Color data

The `color` data type is used to store colors in a program, and the `color()` function is used to assign a `color` variable. The `color()` function can create gray values, gray values with transparency, color values, and color values with transparency. Variables of the `color` data type can store all of these configurations:

```
color(gray)
color(gray, alpha)
color(value1, value2, value3)
color(value1, value2, value3, alpha)
```

The parameters of the `color()` function define a color. The *gray* parameter used alone or with *alpha* defines tones ranging from white to black. The *alpha* parameter defines transparency with values ranging from 0 (transparent) to 255 (opaque). The *value1*, *value2*, and *value3* parameters define values for the different components. Variables of the `color` data type are defined and assigned in the same way as the `int` and `float` data types discussed in Data 1 (p. 37).

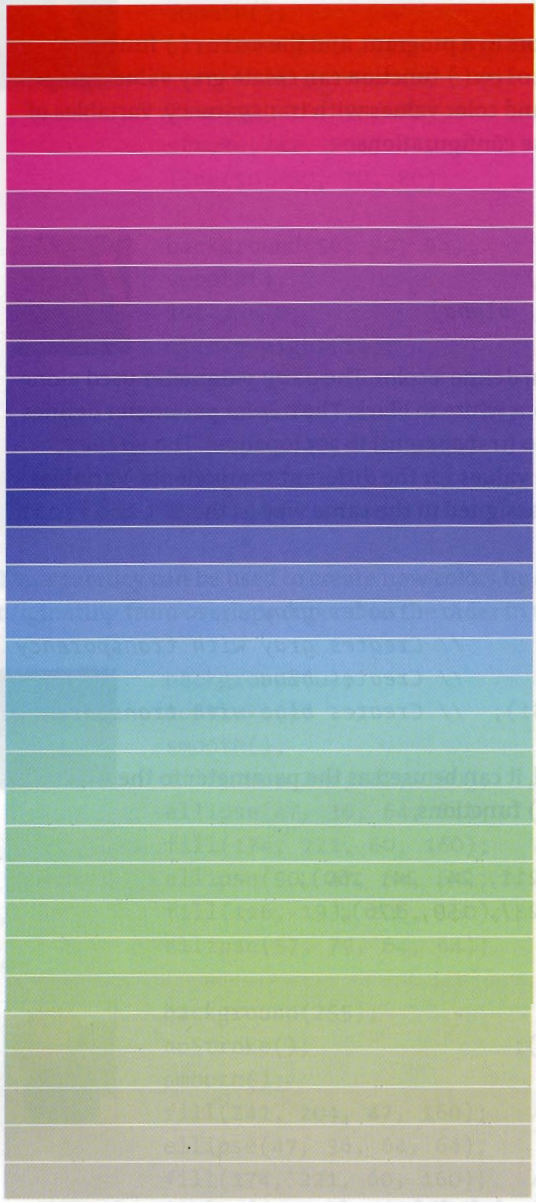
```
color c1 = color(51);              // Creates gray                          9-11
color c2 = color(51, 204);         // Creates gray with transparency
color c3 = color(51, 102, 153);    // Creates blue
color c4 = color(51, 102, 153, 51); // Creates blue with transparency
```

After a `color` variable has been defined, it can be used as the parameter to the `background()`, `fill()`, and `stroke()` functions.

```
color ruby = color(211, 24, 24, 160);                                        9-12
color pink = color(237, 159, 176);
background(pink);
noStroke();
fill(ruby);
rect(35, 0, 20, 100);
```

RGB, HSB

Processing uses the RGB color model as its default for working with color, but the HSB specification can be used instead to define colors in terms of their hue, saturation, and brightness. The hue of a color is what most people normally think of as the color name: yellow, red, blue, orange, green, violet. A pure hue is an undiluted color at its most intense. The saturation is the degree of purity in a color. It is the continuum from the undiluted, pure hue to its most diluted and dull. The brightness of a color is its relation to light and dark.



RGB			HSB			HEX
255	0	0	360	100	100	#FF0000
252	9	45	351	96	99	#FC0A2E
249	16	85	342	93	98	#F91157
249	23	126	332	90	98	#F91881
246	31	160	323	87	97	#F720A4
244	38	192	314	84	96	#F427C4
244	45	226	304	81	96	#F42EE7
226	51	237	295	78	95	#E235F2
196	58	237	285	75	95	#C43CF2
171	67	234	276	71	94	#AB45EF
148	73	232	267	68	93	#944BED
126	81	232	257	65	93	#7E53ED
108	87	229	248	62	92	#6C59EA
95	95	227	239	59	91	#5F61E8
102	122	227	229	56	91	#667DE8
107	145	224	220	53	90	#6B94E5
114	168	224	210	50	90	#72ACE5
122	186	221	201	46	89	#7ABEE2
127	200	219	192	43	88	#7FCDE0
134	216	219	182	40	88	#86DDE0
139	216	207	173	37	87	#8BDDD4
144	214	195	164	34	86	#90DBC7
151	214	185	154	31	86	#97DBBD
156	211	177	145	28	85	#9CD8B5
162	211	172	135	25	85	#A2D8B0
169	209	169	126	21	84	#A9D6AD
175	206	169	117	18	83	#AFD3AD
185	206	175	107	15	83	#BAD3B3
192	204	180	98	12	82	#C1D1B8
197	201	183	89	9	81	#C5CEBB
202	201	190	79	6	81	#CACEC2
202	200	193	70	3	80	#CACCC5

Color by numbers

Every color within a program is set by numbers, and there are more than 16 million colors to choose from. This diagram presents a few colors and their corresponding numbers for the RGB and HSB color models. The RGB column is in relation to `colorMode(RGB, 255)` *and the HSB column is in relation to* `colorMode(HSB, 360, 100, 100)`.

The `colorMode()` function sets the color space for a program:

```
colorMode(mode)
colorMode(mode, range)
colorMode(mode, range1, range2, range3)
```

The parameters to `colorMode()` change the way Processing interprets color data. The *mode* parameter can be either RGB or HSB. The range parameters allow Processing to use different values than the default of 0 to 255. A range of values frequently used in computer graphics is between 0.0 and 1.0. Either a single range parameter sets the range for all the color components, or the *range1*, *range2*, and *range3* parameters set the range for each—either red, green, blue or hue, saturation, brightness, depending on the value of the *mode* parameter.

```
// Set the range for the red, green, and blue values from 0.0 to 1.0
colorMode(RGB, 1.0);
```

9-13

A useful setting for HSB mode is to set the *range1*, *range2*, and *range3* parameters respectively to 360, 100, and 100. The hue values from 0 to 360 are the degrees around the color wheel, and the saturation and brightness values from 0 to 100 are percentages. This setting matches the values used in many color selectors and therefore makes it easy to transfer color data between other programs and Processing:

```
// Set the range for the hue to values from 0 to 360 and the
// saturation and brightness to values between 0 and 100
colorMode(HSB, 360, 100, 100);
```

9-14

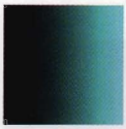
The following examples reveal the differences between hue, saturation, and brightness.

```
// Change the hue, saturation and brightness constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(i*2.5, 255, 255);
  line(i, 0, i, 100);
}
```

9-15

```
// Change the saturation, hue and brightness constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(132, i*2.5, 204);
  line(i, 0, i, 100);
}
```

9-16



9-17

```
// Change the brightness, hue and saturation constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  stroke(132, 108, i*2.5);
  line(i, 0, i, 100);
}
```

9-18

```
// Change the saturation and brightness, hue constant
colorMode(HSB);
for (int i = 0; i < 100; i++) {
  for (int j = 0; j < 100; j++) {
    stroke(132, j*2.5, i*2.5);
    point(i, j);
  }
}
```

It's easy to make smooth transitions between colors by changing the values used for *color()*, *fill()*, and *stroke()*. The HSB model has an enormous advantages over the RGB model when working with code because it's more intuitive. Changing the values of the red, green, and blue components often has unexpected results, while estimating the results of changes to hue, saturation, and brightness follows a more logical path. The following examples show a transition from green to blue. The first example makes this transition using the RGB model. It requires calculating all three color values, and the saturation of the color unexpectedly changes in the middle. The second example makes the transition using the HSB model. Only one number needs to be altered, and the hue changes smoothly and independently from the other color properties.

9-19

```
// Shift from blue to green in RGB mode
colorMode(RGB);
for (int i = 0; i < 100; i++) {
  float r = 61 + (i*0.92);
  float g = 156 + (i*0.48);
  float b = 204 - (i*1.43);
  stroke(r, g, b);
  line(i, 0, i, 100);
}
```

9-20

```
// Shift from blue to green in HSB mode
colorMode(HSB, 360, 100, 100);
for (int i = 0; i < 100; i++) {
  float newHue = 200 - (i*1.2);
  stroke(newHue, 70, 80);
  line(i, 0, i, 100);
}
```

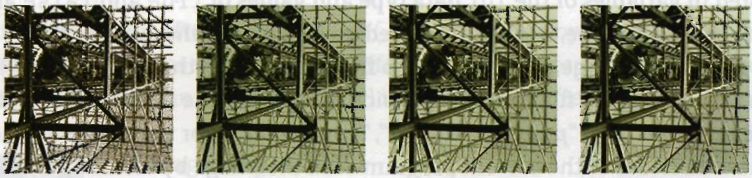
Image 1: Display, Tint

This unit introduces loading and displaying images.

Syntax introduced:

```
PImage, loadImage(), image()
tint(), noTint()
```

Digital photographs are fundamentally different from analog photographs captured on film. Like computer screens, digital photos are rectangular grids of color. The dimensions of digital images are measured in units of pixels. If an image is 320 pixels wide and 240 pixels high, it has 76,800 total pixels. If an image is 1280 pixels wide and 1024 pixels wide, the total number of pixels is an impressive 1,310,720 (1.3 megapixels). Every digital image has a color depth. The color depth refers to the number of bits (p. 669) used to store each pixel. If the color depth of an image is 1, each pixel can be one of two values, for example, black or white. If the color depth is 4, each pixel can be one of 16 values. If the color depth of an image is 8, each pixel can be one of 256 values. Looking at the same image displayed with different color depths reveals how this affects the image's appearance:



1-bit (1 color) 2-bit (4 colors) 4-bit (16 colors) 8-bit (256 colors)

When the Apple Macintosh computer was introduced in 1984, it had a black-and-white screen. Since then, the reproduction of color on screen has rapidly improved. Many contemporary screens have a color depth of 24, which means each pixel can be one of 16,777,216 available colors. This number is typically referred to as "millions of colors."

Digital images are comprised of numbers representing colors. The file format of an image determines how the numbers are ordered in the file. Some file formats store the color data in mathematically complex arrangements to compress the data and reduce the size of the resulting file. A program that loads an image file must know the file format of the image so it can translate the file's data into the expected image. Different types of digital image formats serve specific needs. Processing can load GIF, JPEG, and PNG images, along with some other formats as described in the reference. If you don't already have your images in one of these formats, you can convert other types of digital images to these formats with programs such as GIMP or Adobe Photoshop. Refer to the documentation for these programs if you're unsure how to convert images.

How do you know which image format to use? They all have obscure names that don't help in making this decision, but each format's advantages becomes clear through comparison:

Format	Extension	Color depth	Transparency
GIF	.gif	1-bit to 8-bit	1-bit
JPEG	.jpg	24-bit	None
PNG	.png	1-bit to 24-bit	8-bit

If you are displaying your work on the Internet, image compression becomes an important issue. GIF images are useful for simple graphics with a limited number of colors and transparency. PNG images have similar characteristics but support the full range of colors and transparency. The JPEG format works well for photos, and JPEG files will be smaller than most images saved as PNG. This is because JPEG is a "lossy" format, which means it sacrifices some image quality to reduce file size.

Display

Processing can load images, display them on the screen, and change their size, position, opacity, and tint. There's a data type for images called `PImage`. The same way that integers are stored in variables of the `int` data type and values of `true` and `false` are stored in the `boolean` data type, images are stored in variables of the `PImage` data type. Before displaying an image, it's necessary to first load it with the `loadImage()` function. Be sure to include the file format extension as a part of the name and to put the entire name in quotes (e.g., *"pup.gif"*, *"kat.jpg"*, *"ignatz.png"*). For the image to load, it must be in the data folder of the current program. Add the image by selecting the "Add File" option in the Sketch menu of the Processing environment. Navigate to the image's location on your computer, select the image's icon or name, and click "Open" to add it to the sketch's data folder. As a shortcut, you can also drag and drop an image to the Processing window. To make sure the image was added, select "Show Sketch Folder" from the Sketch menu. The image will be inside the *data* folder. With the image file in the right place, you can load and then display it with the `image()` function:

```
image(name, x, y)
image(name, x, y, width, height)
```

The parameters for `image()` determine the image to draw and its position and size. The *`name`* parameter must be a `PImage` variable. The *x* and *y* parameters set the position of the upper-left corner of the image. The image will display at its actual size (in units of pixels), but you can change the size by adding the *`width`* and *`height`* parameters. Be careful to use the correct capitalization when loading images. If the image is *arch.jpg*, trying to load *Arch.jpg* or *arch.JPG* will create an error. Also, avoid the use of spaces in image names, which can cause problems.

10-01

```
PImage img;
// Image must be in the sketch's "data" folder
img = loadImage("arch.jpg");
image(img, 0, 0);
```

10-02

```
PImage img;
// Image must be in the sketch's "data" folder
img = loadImage("arch.jpg");
image(img, 20, 20, 60, 60);
```

Image color, Transparency

Images are colored with the `tint()` function. This function is used the same way as `fill()` and `stroke()`, but it affects only images:

```
tint(gray)
tint(gray, alpha)
tint(value1, value2, value3)
tint(value1, value2, value3, alpha)
tint(color)
```

All images drawn after running `tint()` will be tinted by the color specified in the parameters. This has no permanent effect on the images, and running the `noTint()` function disables the coloration for all images drawn after it is run.

10-03

```
PImage img;
img = loadImage("arch.jpg");
tint(102); // Tint gray
image(img, 0, 0);
noTint(); // Disable tint
image(img, 50, 0);
```

10-04

```
PImage img;
img = loadImage("arch.jpg");
tint(0, 153, 204); // Tint blue
image(img, 0, 0);
noTint(); // Disable tint
image(img, 50, 0);
```

10-05

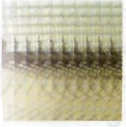
```
color yellow = color(220, 214, 41);
color green = color(110, 164, 32);
color tan = color(180, 177, 132);
PImage img;
img = loadImage("arch.jpg");
tint(yellow);
image(img, 0, 0);
tint(green);
image(img, 33, 0);
tint(tan);
image(img, 66, 0);
```

The parameters for `tint()` follow the color space determined by the `colorMode()` function (remember, the default color mode is RGB, with all values ranging from 0 to 255). If the color mode is changed to HSB or a different range, the tint values should be specified relative to that mode.

To make an image transparent without changing its color, set the tint to white. The value will depend on the current color mode, but the default white value is 255.

10-06

```
PImage img;
img = loadImage("arch.jpg");
background(255);
tint(255, 102);  // Alpha to 102 without changing the tint
image(img, 0, 0, 100, 100);
tint(255, 204, 0, 153);  // Tint to yellow, alpha to 153
image(img, 20, 20, 100, 100);
```



10-07

```
PImage img;
img = loadImage("arch.jpg");
background(255);
tint(255, 51);
// Draw the image 10 times, moving each to the right
for (int i = 0; i < 10; i++) {
  image(img, i*10, 0);
}
```

GIF and PNG images retain their transparency when loaded and displayed in Processing. This allows anything drawn before the image to be visible through the transparent sections of the image. GIF images have only 1-bit transparency, meaning each pixel can only be completely opaque or completely transparent. The PNG format supports 8-bit transparency, meaning there are 256 levels of opacity.

10-08

```
// Loads a GIF image with 1-bit transparency
PImage img;
img = loadImage("archTrans.gif");
background(255);
image(img, 0, 0);
image(img, -20, 0);
```

10-09

```
// Loads a PNG image with 8-bit transparency
PImage img;
img = loadImage("arch.png");
background(255);
image(img, 0, 0);
image(img, -20, 0);
```

Exercises

1. *Draw two images in the display window.*
2. *Draw three images in the display window, each with a different tint.*
3. *Load a GIF or PNG image with transparency and create a collage by layering the image.*

Math 3: Trigonometry

This unit introduces the basics of trigonometry and how to utilize it for generating form.

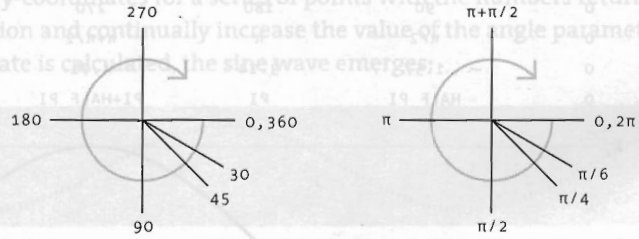
Syntax introduced:

```
PI, QUARTER_PI, HALF_PI, TWO_PI, radians(), degrees()
sin(), cos(), arc()
```

Trigonometry defines the relationships between the sides and angles of triangles. The trigonometric functions sine and cosine generate repeating numbers that can be used to draw waves, circles, arcs, and spirals.

Angles, Waves

Degrees are a common way to measure angles. A right angle is 90°, halfway around a circle is 180°, and the full circle is 360°. In working with trigonometry, angles are measured in units called radians. Using radians, the angle values are expressed in relation to the mathematical value π, written in Latin characters as "pi" and pronounced "pie." In terms of radians, a right angle is π/2, halfway around a circle is simply π, and the full circle is 2π.



Degree values Radian values

The numerical value of π is a constant thought to be be infinitely long and without a repeating pattern. It is the ratio of the circumference of a circle to its diameter. When writing Processing code, use the mathematical constant `PI` to represent this number. Other commonly used values of π are expressed with the constants `QUARTER_PI`, `HALF_PI`, and `TWO_PI`. Run the following line of code to see the value of π to 8 significant digits.

```
println(PI);  // Prints the value of PI to the text area
```

14-01

In casual use, the numerical value of π is 3.14, and 2π is 6.28. Angles can be converted from degrees to radians with the `radians()` function, or vice versa using `degrees()`.

This short program demonstrates the conversions between these representations:

14-02

```
float r1 = radians(90);
float r1 = radians(180);
println(r1);  // Prints "1.5707964"
println(r2);  // Prints "3.1415927"
float d1 = degrees(PI);
float d2 = degrees(TWO_PI);
println(d1);  // Prints "180.0"
println(d2);  // Prints "360.0"
```

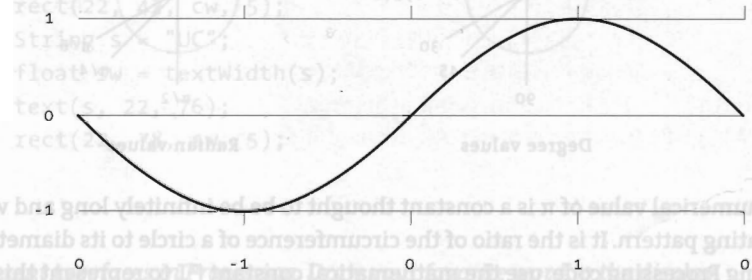
If you prefer working with degrees, use the `radians()` function in your programs to convert the degree values for use with functions that require radian values.

The `sin()` and `cos()` functions are used to determine the sine and cosine value of any angle. Each of these functions requires one parameter:

```
sin(angle)
cos(angle)
```

The *angle* parameter is always specified as a radian value. The values returned from these functions are always between the floating-point values of -1.0 and 1.0. The relationship between sine values and angles are shown here:

Degrees	0	90	180	270	360
Radians	0	π/2	π	π+π/2	2π
Decimal radians	0	1.57	3.14	4.71	6.28
Constants	0	HALF_PI	PI	PI+HALF_PI	TWO_PI



As angles increase in value, the sine values repeat. At the angle 0.0, the value of sine is also 0.0, and this value decreases as the angle increases. When the angle reaches 90.0° (π/2), the sine value increases until is it zero again at the angle 180.0° (π), then it continues to increase until the angle reaches 270.0° (π + π/2), at which point it begins decreasing until the angle reaches 360.0° (2π). At this point, the values repeat the cycle. The sine values can been seen by putting a `sin()` function inside a `for` structure and iterating while changing the angle value:

14-03

```
for (float angle = 0; angle < TWO_PI; angle += PI/24.0) {
  println(sin(angle));
}
```

Because the values from `sin()` are numbers between -1.0 and 1.0, they are easy to use in controlling a composition. Multiplying the numbers by 50.0, for example, will return values between -50.0 and 50.0.

14-04

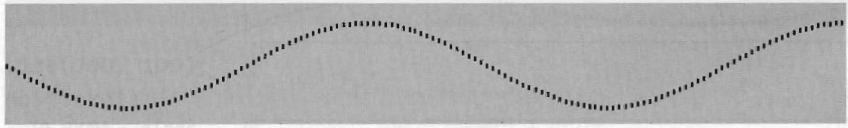
```
for (float angle = 0; angle < TWO_PI; angle += PI/24.0) {
  println(sin(angle) * 50.0);
}
```

To convert the sine values to a range of positive numbers, first add the value 1.0 to create numbers between 0.0 and 2.0. Divide that number by 2.0 to get a number between 0.0 and 1.0, which can then be simply remapped to any range. Alternatively, the `map()` function can be used to convert the values from `sin()` to any range. In this example, the values from `sin()` are put into the range between 0 and 1000.

14-05

```
for (float angle = 0; angle < TWO_PI; angle += PI/24.0) {
  float newValue = map(sin(angle), -1, 1, 0, 1000);
  println(newValue);
}
```

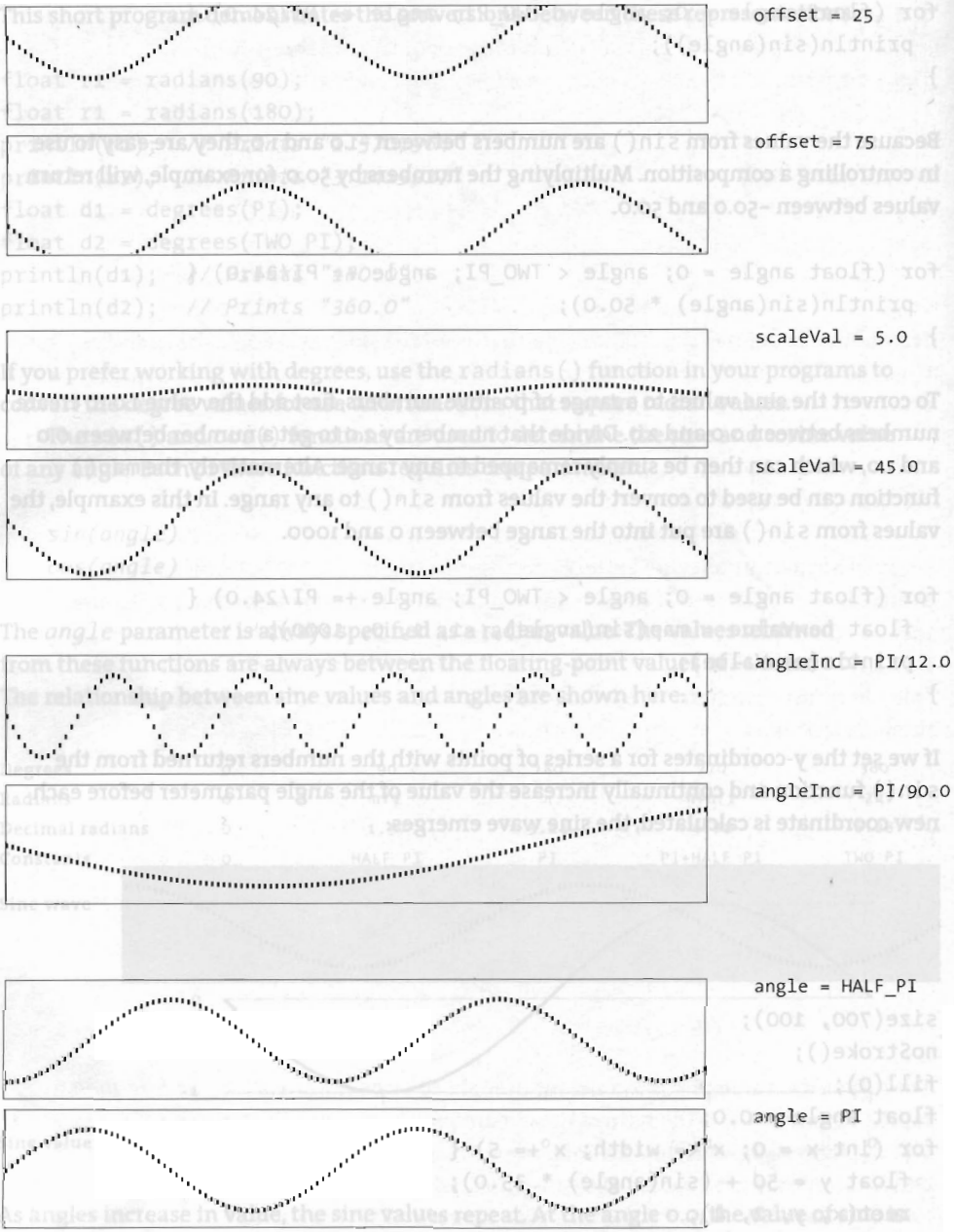
If we set the y-coordinates for a series of points with the numbers returned from the `sin()` function and continually increase the value of the angle parameter before each new coordinate is calculated, the sine wave emerges:



14-06

```
size(700, 100);
noStroke();
fill(0);
float angle = 0.0;
for (int x = 0; x <= width; x += 5) {
  float y = 50 + (sin(angle) * 35.0);
  rect(x, y, 2, 4);
  angle += PI/40.0;
}
```

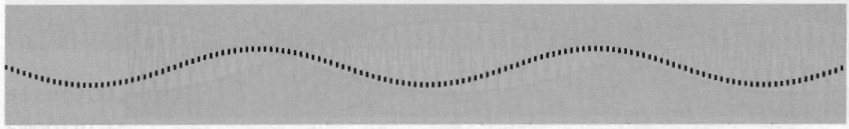
Replacing some fixed numbers in the previous program with variables allows you to control the waveform by simply changing the values of the variables. The `offset` variable controls the y-coordinates of the wave, the `scaleVal` variable controls the



Modulating a sine wave

Different values for the variables in code 14-07 create a range of waves. Notice how each variable affects a different attribute of the wave.

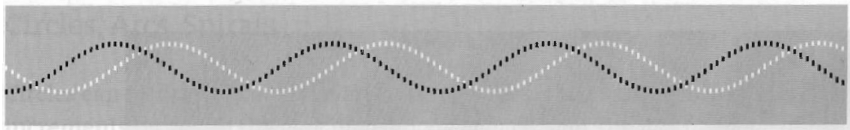
height of the wave, and the `angleInc` variable controls the speed at which the angle increases, thereby creating a wave with a higher or lower frequency.



14-07

```
size(700, 100);
noStroke();
smooth();
fill(0);
float offset = 50.0;      // Y offset
float scaleVal = 35.0;    // Scale value for the wave magnitude
float angleInc = PI/28.0; // Increment between the next angle
float angle = 0.0;        // Angle to receive sine values from
for (int x = 0; x <= width; x += 5) {
  float y = offset + (sin(angle) * scaleVal);
  rect(x, y, 2, 4);
  angle += angleInc;
}
```

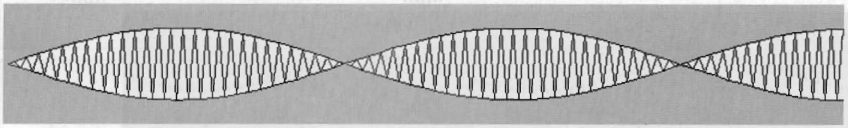
The `cos()` function returns values in the same range and pattern as `sin()`, but the numbers are offset by π/2 radians (90°).



14-08

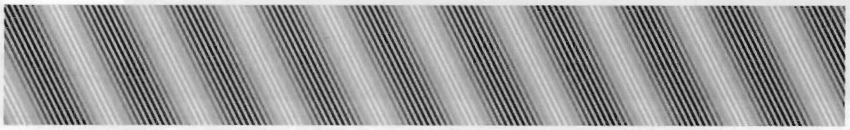
```
size(700, 100);
noStroke();
smooth();
float offset = 50.0;
float scaleVal = 20.0;
float angleInc = PI/18.0;
float angle = 0.0;
for (int x = 0; x <= width; x += 5) {
  float y = offset + (sin(angle) * scaleVal);
  fill(255);
  rect(x, y, 2, 4);
  y = offset + (cos(angle) * scaleVal);
  fill(0);
  rect(x, y, 2, 4);
  angle += angleInc;
}
```

The following examples demonstrate ways to use the numbers from the `sin()` function to generate shapes.



14-09

```
size(700, 100);
float offset = 50;
float scaleVal = 30.0;
float angleInc = PI/56.0;
float angle = 0.0;
beginShape(TRIANGLE_STRIP);
for (int x = 4 ; x <= width+5; x += 5) {
  float y = sin(angle) * scaleVal;
  if ((x % 2) == 0) {  // Every other time through the loop
    vertex(x, offset + y);
  } else {
    vertex(x, offset - y);
  }
  angle += angleInc;
}
endShape();
```



14-10

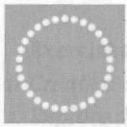
```
size(700, 100);
smooth();
strokeWeight(2);
float offset = 126.0;
float scaleVal = 126.0;
float angleInc = 0.42;
float angle = 0.0;
for (int x = -52; x <= width; x += 5) {
  float y = offset + (sin(angle) * scaleVal);
  stroke(y);
  line(x, 0, x+50, height);
  angle += angleInc;
}
```

14-11

```
size(700, 100);
smooth();
fill(255, 20);
float scaleVal = 18.0;
float angleInc = PI/28.0;
float angle = 0.0;
for (int offset = -10; offset < width+10; offset += 5) {
  for (int y = 0; y <= height; y += 2) {
    float x = offset + (sin(angle) * scaleVal);
    noStroke();
    ellipse(x, y, 10, 10);
    stroke(0);
    point(x, y);
    angle += angleInc;
  }
  angle += PI;
}
```

Circles, Arcs, Spirals

Circles can be drawn from sine and cosine waves. The example below has an angle that increments by 12°, all the way up to 360°. On each step, the `cos()` value of the angle is used to draw the x-coordinate, and the `sin()` value draws the y-coordinate. Because `sin()` and `cos()` return numbers between -1.0 and 1.0, the result is multiplied by the `radius` variable to draw a circle with radius 38. Adding 50 to the `x` and `y` positions sets the center of the circle at (50,50).



14-12

```
noStroke();
smooth();
int radius = 38;
for (int deg = 0; deg < 360; deg += 12) {
  float angle = radians(deg);
  float x = 50 + (cos(angle) * radius);
  float y = 50 + (sin(angle) * radius);
  ellipse(x, y, 6, 6);
}
```

If the angle is incremented only part of the way around the circle, an arc is drawn. For example, changing line 4 in the preceding program gives the following result:

14-13

```
noStroke();
smooth();
int radius = 38;
for (int deg = 0; deg < 220; deg += 12) {
  float angle = radians(deg);
  float x = 50 + (cos(angle) * radius);
  float y = 50 + (sin(angle) * radius);
  ellipse(x, y, 6, 6);
}
```

To simplify drawing arcs, Processing includes an `arc()` function:

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

Arcs are drawn along the outer edge of an ellipse defined by the `x`, `y`, `width`, and `height` parameters. The `start` and `stop` parameters specify the angles needed to draw the arc form in units of radians. The following examples show the function in use.

14-14

```
strokeWeight(2);
arc(50, 55, 50, 50, 0, HALF_PI);
arc(50, 55, 60, 60, HALF_PI, PI);
arc(50, 55, 70, 70, PI, TWO_PI - HALF_PI);
noFill();
arc(50, 55, 80, 80, TWO_PI - HALF_PI, TWO_PI);
```

14-15

```
smooth();
noFill();
randomSeed(0);
strokeWeight(10);
stroke(0, 150);
for (int i = 0; i < 160; i += 10) {
  float begin = radians(i);
  float end = begin + HALF_PI;
  arc(67, 37, i, i, begin, end);
}
```

To create a spiral, multiply the sine and cosine values by increasing or decreasing scalar values. In the following examples, the spiral grows as the `radius` variable increases:

14-16

```
noStroke();
smooth();
float radius = 1.0;
for (int deg = 0; deg < 360*6; deg += 11) {
  float angle = radians(deg);
  float x = 75 + (cos(angle) * radius);
  float y = 42 + (sin(angle) * radius);
  ellipse(x, y, 6, 6);
  radius = radius + 0.34;
}
```

14-17

```
smooth();
float radius = 0.15;
float cx = 33;  // Center x- and y-coordinates
float cy = 66;
float px = cx;  // Start with center as the
float py = cy;  // previous coordinate
for (int deg = 0; deg < 360*5; deg += 12) {
  float angle = radians(deg);
  float x = cx + (cos(angle) * radius);
  float y = cy + (sin(angle) * radius);
  line(px, py, x, y);
  radius = radius * 1.05;
  px = x;
  py = y;
}
```

The content of this unit is applied to controlling movement in Motion 2 (p. 291).

Exercises

1. *Create a composition with the data generated using* `sin()`.
2. *Explore drawing circles and arcs with* `sin()` *and* `cos()`. *Develop a composition from the results of the exploration.*
3. *Generate a series of spirals and organize them into a composition.*

Math 4: Random

This unit introduces the basics of trigonometry and random numbers and explains how to utilize them for generating form.

Syntax introduced:
`random(), randomSeed(), noise(), noiseSeed()`

Random compositional choices have a long history, particularly in modern art. In 1913 Marcel Duchamp's *3 Stoppages Étalon* employed the curves of dropped threads to derive novel units of measurement. Jean Arp used chance operations to define the position of elements in his collages. The composer John Cage sometimes tossed coins to determine the order and duration of notes in his scores. Artists integrate chance, randomness, and noise into their work either as a creative exercise or as a way of relinquishing some control to an external force. Actions like dropping, throwing, rolling, etc., deprive the artists of certain aspects of decisions. The world's chaos can be channeled into making images and objects with physical media. In contrast, computers are machines that make consistent and accurate calculations and must therefore simulate random numbers to approximate the kind of chance operations used in nondigital art.

There is an obvious contrast between rigid structure and complete chaos, and some of the most satisfying aesthetic experiences are created by infusing one with the other. The tension between order and chaos can actively engage our attention:

If a composition is obviously ordered, it will not hold attention beyond a quick glance.

Conversely, if a composition is entirely chaotic, it will also not retain one's gaze.

A balance between the two can yield a more satisfying result.

Unexpected values

The `random()` function is used to create unpredictable values within the range specified by its parameters.

```
random(high)
random(low, high)
```

When one parameter is passed to the function, it returns a `float` from zero up to (but not including) the value of the parameter. The function call `random(5.0)` returns

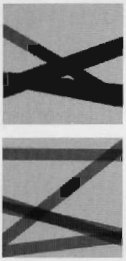
values from 0.0 up to 5.0. If two parameters are used, the function returns a value between the two parameters. Running `random(-5.0, 10.2)` returns values from -5.0 up to 10.2.

The numbers returned from `random()` are always floating-point values; therefore, they cannot be assigned to an `int` variable. The `int()` function can be used to convert a `float` value to an `int`.

```
float f = random(5.2);       // Assign f a float value from 0 to 5.2
int i = random(5.2);         // ERROR! Can't assign a float to an int
int j = int(random(5.2));    // Assign j an int value from 0 to 5
```

15-01

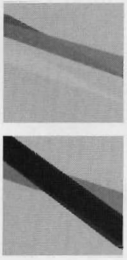
Because the numbers returned from `random()` are not predictable, each time the program is run, the result is different. The numbers from this function can be used to control almost any aspect of a program.



```
smooth();
strokeWeight(10);
stroke(0, 130);
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
```

15-02

The version of `random()` with two parameters provides more control over the results of the function. The previous example has been modified so the lines always progress from the upper-left to the lower-right, but the precise position is a chance operation. Storing the results of `random()` into a variable makes it possible to use the value more than once in the program. This program uses the random value `r` to set both the y-coordinate of the first point of the line and its stroke value.



```
smooth();
strokeWeight(20);
stroke(0, 230);
float r = random(5, 45);
stroke(r * 5.6, 230);
line(0, r, 100, random(55, 95));
r = random(5, 45);
stroke(r * 5.6, 230);
line(0, r, 100, random(55, 95));
r = random(5, 45);
stroke(r * 5.6, 230);
line(0, r, 100, random(55, 95));
```

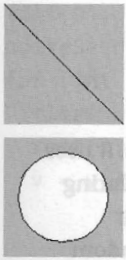
15-03

Using `random()` within a `for` structure is an easy way to generate random numbers.

15-04

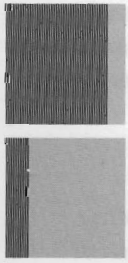
```
background(0);
stroke(255, 60);
for (int i = 0; i < 100; i++) {
  float r = random(10);
  strokeWeight(r);
  float offset = r * 5.0;
  line(i-20, 100, i+offset, 0);
}
```

To use random values to determine the flow of the program, you can place the `random()` function in a relational expression. In the first example below, either a line or an ellipse is drawn, depending on whether the result of the `random()` function is less than 50 or greater than or equal to 50, respectively. In the second example, between 1 and 50 vertical lines are drawn according to the result of the `random()` function.



15-05

```
float r = random(100);
if (r < 50.0) {
  line(0, 0, 100, 100);
} else {
  ellipse(50, 50, 75, 75);
}
```



15-06

```
int num = int(random(50)) + 1;
for (int i = 0; i < num; i++) {
  line(i * 2, 0, i * 2, 100);
}
```

It's sometimes desirable to include unpredictable numbers in your programs but to force the same sequence of numbers each time the program is run. The `randomSeed()` function is the key to producing such numbers.

randomSeed(value)

The *value* parameter must be an `int`. Use the same *value* parameter in a program each time it is run to force the same random numbers to be produced in the same order. There is a unique sequence of random values for every integer value. You might find that

using the number 1843 as the *value* parameter produces numbers that suit your needs, while the number 258 will not.

The following program is a slight variation on code 15-04. Adding `randomSeed()` ensures it will produce the same values every time it is run. Change the *value* parameter assigned to `randomSeed()` to generate a different set of numbers and thereby change the image produced by the program.

s=6

s=12

15-07

```
int s = 6;          // Seed value
background(0);
stroke(255, 60);
randomSeed(s);      // Produce the same numbers each time
for (int i = 0; i < 100; i++) {
  float r = random(10);
  strokeWeight(r);
  float offset = r * 5;
  line(i-20, 100, i+offset, 0);
}
```

Noise

The `noise()` function is a more controllable way to create unexpected values. It uses the Perlin Noise technique, developed by Ken Perlin.[1] Originally used for simulating natural textures through subtle irregularities, Perlin Noise is now also used for generating shapes and realistic motion. It works by interpolating between random values to create smoother transitions than the numbers returned from `random()`. The noise function has between one and three parameters:

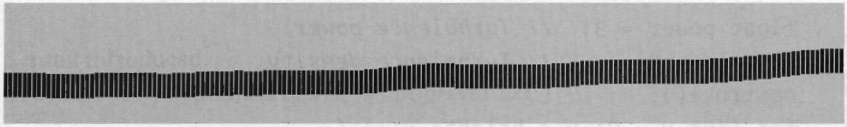
```
noise(x)
noise(x, y)
noise(x, y, z)
```

The version of the function with one parameter is used to create a single sequence of random numbers. Additional parameters produce noise in more dimensions. For example, the version with two parameters can be used to create a two-dimensional texture. The version with three parameters can be used to create a three-dimensional shape or texture or an animated two-dimensional texture. Regardless of the number or value of the parameters, this function always returns values between 0.0 and 1.0. If other values are desired, an equation can be applied to the result to change the range (p. 81).

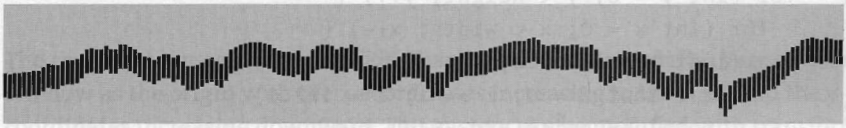
The numbers returned by noise can be made closer to or farther from the previous value by way of changes in the rate at which the parameter increases. As a general rule, the smaller the difference, the smoother the resulting noise sequence will be. A small change generates numbers that are closer to the previous value than a large increase would. Steps of 0.005–0.03 work best for most applications, but this will differ

depending on use. The following program uses the `inc` variable to define the difference between each number. Notice the differences between the results as the value of `inc` increases. The `noiseSeed()` function, which works like `randomSeed()`, is used to produce the same sequence of numbers each time the program runs.

inc=0.01



inc=0.1

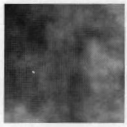


15-08

```
size(600, 100);
float v = 0.0;
float inc = 0.1;
noStroke();
fill(0);
noiseSeed(0);
for (int i = 0; i < width; i = i+4) {
  float n = noise(v) * 70.0;
  rect(i, 10 + n, 3, 20);
  v = v + inc;
}
```

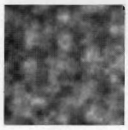
Add a second parameter to `noise()` to open the possibility of creating a two-dimensional texture. In the following example, embedded `for` structures are used to generate continuous noise values on the x-axis and y-axis. The values returned from `noise()` are used to set the gray values for a grid of points drawn to the screen.

inc=0.04



inc=0.02

inc=0.1

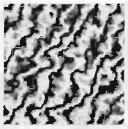


15-09

```
float xnoise = 0.0;
float ynoise = 0.0;
float inc = 0.04;
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    float gray = noise(xnoise, ynoise) * 255;
    stroke(gray);
    point(x, y);
    xnoise = xnoise + inc;
  }
  xnoise = 0;
  ynoise = ynoise + inc;
}
```

Diverse textures can be created using `noise()` in collaboration with `sin()`. The following example deforms a regular sequence of bars created with `sin()` into a texture reminiscent of one found in nature. The `power` variable sets the amount the texture deforms from the lines and the `density` parameter sets the granularity of the texture.

d=8



d=32

d=128

15-10

```
float power = 3;  // Turbulence power
float d = 8;      // Turbulence density
noStroke();
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    float total = 0.0;
    for (float i = d; i >= 1; i = i/2.0) {
      total += noise(x/d, y/d) * d;
    }
    float turbulence = 128.0 * total / d;
    float base = (x * 0.2) + (y * 0.12);
    float offset = base + (power * turbulence / 256.0);
    float gray = abs(sin(offset)) * 256.0;
    stroke(gray);
    point(x, y);
  }
}
```

Examples showing noise used for motion are given in Motion 2 (p. 291).

Exercises

1. *Use three variables assigned to random values to create a composition that is different every time the program is run.*
2. *Create a composition using a `for` structure and `random()` to make a composition of a different density every time the program is run.*
3. *Use `noise()` and `noiseSeed()` to create the same irregular shape every time a program is run.*

Notes

1. Perlin Noise was developed in the 1980s, and in 1997 Perlin received an Academy Award for Technical Achievement for this research. See *http://mrl.nyu.edu/~perlin/doc/oscar.html* and *http://www.noisemachine.com/talk1*.

Transform 1: Translate, Matrices

This unit introduces coordinate system transformations and explains how to control their scope.

Syntax introduced:
`translate(), pushMatrix(), popMatrix()`

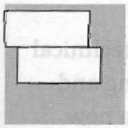
The coordinate system introduced in Shape 1 uses the upper-left corner of the display window as the origin with the x-coordinates increasing to the right and the y-coordinates increasing downward. This system can be modified with transformations. The coordinates can be translated, rotated, and scaled so shapes are drawn to the display window with a different position, orientation, and size.

Translation

The `translate()` function moves the origin from the upper-left corner of the screen to another location. It has two parameters. The first is the x-coordinate offset and the second is the y-coordinate offset:

```
translate(x, y)
```

The values of the *x* and *y* parameters are added to any shapes drawn after the function is run. If 10 is used as the *x* parameter and 30 is used as the *y* parameter, a point drawn at coordinate (0,5) will instead be drawn at coordinate (10,35). Only elements drawn after the transformation are affected. The following examples show how this works.



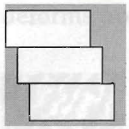
16-01

```
// The same rectangle is drawn, but only the second is
// affected by translate() because it is drawn after
rect(0, 5, 70, 30);
translate(10, 30); // Shifts 10 pixels right and 30 down
rect(0, 5, 70, 30);
```

16-02

```
// A negative number used as a parameter to translate()
// moves the coordinates in the opposite direction
rect(0, 5, 70, 30);
translate(10, -10); // Shifts 10 pixels right and up
rect(0, 5, 70, 30);
```

The translate() function is additive. If translate(10,30) is run twice, all the elements drawn after will display with an x-offset of 20 and a y-offset of 60.



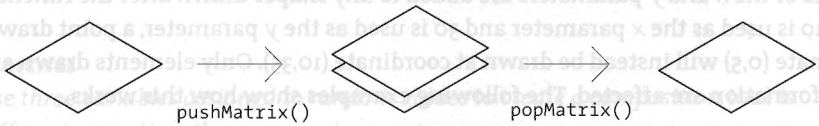
16-03

```
rect(0, 5, 70, 30);
translate(10, 30);  // Shifts 10 pixels right and 30 down
rect(0, 5, 70, 30);
translate(10, 30);  // Shifts everything again for a total
rect(0, 5, 70, 30); // 20 pixels right and 60 down
```

Controlling transformations

The transformation matrix is a set of numbers that defines how geometry is drawn to the screen. Transformation functions such as translate() alter the numbers in this matrix and cause the geometry to draw differently. In the previous examples, we saw how transformations accumulate as the program runs. The pushMatrix() function records the current state of all transformations so that a program can return to it later. To return to the previous state, use popMatrix().

Think of each matrix as a sheet of paper with the current list of transformations (translate, rotate, scale) written on the surface. When a function such as translate() is run, it is added to the paper. To save the current matrix for later use, add a new sheet of paper to the top of the pile and copy the information from the sheet below. Any new changes are made to the top sheet of paper, preserving the numbers on the sheet(s) below. To return to a previous coordinate matrix, simply remove and discard the top sheet of paper to reveal the saved transformations below:

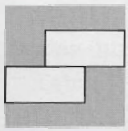


This is essentially how coordinate matrices are updated and stored, but more technical terms are used. Adding a sheet of paper is pushing, removing a sheet is popping and the pile of pages is called a stack. The pushMatrix() function is used to add a new coordinate matrix to the stack, and popMatrix() is used to remove one from the stack. Each pushMatrix() must have a corresponding popMatrix(). The function pushMatrix() cannot be used without popMatrix() and vice versa.

Compare the two examples below. Both draw the same rectangles, but with different results. The second example employs pushMatrix() and popMatrix() to isolate the effects of the translate() function to apply only to the first rectangle. Because the other rectangle is drawn after the call to popMatrix() it draws from its x-coordinate without being affected by the translation.

```
translate(33, 0);  // Shift 33 pixels right
rect(0, 20, 66, 30);
rect(0, 50, 66, 30);
```

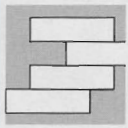
16-04



```
pushMatrix();
translate(33, 0);  // Shift 33 pixels right
rect(0, 20, 66, 30);
popMatrix();  // Remove the shift
// This shape is not affected by translate() because
// the transformation is isolated between the pushMatrix()
// and popMatrix()
rect(0, 50, 66, 30);
```

16-05

Embedding the `pushMatrix()` and `popMatrix()` functions can further control their range. In the following example, the first rectangle is affected by the first translation, the second rectangle is affected by the first and second translations, and the third rectangle is only affected by the first translation because the second translation is isolated with a `pushMatrix()` and `popMatrix()` pair. The fourth rectangle is not affected by any of the translations because the `popMatrix()` on the second-to-last line cancels the `pushMatrix()` on the first line.



```
pushMatrix();
translate(20, 0);
rect(0, 10, 70, 20);  // Draws at (20, 30)
pushMatrix();
translate(30, 0);
rect(0, 30, 70, 20);  // Draws at (50, 30)
popMatrix();
rect(0, 50, 70, 20);  // Draws at (20, 50)
popMatrix();
rect(0, 70, 70, 20);  // Draws at (0, 70)
```

16-06

The transformation functions for rotating and scaling are introduced in Transform 2 (p. 137).

Exercises

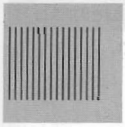
1. *Use `translate()` to reposition a shape.*
2. *Use multiple translations to reposition a series of shapes.*
3. *Use `pushMatrix()` and `popMatrix()` to rearrange the composition from exercise 2.*

22-05 cont.

```
void face(int x, int y, int gap) {
  line(x, 0, x, y);                    // Nose Bridge
  line(x, y, x+gap, y);                // Nose
  line(x+gap, y, x+gap, 100);
  int mouthY = height - (height-y)/2;
  line(x, mouthY, x+gap, mouthY);      // Mouth
  noStroke();
  ellipse(x-gap/2, y/2, 5, 5);         // Left eye
  ellipse(x+gap, y/2, 5, 5);           // Right eye
}
```

Recursion

A common example of recursion is standing between two mirrors to see infinite reflections. In software, recursion means that a function can call itself within its own block. To prevent this from continuing forever, it's necessary to have some way for the function to exit. The following two programs produce the same result by different means, the first using a `for` structure and the second using recursion.



22-06

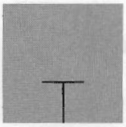
```
int x = 5;
for (int num = 15; num >= 0; num -= 1) {
  line(x, 20, x, 80);
  x += 5;
}
```

22-07

```
void setup() {
  drawLines(5, 15);
}

void drawLines(int x, int num) {
  line(x, 20, x, 80);
  if (num > 0) {
    drawLines(x+5, num-1);
  }
}
```

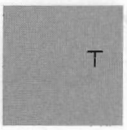
The recursive example uses more of the computer's resources to complete the task. For such a simple calculation, using the `for` structure is advised, but the recursive approach opens other possibilities. The following two examples utilize the custom `drawT()` function to show the effects of recursion.

x=50
y=100
a=35



x=24
y=65
a=45

x=76
y=50
a=12

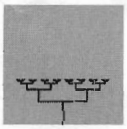


```
int x = 50;   // X-coordinate of the center
int y = 100;  // Y-coordinate of the bottom
int a = 35;   // Half the width of the top bar

void setup() {
  size(100, 100);
  noLoop();
}

void draw() {
  drawT(x, y, a);
}

void drawT(int xpos, int ypos, int apex) {
  line(xpos, ypos, xpos, ypos-apex);
  line(xpos-(apex/2), ypos-apex, xpos+(apex/2), ypos-apex);
}
```

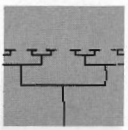
22-08

The `drawT()` function is made recursive by the inclusion of a call to itself within the function block. A fourth parameter called `num` is added to set the number of recursions. This value is decremented by 1 each time the function calls itself. When the value is no longer greater than 0, the recursion stops and the image is drawn to the screen.

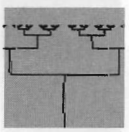
x=50
a=20
n=8



x=50
a=35
n=3



x=50
a=45
n=12



```
int x = 50;   // X-coordinate of the center
int y = 100;  // Y-coordinate of the bottom
int a = 35;   // Half the width of the top bar
int n = 3;    // Number of recursions

void setup() {
  size(100, 100);
  noLoop();
}

void draw() {
  drawT(x, y, a, n);
}

void drawT(int x, int y, int apex, int num) {
  line(x, y, x, y-apex);
  line(x-apex, y-apex, x+apex, y-apex);
  // This relational expression must eventually be
  // false to stop the recursion and draw the lines
  if (num > 0) {
    drawT(x-apex, y-apex, apex/2, num-1);
```

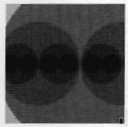
22-09

```
    drawT(x+apex, y-apex, apex/2, num-1);
  }
}
```

22-09 cont.

A binary tree structure (one that has two branches from each node) like the one above can be visualized in different ways. This program draws a circle at every node. The y-coordinate for each node is the same, and the radius for each circle is halved at each layer.

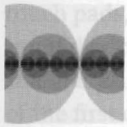
x=63
r=70
n=4



22-10

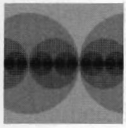
```
int x = 63;  // X-coordinate
int r = 85;  // Starting radius
int n = 6;   // Number of recursions
```

x=63
r=100
n=8



```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
}
```

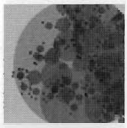
x=63
r=85
n=6



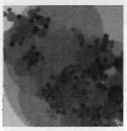
```
void draw() {
  drawCircle(63, 85, 6);
}

void drawCircle(int x, int radius, int num) {
  float tt = 126 * num/4.0;
  fill(tt);
  ellipse(x, 50, radius*2, radius*2);
  if (num > 1) {
    num = num - 1;
    drawCircle(x - radius/2, radius/2, num);
    drawCircle(x + radius/2, radius/2, num);
  }
}
```

A slight modification yields a radical alteration of the form. Circles in every subsequent layer are given random positions relative to the previous position. The resulting images have a balance between order and disorder. At each level of recursion, the size of the circles decrease, their distance from the previous level decreases, and their values grow darker. Change the number used as the parameter to `randomSeed()` (p. 129) to produce a different composition.

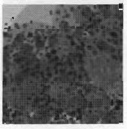
r=55
n=6
rs=18



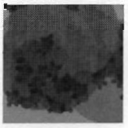
r=65
n=6
rs=22



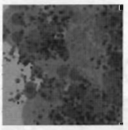
r=65
n=7
rs=22



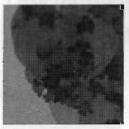
r=90
n=6
rs=24



r=80
n=7
rs=12



r=90
n=6
rs=26



```
int x = 63;   // X-coordinate
int y = 50;   // Y-coordinate
int r = 80;   // Starting radius
int n = 7;    // Number of recursions
int rs = 12;  // Random seed value

void setup() {
  size(100, 100);
  noStroke();
  smooth();
  noLoop();
  randomSeed(rs);
}

void draw() {
  drawCircle(x, y, r, n);
}

void drawCircle(float x, float y, int radius, int num) {
  float value = 126 * num / 6.0;
  fill(value, 153);
  ellipse(x, y, radius*2, radius*2);
  if (num > 1) {
    num = num - 1;
    int branches = int(random(2, 6));
    for (int i = 0; i < branches; i++) {
      float a = random(0, TWO_PI);
      float newx = x + cos(a) * 6.0 * num;
      float newy = y + sin(a) * 6.0 * num;
      drawCircle(newx, newy, radius/2, num);
    }
  }
}
```

Exercises

1. *Write your own function to draw a parameterized arch.*
2. *Create a function for drawing a chair. Use two parameters to change its position and two more to change the shape. Using your function, draw 9 chairs in the display window in a regular 3 × 3 matrix. Use different parameters to give each chair drawn a unique shape.*
3. *Modify code 22-04 to create a sequence of different compositions.*