8/Motion

Like a flip book, animation on screen is created by drawing an image, then drawing a slightly different image, then another, and so on. The illusion of fluid motion is created by *persistence of vision*. When a set of similar images is presented at a fast enough rate, our brains translate these images into motion.

Frames

To create smooth motion, Processing tries to run the code inside `draw()` at 60 frames each second. A *frame* is one trip through the `draw()` and the *frame rate* is how many frames are drawn each second. Therefore, a program that draws 60 frames each second means the program runs the entire code inside `draw()` 60 times each second.

Example 8-1: See the Frame Rate

To confirm the frame rate, run this program and watch the values print to the Console (the `frameRate` variable keeps track of the program's speed):

```
void draw() {
  println(frameRate);
}
```

Example 8-2: Set the Frame Rate

The `frameRate()` function changes the speed at which the program runs. To see the result, uncomment different versions of `frameRate()` in this example:

```
void setup() {
  frameRate(30);    // Thirty frames each second
  //frameRate(12);  // Twelve frames each second
  //frameRate(2);   // Two frames each second
  //frameRate(0.5); // One frame every two seconds
}

void draw() {
  println(frameRate);
}
```

Processing *tries* to run the code at 60 frames each second, but if it takes longer than 1/60th of a second to run the `draw()` method, then the frame rate will decrease. The `frameRate()` function specifies only the maximum frame rate, and the actual frame rate for any program depends on the computer that is running the code.

Speed and Direction

To create fluid motion examples, we use a data type called `float`. This type of variable stores numbers with decimal places, which provide more resolution for working with motion. For instance, when using `int`, the slowest you can move each frame is one pixel at a time (1, 2, 3, 4, . . .), but with `float`, you can move as slowly as you want (1.01, 1.01, 1.02, 1.03, . . .).

Example 8-3: Move a Shape

The following example moves a shape from left to right by updating the `x` variable:

```
int radius = 40;
float x = -radius;
float speed = 0.5;

void setup() {
  size(240, 120);
  ellipseMode(RADIUS);
}

void draw() {
  background(0);
  x += speed; // Increase the value of x
  arc(x, 60, radius, radius, 0.52, 5.76);
}
```

When you run this code, you'll notice the shape moves off the right of the screen when the value of the `x` variable is greater than the width of the window. The value of `x` continues to increase, but the shape is no longer visible.

Example 8-4: Wrap Around

There are many alternatives to this behavior, which you can choose from according to your preference. First, we'll extend the code to show how to move the shape back to the left edge of the screen after it disappears off the right. In this case, picture the screen as a flattened cylinder, with the shape moving around the outside to return to its starting point:

```
int radius = 40;
float x = -radius;
float speed = 0.5;

void setup() {
  size(240, 120);
  ellipseMode(RADIUS);
}

void draw() {
  background(0);
  x += speed;  // Increase the value of x
  if (x > width+radius) {  // If the shape is off screen,
    x = -radius;  // move to the left edge
  }
  arc(x, 60, radius, radius, 0.52, 5.76);
}
```

On each trip through `draw()`, the code tests to see if the value of `x` has increased beyond the width of the screen (plus the radius of the shape). If it has, we set the value of `x` to a negative value, so that as it continues to increase, it will enter the screen from the left. See Figure 8-1 for a diagram of how it works.

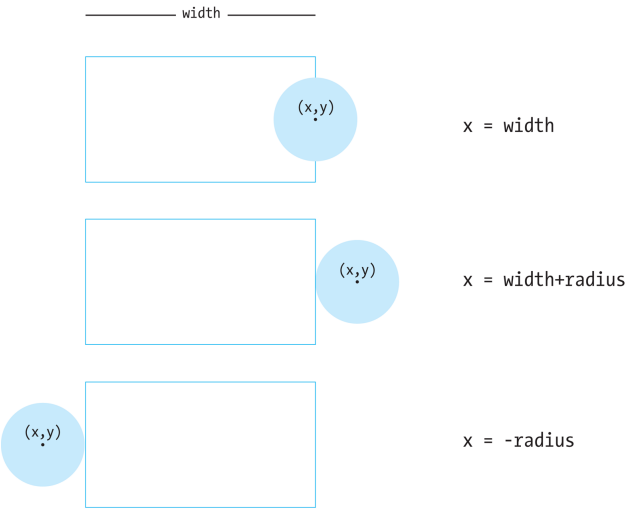
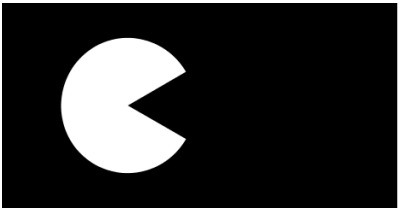
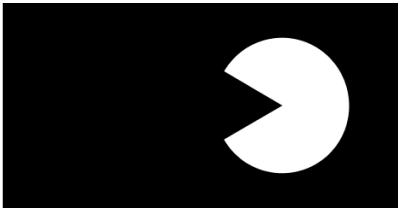


Figure 8-1. *Testing for the edges of the window*

Example 8-5: Bounce Off the Wall

In this example, we'll extend Example 8-3 on page 104 to have the shape change directions when it hits an edge, instead of wrapping around to the left. To make this happen, we add a new variable to store the direction of the shape. A direction value of 1 moves the shape to the right, and a value of –1 moves the shape to the left:





```
int radius = 40;
float x = 110;
float speed = 0.5;
int direction = 1;

void setup() {
  size(240, 120);
  ellipseMode(RADIUS);
}

void draw() {
  background(0);
  x += speed * direction;
  if ((x > width-radius) || (x < radius)) {
    direction = -direction; // Flip direction
  }
  if (direction == 1) {
    arc(x, 60, radius, radius, 0.52, 5.76); // Face right
  } else {
    arc(x, 60, radius, radius, 3.67, 8.9);  // Face left
  }
}
```

When the shape reaches an edge, this code flips the shape's direction by changing the sign of the `direction` variable. For example, if the `direction` variable is positive when the shape reaches an edge, the code flips it to negative.

Tweening

Sometimes you want to animate a shape to go from one point on screen to another. With a few lines of code, you can set up the start position and the stop position, then calculate the in-between (*tween*) positions at each frame.

Example 8-6: Calculate Tween Positions

To make this example code modular, we've created a group of variables at the top. Run the code a few times and change the values to see how this code can move a shape from any location to any other at a range of speeds. Change the `step` variable to alter the speed:

```
int startX = 20;      // Initial x coordinate
int stopX = 160;      // Final x coordinate
int startY = 30;      // Initial y coordinate
int stopY = 80;       // Final y coordinate
float x = startX;     // Current x coordinate
float y = startY;     // Current y coordinate
float step = 0.005;   // Size of each step (0.0 to 1.0)
float pct = 0.0;      // Percentage traveled (0.0 to 1.0)

void setup() {
  size(240, 120);
}

void draw() {
  background(0);
  if (pct < 1.0) {
    x = startX + ((stopX-startX) * pct);
    y = startY + ((stopY-startY) * pct);
    pct += step;
  }
```

```
  ellipse(x, y, 20, 20);
}
```

Random

Unlike the smooth, linear motion common to computer graphics, motion in the physical world is usually idiosyncratic. For instance, think of a leaf floating to the ground, or an ant crawling over rough terrain. We can simulate the unpredictable qualities of the world by generating random numbers. The `random()` function calculates these values; we can set a range to tune the amount of disarray in a program.

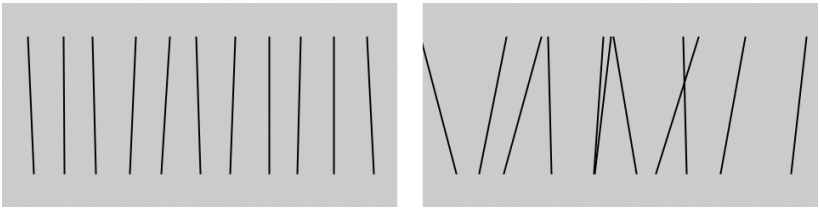
Example 8-7: Generate Random Values

The following short example prints random values to the Console, with the range limited by the position of the mouse. The `random()` function always returns a floating-point value, so be sure the variable on the left side of the assignment operator (`=`) is a `float` as it is here:

```
void draw() {
  float r = random(0, mouseX);
  println(r);
}
```

Example 8-8: Draw Randomly

The following example builds on Example 8-7 on page 109; it uses the values from `random()` to change the position of lines on screen. When the mouse is at the left of the screen, the change is small; as it moves to the right, the values from `random()` increase and the movement becomes more exaggerated. Because the `random()` function is inside the `for` loop, a new random value is calculated for each point of every line:



```
void setup() {
  size(240, 120);
}

void draw() {
  background(204);
  for (int x = 20; x < width; x += 20) {
    float mx = mouseX / 10;
    float offsetA = random(-mx, mx);
    float offsetB = random(-mx, mx);
    line(x + offsetA, 20, x - offsetB, 100);
  }
}
```

Example 8-9: Move Shapes Randomly

When used to move shapes around on screen, random values can generate images that are more natural in appearance. In the following example, the position of the circle is modified by random values on each trip through `draw()`. Because the `background()` function is not used, past locations are traced:

```
float speed = 2.5;
int diameter = 20;
float x;
float y;

void setup() {
  size(240, 120);
  x = width/2;
```

```
  y = height/2;
}

void draw() {
  x += random(-speed, speed);
  y += random(-speed, speed);
  ellipse(x, y, diameter, diameter);
}
```

If you watch this example long enough, you may see the circle leave the window and come back. This is left to chance, but we could add a few `if` structures or use the `constrain()` function to keep the circle from leaving the screen. The `constrain()` function limits a value to a specific range, which can be used to keep `x` and `y` within the boundaries of the Display Window. By replacing the `draw()` in the preceding code with the following, you'll ensure that the ellipse will remain on the screen:

```
void draw() {
  x += random(-speed, speed);
  y += random(-speed, speed);
  x = constrain(x, 0, width);
  y = constrain(y, 0, height);
  ellipse(x, y, diameter, diameter);
}
```

The `randomSeed()` function can be used to force `random()` to produce the same sequence of numbers each time a program is run. This is described further in the *Processing Reference*.

Timers

Every Processing program counts the amount of time that has passed since it was started. It counts in milliseconds (thousandths of a second), so after 1 second, the counter is at 1,000; after 5 seconds, it's at 5,000; and after 1 minute, it's at 60,000. We can use this counter to trigger animations at specific times. The `millis()` function returns this counter value.

Example 8-10: Time Passes

You can watch the time pass when you run this program:

```
void draw() {
  int timer = millis();
  println(timer);
}
```

Example 8-11: Triggering Timed Events

When paired with an `if` block, the values from `millis()` can be used to sequence animation and events within a program. For instance, after two seconds have elapsed, the code inside the `if` block can trigger a change. In this example, variables called `time1` and `time2` determine when to change the value of the `x` variable:

```
int time1 = 2000;
int time2 = 4000;
float x = 0;

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

void draw() {
  int currentTime = millis();
  background(204);
  if (currentTime > time2) {
    x -= 0.5;
  } else if (currentTime > time1) {
    x += 2;
  }
  ellipse(x, 60, 90, 90);
}
```

Circular

If you're a trigonometry ace, you already know how amazing the *sine* and *cosine* functions are. If you're not, we hope the next examples will trigger your interest. We won't discuss the math in detail here, but we'll show a few applications to generate fluid motion.

Figure 8-2 shows a visualization of sine wave values and how they relate to angles. At the top and bottom of the wave, notice how the rate of change (the change on the vertical axis) slows down, stops, then switches direction. It's this quality of the curve that generates interesting motion.

The `sin()` and `cos()` functions in Processing return values between –1 and 1 for the sine or cosine of the specified angle. Like `arc()`, the angles must be given in radian values (see Example 3-7 on page 18 and Example 3-8 on page 19 for a reminder of how radians work). To be useful for drawing, the `float` values returned by `sin()` and `cos()` are usually multiplied by a larger value.

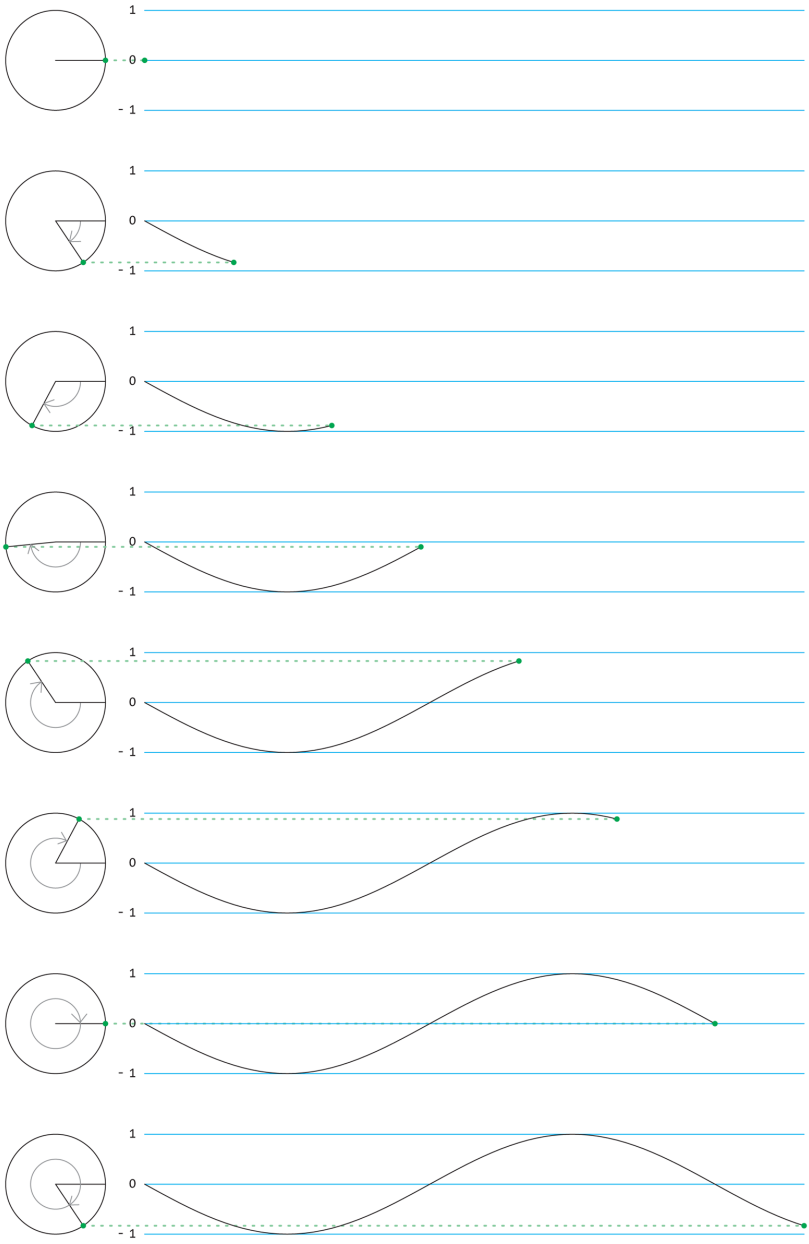


Figure 8-2. *A sine wave is created by tracing the sine values of an angle that moves around a circle*

Example 8-12: Sine Wave Values

This example shows how values for `sin()` cycle from –1 to 1 as the angle increases. With the `map()` function, the `sinval` variable is converted from this range to values from 0 and 255. This new value is used to set the background color of the window:

```
float angle = 0.0;

void draw() {
  float sinval = sin(angle);
  println(sinval);
  float gray = map(sinval, -1, 1, 0, 255);
  background(gray);
  angle += 0.1;
}
```

Example 8-13: Sine Wave Movement

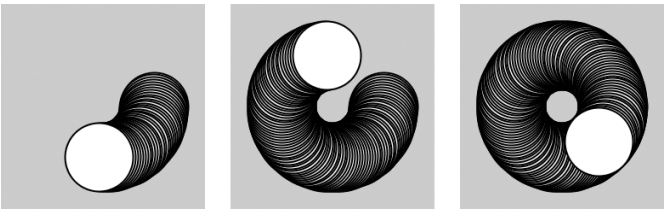
This example shows how these values can be converted into movement:

```
float angle = 0.0;
float offset = 60;
float scalar = 40;
float speed = 0.05;

void setup() {
  size(240, 120);
}

void draw() {
  background(0);
  float y1 = offset + sin(angle) * scalar;
  float y2 = offset + sin(angle + 0.4) * scalar;
  float y3 = offset + sin(angle + 0.8) * scalar;
  ellipse( 80, y1, 40, 40);
  ellipse(120, y2, 40, 40);
```

```
  ellipse(160, y3, 40, 40);
  angle += speed;
}
```

Example 8-14: Circular Motion

When `sin()` and `cos()` are used together, they can produce circular motion. The `cos()` values provide the *x* coordinates, and the `sin()` values provide the *y* coordinates. Both are multiplied by a variable named `scalar` to change the radius of the movement and summed with an offset value to set the center of the circular motion:

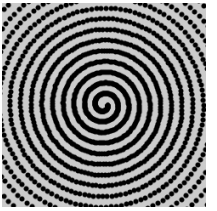


```
float angle = 0.0;
float offset = 60;
float scalar = 30;
float speed = 0.05;

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

void draw() {
  float x = offset + cos(angle) * scalar;
  float y = offset + sin(angle) * scalar;
  ellipse( x, y, 40, 40);
  angle += speed;
}
```

Example 8-15: Spirals

A slight change made to increase the `scalar` value at each frame produces a spiral, rather than a circle:





```
float angle = 0.0;
float offset = 60;
float scalar = 2;
float speed = 0.05;

void setup() {
  size(120, 120);
  fill(0);
}

void draw() {
  float x = offset + cos(angle) * scalar;
  float y = offset + sin(angle) * scalar;
  ellipse( x, y, 2, 2);
  angle += speed;
  scalar += speed;
}
```