the volume for each sound moment.

```
import ddf.minim.*;
Minim minim;
AudioInput in;
AudioPlayer player;

void setup ()
{
  size(512, 300);
  minim = new Minim(this);
  in = minim.getLineIn(Minim.STEREO,1 1024);
  stroke(255);
  fill(100);
  frameRate(10);
}

void draw ()
{
  float ly,  y=0;

  background(0);
  ly =  0;
2 for (int i=0; i<1024; i=i+2)
  {
 3 ly = in.left.get(i)*100+height/2;
    if (i!=0) line (i,y, i+2, ly);
    y = ly;
  }

}
// always close Minim audio classes when you are done
with them
void stop ()
{
  in.close();
  minim.stop();
  super.stop();
}
```





Sketch 76: Controlling a Graphic with Sound

PC-based music players frequently offer a set of visualizers that present abstract moving images that change in coordination with the music, as shown in *Figure 76-1*. Sketch 75 is a visualizer that displays the actual signal, which can be useful for signal analysis and editing, but the purpose of music player visualizations is to entertain by presenting interesting images. This sketch represents one attempt to implement such a visualizer.

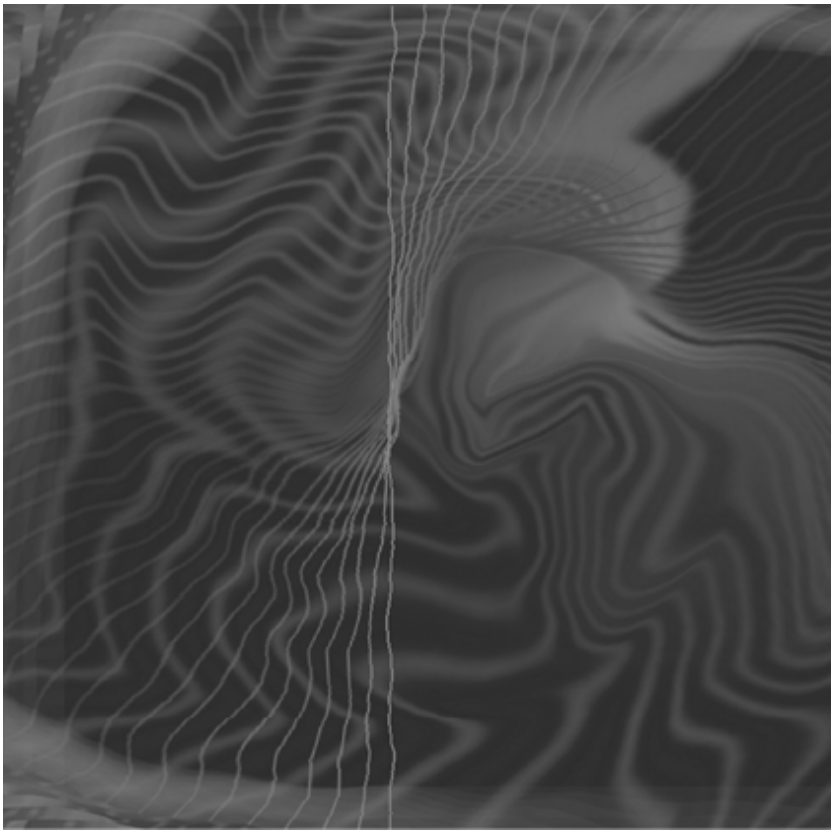


Figure 76-1: An example visualizer

There are many ways to control images using music, but the underlying idea is to pull numbers from the sound data and use them as parameters to some graphical model so the display reacts to the actual sound. Beyond the raw sound data points described in the previous sketch, we want to measure values that indicate changes in the sound so that the display is dynamic. The difference between two consecutive values is one measure. These numbers would tend to be similar to each other, so two values at a fixed time from

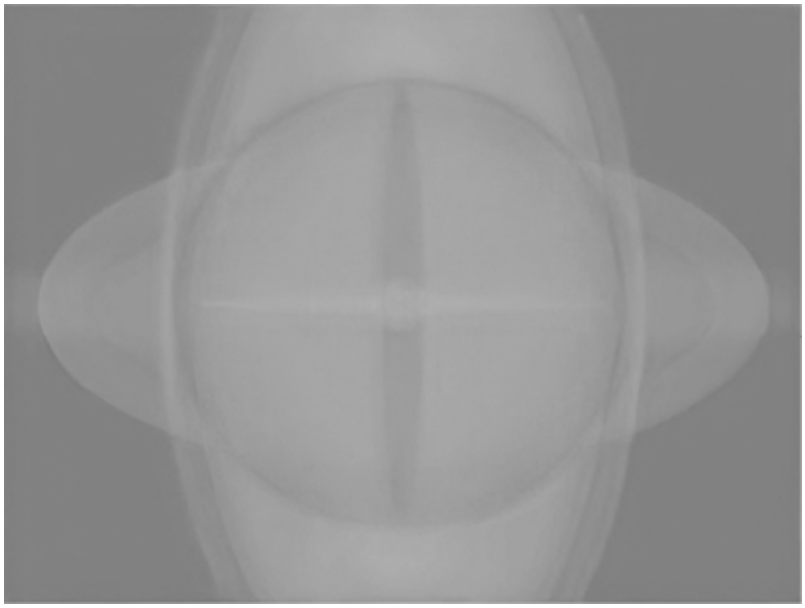
each other might give a better range of numbers. Another idea would be to use the difference between the left and right channels. More complicated measurements include the difference between a data value and the average for a short time or the difference between the maximum and minimum values over a time period.

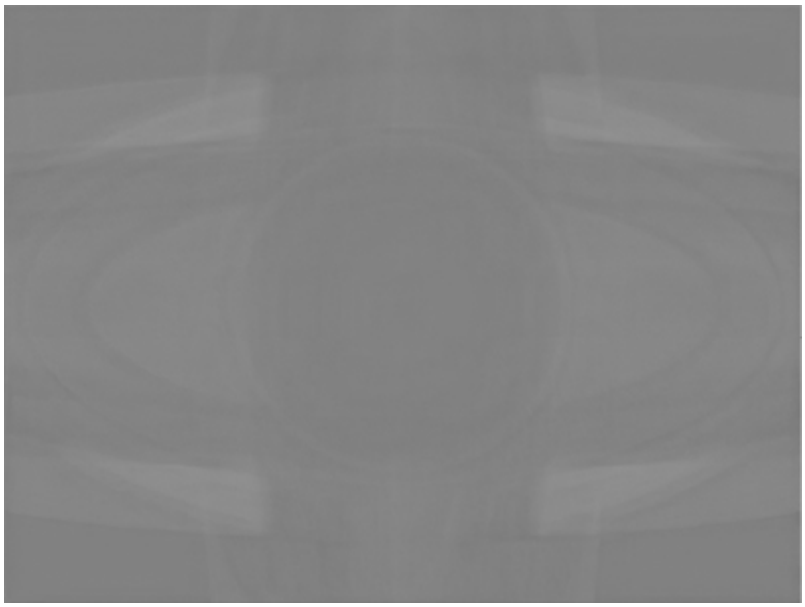
Once we decide which measurements to use, what will we use the values for? This depends on the visual effect we desire. They could represent x, y positions, colors, speed, or even shape parameters.

This sketch will use ellipses as the basis for the display. The data from the left and right channels of the current buffer will define the width and height parameters of an ellipse to be drawn at the center of the screen. The size of the ellipse will increase by five pixels for each frame, so it will grow from the center outwards 2. The color of the ellipse will be related to the difference between the current left data value and the corresponding left data value from the previous buffer 4; this means that color is a function of variation over time. By drawing each ellipse with a transparency (alpha) value of 30, we can make the colors blend into each other. Because we're using transparency, we should display the largest ellipses first, and then smaller ones, or the smaller ones could be overwhelmed by ones drawn above them. We must maintain a set of parameters for these ellipses so that we can display all of them correctly each iteration, and we do this by saving them in a set of arrays: `colors`, `hsize`, and `vsize` for the ellipse color and size.

Start the program and then play a sound file with another program on your PC. The sketch extracts the numeric parameters from the sound 3 and displays the corresponding ellipses each frame 1. The visual is surprisingly interesting given the simplicity of the method.

```
import ddf.minim.*;
final int MAXOBJECTS = 50;
color colors[] = new color[MAXOBJECTS];
int hsize[] = new int[MAXOBJECTS];
int vsize[] = new int[MAXOBJECTS];
int last = 0;
float lastd=0, dl, dr, d;

Minim minim;
AudioInput in;
void setup ()
{
  size (400, 300);
  minim = new Minim(this);
  in = minim.getLineIn(Minim.STEREO, 1024);
  ellipseMode (CENTER); colorMode(HSB);
  noStroke(); frameRate(50);
}

void draw ()
{
  background(128);
  for (int i=MAXOBJECTS-1; i>=0; i--)
  {
    fill (colors[i]);
1 ellipse (200, 150, vsize[i], hsize[i]);
2 vsize[i] += 5; hsize[i] += 5;
  }
3 dl = ((in.left.get(0)+1)/2) *100;
  dr = ((in.right.get(0)+1)/2) *100;
  if (dl>dr)
  {
    vsize[last] = (int)(dl-dr)*200; hsize[last] = 1;
  }
  else
  {
    vsize[last] = 1;  hsize[last] = (int)(dr-dl)*200;
  }
4 colors[last] = color(abs(lastd-dl)*100, 200, 250, 30);
  lastd = dl;
  last = (last + 1)%MAXOBJECTS;
}
```





Sketch 77: Positional Sound

Because humans have two ears, we can roughly identify the location of a sound. We do this partly by using the difference in time of arrival and the volume of the sound at each ear. A sound is louder in the ear that is nearest to the source, and we can use this fact to simulate positional sound using a computer. In this sketch, we'll play a sound and let the user select a listening position in the center of the sketch window. The user can move about, changing the angle they are facing with the A and D keys and stepping forward and backward using W and S.

When the user is facing exactly toward or away from a sound source, the loudness in each ear should be about equal. When they are facing so that the left ear is pointing to the source, the volume in the left ear is loudest and in the right ear it is the quietest, and vice versa when the right ear is facing the sound. With this in mind, we can map volumes from loudest in the left to equal to loudest in the right as a function of the way the listener is facing.

Imagine an angle made between the listener's position, the source position, and the x-axis, labeled θ in *Figure 77-1*. The angle that the listener is facing combines with the angle between the listener and the object to determine how loud the sound will seem in each ear, and thus determines how loud we should play the sound from each speaker to simulate positional sound.

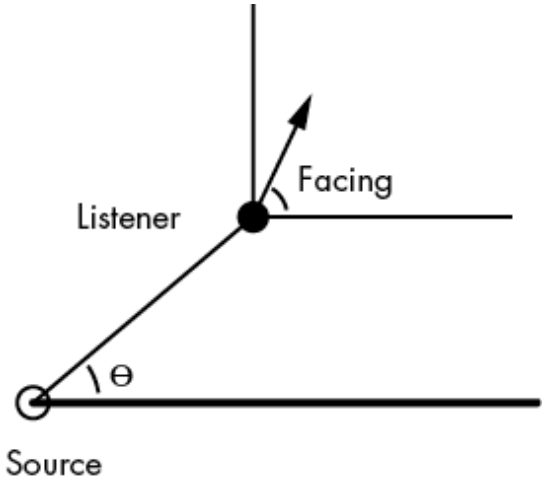


Figure 77-1: Geometry of positional audio

The angle θ is determined using trigonometry as the arctangent of the difference in x over the difference in y 3, or the following, where the `atan2` function handles the case where the angle is vertical:

```
θ = atan2(y1-y0, x1-x0)
```

The difference between the facing angle and θ (theta) defines an angle that controls the volume between two stereo channels being played, via the `setPan()` function. A parameter of −1 means full left channel, 0 means a balance, and +1 means full right. A bit of fiddling on paper shows that a 0-degree angle to the source should correspond to a pan of 0, 90 degrees has a pan of −1, 180 degrees has a pan of 0, and 270 degrees has a pan of +1. These are the extreme points of the function `-sin(facing-theta)`, so this value is passed to `setPan()`.

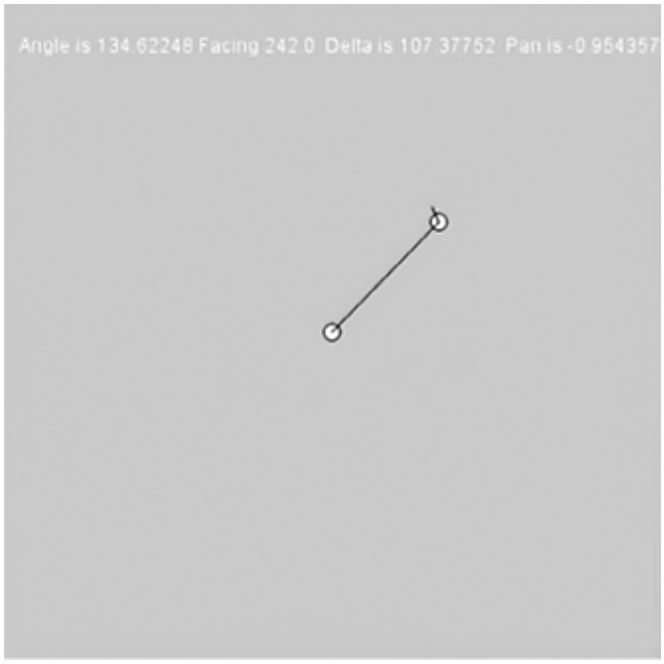
In summary, the sound file (a simple tone) starts playing 1; the sound source is initially located at (200, 200) 2, and the user is initially at (300, 200) but can rotate and move. The volume of the sound played in each speaker is set by determining the angle θ, computing `delta = facing-theta`, and setting the pan to `-sin(delta)` 4.

NOTE

This sketch does not consider the decrease in sound volume as a function of distance `d` from the source. To incorporate the volume decrease, try multiplying the volume by a factor of $1/d^2$.

```
import ddf.minim.*;
float facing=0, delta=0;
float x=300, y=200, dx=1, dy=0;
AudioPlayer player;
Minim minim;
void setup ()
{
  size(400, 400);
  minim = new Minim(this);
  player = minim.loadFile ("sound.mp3");
1 player.loop();
}

void draw ()
{
  background(200);
  ellipse (200, 200, 10, 10);  ellipse (x, y, 10, 10);
2 line (x, y, 200, 200);  line (x, y, x+10*dx, y+10*dy);
  text ("Angle is "+theta (x, y, 200, 200)+" Facing "+
      facing+"  Delta is "+delta+"  Pan is "+(-
sin(radians(delta))), 10, 30);
}

float theta (float x0, float y0, float x1, float y1)
{
  float x;
3 x = (float)atan2(y1-y0, x1-x0);
  if (x<0) x = x + 2*PI;
  return degrees(x);
}

void keyPressed ()
{
  if (key == 'w')   // Move 'forward'
  {  x += 5*dx;   y += 5*dy;  }
  else if (key == 's')  // Move 'backward'
  {  x -= 5*dx;    y -= 5*dy;   }
  else if (key == 'a') facing = facing - 1.0;   // Turn
left
  else if (key == 'd') facing = facing + 1.0;   // Turn
right
  if (facing < 0) facing = facing + 360;
  else if (facing>360) facing = facing - 360;
  dx = cos(radians(facing)); dy = sin(radians(facing));
4 delta = facing - theta(x, y, 200, 200);
```

```
  player.setPan (-sin(radians(delta)));
}
```



Sketch 78: Synthetic Sounds

This sketch will implement a small sound synthesizer. It will only have eight keys, more like a child's toy piano, but it will be functional and can serve as the basis for more complex sound synthesis projects.

`Minim` provides a type (a class) named `AudioOutput` that allows us to display signals, not just sound files, on the PC hardware. It allows the playing of a note, although not exactly musical notes as normally understood. A *note* in this context is a digital audio signal having a specific frequency.

The name of the `AudioOutput` variable in the sketch is `out`, and it is initialized 1 as the following:

```
out = minim.getLineOut(Minim.STEREO);
```

This call allocates a new instance of `AudioOut` that is accessible from the variable `out`. To play a note, call the `playNote()` function 2:

```
out.playNote(440.0);
```

This sends a sine wave with a frequency of 440 Hz (the musical note A) to the sound card. `playNote()` can be called with nearly any frequency, because the "notes" are just snippets of a sine wave.

Unfortunately, the `AudioOutput` object likes to impose a specified duration on a note, so the note plays for what the system believes to be a single unit of time. To imitate a musical instrument played by a human who can vary the duration, we need to call `playNote()` with more parameters:

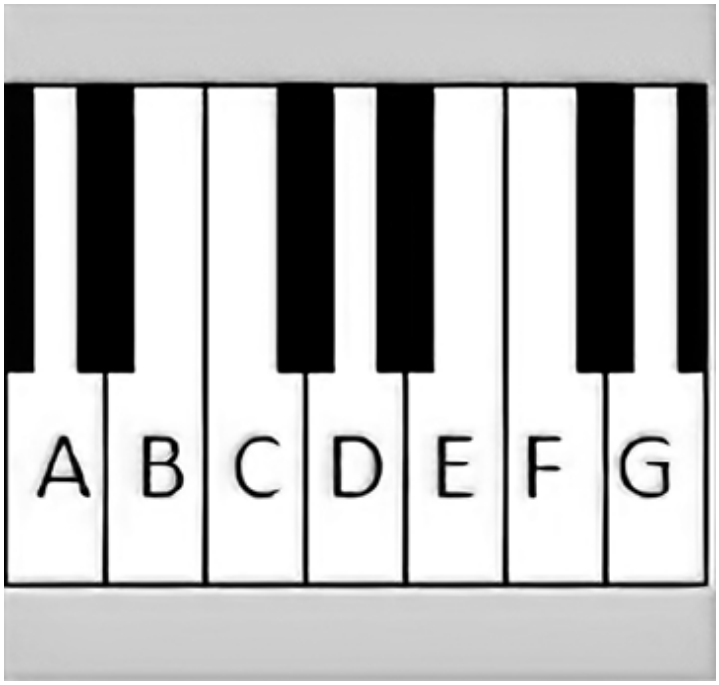
```
out.playNote(0, 1000, 493.9);
```

In this example, 0 is the time until the note is to be played (immediately), 1,000 is the duration, and the final parameter is the frequency; 1,000 units is a long time.

The sketch displays a simple piano image with labeled keys. When the user clicks the mouse on one of the graphical piano keys, the program plays that note 2; the value of the x position of the mouse tells us what the note is

(in `mousePressed()`). When the mouse button is released, the program creates a new `AudioOutput` 3 so that the old note stops playing and a new one can start (in `mouseReleased()`).

```
import ddf.minim.*;
import ddf.minim.signals.*;

Minim minim;
AudioOutput out;
PImage piano;

void setup ()
{
  size(100,100);
  surface.setResizable(true);
  piano = loadImage ("piano.png");
  surface.setSize(piano.width, piano.height);
  minim = new Minim(this);
1 out = minim.getLineOut(Minim.STEREO);
}

void draw ()
{
  image (piano,0,0);
}

void mousePressed ()
{
2 if (mouseX<20) out.playNote(0, 1, 440.0);
  else if (mouseX < 38) out.playNote(0, 1, 493.9);
  else if (mouseX < 57) out.playNote(0, 1, 523.3);
  else if (mouseX < 77) out.playNote(0, 1, 587.3);
  else if (mouseX < 95) out.playNote(0, 1, 659.3);
  else if (mouseX < 114) out.playNote(0, 1, 698.5);
  else if (mouseX < 134) out.playNote(0, 1, 784.0);
}

void mouseReleased ()
{
3 out = minim.getLineOut(Minim.STEREO);
}

void keyPressed ()
{
  out.close();
  super.stop();
  exit();
}
```


A
B
C
D
E
F
G

Sketch 79: Recording and Saving Sound

This sketch captures the audio currently playing on the computer and saves it in a file in *.wav* format. This would permit recording sound from Skype calls, websites, and podcasts, to name a few.

In Sketches 75 and 76 we used `Minim` and an `AudioInput` object to access the currently playing sound for visualization. In this case, the next step is to create an `AudioRecorder`, which takes as a parameter an input from which we can collect sound; that is, the `AudioInput` object connected to the currently playing sound.

An `AudioInput` has three functions (methods) of importance:

`beginRecord()` Start saving audio samples.

`endRecord()` Stop saving the audio samples.

`save()` Store the saved samples as an audio file.

How much audio data we can save depends on the memory available on the computer.

The sketch opens a window and displays the playing sound signal as in Sketch 75. If the user types the R character 2 (handled by `keyReleased()`), we call `beginRecord()` and start saving data. When the user types Q 3, we call `endRecord()` and the recording stops. If the user types S, we call `save()` 4.

We specify the file used to save the data as a parameter on the creation of the `AudioRecorder` 1:

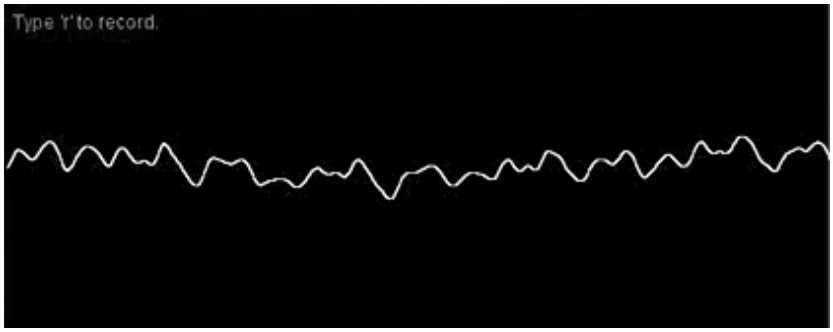
```
recorder = minim.createRecorder(input, "processing.wav",
true);
```

Here, `input` is the already existing `AudioInput` object, *processing.wav* is the file where we'll save the sound data, and the final parameter represents whether or not the recording is *buffered*, which is to say whether the data is saved in memory or written directly to the file. If it's not buffered, the system opens the file when recording begins. Otherwise the system opens the file when we write the data.

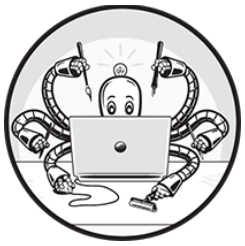
A small change to this code would allow the user to save to a different file each time they start and stop recording. This could be useful for voice recording, such as reading scripts or reading books to tape.

```
import ddf.minim.*;
Minim minim;
AudioInput input;
AudioRecorder recorder;
void setup ()
{
  size(512, 200);
  minim = new Minim(this);
  input = minim.getLineIn(Minim.STEREO, 1024);
1 recorder = minim.createRecorder(input,
"processing.wav", true);
}

void draw ()
{
  background(0);  stroke(255);
  for(int i = 0; i < input.left.size()-1; i++)
    line (i,input.left.get(i)*100+height/2, i+1,
          input.left.get(i+1)*100+height/2);

  if (recorder.isRecording())
  {
    fill(255, 0, 0);
    text("Recording. Type 'q' to quit recording.", 5,
15);
  } else
  {
    fill(0, 255, 0);
    text("Type 'r' to record.", 5, 15);
  }
}

void keyReleased ()
{
2 if ( key == 'r' && !recorder.isRecording())
recorder.beginRecord();
3 else if (key == 'q' && recorder.isRecording())
recorder.endRecord();
4 else if ( key == 's')  recorder.save();
  else if ( key == '0') stop();
}

void stop ()
{
  input.close();
```

```
  minim.stop();
  super.stop();
  exit();
}
```



10
WORKING WITH VIDEO



Sketch 80: Playing a Video

We can use Processing to play videos but, as was the situation with audio, Processing does not have its own facility for doing so. Instead, we use the `Movie` class from the `processing.video` library, which in turn uses the underlying Java-based video functions. As a first example, this sketch will load and display a short video.

First, we import the `processing.video` library 1 as the first line in the program:

```
import processing.video.*;
```

Now we can declare an instance of the `Movie` class 2, one for each movie we want to play:

```
Movie movie;
```

We load the video file when we initialize the class instance by calling its constructor (see Sketch 43), specifying the name of the file as a parameter 3:

```
movie = new Movie(this, "car.avi");
```

In the `setup()` function, we begin reading the video from the file by calling the `movie.play()` function (which doesn't just play the video, as you'd expect). A video is a sequence of compressed images or frames, just like an animation, and each one can take some significant time to read and decode. After we call `play()`, the system tries to read frames from the file, and when one is ready, the `available()` function returns `true`. We can then acquire the frame using `read()`. Like a `PGraphics` object, a `Movie` object can be treated as an image and displayed using the `image()` function. Thus, this is the process for displaying a movie 4:

```
if (movie.available())
{
  movie.read();
```

```
  image (movie, 0, 0);
}
```

If no new frame were available, `read()` would not be called, and the previously read frame would be displayed in its place. This is usually not noticeable.

The `Movie` class plays the sound with the movie.

The sketch also prints relevant information at the top of the window. It counts the number of frames read in and displays that number. It also displays the *time count*, which is the number of seconds that have been played so far, retrieved using the `movie.time()` function call 5. When the movie is complete, as indicated by `movie.time() >= movie.duration()` 6, the counters reset and the movie resumes playing from the first frame by calling `movie.jump(0)`. The `jump(t)` function call moves the current frame to the one at time `t`. Playing in a loop could also be accomplished by calling `movie.loop()` instead of `movie.play()`. In that case, the replaying of the movie from location 0 would be automatic.

```
1 import processing.video.*;

  boolean playing = true;
2 Movie movie;
  int frame = 1;

  void setup ()
  {
 3 movie = new Movie(this, "car.avi"); // Create the
  instance of Movie
    size(320, 300);
    movie.play();    // Start playing
  }

  void draw ()
  {
 4 if (movie.available())   // New frame?
    {
      movie.read();           // Read it
      frame = frame + 1;      // Count it
    }
    image(movie,0,0);         // Display the current frame
 5 text ("Frame "+frame+"  Time: "+ (float)((int)
  (movie.time()*100))/100, 10, 20);
 6 if (movie.time() >= movie.duration())  // End
    {
      frame = 1; movie.jump(0);  // Restart the counters.
  Rewind.
    }
  }

  void mouseReleased ()
  {
    if (playing)
    {
      movie.pause();
      playing = false;
    } else
    {
      movie.play();
      playing = true;
    }
  }
```


Frame 147 Time: 7.62

Sketch 81: Playing a Video with a Jog Wheel

A *jog wheel* (or *shuttle dial*) is a device, often circular, that allows the user to advance or back through a video. Turning it clockwise moves the video forward by individual frames, and turning it counterclockwise moves the video backward. Editors often use this for editing where the video needs to be positioned frame by frame. This sketch will implement an approximation of this *jogging* process. The video will begin to play, and the user can adjust the speed and direction of play using the mouse. At any point, the user can stop the video and back up slowly to arrive at any specific frame.

To do this we have to address the problem of how to play a video backward. The `jump()` function permits the positioning of the video at any moment in time 2. The time of any particular frame depends on the frame rate, which is the number of frames played per second. Given a frame rate of `rate`, we know that each frame lasts 1/`rate` seconds. The final frame occurs at `duration()` seconds from the start, so positioning at the frame before that could be done with the following call:

```
movie.jump (movie.duration-(1/rate))
```

The frame before that one is at `movie.jump (movie.duration-(1/rate)*2)` and so on. Simply step backward through the frames in this way, read the frame, and display it.

In the sketch, we store the time of the current frame in a `time` variable, and the time between frames is in the variable `ftime`. We will use the mouse to control the speed with which the video will be displayed. A mouse click in the middle of the screen sets the speed to 0 by setting `ftime` to 0. A click on the right sets `ftime` to a value in proportion to the distance from the middle, and it moves the video forward; a click on the left sets `ftime` to a value that moves the video backward. Initially `ftime = 1/rate`, but this becomes −3 times that for a far left click and +3 times that for a far right click. This is the whole calculation 3:

```
ftime = 3*((float)(width/2 - mouseX)/(width/2))/rate;
```

A minor problem occurs at the end of the video, which is really the beginning if it is playing in reverse. Time is set to 0 if the end is found while moving forward, and it is set to `duration()-ftime` if the beginning is found while moving backward.

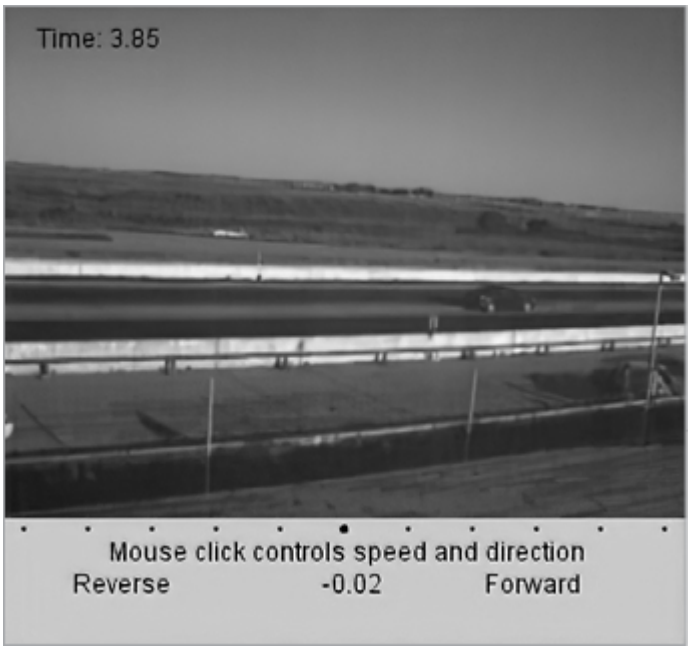
The basic display process 1 occurs within `draw()` and is as follows:

```
if (movie.available())  movie.read();  // Read a frame if one
is there
image(movie,0,0);                      // Display it
time = time - ftime;                   // Advance/retard the
time value
movie.jump(time);                      // Set frame to the
one at that time
```

The sketch displays a simple calibration to allow the user to select a speed, and it also displays the value of `ftime`.

```
import processing.video.*;

boolean playing = true;
Movie movie;
float time = 1, ftime = 1, rate = 20;

void setup ()
{
  movie = new Movie(this, "car.avi");
  size(320, 300);
  ftime = 1.0/rate;
  time = movie.duration() - ftime;
  movie.play();    // Start playing
  fill (0);
}

void draw ()
{
  background(200);
1 if (movie.available())  movie.read();          // Read
it
  image(movie,0,0);         // Display the current frame
  text ("  Time: "+ (float)((int)
(movie.time()*100))/100, 10, 20);
  time = time - ftime;     // Control which is the next
frame
  movie.jump(time);
  if ( (time < 0) || (time>movie.duration()) )
  {
    if (time<0) time = movie.duration() - ftime;
    else time = 0;
 2 movie.jump(time);      // Restart the counters.
Rewind.
  }
  text ("Mouse click controls speed and direction", 50,
260);
  text (" Reverse
Forward", 30, 275);
  ellipse (160, 245, 3, 3);
  for (int i=160; i<320; i=i+30) ellipse (i, 245, 1,
1);
  for (int i=160; i>0; i=i-30) ellipse (i, 245, 1, 1);
  text (""+  -((int)(ftime*100))/100.0, 150, 275);
}
```

```
void mouseReleased ()
{
3 if (mouseX < width/2) ftime = 3*((float)(width/2 -
mouseX)/(width/2))/rate;
  else  ftime = -3*(float)(mouseX-
(width/2))/(width/2)/rate;

}
```



Sketch 82: Saving Still Frames from a Video

This sketch will allow the user to save a set of still image frames from a video. The video is played in a loop so that the user can select all of the frames they need. Clicking the mouse will start saving images, and clicking again will stop it.

Saving frames is accomplished using the `save()` function of the `Movie` class object. If `movie` is a `Movie` object, the following call saves the current frame in the named file as the type indicated by the file extension:

```
movie.save("name.jpg");
```

This is the same way we save `PImage` pictures. In this case, we save a JPEG, but GIF, PNG, and other file formats work too.

To save multiple frames without overwriting the same file each time, we might use the number of stills that we have already saved, stored in the variable `v`, in the filename, as follows:

```
movie.save("frame"+v+".jpg");
```

This means that the filenames would be *frame1.jpg*, *frame2.jpg*, and so on.

With this labeling scheme, however, there's no way to tell where one saved sequence ends and the next one begins. This sketch solves that problem by using the variable `nclicks` in conjunction with `v`. When the user clicks the mouse while the frames are being saved, then saving ceases, `nclicks` is incremented, and `v` is reset. We build a filename using the frame count and a letter that is relative to the `nclicks` variable: `nclicks` = 0 adds the letter "`a`" to the name, `nclicks` = 1 adds "`b`" to the name, and so on. The file for each frame is actually saved as follows 1:

```
movie.save("frame"+char(nclicks+int('a'))+v+".jpg");
```

The first sequence would be *framea1.jpg*, *framea2.jpg*, . . . and the second would be *frameb1.jpg* and so on.

The sketch draws the time on the screen, but this is for the user—it will not appear on the saved image.

Another way to save video frames is to display them in the sketch window and then save the sketch window as an image. If we did that in this case, the time drawn on the window would in fact be saved to the file with the image.

```
import processing.video.*;

boolean saving = false;
Movie movie;
float time = 1, rate = 20;
int frame = 1, v = 0;
int nclicks = 0;

void setup ()
{
  movie = new Movie(this, "car.avi"); // Create the
instance of Movie
  size(320, 300);
  movie.frameRate(rate);
  movie.play();    // Start playing
  fill (0);
}

void draw ()
{
  if (saving) background (0, 200, 20);
  else background(200);
  if (movie.available())
  {
    movie.read();             // Read it
    image(movie,0,0);         // Display the current
frame
    if (saving)
    {
      v++;
   1 movie.save("frame"+
char(nclicks+int('a'))+v+".jpg");
      frame = frame + 1;
    }
  } else image(movie,0,0);

  text ("  Time: "+ (float)((int)
(movie.time()*100))/100, 10, 20);
  if (saving) text ("Saving file "+frame, 20, 275);
  if ( movie.time() >= movie.duration() )
    movie.jump(0);  // Restart the video
}

void mouseReleased ()
{
  saving = !saving;
```

```
  if (!saving) {
    nclicks = nclicks + 1;
    v = 0;
  }
  if (nclicks > 25) nclicks = 0;
}
```

Sketch 83: Processing Video in Real Time

Some applications process or analyze a video frame by frame, and it is not necessary to see the result in real time. For example, it is possible to analyze a batter's swing by capturing a video, enhancing relevant portions in each frame, and then putting the enhanced frames back in video form. It is even possible, when the analysis of each frame does not require too much computational effort, to do the processing as the video is playing and see the result as the action is going on.

In this sketch, the video that we used in the previous two sketches will be converted to grayscale and then *thresholded* in real time, just as we did in Sketch 23 for a still image.

Recall that we can treat a `Movie` object just like a `PImage` (they have the same local functions). We extract each pixel `p` in the movie image using `movie.loadPixels()` 1 and calculate a brightness or grey level by averaging the color components: `(red(p)+green(p)+blue(p))/3` 2. If this value is less than a threshold, the corresponding pixel in the display image is set to black; otherwise it is set to white. In this sketch, the threshold value is 100. The result is a video that displays only black and white pixels.

The setup is the same as before, but we also create a second image the size of a video frame (named `display`) that will hold a processed copy of each frame as it is displayed. The `draw()` function reads a frame when it is ready and then calls a local `thresh()` function to calculate a thresholded image. After `thresh()` has created a thresholded version of the movie image, both are displayed, one above the other, and both versions play simultaneously.

The result in this case is unimpressive, but it does give an idea of what we could do. For example, if we choose the threshold carefully, it might be possible to show only the motion of the car in the scene, removing the background clutter.

In other videos, we could locate faces, enhance and read license plates on moving cars, or inspect and count apples moving past the camera on a conveyor belt. These are problems in *computer vision*, and Processing is a

good tool for building computer vision systems because of the ease with which it deals with images.

NOTE

A tool called SimpleOpenNI is available for download from https://code.google.com/p/simple-openni/. It allows Processing to communicate with Kinect devices, which in turn allows us to acquire 3D images in real time. Microsoft uses this to build computer games, but there are many other potential uses, like the computer vision problems just described.

```
import processing.video.*;

PImage display;
Movie movie;


void setup ()
{  // Create the instance of Movie
  movie = new Movie(this, "car.avi");
  size(320, 480);
  movie.play();    // Start playing
  movie.frameRate(15);
  fill (0);
  display = createImage (320, 300, RGB);
}

void draw ()
{
  background (255);
  if (movie.available())
  {
    movie.read();// Read it
    thresh();
  }
  image(display,0,0);
  image (movie, 0, 240);

  text ("  Time: "+ (float)((int)
(movie.time()*100))/100, 10, 20);
}
void thresh ()
{
  color p,q;
1 movie.loadPixels();
  for (int i=0; i<movie.pixels.length; i++)
  {
    p = movie.pixels[i];
 2 if ((red(p)+green(p)+blue(p))/3 < 100) q =
color(0,0,0);
    else q = color(255,255,2525);
    display.pixels[i] = q;
   }
   display.updatePixels();
}
```


Time: 0.77

Sketch 84: Capturing Video from a Webcam

Webcams are present on most computers and almost all laptops. The previous sketches dealt with video that had already been *captured*, in the sense that a video file was available to be displayed or processed. This sketch will capture live video data from a webcam and display it in grayscale.

The `Capture` class deals with cameras and image/video capture. To use it, first declare an instance 1:

```
Capture camera;
```

Then initialize it using the class constructor. The class constructor may take only the parameter `this`, or `this` and a device specifier 2:

```
Camera = new Capture (this);
camera = new Capture (this, myCamera);
```

The `myCamera` variable is a device specifier string of the following form:

```
"name=USB2.0 HD UVC WebCam,size=160x120,fps=15"
```

Much of the information in this string has an obvious meaning, and most is not absolutely necessary. If you know that the camera has a resolution of 640×480, the following call will open the camera:

```
camera = new Capture (this, "size=640x480");
```

Image capture begins with a call to `start()` 3:

```
camera.start();
```

As when playing a video, a frame is available when `camera.available()` returns `true`. The camera instance can now be treated like a `PImage` and be displayed with a call to `image()`.

This sketch copies the camera image into a `PImage` variable, `display` 4. The function `grey()` converts the color image into a grey one, which is displayed in place of the original. The result is a moving grayscale image of what is being captured by the camera. Be patient—it can take some time to open the camera device.

The `Capture` class function `list()` looks at the camera devices available on the computer and returns a list of descriptors that can be used in the constructor. So, if this line

```
String[] cameras = Capture.list();
```

were to be followed by this

```
for (int i=0; i<cameras.length; i++)
  println (cameras[i]);
```

then a list of available cameras would be printed to the window. We could select one and use the index for it in the code to select it from the `cameras[]` array. For instance, you could search for a camera that is 640×480 at 130 frames per second and find it as camera `i` in the list. Then you could use the selector you want by indexing the array:

```
camera = new Capture (this, cameras[i]);
```

```
import processing.video.*;
Capture camera;
PImage display;
void setup ()
{
  String[] cameras = Capture.list();
    if (cameras.length == 0)
    {
      println("There are no cameras.");
      exit();
    }

  camera = new Capture(this, cameras[0]);
   display = createImage (640, 480, RGB);
  camera.start();
   size (640, 480);
}

void draw ()
{
  if (camera.available() == true) camera.read();
  display.copy (camera, 0,0,640, 480, 0,0,640, 480);
   grey();
   image(display, 0, 0);  //  set(0, 0, camera);
}

void grey ()
{
  color p;
  int k;

  for (int i=0; i<display.pixels.length; i=i+1)
  {
    p = display.pixels[i];
    k = (int)((red(p)+green(p)+blue(p))/3);
    display.pixels[i] = color(k,k,k);
  }
}
```

Sketch 85: Mapping Live Video as a Texture

In the previous sketches, you saw that a `Movie` object can be treated as a `PImage` for display purposes and even for extracting pixels from a video frame. This sketch shows the use of a video as a texture for a 3D surface, again like a `PImage`. The idea is to paint a four-cornered plane (a quad) with a movie so that the video plays on a 3D plane and is foreshortened as the user's point of view changes.

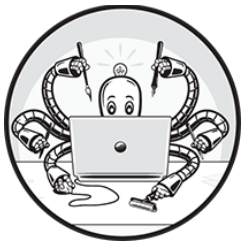
The first part of the sketch sets up the webcam (as before), establishes the `camera` variable as a source of images, and establishes P3D as the current renderer. When executing, the system requires a few seconds to figure out what cameras are attached and which one to use. We do all of this, including starting the camera, by calling `start()` 1 in `setup()`.

In `draw()`, the first thing is to check if there is a new image available. If so, we read it; if not, then the previous image remains as the current one 2. Then we establish a 3D environment, with a call to `camera` setting up the viewpoint 3. We draw a quad in the 3D space and use the webcam as a texture 4. The viewpoint oscillates a little bit (x between −30 and 100) 5 to show that the view is changing.

The effect is that the quad seems to continuously change location and orientation while the live video plays within the quad. An interesting variation on this would be to draw a rotating cube with the video mapped on all faces. This would show nothing new, but it would take more code.

```
import processing.video.*;
Capture camera;
float sx=30., sy=40., sz=12.;
int eyex=30, eyey=50, eyez=60;
int cx=20, cy=30, cz=12, dx=-1;

void setup ()
{
  String[] cameras = Capture.list();
  size (640, 480, P3D);

  if (cameras.length == 0)
  {
    println("There are no cameras.");
    exit();
  }

  camera = new Capture(this, cameras[0]);
1 camera.start();
}

void draw ()
{
2 if (camera.available() == true) camera.read();
  background(255);
3 camera(eyex, eyey, eyez, cx, cy, cz, 0, 1, 0);
  textureMode(NORMAL);
  beginShape (QUAD);
4 texture (camera);
  vertex (0., 0., 0.,  0., 0.);
  vertex (sx, 0., 0.,  1., 0.);
  vertex (sx, sy, 0.,  1., 1.);
  vertex (0., sy, 0.,  0., 1.);
  endShape (CLOSE);
  eyex = eyex + dx;
5 if (eyex < -30) dx = -dx;
  if (eyex > 100) dx = -dx;
}
```

11
MEASURING AND SIMULATING TIME



Sketch 86: Displaying a Clock

Time in a computer program can mean many things. There is *execution time*, which is the number of CPU cycles used by a program to a particular point. There is *process time*, or the amount of time that a program has been active. There is *real time*, which is the time on your watch. We can also call that *clock time*. This sketch will acquire the clock time from the computer system and display it as the hands of a traditional clock.

Getting the time of day from Processing is easy. These are the basic functions:

`hour()`: Returns the current hour in the day using a 24-hour clock.

`minute()`: Returns the number of minutes past the hour.

`second()`: Returns the number of seconds into the current minute.

The clock will be a circle, and there will be three linear indicators (hands): a second hand, a minute hand, and an hour hand. Since there are 60 seconds in a minute, the second hand will rotate about its center point by 360/60, or 6 degrees each second. The same is true of the minute hand; since there are 60 seconds per minute and 60 minutes in an hour, it rotates 6 degrees per minute. The origin for drawing the second hand is the clock's center, but the other endpoint is not known, only the angle. If the length of the second hand is *r*, then the second point can be determined with trigonometry, as seen in *Figure 86-1*.

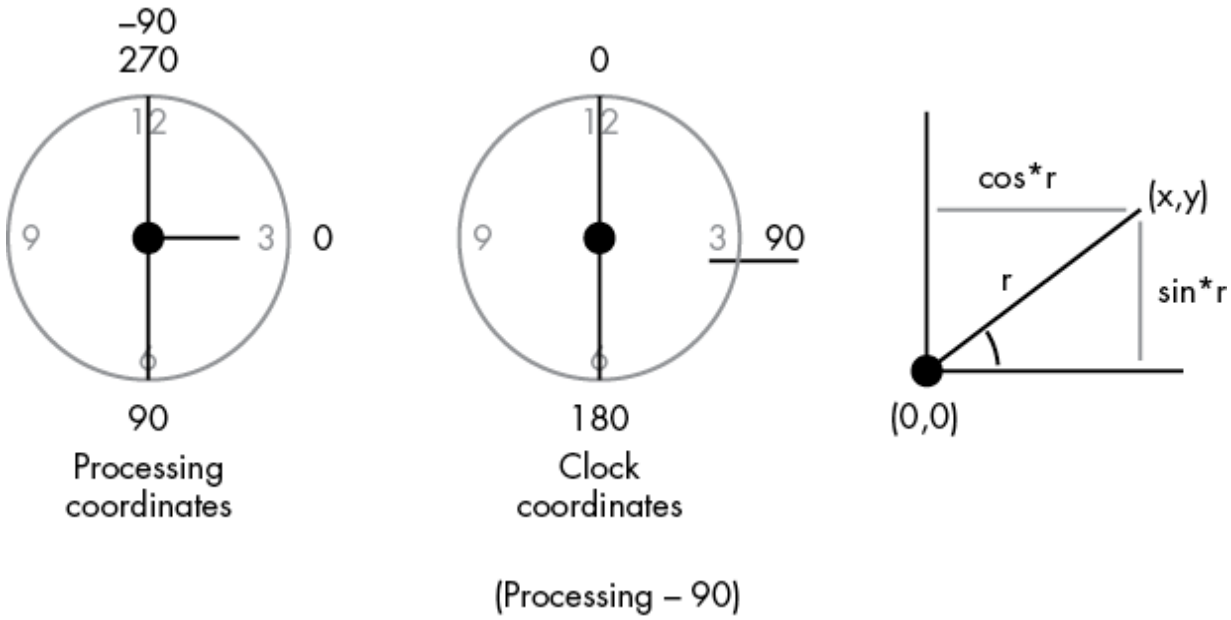


Figure 86-1: Determining the position of a clock hand

The angle as defined by Processing is not the same as that for a clock. On a clock, vertical represents 0, whereas in Processing that is −90 degrees. Drawing the second hand with (`cx`, `cy`) as the center point and with a length of `r` would be done as follows, where the variable `s` is the number of seconds 1:

```
s = radians(second()*6 - 90.0);
line (cx, cy, cx + cos(s)*sr, cy+sin(s)*sr);
```

The same scheme works for the minute hand, which is shorter. The hour hand should be shorter still, and the `hour()` value is divided by 2 if it exceeds 12. Also, there are only 12 hours in the 360-degree cycle, not 60, so each hour amounts to 30 degrees. The hour hand moves continuously around the face and does not jump when the hour changes, so each minute that passes should move the hour hand a little bit; 30 degrees (1 hour) is 60 minutes, so each minute moves the hour hand by 0.5 degrees 2. This is the code:

```
h = radians(hour()*30.0-90.0) + radians(minute()*0.5);
```

```
int cx=259, cy=380;
float hr = 8;
float mr = 15;
float sr = 20.0;
PImage clock;

void setup ()
{
  size(100,100);
  surface.setResizable(true);
  clock = loadImage ("clock.jpg");
  surface.setSize(clock.width, clock.height);
}

void draw ()
{
  float s, m, h;
  float angle, x, y;

  background(200);
  image (clock, 0, 0);

1 s = radians(second()*6 - 90.0);
  m = radians(minute()*6 - 90.0);
  h = hour();

  if (h > 12) h = h - 12;
2 h = radians(hour()*30.0-90.0) + radians(minute()*0.5);

  stroke(21);   // Draw the hands
  strokeWeight (2);
  line (cx, cy, cx + cos(s)*sr, cy + sin(s)*sr);
  line (cx, cy, cx + cos(m)*mr, cy + sin(m)*mr);
  strokeWeight(3);
  line (cx, cy, cx + cos(h)*hr, cy + sin(h)*hr);
}
```

Sketch 87: Time Differences—Measuring Reaction Time

Measuring the time between two events is the subject of this sketch: in particular, the time between a prompt by the computer and a response by the user, the *reaction time*. A typical (average) reaction time for a human is about 0.215 seconds. That is, between the time that a light goes on and the time that someone can press a button in response, an average of 215 milliseconds will pass.

This sketch measures reaction time by having the user click the mouse as quickly as they can when the background changes from grey to green. The background then changes back to grey, and the cycle repeats five times. The program measures the time between the screen turning green and the mouse click using the `millis()` function, and it averages the five trials to get a more precise measurement.

We use `millis()` because the function used in the previous sketch to move the second hand, `second()`, only returns whole seconds. `millis()` returns the number of milliseconds (1/1,000 seconds) since the sketch started executing. On the face of it, that value does not seem to have much meaning, but it does mean that the time difference between two events can be measured pretty accurately. Simply call `millis()` 1 when the first event happens, save the value, call it again when the second event occurs 2, and subtract the two.

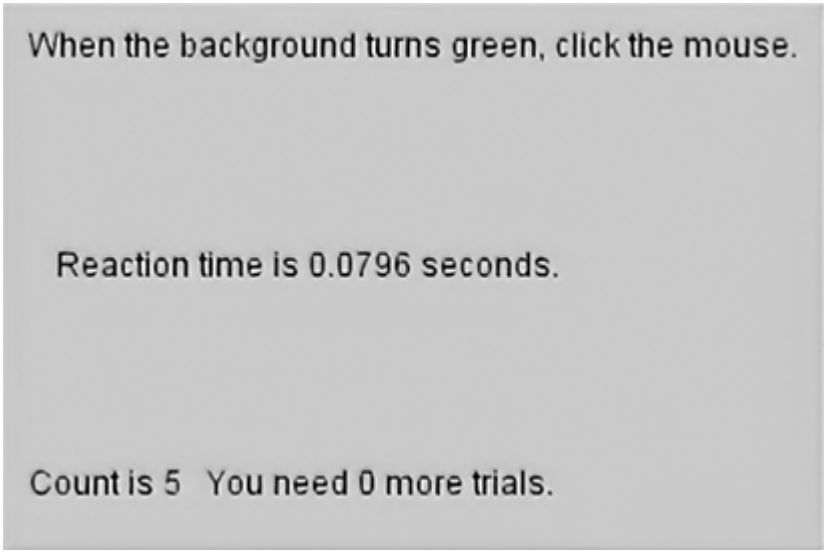
The `millis()` function can be used for other purposes, not the least of which is to determine how long it takes for a particular loop or function to execute. This sort of measurement is important when a program takes too long and the programmer needs to find ways to speed it up. Measuring one call to a function would not likely work, because most functions execute too quickly to measure, even slow functions. Instead, we put a function to be tested within a loop and execute it many times. We divide the time required to execute the loop by the number of iterations to determine the time needed for a single execution. Here is how the function `get(12,100)` could be timed:

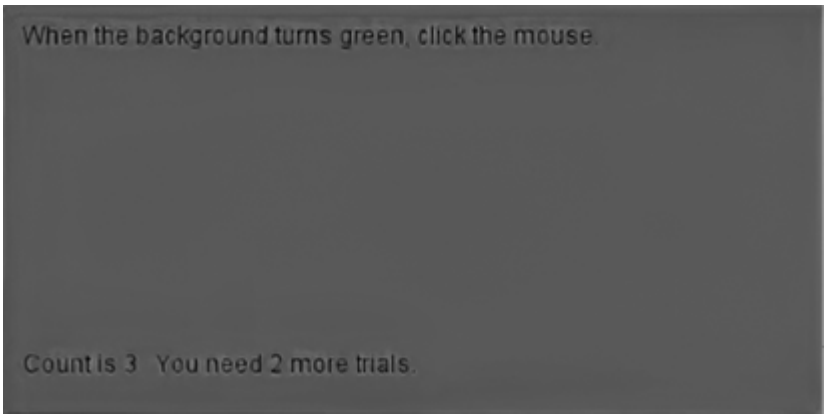
```
t1 = millis();
for (int i=0; i<100000000; i++)    y = get(12,100);
t2 = millis();
println ("Time was "+(t2-t1)+" or "+((t2-t1)/100000000.0));
```

The times obtained vary, so taking an average over many trials should give a more accurate result. Execution times may change depending on what other programs are executing at the same time or how many virtual memory page faults occurred.

```
float m0,m1, sum=0;
int wait = 0, count=0;
boolean timing = false;

void setup ()
{
  size (400, 200);
  fill (0);
}

void draw ()
{
  if (timing) background(0,200,0);
  else background(200);
  text ("Count is "+count+"   You need "+(5-count)+"
more trials.", 10, 180);
  text ("When the background turns green, click the
mouse.", 10, 20);
  wait = wait + 1;
  if (wait > random (5000) && !timing)
  {
    background(0, 200, 0);
    timing = true;
1   m0 = millis();
  }
  if (count == 5)
  {
    noLoop();
    sum = sum/count;
    text ("Reaction time is "+sum/1000 + " seconds.",
20, 100);
  }
}

void mousePressed ()
{
  if (timing)
  {
2   m1 = millis();
    timing = false;
    sum = sum + (m1-m0);

    count = count + 1;
    wait = 0;
```

```
    }
}
```


When the background turns green, click the mouse.
Reaction time is 0.0796 seconds.
Count is 5 You need 0 more trials.


When the background turns green, click the mouse.
Count is 3 You need 2 more trials.

Sketch 88: M/M/1 Queue—Time in Simulations

A *single-server queuing system*, or *M/M/1 queuing system*, is like a bank teller. Customers arrive at random times to the teller for service. The service requires some random amount of time, and then the customer departs. If the teller is busy with a customer when another one arrives, the new arrival waits in a *queue* or *waiting line*. When a departure occurs, the next customer in line is served; if there is no one in the queue, the teller (the *server*) becomes idle. This system resembles many that we see in real life: grocery checkouts, gas stations, waiting for a bus, even air traffic and ships arriving in a port.

This sketch simulates one server and one queue, but it can be adapted to do more, and it calculates the average queue length. The value in doing a simulation of such a system is in finding out how long the queue becomes, how much time a client spends in the queue, what percentage of the time the server is busy, and so on. All of this concerns costs and wasted time.

In the real world, time is continuous, but on the computer, that is not possible. Instead, the time of the simulation takes on discrete values: time = 0, time = 1.5, time = 3.99, and so on. When the simulation starts, we set the variable `time` to the time of the first arrival 1, and the time after that will be the time of the event being processed. This is known as a *next event* simulation: the current time in the simulation keeps jumping ahead to the time of the next event (arrival or departure) that occurs.

Arrivals happen at random times according to a particular probability distribution. When an arrival happens, it (the customer) enters the queue for the service (teller). If there is no queue, it gets served immediately; otherwise it must wait. When it gets to the server (the teller), it will require some random amount of time to be served, and then it will leave. Here are the steps to handle each event:

Arrival	Departure
1. Place the arrival into the queue 2.	1. Remove the job from the queue 3.
2. Is the server busy?	2. Queue empty?

3. If not, start the server.	3. If so, the server becomes idle.
4. Schedule the next arrival.	4. If not, schedule a departure.

The queue is an array holding numbers. Adding to the queue means placing a new value (the randomly generated service time for the job) at the end of the queue. When a value departs the queue, it means removing the first element and moving each consecutive value forward by one place. The function `into(t)` 5 inserts time `t` into the queue, whereas `out()` 6 removes the front element from the queue. The queue is empty (or the system is idle) if there is nothing in the queue 4.

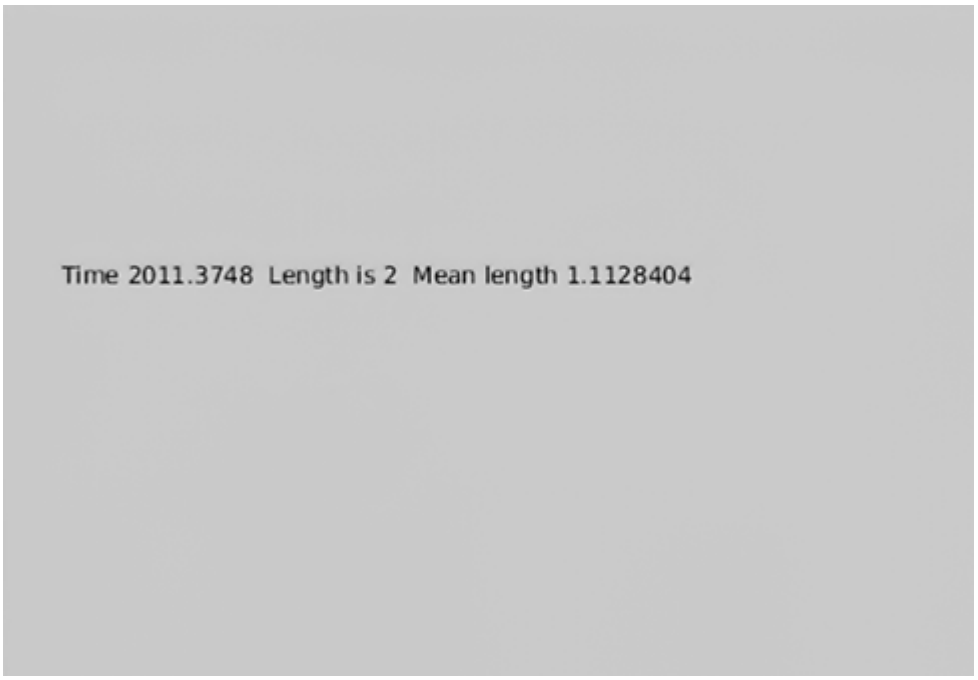
The statistical distribution of times between arrivals and departures is according to the *negative exponential distribution*. If the average time between arrivals is μ, then this will be the time of the next arrival in the simulation:

$$-\mu * \log(\texttt{random(1)})$$

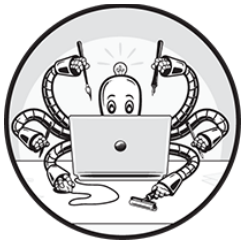
A similar situation exists for departures.

```
float queue[] = new float[200], end=2.0e3;
float miaTime=16.0, msTime=8.0,arrival,departure;
int Nqueue=0, nq;
float qsum=0.0, time=0.0;
void setup ()
{
  size (500, 350);
  fill(0);
1 arrival=-miaTime*log(random(1));
  departure = end*2;
}
void draw ()
{
  if (time>end) return;
  background(200);
  if (arrival<departure)
  {  // An arrival
2 into(time + -msTime*log(random(1)));
    arrival = time + -miaTime*log(random(1));
    if (departure>end) departure = queue[Nqueue-1];
    time = arrival;
  } else if (departure<end)
  { // A departure
3 out();
    time = departure;
4 if (Nqueue>0) departure = queue[Nqueue-1];
    else departure = end*2;
  }
  fill (0); text ("Time "+time+"  Length is "+Nqueue+"
Mean length "+(qsum/nq),30,145);
  nq += 1; qsum += Nqueue;
}

5 void into(float t)
  {
    queue[Nqueue] = t;
    Nqueue = Nqueue + 1;
  }

6 void out ()
  {
    if (Nqueue <= 0) return;
    for (int i=0; i<Nqueue-1; i++) queue[i] = queue[i+1];
```

```
  Nqueue = Nqueue - 1;
}
```



12
CREATING SIMULATIONS AND GAMES



Sketch 89: Predator-Prey Simulation

Picture rabbits and coyotes living together in their natural environment. They don't get along in the traditional sense: coyotes tend to eat rabbits when they can. Rabbits can breed very quickly, whereas coyotes cannot. And, of course, if rabbits are the only prey, then if the rabbits should all perish, the coyotes will also, soon after. This is a *predator-prey* relationship, and it can be simply modeled when there is only one species in each group.

Mathematically the predator-prey relationship is represented by a pair of *differential equations* (don't worry, very little actual math here) that look like this:

$$\frac{dx}{dt} = \alpha x - \beta xy$$

$$\frac{dy}{dt} = \delta xy - \gamma y$$

Here $\frac{dy}{dt}$ represents how fast the coyote population is growing, and $\frac{dx}{dt}$ is how fast the rabbit population is growing. The solution to these equations, known as the Lotka-Volterra equations, is not important. The program will simulate them. In these equations, the variables are as follows:

x: The number of rabbits (prey animals)

y: The number of coyotes (predator species)

α: The rate at which the rabbit population grows, unfettered

β: The rate at which prey and predators meet and a rabbit dies as a result

γ: The rate of death of predators due to natural causes or moving away

δ: The rate of growth of the predator population

The simulation will begin with specified values of the four variables α, β, γ, and δ, (`alpha`, `beta`, `gamma`, and `delta`) and will have known initial population sizes. It will then compute a new population each time `draw()`

executes, based on the preceding equations. This is the critical code for the rabbits 3:

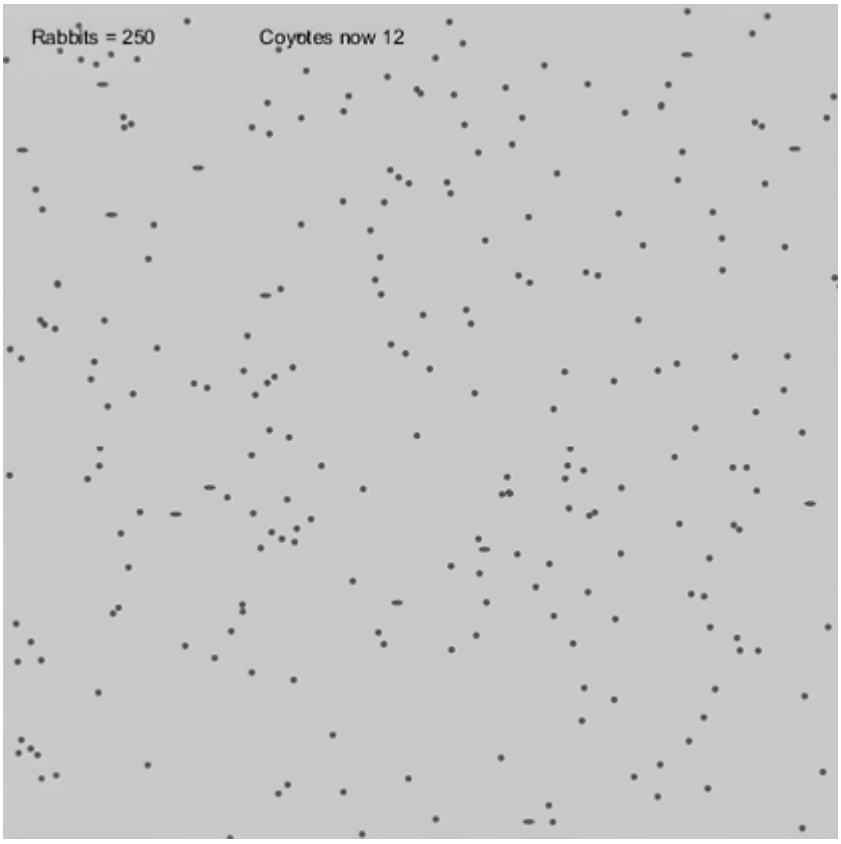
```
dr = alpha*Nrabbits - beta*Nrabbits*Ncoyotes;
Nrabbits = (int)(Nrabbits + dr);
```

And for the coyotes 4:

```
dc = delta*Nrabbits*Ncoyotes - gamma*Ncoyotes;
Ncoyotes = (int)(Ncoyotes + dc);
```

The populations are then rendered graphically in the window. We draw each rabbit as a green circle someplace on the screen (position does not matter) 1, and each coyote is a red circle 2. We can observe the relative populations increase and decrease as the predator population and the prey population change. If all of the prey die, the predators do too; if all of the predators die, the prey grows without limit.

```
int Nrabbits=190, Ncoyotes=16;
float time=0, alpha=.19, beta=0.008, gamma=0.15,
delta=.0005;

void setup ()
{
  size (500, 500);
  frameRate(2);
  noStroke();
}

void draw ()
{
  background(200);
  fill (0,200,0);
1 for (int i=0; i<Nrabbits; i++)
    ellipse (random(width),random(height),2,2);
  fill (200,0,0);
2 for (int i=0; i<Ncoyotes; i++)
    ellipse (random(width),random(height),4,2);
  prey();
  predator();
}

void prey ()
{
  float dr=0.0;
3 dr = alpha*Nrabbits - beta*Nrabbits*Ncoyotes;

  Nrabbits = (int)(Nrabbits + dr);
  if(Nrabbits<0) Nrabbits = 0;
  fill(0);
  text ("  Rabbits = "+Nrabbits, 10, 25);
}

void predator ()
{
  float dc=0.0;
4 dc = delta*Nrabbits*Ncoyotes - gamma*Ncoyotes;
  Ncoyotes = (int)(Ncoyotes + dc);
  fill (0);
  text (" Coyotes now "+Ncoyotes, 150, 25);
}
```


Rabbits = 250
Coyotes now 12

Sketch 90: Flocking Behavior

Craig Reynolds created a system he called Boids in 1986. It was a simulation of the behavior of birds when in a flock, or fish in a school. A *flock* is a collection of distinct objects of the same kind. They move, and wish to end up in the same place. They also don't wish to hit each other. The simulation involves knowing where each object is, how fast it is moving, and what direction it is traveling, and then updating the position of each object iteratively. Three rules make the objects a flock:

Separation Objects try to maintain a small distance between themselves and their neighbors. During each iteration, an object moves away (if possible) from any neighbor nearer than a distance *d*.

Alignment Objects try to match velocities with nearby objects. This keeps them moving in a similar direction and keeps them from spreading out too much. We compute a local velocity as seen from the object and then add a fraction of that to the object's velocity for the next iteration.

Cohesion An object will try to move toward the center of mass of its neighbors. This keeps them in a group. We find the center of mass, not including the current object itself, and move the object a fraction (1 percent to 3 percent) of the way toward that point.

Each position is stored as a vector (`PVector` object) that has an x and y component. The vector array `FlockV` stores the velocity of each object. The `draw()` function calls functions that move and then draw the flock. `match()` computes a new velocity, trying to match neighbors 2; `toCenter()` moves each object toward the center of mass 3; and `away()` attempts to keep the spacing between objects 4. We call each of the three for each object during each iteration. Each of these functions returns a value that we add to the object's position 1. Objects are small circles, and they will follow the mouse as we move it.

NOTE

Read more about Craig Reynolds at https://www.red3d.com/cwr/.

```
final int N = 42;
PVector flock[]=new PVector[N],
flockV[]=new PVector[N];

void setup ()
{
  size (500,400);
  for (int i=0; i<N; i++)
  {
   flock[i]= new PVector(random(width),
     random(height));
   flockV[i] = new PVector(0, 0, 0);
  }
  noStroke(); fill (255); frameRate(15);
}

void draw ()
{
   background (0);
   drawFlock();
   moveFlock();
 }

void drawFlock ()
{
   for (int i=0; i<N; i++)
         ellipse(flock[i].x, flock[i].y, 5,5);
}

void moveFlock ()
{
  PVector c = new PVector(0,0);
  for (int i=0; i<N; i++)
  {
    c = toCenter (i);
    c.add(away(i));
    c.add(match(i));
    flockV[i].add(c);
1 flock[i].add(flockV[i]);
    flockV[i].normalize();
    flockV[i].mult(6);
  }
}

2 PVector match (int b)
```

```
{
  PVector c=new PVector (0,0);
  for (int i=0; i<N; i++)
    if (i!=b) c.add(flockV[i]);
  c.div(N-1);
  c.sub(flockV[b]);
  c.div(8);
  return c;
}
```

3
```
PVector toCenter (int b)
{
  PVector c = new PVector(0,0,0);
  for (int i=0; i<N; i++)     // Find center of mass
        if (i!=b) c.add(flock[i]);
  c.div(N-1);
  c.sub(flock[b]);
  c.x -= 2*(flock[b].x-mouseX);
  c.y -= 2*(flock[b].y-mouseY);
  c.normalize();
  return c;
}
```

4
```
PVector away (int b)
{
  PVector r=new PVector (0,0),q=new PVector
    (flock[b].x, flock[b].y);
  for (int i=0; i<N; i++)
    if (flock[b].dist(flock[i]) < 100)
    {
        q.set(flock[b]);
        q.sub(flock[i]);
        r.sub(q);
    }
  r.normalize();
  r.mult(-.5);
  return r;
}
```

Sketch 91: Simulating the Aurora

Among objects that are difficult to render on a computer, the northern lights, or aurora, is near the top of the list. They flicker and roll, the colors change, the shape changes at various speeds, and they generally have no one specific shape. There have been efforts to draw them with more or less success; this sketch is one of those attempts.

There are many shapes that the aurora can take, and we will only attempt to draw one of those in this sketch: the typical curtain type, one example of which appears in *Figure 91-1*.

Figure 91-1: Red and green aurora

The sketch will make the color change slowly as a function of y position. Starting with a red value at the bottom of the auroral curtain, the hue will

increase in pixels above. Starting with a hue value of `h`=15, the hue increases according to this equation 2:

```
h = h + random(.87);
```

Thus, the hue increases at a random rate, but it always increases as the y-coordinate changes. At the very top of the curtain, the brightness will decrease, fading the color away.

Next, notice that the aurora appears to consist of vertical strokes and is banded horizontally. This is accomplished in the program by changing the saturation of the pixels periodically as a function of the x-coordinate 1. This is the code, where `i` is the horizontal position and `s` is the saturation:

```
if (i%3 == 0) s = 220+random(20)-10;
else if (i%2 == 0) s = 210+random(20)-10;
else s = 200+random(20)-10;
```

`i%3` is the remainder when `i` is divided by 3, so there is a somewhat random variation in the saturation, giving darker bands.

The curtain effect is accomplished using a sine function to locate the pixels vertically. For a basic coordinate (`i`, `j`) the actual pixel will be at `(i,j-bb*sin(a*i))`, where the parameters `a` and `bb` change by a small and random amount during each iteration 3. This makes the curtain move.

The visual effect is enhanced using a pair of images. An image of stars is used as a background, mimicking the night sky. We draw the aurora over this, followed by a foreground image of trees and shrubs. This image is a *stencil*, with black objects on a transparent background. The result is a pleasing interpretation of the aurora, although it is far from perfect, and much work could be done to improve the realism.

```
float a=.02, bb=10;
PImage foreground, background;
void setup ()
{
  foreground=loadImage("trees.gif");
  background=loadImage("stars.jpg");
  size (400, 224);
  colorMode(HSB);
}

void draw ()
{
  float h, s, b=250, dt=0;
  image (background, 0, 0);
  for (int i=0; i<390; i++)
  {
1 if (i%3 == 0) s = 220+random(20)-10;
    else if (i%2 == 0) s = 210+random(20)-10;
    else s = 200+random(20)-10;
    h = 15;
    for (int j=130; j>30; j--)
    {
      if (j<=100) dt = (100-j)*3;
      else dt = 0;
      stroke (h, s, b, 200-dt);
2   h = h + random(.87);
3   point (i,j-bb*sin(a*i));
    }
    a = a + random(0.001)-0.0005;
    bb = bb + random(1)-0.5;
    if (bb>16) bb = 15;
    if (bb<-10) bb = 0;
    if (a<-0.1 || a>0.1) a = 0;
  }
  image (foreground, 0, 0);
}
```

Sketch 92: A Dynamic Advertisement

On video screens all over the world, we see public advertising. In airports, shopping malls, and even schools, promotional material of all kinds is presented to a captive audience. Video is a convenient medium, since the price of large plasma and LCD screens has dropped below $10 per inch. Video is a more dynamic medium as well, allowing ads that move and multiple presentations in sequence, something that printed posters and billboards can't do.

The technology connected with video ads is well known too (Biteable Ad Maker, InVideo, even Adobe Premiere), and it's available on the computer desktop. This sketch is one example of a simple advertisement—for a Tex-Mex restaurant. It is loosely based on a collection of actual video presentations seen in airports in North America.

First we need a good image of the subject (the product): a burrito. The image used here is publicly available (*https://commons.wikimedia.org/wiki/File:Carne-asada-burrito.jpg*), but in general such images are professional photographs taken at high resolution. In the sketch, this image is 800×431 pixels. We reduce it to a smaller size, 770×401, or 30 pixels smaller in each dimension. This is so we can slowly move the image for a more dynamic presentation. We display the image using this statement 1, where `xoff` and `yoff` are pixel offsets for positioning the image before display:

```
image (ad1, xoff, yoff);
```

These offsets change in each frame by a small amount, up to a maximum of 30 pixels, at which time the displacement reverses direction 2:

```
xoff += dx; yoff += dy;
if (xoff <= -30 || xoff > 0) dx = -dx;
if (yoff <= -30 || yoff > 0) dy = -dy;
```

The values of `dx` and `dy` are very small, 0.05 and 0.03 respectively. They differ in value so that the image appears to move in a vaguely elliptical manner.

The text is displayed over the image in a fixed position, reinforcing the motion of the image. The text at the bottom remains the same throughout, but the text at the top changes. The implementation has two stages: if the variable `stage = 0`, we display the first text string (“It takes us hours to make it”) 3. After 850 frames (about 28 seconds) have passed, we increment the variable `stage`, and as a result we display the second string (“It takes you five minutes to eat it”) 4. After 900 more frames, `stage` becomes 1 again and the cycle repeats.

We could allow an arbitrary number of stages to allow for the presentation of multiple distinct messages and images, and in a random sequence.

```
PImage ad1;
float xoff=0, yoff=0;
float dx=-.05, dy=.03;
int stage = 0, count = 0;

void setup ()
{
  size(100,100);
  surface.setResizable(true);
  ad1 = loadImage ("burrito.jpg");
  surface.setSize (ad1.width-30, ad1.height-30);
}

void draw ()
{
  noStroke();
1 image (ad1, xoff, yoff);
  xoff += dx; yoff += dy;

2 if (xoff <= -30 || xoff > 0) dx = -dx;
  if (yoff <= -30 || yoff > 0) dy = -dy;
  fill (150, 150, 90);
  rect (0, height-50, width, 90);
  triangle (width-260, height, width, height-120,
width, height);
  fill(200);
  textSize(30);
  text ("Organically raised, no additives. Only the
best.", 40, height-20);

3 if (stage == 0)
  {
    fill (30);
    text ("It takes us hours to make it.", 40, 90);
    count += 1;
    if (count > 850)
    {   count = 0; stage = 1; }
  } else if (stage == 1)
4 {
    fill (30, 100, 100);
    text ("It takes you five minutes to eat it.", 40,
90);
    count += 1;
    if (count > 900)
    {  count = 0; stage = 0;
```

```
        }
    }
}
```


It takes us hours to make it.
Organically raised, no additives. Only the best.

Sketch 93: Nim

Nim is a game so old that its origins are lost to history. It was likely invented in China, and it is one of the oldest games known. It was also one of the first games to have a computer or electronic implementation and has been the frequent subject of assignments in computer programming classes. The game starts with three rows of objects, such as matches or coins, and there are a different number of objects in each row. A player may remove as many objects from one row as they choose, but they must remove at least one and must take them only from one row. Players take turns removing objects, and the player taking the final one is the winner.

This sketch will implement the game using 9, 7, and 5 coins, and it will play one side.

Setting the stage for the game play involves reading an image for the object, in this case a penny, and drawing the correct number of them in the window. When the human player clicks the mouse over one of the coins, that coin and all of the coins to the left are removed, and the remaining ones will move left. Then the computer will remove some coins.

There are three rows 100 pixels apart, so when the player clicks the mouse, the row index is simply `i = (mouseY/100)-1`. The number of coins removed is the number of coins to the left, which in the case of the sketch is `j = (mouseX-10)/45+1` because of how we drew them (45 pixels apart, indented 10 from the left) 1. An array named `val` contains the number of coins in each row, so when the user clicks the mouse, this is the action 2:

```
val[i] = val[i] - j;
```

This reduces the number of coins in row `(mouseY/100)-1` by `(mouseX-10)/45+1`.

Then it is the computer's turn. There is a strategy that will permit the computer to almost always win, as long as the user makes the first move. It involves computing a *parity* value and making a move to ensure that we maintain that parity value. Consider the initial state and the state after taking two coins from row 1:

	Before	After
Row 1	5 = 0 1 0 1	3 = 0 0 1 1
Row 2	7 = 0 1 1 1	7 = 0 1 1 1
Row 3	9 = 1 0 0 1	9 = 1 0 0 1
Parity	1 0 1 1	1 1 0 1

The parity is determined by looking at each digit in the binary representation of the values. In each column position, the parity bit for that column is 1 if the number of 1 bits in the column is odd and 0 if it is even. We can calculate this using the *exclusive OR* operator, which in Processing is "^", like so: `val[0]^val[1]^val[2]` 3.

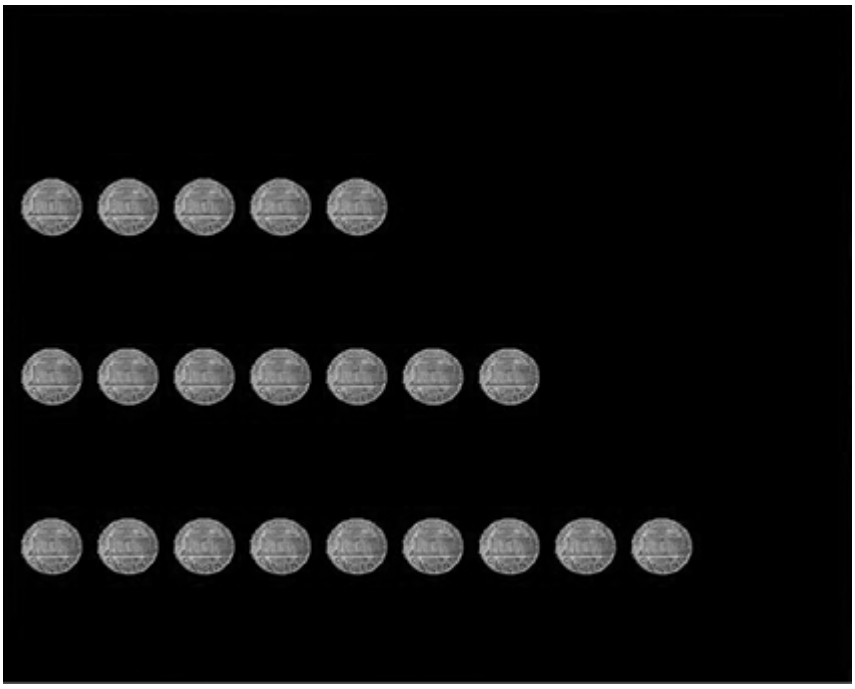
The strategy in Nim is to make a move that makes the parity value 0. It turns out that this is always possible; in the preceding situation, the computer might remove 5 coins from row 3 giving this state:

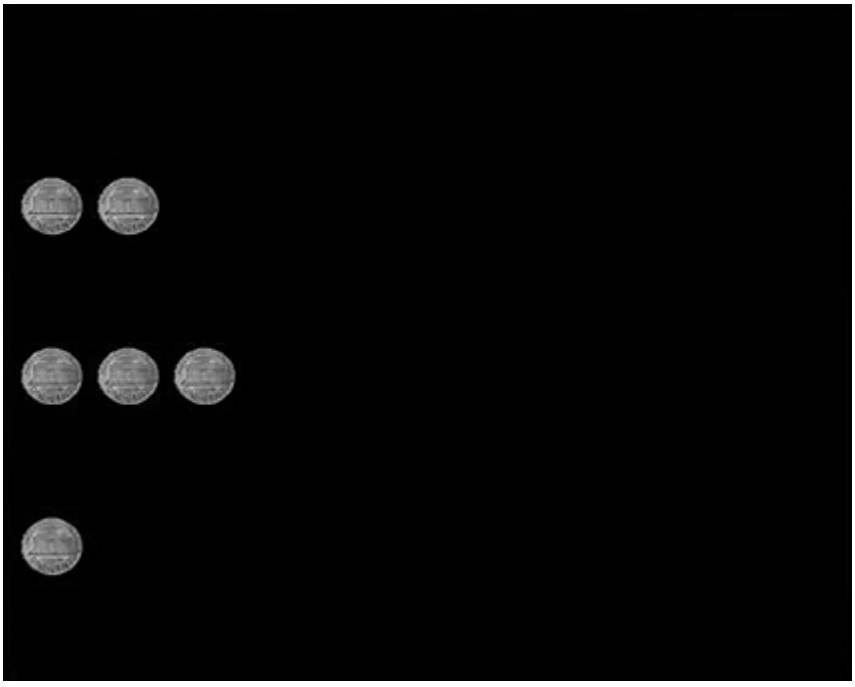
Row 1	3 = 0 0 1 1
Row 2	7 = 0 1 1 1
Row 3	4 = 0 1 0 0
Parity	0 0 0 0

This is what the sketch does after every move the player makes: computes the parity of all possible moves until it finds one with 0 parity 4.

```
PImage piece;
int val[] = {5, 7, 9}, i, j;

void setup ()
{
  size (500, 400);
  piece = loadImage ("coin.gif");
  frameRate (0.5);
}

void draw ()
{
  background (0);
  for (int j=0; j<3; j++)
    for (int i=0; i<val[j]; i=i+1)
1     image (piece, i*45+10, (j+1)*100);
}

void mouseReleased ()
{
  i = (mouseY/100)-1; j = (mouseX-10)/45+1;
  if (i<0) return;
2 if (j<=val[i]) val[i] = val[i] - j;
  draw(); move();
  if (val[0]+val[1]+val[2] == 0)
  { draw(); text ("Computer wins.",20,300);
    noLoop(); return;
  }
}

3 int eval() { return val[0]^val[1]^val[2]; }

void move()
{
  if (val[0]+val[1]+val[2] == 0)
  { text ("You Win.",20,300); noLoop(); return; }
  for(int i=0; i<3; i++)
    for (int j=1; j<=val[i]; j++)
    {
      val[i] = val[i] - j;
      if 4eval() == 0) return;
      val[i] = val[i] + j;
    }
  text ("Computer resigns- you win.", 20, 300);
```

```
  noLoop();
}
```





Sketch 94: Pathfinding

Pathfinding amounts to finding a route from one place to another in two or three dimensions. Potential routes could be blocked by walls, rivers, wires, or a host of other obstacles. Of course, it is the *best* route that is desired, where "best" can be based on many factors, such as physical distance, time, or cost. In circuit design, we use pathfinding to create a connection between circuit elements. In computer games, it finds a path to get a game object from one place to another. This sketch will implement a basic pathfinding method in two dimensions.

The method begins at some initial point, (x, y), and there is a destination or target point to be reached, (x_t, y_t). Each neighbor (x_n, y_n) of (x, y) is *marked* with its distance to (x, y). Then we look at the neighbors of those locations (x_n, y_n) and mark those locations with the distance to (x, y) by adding the distance to the neighbor (x_n, y_n) to the distance of (x_n, y_n) to (x, y). We keep repeating this until we find ourselves at the target pixel (x_t, y_t). Now we know the distance to the start pixel, and the best route can be traced backward following the connected locations having the smallest marked value. A neighbor must be an open space, not an obstacle, in order to be marked, so the route will never pass through obstacles.

The program begins by reading in an image on which obstacles appear in black and the background is white. The start and end positions of the path are specified in the program as x, y coordinates: `startx`, `starty`, and `endx`, `endy` (you can change these to find a different path).

Beginning at the start coordinates, we examine the immediately neighboring pixels 1. The neighbors of any pixel are the ones to the left, right, above, or below. The distance between pixels (x_0, y_0) and (x_1, y_1) is therefore $|x_0 - x_1| + |y_0 - y_1|$ and is an integer. The distance between the start pixel and its neighbors is 1. This way of measuring distance is called *Manhattan distance*; you could adapt the pathfinding method to use other distance measurements as well.

If one of the neighbors is the end of the path, the search is complete 2; otherwise we color the pixel a shade of cyan proportional to its distance

from the start point. We use the red component of the RGB color as the distance, so as the red increases, the color gets brighter. We could instead use a separate 2D array to store distances, especially if floating-point distances are required, such as when calculating Euclidean distances.

Next, we examine all pixels that have the red value 1 (those that are a distance 1 from the start) in the same way, and set their neighbors to 2. Then we set their neighbors to 3, and so on until we reach the end location.

At this point, the distance to the start point is *N*. To trace a route back to the start, we look for a neighbor of the end location that has a value of $N - 1$; any one will do. Mark that location as being on the route, and look for a neighbor of that location that has a value of $N - 2$; mark it and repeat. At any moment there will be many pixels having a particular value, but only ones connected to the path are interesting. The path is complete when we reach the start location. The `drawRoute()` function searches the neighbors of the end pixel for a neighbor with a value of *N*, marks that pixel with a specific color, and then recursively finds a neighbor of that pixel, marks it, and so on 3:

```
set (i,j,color(0,100,200));
drawRoute (i,j,n-1);
```

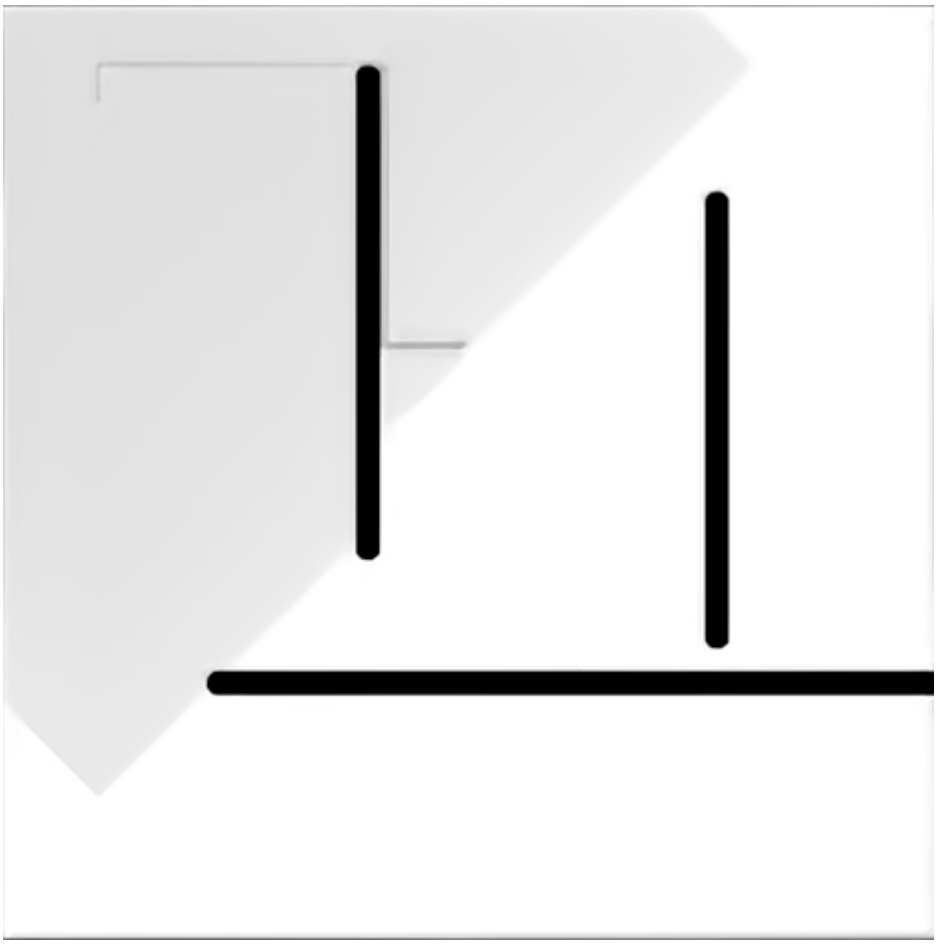
The result is that a path is drawn on the displayed image.

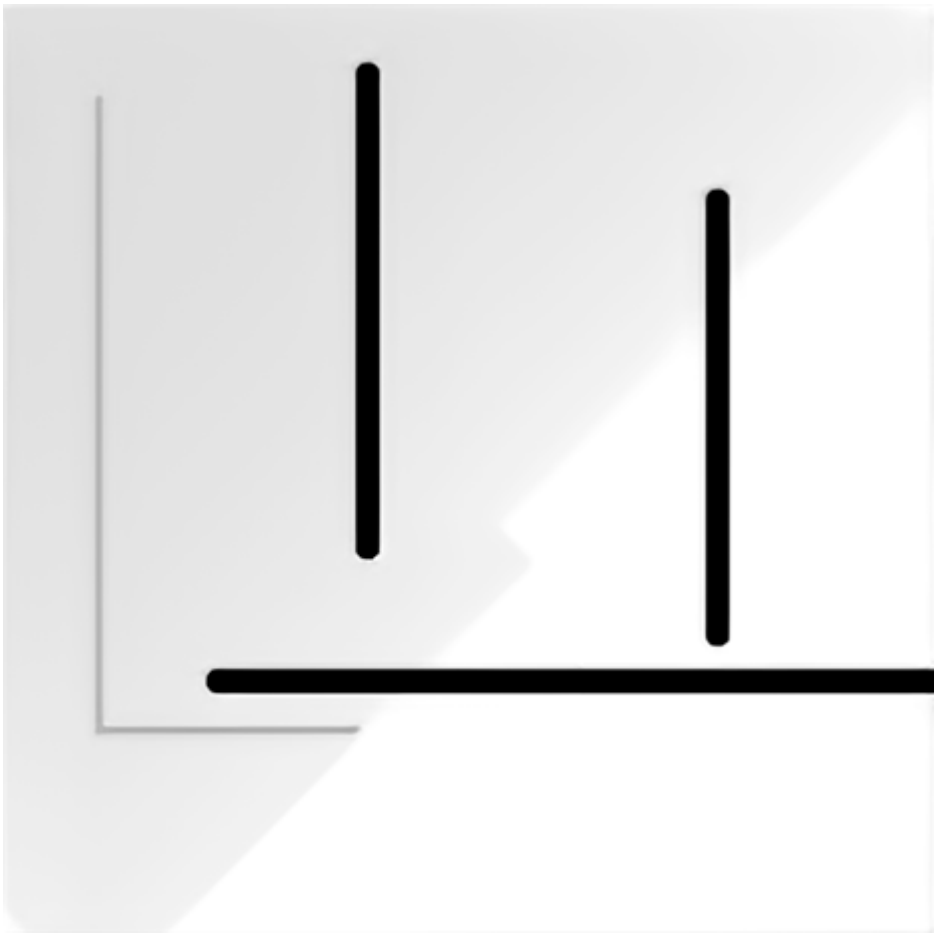
NOTE

The colors indicate distance in this sketch only to illustrate the method. This would not be done in an actual application. Also, this method is generally too slow for many applications, and better (and more complex) algorithms exist. The most commonly used method is the A algorithm.*

```
int startx=20, starty=20;
int endx=99, endy=73;
PImage back;
int stage = 1, n=1;

void setup ()
{
  size (200, 200);
  back = loadImage("plan.png");
  back.set(startx, starty, color(1,1,1));
  image(back, 0, 0);
}

void draw ()
{
  if (stage == 1) step();
  else
    if (drawRoute(endx, endy, n-1)) noLoop();
}
void step ()
{
  for (int i=0; i<width; i++)
    for (int j=0; j<height; j++)
      if (red(get(i,j)) == n)
      {
     1 if(red(get(i-1,j))>n) set(i-1,j,color(n+1, 255,
255));
        if(red(get(i+1,j))>n) set(i+1,j,color(n+1, 255,
255));
        if(red(get(i,j-1))>n) set(i,j-1,color(n+1, 255,
255));
        if(red(get(i,j+1))>n) set(i,j+1,color(n+1, 255,
255));
     2 if (i==endx && j==endy) { stage = 2; return; }
      }
  n=n+1;
}
boolean drawRoute (int x, int y, int n)
{
  for (int i=x-1; i<=x+1; i++)
    for (int j=y-1; j<=y+1; j++)
      if (red(get(i,j)) == n)
        {
       3 set (i,j,color(0,100,200));
          drawRoute (i,j,n-1);
```

```
            return true;
        }
    return false;
}
```





Sketch 95: Metaballs—A Lava Lamp

Figure 95-1: A lava lamp (shown in motion online: https://en.wikipedia.org/wiki/File:Lava_lamp_(oT)_07_ies.ogv)

This sketch represents an attempt to create a dynamic graphical simulation of a lava lamp, a popular item from the 1960s (see *Figure 95-1*). Most North Americans will recognize one, because they have undergone a resurgence in popularity, perhaps due to an interest in retro furnishings. The lamp is a glass container filled with oil. There is an incandescent lamp at the bottom and some colored wax. When the lamp heats up, it melts the wax, and globules slowly rise to the top, changing shape. Cooling globules fall to the bottom, creating a dynamic visual effect as the smooth wax shapes interact.

Each blob in the lamp seems to move on its own, so we'll use a collection of points with x, y coordinates that form the center of each blob, and these points can move about in a 2D area. We'll create the actual blob in an interesting way: each one is a 3D function, and we'll render a 2D view looking down at the part of the 3D blobs that have z (height) values greater

than a threshold, like an aerial view of an island sticking out of the water (*Figure 95-2*). These 3D functions are referred to as *isosurfaces* or *metaballs*.

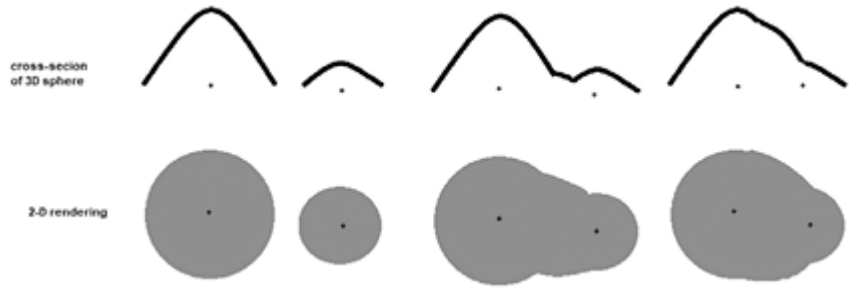


Figure 95-2: How the threshold slices the 3D function

As two metaballs get close to each other, the height of the area where they intersect is the sum of the two objects, and as they get nearer, this region will exceed the z threshold, so it will appear in the 2D rendering (*Figure 95-3*). This creates the illusion of wax blobs interacting.

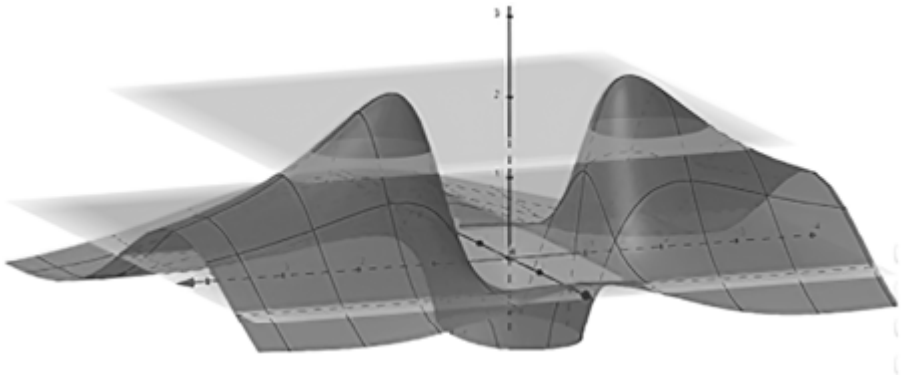


Figure 95-3: How the metaballs add up to produce a blob

We will use a simple function for the metaball: a sphere, as defined by the function named `equation()` 5. It defines a pixel value at any point `x`, `y` with respect to a sphere `k` at some other point, as follows:

```
radius[k] / sqrt( (xx-x[k])*(xx-x[k]) + (yy-y[k])*(yy-y[k]) )
);
```

This sketch has six spheres defined by arrays `x` and `y`, and they move as defined by arrays `dx` and `dy`. The `setup()` function initializes the six spheres. The first one is quite large, does not move, and lies at the bottom of

the region to simulate the large wax reservoir at the bottom of most lamps 1.

The `draw()` function calculates the sum of all spheres at any point in the drawing area 2. In many instances this will be zero, but as the balls get nearer, the sum increases and becomes visible if it is greater than the threshold `MINT`. Visible pixels will be drawn as green, and the background will be yellow. The balls are moved each iteration 3 and can change size randomly 4.

```
int maxMetaballs = 6;
float x[] = new float[maxMetaballs];
float y[] = new float[maxMetaballs];
float dx[] = new float[maxMetaballs];
float dy[] = new float[maxMetaballs];
float radius[] = new float[maxMetaballs];
float MINT = 1.4f, MAXT = 50f;

void setup ()
{
  size(500, 500);
1 for (int i=0; i<maxMetaballs; i=i+1) radius[i] = -1;
  x[0] = 250; y[0] = 850; radius[0] = 400;
  x[1] = 100; y[1] = 300; radius[1] = 20; dx[1] = 0;
dy[1] = -0.55;
  x[2] = 120; y[2] = 100; radius[2] = 30; dx[2] = 0.01;
dy[2] = 0.57;
  x[3] = 90;  y[3] = -330; radius[3] = 23; dx[3] =
-0.01; dy[3] = 0.32;
  x[4] = 320; y[4] = -650; radius[4] = 19; dx[4] =
0.01; dy[4] = 0.4;
  x[5] = 230; y[5] = -800; radius[5] = 24; dx[5] =
-0.01; dy[5] = 0.42;
}

void draw ()
{
  float sum;
  background(230, 220, 40, 90);
  for(int yy = 0; yy < height; yy++)
    for(int xx = 0; xx < width; xx++)
    {
      sum = 0;
   2 for(int i = 0; i < maxMetaballs && radius[i] > 0;
i++)
        sum += equation(xx,yy,i);
      if(sum >= MINT && sum <= MAXT)
        set(xx, yy, color(0,170,50,100));
    }
  for (int i=1; i<maxMetaballs; i=i+1)
  {
 3 if (radius[i] >0)
    {
      y[i] += dy[i];
      if (y[i]>height+6*radius[i] || (y[i]
```

```
<-3*radius[i])&&(dy[i]<0))
      { dy[i] = -dy[i]; x[i] += random (10)-5; }
    }
    ellipse (x[i], y[i], 3, 3);
  }
```

4
```
if (random(500)< 2) radius[(int)random (maxMetaballs)]
+= random (1)-0.5;
}
```

5
```
float equation(float xx, float yy, int k)
{ return (radius[k] / sqrt( (xx-x[k])*(xx-x[k]) + (yy-
y[k])*(yy-y[k]) ) ); }
```

Sketch 96: A Robot Arm

The word *robot* is often associated with a human-shaped mechanical device, but by far the most common robots are more restricted devices with a single function and a small range of motion. An example would be the kind of robot that welds joints or paints cars in factories. These frequently look like an arm, complete with multiple joints and some kind of tool at the end of the arm where the hand would be. This sketch allows a user to move a 2D simulated robot arm using key presses.

The robot in the simulation is typical of the type described, such as the commercially available PUMA. It consists of three linked segments, each of which can be rotated at the joint, as shown in *Figure 96-1*. The joints are the shoulder (jangle1), joined by the bicep to the elbow (jangle2), joined by the forearm to the wrist (jangle3), which connects to the hand. The user controls the angles subtended by the joints using keys: jangle1 is controlled by Q and E, jangle2 by A and D, and jangle3 by Z and C.

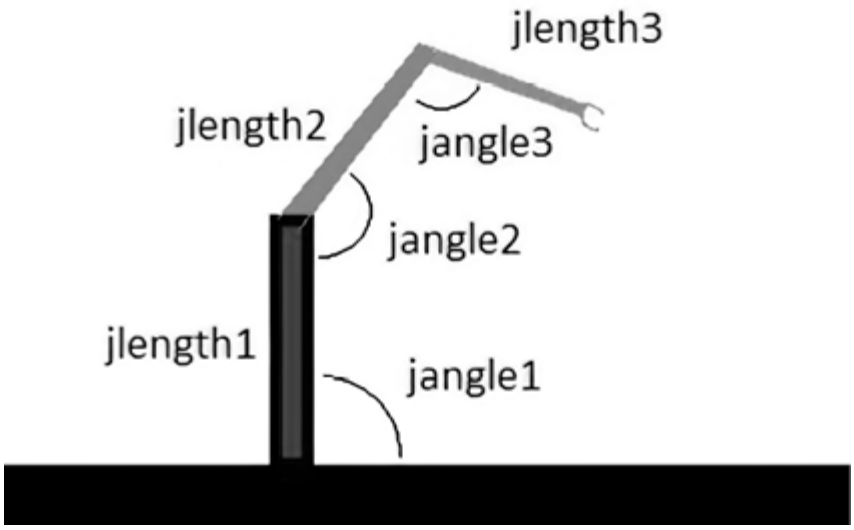


Figure 96-1: Three linked segments forming a robot arm

We'll represent each arm section by an image. The axis of rotation is not the upper-left corner or the center of the image but instead is a point in the image where the joint is connected to the previous one. The angle for any

joint is increased by pressing one key and decreased by another, but because they are connected to each other, the rotations must be relative to the previous section. The rotations are computed from the shoulder down to the hand. Then the hand is drawn at the final rotated location (all three rotations), the forearm is drawn at the location previous (two rotations), and finally the bicep after its rotation. This is accomplished using the Processing functions `pushMatrix()` and `popMatrix()`: the shoulder joint is rotated and then the state pushed 1; the elbow is rotated and pushed 2; the wrist is rotated and drawn. Then we restore the previous state, draw the bicep 3, and then perform one more restoration.

The images that represent the arm parts must be analyzed and the results coded into the program as coordinates. For example, consider the elbow: this is where the bicep (`armA` in the code) meets the forearm (`armB` in the code). The point where they meet has an offset from both images by a different amount, as shown in *Figure 96-2*. For the bicep, the point of contact is (167, 37) as measured from its upper left. The connection to the forearm is at (31, 25) relative to the forearm image, which is its axis of rotation as well. So, to rotate the forearm, we first translate it by (−31, −25) so that it appears to rotate about the correct place. The forearm must be translated when drawn so that the connection on the bicep at (167, 37) aligns with the connection on the forearm at (31, 25), so the next translation is (167 − 30, −(37 − 24)), or (137, −13). We reverse the sign on the y-coordinate because the direction of y is reversed from the usual y-axis in mathematics. The coordinates of each connection point are obtained from the images, and if they change, the points will have to be remeasured.

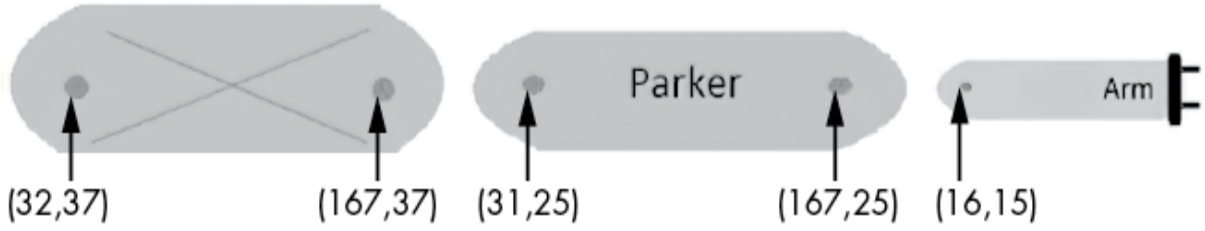
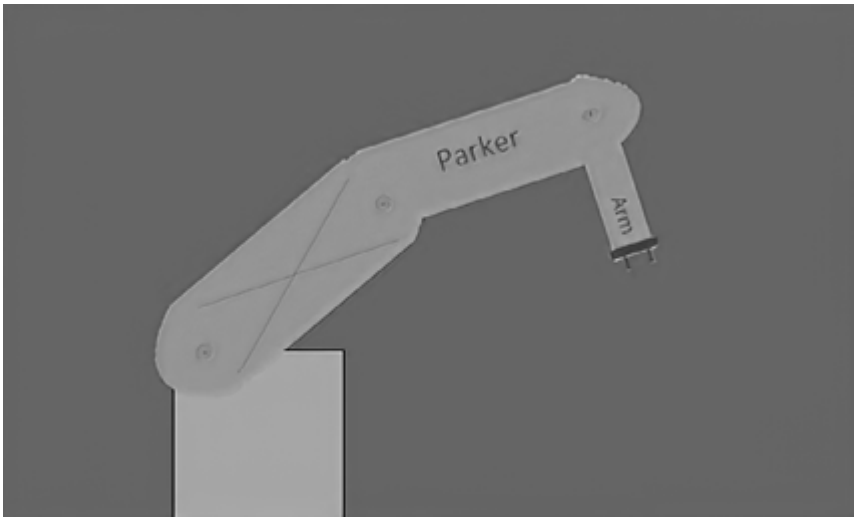
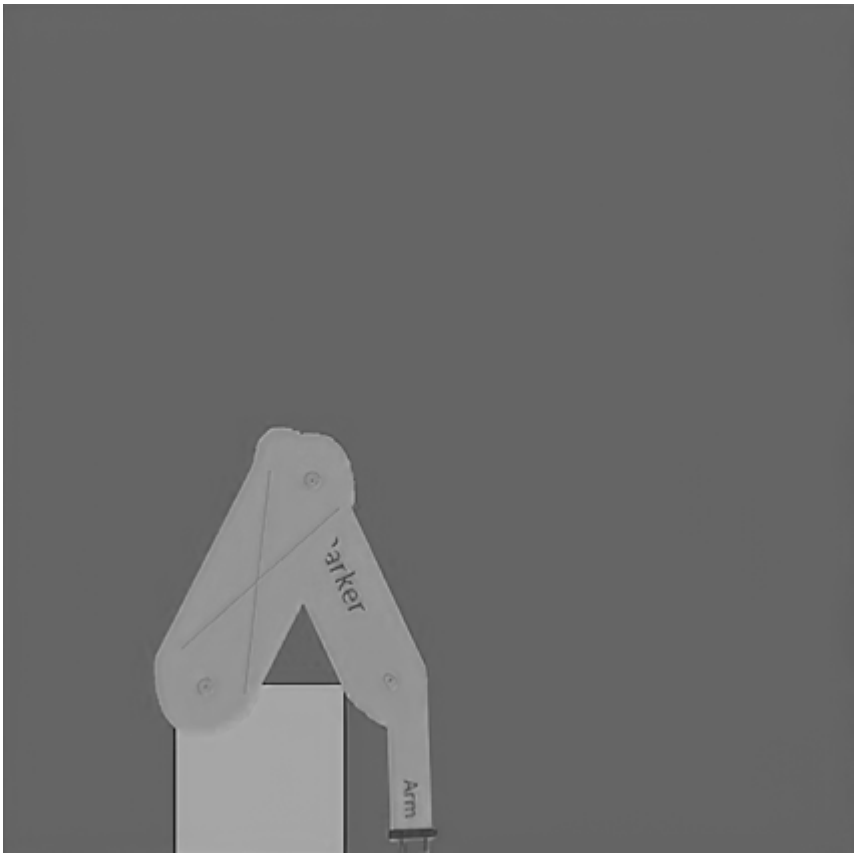


Figure 96-2: The connection points between the arm segments

```
float jangle1=-40, jangle2=22, jangle3=90;
PImage armA, armB, armC;
void setup ()
{
  size(500, 500);
  armA = loadImage ("robotA.gif");
  armB = loadImage ("robotB.gif");
  armC = loadImage ("robotC.gif");
  fill (200, 200, 110);
}

void draw ()
{
  background(100, 100, 100, 1);
  makeArm();
}

void makeArm ()
{
  translate (100, 400);
  rect (0, 0, 100, 100);
  translate (50, 40);
  pushMatrix();
1 translate (-31, -37); rotate (radians(jangle1));
  pushMatrix();
2 translate (137, -13); rotate(radians(jangle2));
  pushMatrix();
3 translate (137, -11); rotate (radians(jangle3));
  translate (-16, -15); image (armC, 0, 0);
  popMatrix();
  translate (-30, -25); image(armB, 0, 0);
  popMatrix();
  translate (-31, -37); image (armA, 0, 0);
  popMatrix();
}
void keyPressed ()
{
  if (key=='q') jangle1 = jangle1-1;
  if (key=='e') jangle1 = jangle1+1;
  if (key=='a') jangle2 = jangle2-1;
  if (key=='d') jangle2 = jangle2+1;
  if (key=='z') jangle3 = jangle3-1;
  if (key=='c') jangle3 = jangle3+1;
}
```


Parker
Arm


arker
Arm

Sketch 97: Lightning

Lightning moves quickly, randomly, and brightly. It would seem to be a difficult thing to capture in a computer graphic sense, and yet because it is in everyone's experience, there are situations where it would be important to be able to draw lightning. This sketch is a basic attempt to do that.

As was the situation with the auroral simulation of Sketch 91, there is a history and literature on the subject of drawing lightning, and a lot of it is based on an effort to model the physical process by which lightning occurs in the real world. This is too complex to reproduce in a small program, but some of the fruits of that work can be useful. Researchers have measured the angle between a streak of lightning and a branch, for instance (about 16 degrees), and also the likelihood of a branch.

This sketch will generate random lightning shapes as small, connected line segments. The length of the segment and an angle from the previous one will be random. A 2D array will hold the various segments for the main and branching parts. The main part is a sequence of line segments in the first part of the array: a line segment with starting point `x[0][i]` and `y[0][i]` connected to segment endpoint `x[0][i+1]`, `y[0][i+1]` 1. A branch will occur at random, with probability 0.11 2, and it will occupy another row of the array, the first branch starting at `x[1][0]`, `y[1][0]`, the second at `x[2][0]`, `y[2][0]`, and so on.

A branch can also terminate, with probability 0.2 3, but the main branch cannot. It will continue until it reaches a y value greater than 205, where it terminates 4. A new lightning stroke will occur later, at a random time and x location 2.

Each time `draw()` is called, a new section of each stroke is created and drawn, so the lightning is a dynamic display. It appears to descend from the top of the image down to the ground, or to the water in this case: a background image of a storm at sea is displayed, and the lightning appears to start in the clouds and strike the water.

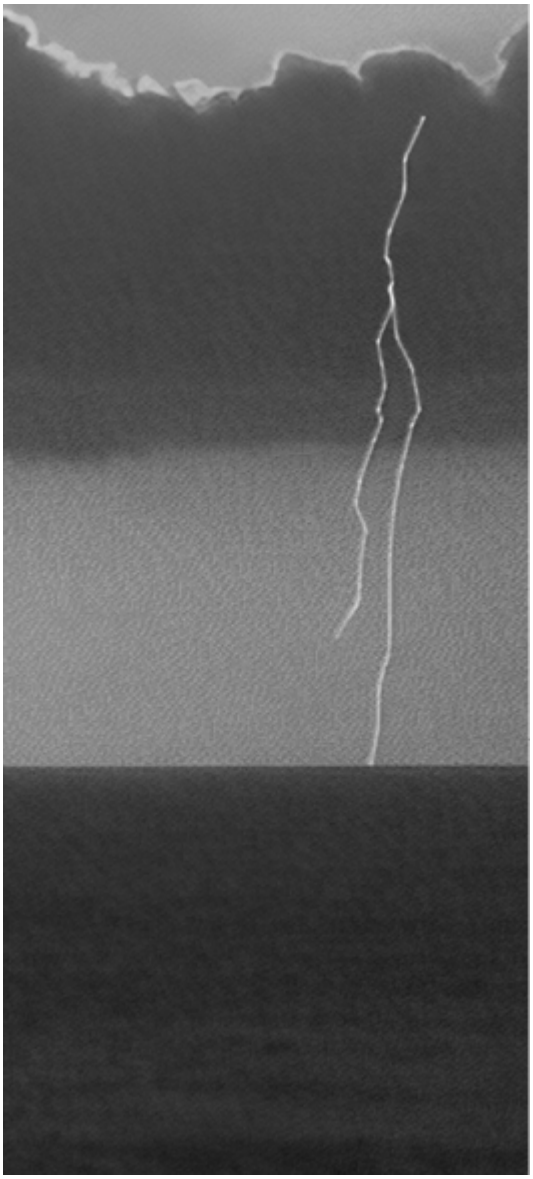
This scheme has some flaws. Sometimes, at random, strokes can appear in what a human would consider an unrealistic way. Branches can pass over

each other, sometimes more than once. This *could* occur in real life, but it does not happen very often. A lightning path usually has a surrounding glow, too, and this is missing from the sketch. Lightning is also a source of illumination and would alter the ambient light in the scene. It is possible to reproduce this effect, but using a static image as the background makes it difficult to change the illumination. Finally, we add to the lightning strokes iteratively and, once they are determined to a specific point, do not change. Lightning paths have been seen to move along their length, not just at the lower end, but the effect is subtle.

The code offers chances for experimentation. We can alter the probabilities of the creation of a new branch or of an existing one being deleted. The length of each section, now random between 0 and 12, and the angle, random between −30 and +30 degrees, can also have a significant visual impact on the result.

```
PImage back;
float x[][] = new float [40][600];
float y[][] = new float [40][600];
int n[] = new int [40], m=0, count=0;
void setup ()
{
  size(100,100);
  surface.setResizable(true);
  back = loadImage ("back.jpg");
  surface.setSize (back.width, back.height);
  m=1; n[0] = 1;
  x[0][0] = 30; x[0][1] = 35;
  y[0][0] = 43; y[0][1] = 47;
}
void draw ()
{
  float a, d;
  stroke (250,255, 250,128);
  image (back, 0, 0);
  if (count<=0)
    if (random(300) < 2)
    {
      m=1; x[0][0] = random(width); x[0][1] = x[0][0];
      y[0][0] = random(50)+12; y[0][1] = y[0][0]; n[0]
= 1; count=1;
    }
  for (int i=0; i<m; i++)
    for (int j=0; j<n[i]; j++)   // Draw existing
      if (x[i][0]>0) line (x[i][j], y[i][j], x[i]
[j+1],y[i][j+1]);
1 for (int i=0; i<m; i++)        // Grow existing
  {
    a = random (60)+60; d = random (12);
    if (x[i][0] < 0) continue;
    x[i][n[i]+1] = x[i][n[i]]+d*cos(radians(a));
    y[i][n[i]+1] = y[i][n[i]]+d*sin(radians(a));
    n[i] = n[i] + 1;
2 if (random (1)<0.11)          // New branch
    {
      a = random (60)+60; d = random (12);
      x[m][0] = x[i][n[i]-1]; y[m][0] = y[i][n[i]-1];
      x[m][1] = x[i][n[i]-1]; y[m][1] = y[i][n[i]-1];
      n[m] = 1; m = m + 1;
    }
3 if (i!=0 && random(1) < 0.20) for (int j=0; j<600;
```

```
j++) x[i][j] = -1;
  }
4 if (y[0][n[0]-1] > 205) { m=0; count = -1; }
}
```



Sketch 98: The Computer Game Breakout

The original game Breakout was designed and built in 1975 by legendary early builder of games Nolan Bushnell, Steve Wozniak (later of Apple fame), and Steve Bristow at Atari. In basic concept, it is a variation on Pong for one player, where the paddle is used to bounce a ball into bricks that vanish when hit. The original game has eight rows of rectangular bricks, with pairs of rows having the same color. The ball bounces off the sides and top of the game screen, and off a brick after it disappears, but is free to pass through the bottom. The player must move the paddle to hit the downward-moving ball to prevent it from disappearing. The player has three turns (that is, they can miss the ball three times) to clear the screen of bricks, and different colored bricks score a different number of points.

This sketch will implement a simplified version of the game. There are three rows of red bricks, all worth the same amount. There is no sound and no high score. The bricks are filled rectangles, 30 pixels by 15 pixels, and the ball is simply a small circle, 3 pixels across. A 2D array, `exists[][]`, is used to keep track of which bricks have been eliminated, and the brick in row `i` column `j` will be drawn if `exists[i][j]` is true. Drawing the bricks is therefore simple 1:

```
for (int i=0; i<Ncols; i++)  // Draw all bricks
  for (int j=0; j<Nrows; j++)
    if (exists[i][j]) rect (i*30+20, j*15+30, 30, 15);
```

The ball is drawn at location `(x, y)` and is moved during each frame by an amount `(dx, dy)` 2. The paddle is simply a line drawn centered at `(px, py)`. Typing the A key moves the paddle left by 10 pixels (`px=px-10`), and typing D moves it right by the same amount. If the ball moves past the coordinate `py` (= 300) and its `x` value is between `px` − 30 and `px` + 30, then the ball changes y-direction (`dy=-dy`) and it appears to bounce. The ball also bounces off the top of the screen (`y==0`) and the sides (`x<0` or `x>width`).

We test the ball against each brick for a collision during each frame; this is done using the absolute coordinates of each brick. If the brick at (i, j) exists, then these are the brick boundaries:

Dimension	Coordinate value	Boundary	Coordinate value	Boundary
X	i*30+20	Left edge	i*30+50	Right edge
Y	j*15+30	Top edge	j*15+45	Bottom edge

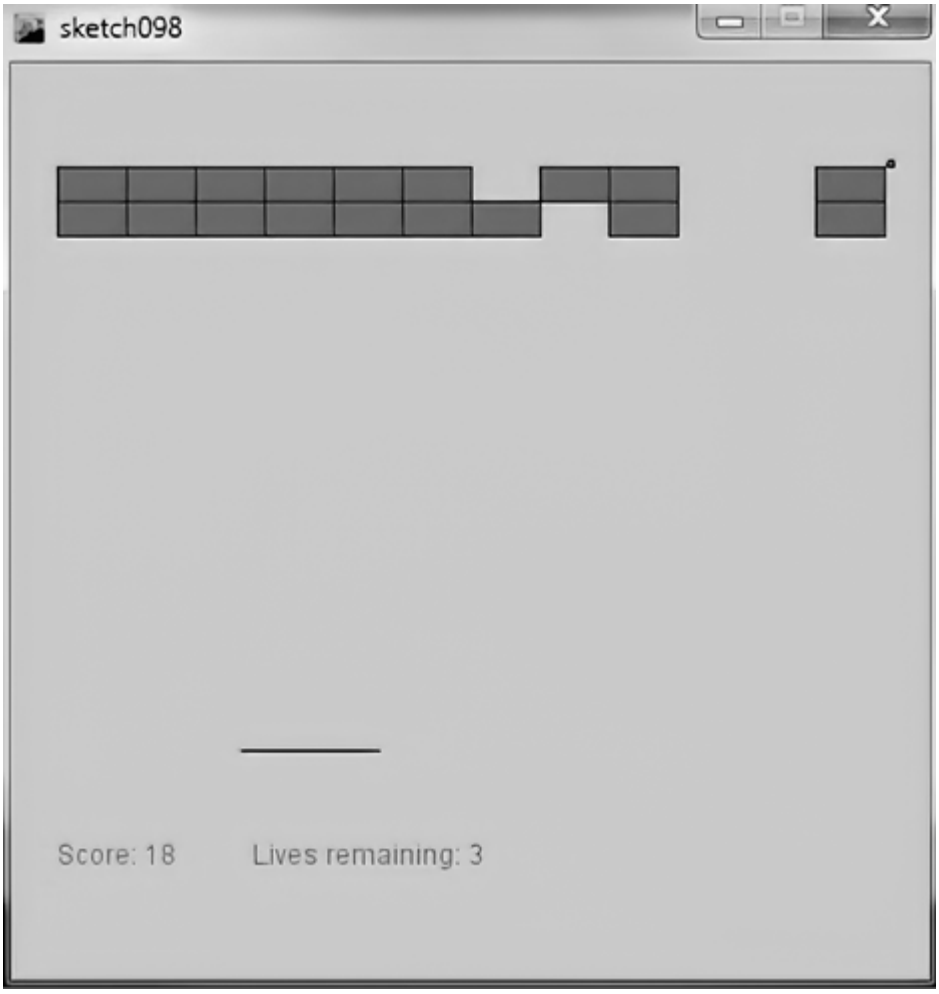
Simply check the ball's coordinates against these values for every brick, and bounce if the ball is inside the brick 3, at the same time setting `exists[i][j]` to `false` and increasing the `score`.

After the ball falls past the bottom, we decrement `life` and the ball is redrawn at a random `x` location at `y` value 150. The game is over when either the value of `life` is 0 or the `score` is the maximum of 36.

This simple version has flaws. The bounce off of the bricks is not dependent on the side of the brick that was hit; the y-direction of the ball is always changed. The bounce off the paddle is always the same, no matter where the point of impact.

```
final int Ncols = 12, Nrows = 3;
boolean exists [][] = new boolean[Ncols][Nrows];
int x, y, dx, dy, px, py, score = 0, life=5;
int direction = 0;
void setup ()
{
  size (400, 400);  fill (200, 40, 40);
  for (int i=0; i<Ncols; i++)  // All bricks exist
    for (int j=0; j<Nrows; j++) exists[i][j] = true;
  x = (int)random(300)+100; y = 150;  // Random X
    start
  dx=1; dy=-2; px = 120; py = 300; // Initial paddle
    position
}

void draw ()
{
  background(200);
1 for (int i=0; i<Ncols; i++)  // Draw all bricks
    for (int j=0; j<Nrows; j++)
      if (exists[i][j]) rect (i*30+20, j*15+30, 30,
        15);
  line (px-30, py, px+30, py);    // Paddle
  ellipse (x, y, 3, 3); move();   // Ball
  text ("Score: "+score+"           Lives remaining:
    "+life,20,350);
  if (score>=36) text (" YOU WIN!",100, 300);
  else if (life <= 0)
  {
    text (" YOU LOSE!",100, 300);
    noLoop(); // Win or lose.
  }
}
void keyPressed ()  // Use the 'a' and 'd' keys to
  move the paddle
{
  if (key == 'a' && px > 30) direction = -4;
  else if (key == 'd' && px<width-30) direction = 4;
  else direction = 0;
}

void keyReleased ()
{
  direction = 0;
}
```

```
2 void move ()  // Move the ball
  {
    x = x + dx; y = y + dy;  // Basic move
    px = px + direction;
    if (x<2|| x>width-2) dx = -dx;  // X bounce?
    if (y>=py-1&&y<=py+1 && (x>=px-30&&x<=px+30)) dy =
      -dy;  // Paddle bounce
    if (y<0) dy = -dy;  //  Y bounce top
3 for (int i=0; i<Ncols; i++)  // Ball hits a brick
      for (int j=0; j<Nrows; j++)
        if(exists[i][j] && x>=i*30+20 && y>=j*15+30 &&
          x<=i*30+50 && y<=j*15+45)
        {
          exists[i][j] = false;  // Brick is destroyed
          dy = -dy; score++;     // Ball bounces, score
        }
    if (y>height)  // Ball through the bottom
    {
      if (score < 36) life--;  y=150;  // Lose a life,
        restart the ball
      x = (int)random(width-100+50);
    }
  }
```


sketch098
Score: 18
Lives remaining: 3

Sketch 99: Midpoint Displacement—Simulating Terrain

This sketch will generate a pseudo-random terrain profile, with a darkening sky and twinkling stars. The heart of this sketch is the *midpoint displacement* method for generating terrain, and while this example is only two-dimensional, it illustrates the more general algorithm pretty well.

The method starts with a line, which in the case of this sketch is a line that runs horizontally across the entire image. Next we select the midpoint of the line, displace it by a random value between `dy` and `-dy`, and create two lines as in *Figure 99-1*.

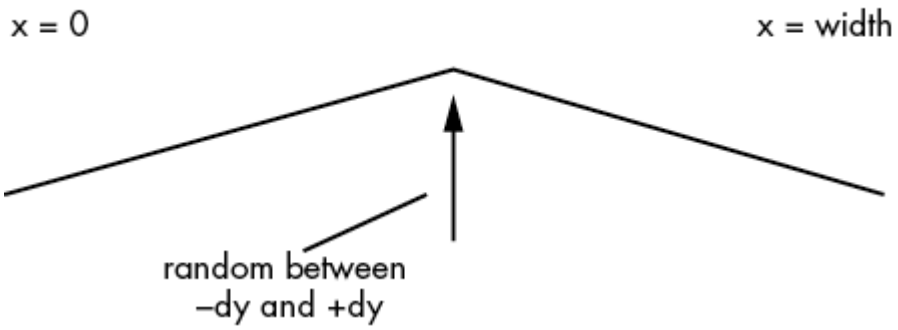


Figure 99-1: Splitting a line

Then we do the same thing again with the two lines just created, except we decrease the value of `dy`. The result is four lines. Each time we generate a new line pair, the resulting segments can be split again using a smaller `dy` value until some termination criterion is reached. In the sketch, the initial value of `dy` is 75, and the splitting process ceases when it becomes smaller than 2.

The splitting process is accomplished by a recursive procedure, `md()` 4:

```
void md (float x0, float y0, float x1, float y1, float dy)
```

Here, (`x0`, `y0`) and (`x1`, `y1`) are the line segment endpoints, and `dy` is the maximum value of the random height change. This procedure finds the

midpoint and calls itself twice, passing the left and right halves of the line and a smaller `dy`. The process continues as illustrated in *Figure 99-2* until we reach the minimum `dy` value.

The line endpoints are then saved in an array pair, `lx[]` and `ly[]`. We don't actually draw the line segments but make a filled region by drawing a line from each endpoint down to the bottom of the window that is as thick as half of the line segment x width 3. The result is a horizon line with a convincing random nature.

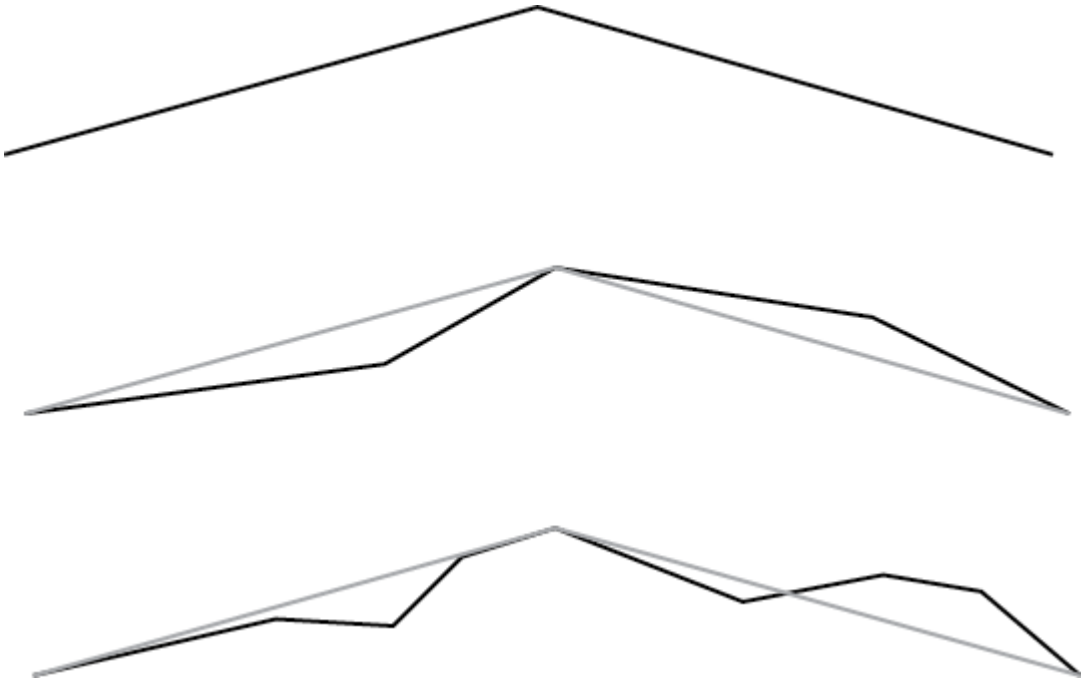


Figure 99-2: Multiple recursive splits create a realistic horizon.

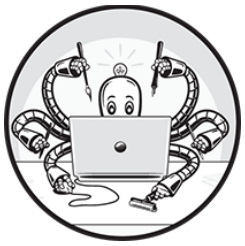
The sky is a set of horizontal lines starting at a color of (50, 50, 240) and decreasing by 1 in the blue value for every two lines drawn 1. This produces a nice deep blue gradient in the sky.

The stars are simply small circles drawn in random locations, but they must appear in the same place during each frame, so the arrays `starx[]` and `stary[]` hold their locations. They don't exactly twinkle, but we draw them with a probability of 99 percent so that once in a while one of the stars is not drawn during a particular frame 2. During any one frame it is likely that at least one star has gone dark. The overall effect is that of an evening sky and a rural landscape.

```
int starx[]=new int[50], stary[]=new int[50];
float lx[]=new float[1000], ly[]=new float[1000];
int n=0;
void setup ()
{
  size(640, 480);
  fill (255);
  for (int i=0; i<50; i++)
  {
    starx[i] = (int)random(width);
    stary[i] = (int)random(height/2);
  }
  md(0,300,width, 300, 75.0);
}

void draw ()
{
  for (int i=0; i<height; i++)
  {
1 stroke(50, 50, (float)(height-i)/2);
    line (0,i, width, i);
  }
  noStroke();
2 for (int i=0; i<50; i++)
    if (random(1)<0.99) ellipse (starx[i], stary[i], 2,
2);
  stroke(0); strokeWeight(lx[1]-lx[0]+1);
3 for (int i=0; i<n; i=i+1)
    line (lx[i],ly[i], lx[i],height);
}

4 void md (float x0, float y0, float x1, float y1, float
dy)
{
  if (dy < 2)
  {
    lx[n] = x0; ly[n] = y0;
    lx[n+1] = x1; ly[n+1] = y1;
    n = n + 2;
  } else
    {
      float d = random(dy+dy)-dy;
      md (x0,y0, x0+(x1-x0)/2, y0+(y1-y0)/2-d, .6*dy);
      md (x0+(x1-x0)/2, y0+(y1-y0)/2-d, x1, y1, .6*dy);
```

```
        }
}
```

13
MAKING YOUR WORK PUBLIC



Sketch 100: Processing on the Web

Processing sketches can usually execute within a browser, requiring little to no modification to make dynamic and interactive web objects. The system that allows this is *Processing.js*; it converts the Processing sketch into JavaScript before running it and displays the result in an HTML5 *canvas*.

There are four steps in running a sketch from the web:

1. Download Processing.js. This means going to a site like *https://processingjs.org/download/* and getting the files *processing.js* and *processing.min.js*.
2. Create the Processing sketch. We'll use Sketch 91, the aurora simulation, as our example. This sketch will be named *sketch100.pde*.
3. Create a web page within which you'll embed the sketch. It must load *processing.min.js* as a script in the header of the page 2:

```
<script src="processing.min.js"></script>
```

4. Create a canvas, specifying *sketch100.pde* as a data processing source 3:

```
<canvas data-processing-sources="sketch100.pde"> </canvas>
```

This will only work properly on a web server, so you need to upload all files to a server and display the page from the internet, or install a server on your computer.

All three files—the HTML source, the sketch, and *processing.min.js*—should be in the same directory on the web server. When the page is loaded, the sketch should run and display results in the canvas.

There may be some other issues depending on the sketch. First, if the sketch uses images, these must be preloaded so that their size and other properties are available when the sketch runs. A `preload` directive must appear in a comment at the beginning of the sketch. For example, in this case, the files `trees.gif` and `stars.jpg` are used 1:

```
/* @pjs preload="trees.gif, stars.jpg"; */
```

Next, be careful if the sketch uses integers. The Processing code is translated into JavaScript, which has no integer type. Integers will become floating-point values. Any program that depends on integer arithmetic (like 5/2 = 2) will not work properly.

Any program that requires a Java library won't work either. Minim is a Java library, and so are the video classes. There are JavaScript variations of these, but using them will require learning how JavaScript works and how to access JavaScript from Processing and vice versa.

The HTML code for the web page follows the code for the sketch on the next page.

```
/* @pjs preload="trees.gif, stars.jpg"; */
float a=.02, bb=10;
PImage foreground, background;

void setup ()
{
  foreground=loadImage("trees.gif");
  background=loadImage("stars.jpg");
  size (400, 224);
  colorMode(HSB);
}

void draw ()
{
  float h, s, b=250, dt=0;
  image (background, 0, 0);
  for (int i=0; i<390; i++)
  {
    if (i%3 == 0) s = 220+random(20)-10;
    else if (i%2 == 0) s = 210+random(20)-10;
    else s = 200+random(20)-10;
    h = 15;
    for (int j=130; j>30; j--)
    {
      if (j<=100) dt = (100-j)*3;
      else dt = 0;
      stroke (h, s, b, 200-dt);
      h = h + random(.87);
      point (i,j-bb*sin(a*i));
    }
    a = a + random(0.001)-0.0005;
    bb = bb + random(1)-0.5;
    if (bb>16) bb = 15;
    if (bb<-10) bb = 0;
    if (a<-0.1 || a>0.1) a = 0;
  }
  image (foreground, 0, 0);
}
<!DOCTYPE html>
<html>
<head>
<title> Sketch 100: Processing on the Web</title>
<script src="processing.min.js"></script>
</head>
<body bgcolor=aa99aa>
```

```
<br><br>
<h1> Sketch 91: Simulating the Aurora
 </h1>
This is sketch 91 adapted for the web. <br>
Among objects that are difficult to render on a
computer, the northern lights or aurora is near the top
of the list. They flicker and roll, the colors change,
the shape changes at various speeds, and they generally
have no one specific shape. There have been efforts to
draw them with more or less success; this sketch is one
of those attempts.<br>
<center>
<canvas data-processing-sources="sketch100.pde">
</canvas>
</center>
<br>
There are many shapes that the aurora can take, and
this sketch will only attempt to draw one of those: the
typical curtain type.<br>
The sketch will make the color change slowly as a
function of Y position. Starting with a red value at
the bottom of the auroral curtain, the hue will
increase in pixels above. Starting at a hue value of
h=15, the hue increases according to:<br>
 <b> h = h + random(.87)  </b><br>
So that the hue increases at a random rate, but always
increases, as the y coordinate changes. At the very top
of the curtain the brightness will decrease also,
fading the color away.<br>
Next, notice that the aurora appears to consist of
vertical strokes and is banded horizontally. This is
accomplished in the program by changing the saturation
of the pixels periodically as a function of X
coordinate. The code is:<br><b>

if (i%3 == 0) s=220+random(20)-10; <br>
else if (i%2 == 0) s = 210+random(20)-10; <br>
else s = 200+random(20)-10; <br></b>

where i is the horizontal position and s is the
saturation. i%3 is the remainder when i is divided by
3, so there is a somewhat random variation in the
saturation, giving darker bands.<br>
The curtain effect is accomplished using a sine
function to locate the pixels vertically. For a basic
coordinate (i,j) the actual pixel will be at (i,j-
bb*sin(a*i)) where the parameters a and bb change by a
```

```
small and random amount during each iteration. This
makes the curtain move.<br>

</body>
</html>
```