```
float y = 20;
line(0, y, width, y);
y = y * 1.6;  // Assign 32.0 to y
line(0, y, width, y);
y = y * 1.6;  // Assign 51.2 to y
line(0, y, width, y);
y = y * 1.6;  // Assign 81.920006 to y
line(0, y, width, y);
```

4-07

The +, -, *, /, and = symbols are probably familiar, but the % is more exotic. The % operator calculates the remainder when one number is divided by another. The %, the code notation for modulus, returns the integer remainder after dividing the number to the left of the symbol by the number to the right.

Expression	Result	Explanation
9 % 3	0	3 goes into 9 three times, with no remainder
9 % 2	1	2 goes into 9 four times, with 1 as the remainder
35 % 4	3	4 goes into 35 eight times, with 3 remaining

Modulus can be explained with an anecdote. After a hard day of work, Casey and Ben were hungry. They went to a restaurant to eat dumplings, but there were only 9 dumplings left so they had to share. If they share equally, how many dumplings can they each eat and how many will remain? It's obvious each can have 4 dumplings and 1 will remain. If there are 4 people and 35 dumplings, each can eat 8 and 3 will remain. In these examples, the modulus value is the number of remaining dumplings.

The modulus operator is often used to keep numbers within a desired range. For example, if a variable is continually increasing (0, 1, 2, 3, 4, 5, 6, 7, etc.), applying the modulus operator can transform this sequence. A continually increasing number can be made to cycle continuously between 0 and 3 by applying %4:

x	0	1	2	3	4	5	6	7	8	9	10
x % 4	0	1	2	3	0	1	2	3	0	1	2

Many examples throughout this book use % in this way.

When working with mathematical operators and variables, it's important to be aware of the data types of the variables you're using. The combination of two integers will always result in an `int`. The combination of two floating-point numbers will always result in a `float`, but when an `int` and `float` are operated on, the result is a `float`.

```
println(4/3);     // Prints "1"
println(4.0/3);   // Prints "1.3333334"
println(4/3.0);   // Prints "1.3333334"
println(4.0/3.0); // Prints "1.3333334"
```

4-08

Structure 2: Continuous

The unit introduces programs that run continuously and explains how to control their speed.

Syntax introduced:
`draw(), frameRate(), frameCount, setup(), noLoop()`

All the programs in preceding units run their code once and the program stops. Programs that animate or respond to live information must run continuously. Continuously running programs can create animation or use the mouse and keyboard for input.

Continuous evaluation

Programs that run continuously must include a `draw()` function. The code inside a `draw()` block runs in a loop until the the stop button is pressed or the window is closed. A program can have only *one* `draw()`. Each time the `draw()` function finishes, it draws a new frame to the display window and then starts running the block again from the first line.

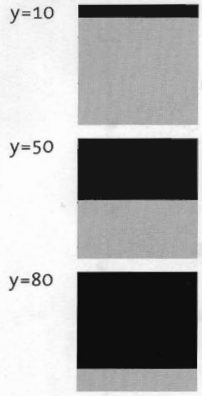
By default, frames are drawn to the screen at 60 frames per second (fps). The `frameRate()` function changes and controls the number of frames displayed each second. The program will always attempt to run at the speed set by the parameter to the `frameRate()` function, but sometimes the ambitions of the programmer exceed the speed of the computer. The `frameRate()` function controls only the maximum frame rate—it can not speed up a program that runs slowly because of equipment limitations.

The `frameCount` variable always contains the number of frames displayed since the program started. A program with `draw()` keeps displaying frames (1, 2, 3, 4, 5, . . .) until it is stopped, the computer is shut down, or the power goes out.

```
// Prints each frame number to the console                    20-01
void draw() {
  println(frameCount);
}
```

```
// Runs at around 4 fps, prints each frame number to the console  20-02
void draw() {
  frameRate(4);
  println(frameCount);
}
```

Changing visual elements from frame to frame creates animation. For example, changing the position of a line each frame will cause it to move:




```
float y = 0.0;                                    20-03

void draw() {
  frameRate(30);
  line(0, y, 100, y);
  y = y + 0.5;
}
```

When this code runs, the variables are replaced with their current values and the statements are run in this order:

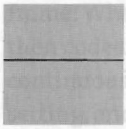
```
float y = 0.0
                                    Enter draw()
frameRate(30)
line(0, 0.0, 100, 0.0)
y = 0.5
                                    Enter draw() for the second time
frameRate(30)
line(0, 0.5, 100, 0.5)
y = 1.0
                                    Enter draw() for the third time
frameRate(30)
line(0, 1.0, 100, 1.0)
y = 1.5
Etc...
```

The variable y must be declared outside `draw()` for this program to move the line each frame. If the variable is declared inside `draw()`, it will be re-created each time the `draw()` block is run and reassigned to the same value, placing the line in the same position.

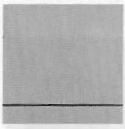
The background of the display window does not refresh automatically, so lines will simply accumulate. To clear the display window at each frame, insert a `background()` function at the beginning of the `draw()` function. The `background()` function fills the entire display window with the specified color. It overwrites every pixel in the display window each time it is run. If the `background()` is not placed at the top of `draw()`, it will cover any element drawn earlier.

y=10

y=46



y=84



20-04

```
float y = 0.0;

void draw() {
  frameRate(30);
  background(204);
  y = y + 0.5;
  line(0, y, 100, y);
}
```

The variable that controls the line position can be used for other purposes. In the next example, it's also used to set the color of the background.

y=20

y=49

y=88

20-05

```
float y = 0.0;

void draw() {
  frameRate(30);
  background(y * 2.5);
  y = y + 0.5;
  line(0, y, 100, y);
}
```

After a few seconds, the line moves off the bottom edge of the display window. An `if` structure can reset the value of the variable so the line position is set back to the top:

y=30

y=88

y=19

20-06

```
float y = 0.0;

void draw() {
  frameRate(30);
  background(y * 2.5);
  y = y + 0.5;
  line(0, y, 100, y);
  if (y > 100) {
    y = 0;
  }
}
```

Drawing 1: Static Forms

This unit discusses drawing in relation to software and presents code for basic drawing programs.

The activity of drawing translates an individual's perception and imagination into visual form. The differences between the drawings of different people demonstrates the fact that every hand and mind is unique. Drawings range from the mechanical grids of Sol LeWitt to the playful lines of Paul Klee to the expressionist figures of Egon Schiele and far beyond. Each surface and instrument offers a different tactile experience. Ink, charcoal, crayon, vellum, and cloth all enable unique sorts of drawing. Less conventional materials, such as chocolate, dirt, glue, and light, have been used with significant results. Materials can be applied with fingers, toes, or elbows or with utensils like pencils, brushes, and sticks. Every choice of material and application influences the viewer's perception of what the composition communicates.

The idea of drawing with computers dates back to the 1960s. Ivan Sutherland created the remarkable Sketchpad software for his PhD dissertation in 1963. Sketchpad, the progenitor of computer-aided drawing software (CAD) such as Autodesk's AutoCAD and Adobe Illustrator, used the newly invented light pen input device that made it possible to draw directly to the screen. The software had features to convert imprecise marks into perfect straight lines, arcs, and circles. It could also constrain marks to make them identical, parallel, or perpendicular. Most example drawings demonstrated the features and accuracy of the system for making technical drawings, but Sutherland also discussed the use of his software for other purposes and created an example of an animated portrait.

Logo is another innovative software drawing system with origins in the 1960s. Seymour Papert developed Logo's turtle graphics as a way to get children thinking about geometry. Logo uses text commands to control a turtle on the screen that leaves a trail as it travels. The command `RT 90` turns the turtle 90° to the right, and `FD 100` moves the turtle forward 100 units. One early Logo implementation employed a robotic turtle named Irving that moved around the room according to the children's instructions.

In contrast to the interactive approach of Sketchpad and Logo, most early software drawing systems translated input directly from code to paper. Software drawing pioneers of the early 1960s included A. Michael Noll, Frieder Nake, Georg Nees, and Charles Csuri. Their images were realized with computer-driven plotters, a common output device of that time. A plotter is a pen attached to a moving mechanical arm controlled by motors through a computer. Many drawings from this period utilized the technology in a way that showcased the precision of the tools. Another wave of individuals utilizing software plotters emerged in the 1970s and 1980s. These artists included Manfred Mohr, Jean-Pierre Hébert, and Mark Wilson.

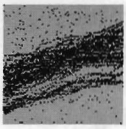
Bridging the era of plotters to present-day technologies, Harold Cohen's *AARON* is arguably the most sophisticated drawing software ever written. The software has undergone continuous development since it was first created in 1973. *AARON*'s drawings have been featured in some of the world's most prominent museums, including the Tate Gallery in London and the Stedelijk Museum in Amsterdam. *AARON* initially created abstract drawings and over the years has been refined to add rocks, then plants, and finally people. Cohen has encoded his ideas about drawing as a set of rules that comprise the *AARON* software. The program operates autonomously and makes a unique drawing each time it is run. The software makes every composition decision and produces drawings that often surprise Cohen.

Contemporary artists continue to write innovative software to enable unique approaches to drawing. Input tools can severely limit drawing with software. The quality of drawing with a device such as the mouse is constrained by the small amount of information transferred from the hand into the software. The hand, with its strength and flexibility, is capable of gestures of the smallest nuance in pressure and direction, but the mouse receives only position information. Such limitations were diminished with the introduction of drawing tablets and stylus devices capable of reading pressure and direction, but no matter how much these devices improve, they will only approximate the quality of using physical instruments and media.

Rather than applying a "traditional" model of drawing to the software medium, another approach is to address the possibilities tangential to the constraints. When using a new medium, why constrain oneself to imitating other media? In the context of this book, we consider the potential of software in contrast to physical media and address drawing methods that are unique to software.

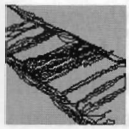
Simple tools

The easiest way to draw with Processing is to not include the `background()` function inside `draw()`. This omission allows the display window to accumulate pixels from frame to frame.



24-01

```
// Draw dots at the position of the cursor
void setup() {
  size(100, 100);
}
void draw() {
  point(mouseX, mouseY);
}
```



24-02

```
// Draw from the previous mouse location to the current
// mouse location to create a continuous line

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

void draw() {
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

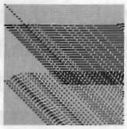
24-03

```
// Draw a line only when a mouse button is pressed

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

void draw() {
  if (mousePressed == true) {
    line(mouseX, mouseY, pmouseX, pmouseY);
  }
}
```

24-04

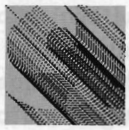
```
// Draw lines with different gray values when a mouse
// button is pressed or not pressed

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

void draw() {
  if (mousePressed == true) {  // If mouse is pressed,
    stroke(255);               // set the stroke to white
  } else {                     // Otherwise,
    stroke(0);                 // set to black
  }
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

Drawing with software is not restricted to making a single mark that follows the cursor. A `for` structure makes it possible to create more complex drawings with just a few lines of code. The following examples use a `for` structure to draw many elements to the screen at each frame.



24-05

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

void draw() {
  for (int i = 0; i < 50; i += 2) {
    point(mouseX+i, mouseY+i);
  }
}
```



24-06

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

void draw() {
  for (int i = -14; i <= 14; i += 2) {
    point(mouseX+i, mouseY);
  }
}
```

24-07

```
void setup() {
  size(100, 100);
  noStroke();
  fill(255, 40);
  background(0);
}

void draw() {
  if (mousePressed == true) {
    fill(0, 26);
  } else {
    fill(255, 26);
  }
  for (int i = 0; i < 6; i++) {
    ellipse(mouseX + i*i, mouseY, i, i);
  }
}
```

Drawing with images

Images can also be used as drawing tools. If an image is positioned in relation to the cursor at each frame, its pixels can be used to create visually complex compositions. In the following examples, the image follows the position of the cursor.

24-08

```
// Draw with an image sliver

PImage lineImage;

void setup() {
  size(100, 100);
  // This image is 100 pixels wide, but one pixel tall
  lineImage = loadImage("imageline.jpg");
}

void draw() {
  image(lineImage, mouseX-lineImage.width/2, mouseY);
}
```

24-09

```
// Draw with an image that has transparency

PImage alphaImg;

void setup() {
  size(100, 100);
  // This image is partially transparent
  alphaImg = loadImage("alphaArch.png");
}

void draw() {
  int ix = mouseX - alphaImg.width/2;
  int iy = mouseY - alphaImg.height/2;
  image(alphaImg, ix, iy);
}
```

Exercises

1. *Make a custom drawing tool that changes its color when a mouse button is pressed.*
2. *Make a custom drawing tool that changes its form when a mouse button is pressed.*
3. *Load an image and use it as a drawing tool.*

Image 2: Animation

This unit introduces techniques for displaying sequences of images successively, creating animation.

Animation occurs when a series of images, each slightly different, are presented in quick succession. A diverse medium with a century of history, animation has progressed from the initial experiments of Winsor McCay to the commercial and realistic innovations of early Walt Disney studio productions, to the experimental films by such animators as Lotte Reiniger and James Whitney in the mid-twentieth century. The high volume of animated special effects in live-action film and the deluge of animated children's films are changing the role of animation in popular culture.

There's a long history of using software to extend the boundaries of animation. Some of the first computer graphics were presented as animation on film during the 1960s. Because of the cost and expertise required to make these films, they emerged from high-profile research facilities such as Bell Laboratories and IBM's Scientific Center. Kenneth C. Knowlton, then a researcher at Bell Labs, is an important protagonist in the story of early computer animation. He worked separately with artists Stan VanDerBeek and Lillian Schwartz to produce some of the first films made using computer graphics. VanDerBeek and Knowlton's *Poem Field* films, produced throughout the 1960s, utilized Knowlton's BEFLIX code and punch cards to produce permutations of visual micropatterns. Schwartz and Knowlton's *Pixillation* (1970) featured a wide range of effects made by contrasting geometric forms with organic motion. John Whitney worked in collaboration with Jack Citron at IBM to make a number of films including the innovative *Permutations*. This film expresses Whitney's ideas about relationships to music and abstract form by permuting an array of dots into infinite kinetic patterns. Other artists working with computer animation around this time were Larry Cuba, Peter Foldes, and John Stehura.

The paths of contemporary animation and software development often overlap. The 3D visualization of the Death Star in *Star Wars* (1977) was one of the first uses of computer-generated animation in a feature film. Custom software was written to produce a wire-frame fly-through of the massive ship. Interest in computer animation briefly peaked with Disney's *Tron* in 1982, but soon receded due to the film's commercial failure. The industry gradually rebuilt itself into its current role as a major force in contemporary film. Pixar, the hugely successful animation studio that produced *Toy Story* and *The Incredibles*, operated for many years as a software development company. Pixar's RenderMan software (1989) enabled the rendering of 3D computer graphics as photorealistic images. RenderMan became an industry standard and Pixar continues to develop custom software for each film. The success of the company's films reflects its successful marriage of technical virtuosity and masterful storytelling.

Creating unique and experimental animation with software is no longer restricted

to research labs and film studios. The Internet has become a vast repository for experimental software animation. In the late 1990s, *Turux* was created by Lia and Dextro. This online collection of intricate animated images and sounds synthesizes a digital glitch aesthetic with organic qualities. The drawings continually change and sometimes respond to viewer input.

James Paterson (p. 165), a Canadian animator, develops *Presstube.com*, where he produces thousands of drawings, typically organizing tight loops of elements that materialize and dissipate. This technique allows him to arrange these loops in nearly any order while maintaining a fluid progression of growth and decay. The sequences of David Crawford's *Stop Motion Studies* are series of photographs—typically of people in subways in different cities around the world. These photographs are taken in quick succession; presented again in a nonlinear sequence of animated frames, they reveal an incredibly complex and subtle range of human gesture.

Sequential images

Before a series of images can be presented sequentially in a program, all the images must first be loaded. The image variables should be declared outside of `setup()` and `draw()` and then assigned within `setup()`. The following program loads these twelve images from James Paterson . . .

. . . and then draws them to the display window in numeric order.

frame=0

frame=3

frame=11

```
int numFrames = 12;  // The number of animation frames
int frame = 0;  // The frame to display
PImage[] images = new PImage[numFrames];  // Image array

void setup() {
  size(100, 100);
  frameRate(30);  // Maximum 30 frames per second
  images[0] = loadImage("ani-000.gif");
  images[1] = loadImage("ani-001.gif");
  images[2] = loadImage("ani-002.gif");
  images[3] = loadImage("ani-003.gif");
  images[4] = loadImage("ani-004.gif");
  images[5] = loadImage("ani-005.gif");
  images[6] = loadImage("ani-006.gif");
  images[7] = loadImage("ani-007.gif");
  images[8] = loadImage("ani-008.gif");
  images[9] = loadImage("ani-009.gif");
```

```
    images[10] = loadImage("ani-010.gif");
    images[11] = loadImage("ani-011.gif");
  }

  void draw() {
    frame++;
    if (frame == numFrames) {
      frame = 0;
    }
    image(images[frame], 0, 0);
  }
```

34-01 cont.

The next example shows an alternative way of loading images by utilizing a `for` structure. These lines of code can load between 1 and 999 images by changing the value of the `numFrames` variable. This shortens the code that flips through each image and returns to the first image at the end of the animation. The `nf()` function (p. 422) on line 11 is used to format the name of the image to be loaded. The names of frames with small numbers are prefaced with zeros so that the images remain in the correct sequence in their folder. For example, instead of *ani-1.gif*, the file is named *ani-001.gif*. The `nf()` function pads the small numbers created in a `for` structure with zeros on the left of the number, so 1 becomes 001, 2 becomes 002, etc. The % operator (p. 45) on line 18 uses the `frameCount` variable to make the `frame` variable increase by 1 each frame and return to 0 once it exceeds 11.

34-02

```
int numFrames = 12;  // The number of animation frames
PImage[] images = new PImage[numFrames];  // Image array

void setup() {
  size(100, 100);
  frameRate(30);  // Maximum 30 frames per second
  // Automate the image loading procedure. Numbers less than 100
  // need an extra zero added to fit the names of the files.
  for (int i = 0; i < images.length; i++) {
    // Construct the name of the image to load
    String imageName = "ani-" + nf(i, 3) + ".gif";
    images[i] = loadImage(imageName);
  }
}

void draw() {
  // Calculate the frame to display, use % to cycle through frames
  int frame = frameCount % numFrames;
  image(images[frame], 0, 0);
}
```

Displaying the images in random order and for different amounts of time enhances the visual interest of a few frames of animation. Replaying a sequence at irregular intervals, in a random order with random timing, can give the appearance of more different frames than actually exist.

34-03

```
int numFrames = 5;  // The number of animation frames
PImage[] images = new PImage[numFrames];

void setup() {
  size(100, 100);
  for (int i = 0; i < images.length; i++) {
    String imageName = "ani-" + nf(i, 3) + ".gif";
    images[i] = loadImage(imageName);
  }
}

void draw() {
  int frame = int(random(0, numFrames));  // The frame to display
  image(images[frame], 0, 0);
  frameRate(random(1, 60.0));
}
```

There are many ways to control the speed at which an animation plays. The `frameRate()` function provides the simplest way. Place the `frameRate()` function in `setup()` as seen in the previous example. Use this function to ensure that the software will run at the same speed on other machines.

If you want other elements to move independently of the sequential images, set up a timer and advance the frame only when the timer value grows larger than a predefined value. In the following example, the animation playing in the top of the window is updated each frame and the speed is controlled by the parameter to `frameRate()`. The animation in the bottom frame is updated only twice a second; the timer checks the milliseconds since the last update and changes the frame only if 500 milliseconds (half a second) have elapsed.

34-04

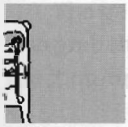
```
int numFrames = 12;    // The number of animation frames
int topFrame = 0;      // The top frame to display
int bottomFrame = 0;   // The bottom frame to display
PImage[] images = new PImage[numFrames];
int lastTime = 0;

void setup() {
  size(100, 100);
  frameRate(30);
  for (int i = 0; i < images.length; i++) {
```

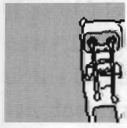
```
    String imageName = "ani-" + nf(i, 3) + ".gif";
    images[i] = loadImage(imageName);
  }
}

void draw() {
  topFrame = (topFrame + 1) % numFrames;
  image(images[topFrame], 0, 0);
  if ((millis() - lastTime) > 500) {
    bottomFrame = (bottomFrame + 1) % numFrames;
    lastTime = millis();
  }
  image(images[bottomFrame], 0, 50);
}
```

34-04 cont.

Images in motion

Moving one image, rather than presenting a sequence, is another approach to animating images. The same techniques for creating movement presented in Motion 1 (p. 279) apply to images. The following example moves an image from left to right, returning it to the left when it disappears off the edge of the screen.





34-05

```
PImage img;
float x;

void setup() {
  size(100, 100);
  img = loadImage("PT-Shifty-0020.gif");
}

void draw() {
  background(204);
  x += 0.5;
  if (x > width) {
    x = -width;
  }
  image(img, x, 0);
}
```

The transformation functions also apply to images. They can be translated, rotated, and scaled over time to produce motion. In this example, an image is drawn to the center of the display window and rotated slowly around its center.



```
PImage img;
float angle;

void setup() {
  size(100, 100);
  img = loadImage("PT-Shifty-0023.gif");
}

void draw() {
  background(204);
  angle += 0.01;
  translate(50, 50);
  rotate(angle);
  image(img, -100, -100);
}
```

Images can also be animated by changing their drawing attributes. In this example, the opacity of an image is increased so that it is brought into view over the background.



```
PImage img;
float opacity = 0;      // Set opacity to the minimum

void setup() {
  size(100, 100);
  img = loadImage("PT-Teddy-0017.gif");
}

void draw() {
  background(0);
  if (opacity < 255) {  // When less than the maximum,
    opacity += 0.5;     // increase opacity
  }
  tint(255, opacity);
  image(img, -25, -75);
}
```

Exercises

1. *Load a sequence of related images into an array and use them to create a linear animation.*
2. *Modify the program for exercise 1 to present each frame of animation at a different rate and in a different sequence.*
3. *Animate an image by changing more than one of its attributes (e.g., size, position, tint).*

Image 3: Pixels

This unit introduces techniques for getting and setting the values for single pixels and groups of pixels.

Syntax introduced:
`get(), set()`

Image 1 (p. 95) defined an image as a rectangular grid of pixels in which each element is a number specifying a color. Because the screen itself is an image, its individual pixels are also defined as numbers. The color values of individual pixels can be read and changed.

Reading pixels

When a Processing program starts, the display window opens at the dimension requested in `size()`. The program gains control over that area of the screen and sets the color value for each pixel. The display window communicates with the operating system, so when the window moves, it takes control of its new area of the screen and gives control of its previous space to the operating system.

The `get()` function can read the color of any pixel in the display window. It can also grab the whole display window or a section of it. There are three versions of this function, one for each use.

```
get()
get(x, y)
get(x, y, width, height)
```

If `get()` is used without parameters, a copy of the entire display window is returned as a `PImage`. The version with two parameters returns the color value of a single pixel at the location specified by the *x* and *y* parameters. A rectangular area of the display window is returned if the additional *width* and *height* parameters are used. If `get()` grabs the entire display window or a section of the window, the returned data must be assigned to a variable of type `PImage`. These images can be redrawn to the screen in different positions and resized.

35-01

```
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
PImage cross = get();    // Get the entire window
image(cross, 0, 50);     // Draw the image in a new position
```

35-02

```
smooth();
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
noStroke();
ellipse(18, 50, 16, 16);
PImage cross = get();              // Get the entire window
image(cross, 42, 30, 40, 40);  // Resize to 40 x 40 pixels
```

35-03

```
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
PImage slice = get(0, 0, 20, 100);  // Get window section
set(18, 0, slice);
set(50, 0, slice);
```

The `get()` function always grabs the pixels in the display window in the same way, regardless of what is drawn to the window. In the previous examples, the `get()` function grabbed the images of the lines after they had been converted to pixels for display on screen. The following example is the same as code 35-01, but a photograph is first loaded into the display window, so `get()` grabs that image.

35-04

```
PImage trees;
trees = loadImage("topanga.jpg");
image(trees, 0, 0);
PImage crop = get();  // Get the entire window
image(crop, 0, 50);   // Draw the image in a new position
```

When used with an x- and y-coordinate, the `get()` function returns values that should be assigned to a variable of the `color` data type. These values can be used to set the color of other pixels or can serve as parameters to `fill()` or `stroke()`. In the following example, the color of one pixel is used to set the color of the rectangle.

35-05

```
PImage trees;
trees = loadImage("topanga.jpg");
noStroke();
image(trees, 0, 0);
color c = get(20, 30);  // Get color at (20, 30)
fill(c);
rect(20, 30, 40, 40);
```

The mouse values can be used as the parameters to the `get()` function. This allows the cursor to select colors from the display window. In the following example, the pixel beneath the cursor is read and defines the fill value for the rectangle on the right.



35-06

```
PImage trees;

void setup() {
  size(100, 100);
  noStroke();
  trees = loadImage("topangaCrop.jpg");
}

void draw() {
  image(trees, 0, 0);
  color c = get(mouseX, mouseY);
  fill(c);
  rect(50, 0, 50, 100);
}
```

The `get()` function can be used within a `for` structure to grab many pixels or groups of pixels. In the following example, the values from each row of pixels in the image are used to set the values for the lines on the right. Run this code and move the mouse up and down to see the relation between the image on the left and the bands of color on the right.



35-07

```
PImage trees;
int y = 0;

void setup() {
  size(100, 100);
  trees = loadImage("topangaCrop.jpg");
}

void draw() {
  image(trees, 0, 0);
  y = constrain(mouseY, 0, 99);
  for (int i = 0; i < 49; i++) {
    color c = get(i, y);
    stroke(c);
    line(i+50, 0, i+50, 100);
  }
  stroke(255);
  line(0, y, 49, y);
}
```

Every PImage variable has its own get() to grab pixels from the image. This allows pixels to be grabbed from an image independently of the pixels in the display window. Because a PImage is an object, the get() function is accessed with the name of the image and the dot operator. In the following example, the pixels are grabbed directly from the image and not from the screen, so white lines drawn to the display window are not grabbed.

35-08

```
PImage trees;
trees = loadImage("topanga.jpg");
stroke(255);
strokeWeight(12);
image(trees, 0, 0);
line(0, 0, width, height);
line(0, height, width, 0);
PImage treesCrop = trees.get(20, 20, 60, 60);
image(treesCrop, 20, 20);
```

Writing pixels

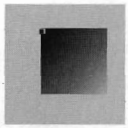
The pixels in Processing's display window can be written directly with the set() function. There are two versions of this function, each with three parameters.

```
set(x, y, color)
set(x, y, image)
```

When the third parameter is a color, set() changes the color of any pixel in the display window. When the third parameter is an image, set() writes an image at the coordinates specified by the *x* and *y* parameters.

35-09

```
color black = color(0);
set(20, 80, black);
set(20, 81, black);
set(20, 82, black);
set(20, 83, black);
```



35-10

```
for (int i = 0; i < 55; i++) {
  for (int j = 0; j < 55; j++) {
    color c = color((i+j) * 1.8);
    set(30+i, 20+j, c);
  }
}
```

The `set()` function can write an image to the display window at any location. Using `set()` to draw an image is faster than using the `image()` function because the pixels are copied directly. However, images drawn with `set()` cannot be resized or tinted, and they are not affected by the transformation functions.

35-11

```
PImage trees;

void setup() {
  size(100, 100);
  trees = loadImage("topangaCrop.jpg");
}

void draw() {
  int x = constrain(mouseX, 0, 50);
  set(x, 0, trees);
}
```

Every `PImage` variable has its own `set()` function to write pixels directly to the image. This allows pixels to be written to an image independently of the pixels in the display window. Because a `PImage` is an object, the `set()` function is run with the name of the image and the dot operator. In the following example, four white pixels are set into the image variable `trees`, and the image is then drawn to the display window.

35-12

```
PImage trees;
trees = loadImage("topangaCrop.jpg");
background(0);
color white = color(255);
trees.set(0, 50, white);
trees.set(1, 50, white);
trees.set(2, 50, white);
trees.set(3, 50, white);
image(trees, 20, 0);
```

Exercises

1. *Load an image and use `get()` to create a collage by overlaying different sections of the same image.*
2. *Load an image and use `mouseX` and `mouseY` to read the value of the pixel beneath the cursor. Use this value to change some aspect of the image.*
3. *Draw a shape in the display window. Copy a section of the window to another by using `get()` and `set()` within a `for` structure.*

Image 4: Filter, Blend, Copy, Mask

This unit introduces techniques for filtering, blending, copying, and masking images.

Syntax introduced:
`filter(), blend(), blendColor(), copy(), mask()`

Digital images have the remarkable potential to be easily reconfigured and combined with other images. Software now simulates and improves upon complex and time-consuming operations formerly completed in a darkroom with light and chemistry. Every pixel in a digital image is a grouping of numbers that can be added, multiplied, or averaged with the numbers from any other pixel. Some of these calculations are based on simple arithmetic and others use the more complex mathematics of signal processing, but the visual results are most important. Software programs such as the GNU Image Manipulation Program (GIMP) and Adobe's Photoshop have made it possible to perform many of the more common and useful calculations without thinking about the math behind the effects. These programs allow users to easily perform technical operations such as converting images from RGB colors to grayscale values, increasing an image's contrast, or tweaking color balance. Such tools also allow users to apply filters that range from the basic to the kitschy and absurd. A filter might blur an image, mimic solarization, or simulate watercolor effects. The actions of filtering, blending, and copying can easily be controlled with code to produce striking changes. These techniques may be too slow for use in real-time animation.

Filtering, Blending

Processing provides functions to filter and blend images in the display window. Each of these functions operates by transforming the pixel values of a single image or by performing an operation to merge pixels between two different images. The `filter()` function has two prototypes:

```
filter(mode)
filter(mode, level)
```

Eight options exist for the *mode* parameter: THRESHOLD, GRAY, INVERT, POSTERIZE, BLUR, OPAQUE, ERODE, or DILATE. Some of these parameters require the *level* parameter and others don't. For example, the THRESHOLD mode converts every pixel in an image to black or white based on whether its value is above or below the value of the *level* parameter.

The following example applies the THRESHOLD filter to an image with the *level* parameter set to 0.3, which signifies that pixels with a gray value greater than 30 percent

BLUR, 1

BLUR, 4

BLUR, 8

BLUR
Executes a Gaussian blur with the level parameter specifying the extent of the blurring

POSTERIZE, 2

POSTERIZE, 4

POSTERIZE, 8

POSTERIZE
Limits each channel of the image to the number of colors specified as the level parameter

THRESHOLD, 0.2

THRESHOLD, 0.5

THRESHOLD, 0.8

THRESHOLD
Converts the image to black-and-white pixels depending on whether they are above or below the threshold defined by the level parameter

INVERT
Sets each pixel to its inverse value

GRAY
Converts any colors in the image to grayscale equivalents

ERODE
Reduces the light areas with the amount defined by the level parameter

DILATE
Increases the light areas with the amount defined by the level parameter

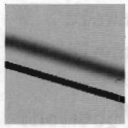
Filtering
The `filter()` function modifies the pixels of the display window and images. The different kinds of filters seen here provide a range of ready-made options, but it's possible to write custom filters using the language elements introduced in Image 5 (p. 355).

of the maximum brightness will be set to white and pixels below that value will be set to black.

```
PImage img = loadImage("topanga.jpg");
image(img, 0, 0);
filter(THRESHOLD, 0.3);
```

39-01

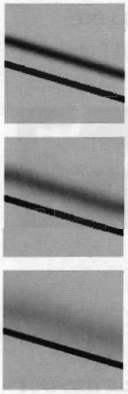
The `filter()` function affects only what has already been drawn. For example, if a program draws two lines and blur is created after one line is drawn, it does not affect the second line:



```
smooth();
strokeWeight(5);
noFill();
line(0, 30, 100, 60);
filter(BLUR, 3);
line(0, 50, 100, 80);
```

39-02

Changing the parameter value of `filter()` with each frame creates movement. The effects of `filter()` are reset each time through `draw()`, but increasing or decreasing the *`level`* parameter results in the filter becoming more or less pronounced as the program runs:



```
float fuzzy = 0.0;

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(5);
  noFill();
}

void draw() {
  background(204);
  if (fuzzy < 16.0) {
    fuzzy += 0.05;
  }
  line(0, 30, 100, 60);
  filter(BLUR, fuzzy);
  line(0, 50, 100, 80);
}
```

39-03

ADD
Additive blending with maximum value of white:
`C = min(A*factor + B, 255)`

SUBTRACT
Subtractive blending with minimum value of black:
`C = max(B - A*factor, 0)`

LIGHTEST
The lightest color is used:
`C = max(A*factor, B)`

DARKEST
The darkest color is used:
`C = min(A*factor, B)`

MULTIPLY
Multiply the colors; result will always be darker:
`C = A * B`

A B C

Blending

The `blend()` function combines two images. Different modes blend in different ways. The equations shown with each description mathematically define each blending technique. The letters A and B are the pixels of the source images, and C is the pixels of the resulting image. The factor is the alpha component (transparency) of the source image. Additional blend modes are documented in the Processing reference.

The PImage class has a `filter()` method that can isolate filtering to a specific image. The following examples show how to use this method on individual images without affecting the display window.

```
PImage img = loadImage("forest.jpg");
image(img, 0, 0);
img.filter(INVERT);
image(img, 50, 0);
```

39-04

The `blend()` function mixes pixels in different ways depending on the mode parameter. The `blend()` function has two different versions.

```
blend(x, y, width, height, dx, dy, dwidth, dheight, mode)
blend(srcImg, x, y, width, height, dx, dy, dwidth, dheight, mode)
```

The *mode* parameter can be BLEND, ADD, SUBTRACT, DARKEST, LIGHTEST, DIFFERENCE, EXCLUSION, MULTIPLY, SCREEN, OVERLAY, HARD_LIGHT, SOFT_LIGHT, DODGE, and BURN. The *x* and *y* parameters are the x- and y-coordinates of the region to copy. The *width* and *height* parameters set the size of the source area. The *dx* and *dy* parameters are the x- and y-coordinate of the destination area. The *dwidth* and *dheight* are the width and height of the destination area. To blend between two images instead of the display window, a second image can be used as the *srcImg* parameter. If the source and destination regions are different sizes, the pixels will be automatically resized to fit the specified target region.

You can blend the image in the display window with itself using any of the different modes. The next example demonstrates blending the window using the ADD mode.

```
background(0);
stroke(153);
strokeWeight(24);
smooth();
line(44, 0, 24, 80);
line(0, 24, 80, 44);
blend(0, 0, 100, 100, 16, 0, 100, 100, ADD);
```

39-05

You can also blend imported images with the display window by including a source image as the first parameter to blend. In this example, the image is seen only through the drawn lines because the background was set to black and the *mode* parameter is DARKEST.

```
PImage img = loadImage("topanga.jpg");
background(0);
stroke(255);
strokeWeight(24);
```

39-06

```
smooth();
line(44, 0, 24, 80);
line(0, 24, 80, 44);
blend(img, 0, 0, 100, 100, 0, 0, 100, 100, DARKEST);
```

39-06 cont.

The PImage class has a blend() method that can be used to blend an image or two images together without affecting the display window. The following example blends the center of the image *forest.jpg* with the center of *airport.jpg* and then displays the modified image variable to the display window.

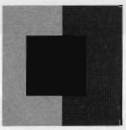
```
PImage img = loadImage("forest.jpg");
PImage img2 = loadImage("airport.jpg");
img.blend(img2, 12, 12, 76, 76, 12, 12, 76, 76, ADD);
image(img, 0, 0);
```

39-07

The blendColor() function is used to blend individual color values.

```
blendColor(c1, c2, mode)
```

The *c1* and *c2* parameters are the color values that create a new color when blended together. The options for the *mode* parameter are the same as the options for the blend() function. Because this unit is printed in black and white, it's not possible to use examples with color values, so the effect is demonstrated in the following example by using gray values.



```
color g1 = color(102);  // Middle gray
color g2 = color(51);   // Dark gray
color g3 = blendColor(g1, g2, MULTIPLY);  // Create black
noStroke();
fill(g1);
rect(50, 0, 50, 100);  // Right rect
fill(g2);
rect(20, 25, 30, 50);  // Left rect
fill(g3);
rect(50, 25, 20, 50);  // Overlay rect
```

39-08

The Processing language includes syntax that makes it easy to write additional custom filters and blending operations. These actions are is discussed further in ƒa (p. 337).

Copying pixels

The `copy()` function has two versions, each of which has a large number of parameters:

```
copy(x, y, width, height, dx, dy, dwidth, dheight)
copy(srcImg, x, y, width, height, dx, dy, dwidth, dheight)
```

The version of `copy()` with eight parameters replicates a region of pixels from the display window in another area of the display window. The version with nine parameters copies all or a portion of the image specified by the *`srcImg`* parameter into the display window. If the source and destination regions are of different sizes, the pixels will automatically be resized to fit the destination width and height. The other parameters are the same as described for `blend()` (p. 351). The `copy()` function differs from the previously discussed `get()` and `set()` functions because it can both get pixels from one location and set them to another. The following two examples demonstrate the function.

39-09

```
PImage img = loadImage("forest.jpg");
image(img, 0, 0);
copy(0, 0, 100, 50, 0, 50, 100, 50);
```

39-10

```
PImage img1, img2;

void setup() {
  size(100, 100);
  img1 = loadImage("forest.jpg");
  img2 = loadImage("airport.jpg");
}

void draw() {
  background(255);
  image(img1, 0, 0);
  int my = constrain(mouseY, 0, 67);
  copy(img2, 0, my, 100, 33, 0, my, 100, 33);
}
```

The `PImage` class also has a `copy()` method. It can be used to copy portions of one image to itself or areas of one image to another. The following example shows this method in action.

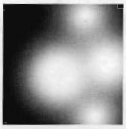
```
PImage img = loadImage("tower.jpg");
img.copy(50, 0, 50, 100, 0, 0, 50, 100);
image(img, 0, 0);
```

39-11

Masking

The `mask()` method of the `PImage` class sets the transparency values of an image based on the contents of another image. The mask image should contain only grayscale data and must be the same size as the image to which it is applied. If the image is not grayscale, it may be converted with the `filter()` function. The light areas of the mask let the original image through, and the dark areas conceal the original. The following example uses `mask()` to composite the images shown below.

airport.jpg



airportmask.jpg

The resulting image and the code to produce it follow:

```
background(255);
PImage img = loadImage("airport.jpg");
PImage maskImg = loadImage("airportmask.jpg");
img.mask(maskImg);
image(img, 0, 0);
```

39-12

Exercises

1. *Load an image and alter it with `filter()`.*
2. *Load three images and combine them with `blend()`.*
3. *Load two images and use `copy()` with `mouseX` and `mouseY` to combine them in a way that reveals the relationship between the images.*

Image 5: Image Processing

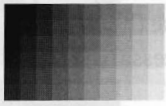
This unit introduces techniques for directly accessing the pixels in an image and explains the use of those techniques in modifying images.

Syntax introduced:

```
pixels[], loadPixels(), updatePixels(), createImage()
```

Image processing is a general term for manipulating and modifying images, whether for the purpose of correcting a defect, improving aesthetic appeal, or facilitating communication. Programs such as GIMP and Adobe Photoshop provide their users with ways to process images including changing the contrast, blurring, and warping. This section explains how some image processing features work to provide a better understanding of their application.

In Processing, each image is stored as a one-dimensional array of colors. When an image is displayed to the screen, each element in the array is drawn as a pixel. The number of elements in the array is determined by multiplying the width of an image by its height. If an image is 100 pixels wide and 100 pixels high, the array will have 10,000 elements. If an image is 200 pixels wide and 2 pixels high, the array will have 400 elements. The first position in the array is the pixel in the upper-left corner of the image, and the last position in the array is the pixel in the lower-right corner. The width and height of an image are used to map each element's position in the one-dimensional array to the two-dimensional position on screen. To make this clear, let's look at an array belonging to a small image of the size 10 × 6 pixels:



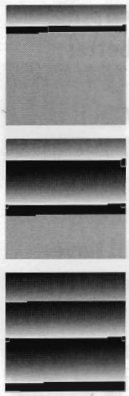
When this image is loaded into Processing, its one-dimensional pixel array contains each row of the two-dimensional image, one after another:

Because the image is 10 × 6 pixels, the array has 60 elements. The first element is at position [0] and the last at position [59]. Storing images in this format makes it easy to apply algorithms to the color values.

Pixels

The `pixels[]` array stores a color value for each pixel of the display window. The `loadPixels()` function must be called before the `pixels[]` array is used. After the pixels have been read or changed, they must be updated using the `updatePixels()` function. Like `beginShape()` and `endShape()`, `loadPixels()` and `updatePixels()` should always appear together.

The following example changes the color of the pixels in the display window by changing one pixel each frame based on the current second value. Over time, the pixels are set in order from left to right and top to bottom. The shift from white to black happens with each minute—when the value from `seconds()` jumps from 59 to 0. When the last pixel in the array is set, the program starts again at the beginning of the array.



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

void draw() {
  float gray = map(second(), 0, 59, 0, 255);
  color c = color(gray);
  int index = frameCount % (width*height);
  loadPixels();
  pixels[index] = c;
  updatePixels();
}
```

The `loadPixels()` and `updatePixels()` functions ensure that the `pixels[]` array is ready to be manipulated and that the changes are updated. Be sure to place them around any block of code that manipulates the array, but use them only when necessary because overuse can make your program run slowly.

Reading and writing data directly to and from the `pixels[]` array is a different way to perform the same action as `get()` and `set()`. The x-coordinate and y-coordinate can be mapped to the corresponding position within the array by multiplying the y-coordinate value by the width of the display window and then adding the x-coordinate value. To calculate the location of any pixel in the array, use the equation (`y * width`) + `x`.

```
// These 3 lines of code are equivalent to: set(25, 50, color(0))
loadPixels();
pixels[50*width + 25] = color(0);
updatePixels();
```

To convert to the opposite direction, divide the pixel's position in the array by the width of the display window to get the y-coordinate, and take the modulo value (p. 45) of the position and the width to get the x-coordinate:

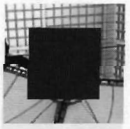
```
// These 3 lines are equivalent to: pixels[5075] = color(0)
int y = 5075 / width;
int x = 5075 % width;
set(x, y, color(0));
```

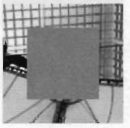
40-03

In programs that manipulate many pixels at a time, reading and writing values to the `pixels[]` array is much faster than using `get()` and `set()`. The following two examples show how to achieve the functionality of `get()` and `set()` using the `pixels[]` array. These examples will actually be slower than using `get()` and `set()`, but later examples will be much faster.

40-04

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

void draw() {
  // Constrain to not exceed the boundary of the array
  int mx = constrain(mouseX, 0, 99);
  int my = constrain(mouseY, 0, 99);
  loadPixels();
  pixels[my*width + mx] = color(0);
  updatePixels();
}
```





40-05

```
PImage arch;

void setup() {
  size(100, 100);
  noStroke();
  arch = loadImage("arch.jpg");
}

void draw() {
  background(arch);
  // Constrain to not exceed the boundary of the array
  int mx = constrain(mouseX, 0, 99);
  int my = constrain(mouseY, 0, 99);
  loadPixels();
  color c = pixels[my*width + mx];
  fill(c);
  rect(20, 20, 60, 60);
}
```

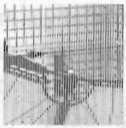
Each image has its own `pixels[]` array that is accessed with the dot operator. This array makes it possible to change an image while leaving the pixels in other images and the display window untouched. In the next example, pixels inside an image are colored black according to the position of the mouse.



40-06

```
PImage arch;

void setup() {
  size(100, 100);
  arch = loadImage("arch.jpg");
}

void draw() {
  background(204);
  int mx = constrain(mouseX, 0, 99);
  int my = constrain(mouseY, 0, 99);
  arch.loadPixels();
  arch.pixels[my*width + mx] = color(0);
  arch.updatePixels();
  image(arch, 50, 0);
}
```

Using the `pixels[]` array rather than the `image()` function to draw the image to the display window provides more control and leaves room for variation in displaying the image. Small calculations modifying the `for` structure and the `pixels[]` array reveal some of the potential of this technique.



40-07

```
PImage arch = loadImage("arch.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i += 2) {
  pixels[i] = arch.pixels[i];
}
updatePixels();
```

40-08

```
PImage arch = loadImage("arch.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i += 3) {
  pixels[i] = arch.pixels[i];
}
updatePixels();
```

```
PImage arch = loadImage("arch.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i++) {
  pixels[i] = arch.pixels[count - i - 1];
}
updatePixels();
```

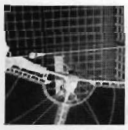
40-09

```
PImage arch = loadImage("arch.jpg");
int count = arch.width * arch.height;
arch.loadPixels();
loadPixels();
for (int i = 0; i < count; i++) {
  pixels[i] = arch.pixels[i/2];
}
updatePixels();
```

40-10

Pixel components

The `red()`, `green()`, and `blue()` functions (pp. 337–338) are used to read the individual color components from each pixel in an image. These components can be changed and then returned to the `pixels[]` array to modify the image. For example, if each value is multiplied by 2, the image will become lighter; if each value is divided by 2, the image will become darker. Using a `for` structure makes it easy to read and change every pixel in the display window. Because the `pixels[]` array is a one-dimensional array, only one `for` structure is necessary to modify every pixel in the image. The following example shows how to invert an image.



```
PImage arch = loadImage("arch.jpg");
background(arch);
loadPixels();
for (int i = 0; i < width*height; i++) {
  color p = pixels[i];          // Grab pixel
  float r = 255 - red(p);       // Modify red value
  float g = 255 - green(p);     // Modify green value
  float b = 255 - blue(p);      // Modify blue value
  pixels[i] = color(r, g, b);   // Assign modified value
}
updatePixels();
```

40-11

Values from the keyboard and the mouse can be used to change the way the `pixels[]` array is altered while the program runs. In the following example, a color image is

converted to gray values by averaging its components. These values are incremented by mouseX to make the image lighter as the mouse moves to the right.

```
PImage arch;

void setup() {
  size(100, 100);
  arch = loadImage("arch.jpg");
}

void draw() {
  background(arch);
  loadPixels();
  for (int i = 0; i < width*height; i++) {
    color p = pixels[i];      // Read color from screen
    float r = red(p);         // Modify red value
    float g = green(p);       // Modify green value
    float b = blue(p);        // Modify blue value
    float bw = (r + g + b) / 3.0;
    bw = constrain(bw + mouseX, 0, 255);
    pixels[i] = color(bw);    // Assign modified value
  }
  updatePixels();
  line(mouseX, 0, mouseX, height);
}
```

The functions for extracting individual color components are accurate and easy to use, but they are slow. When an idea requires using these functions hundreds or thousands of times each frame, they can be replaced with a technique called bit-shifting (p. 673).

Convolution

Another way to modify an image is to change the value of each pixel in relation to the neighboring pixels. This process operates in a similar way as the cellular automata introduced in Simulate 1 (p. 461). A matrix of numbers called a convolution kernel is applied to every pixel in the image—neighboring pixels are multiplied by the corresponding kernel value and added together to set the value of the center pixel. Applying the kernel to every pixel in the image is called convolving the image. This type of math can be performed very efficiently, and in advanced programs such as Photoshop, most of the filters are implemented in this manner.

As an example, let's use a kernel to determine the value for the pixel at coordinate (2,2) in the simple 6 × 6 pixel image shown below. The center of the kernel is first placed at the coordinate, and then each value within the area is multiplied by the

corresponding kernel value and all are added together. The sum is the new value for the pixel at the center of the kernel:

Original image | Enlarged region and gray values | Kernel values | Result

The first expression created by multiplying the image gray values by the kernel values is

```
(255 * 0.111) + (226 * 0.111) + (153 * 0.111) +
(226 * 0.111) + (153 * 0.111) + (  0 * 0.111) +
(153 * 0.111) + (  0 * 0.111) + (  0 * 0.111)
```

This simplifies to

```
28.305 + 25.086 + 16.983 +
25.086 + 16.983 +  0.000 +
16.983 +  0.000 +  0.000
```

This simplifies further to 129.426, is converted to the integer value 129, and becomes the gray value of the pixel.

To convolve the entire image, perform this action for all of the pixels in the image. It's clear that a problem arises when you try to use the kernel at the edges of the image. At the edges, there are no adjacent pixels from which to take values:

Original image | Enlarged region and gray values

To simplify the code in the examples below, this is ignored and only the pixels away from the border are used.

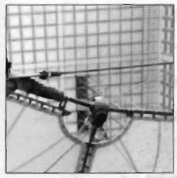
Patterns in the kernel numbers create different types of filters. If all the values for the kernel are positive, it creates what is called a low pass filter. A low pass filter removes areas where extreme differences in the values of adjacent pixels exist. For example, if one pixel is white and the next is black, they will create a gray when averaged together. When applied to an entire image, a low pass filter causes a blur. A mixture of positive and negative values can be used to create a high pass filter. A high pass filter removes areas where there is little difference in value between adjacent pixels. This technique sharpens images. A specific type of high pass filter is used for edge detection. An edge is an area that contains sudden changes in value. Common kernels for edge detection have negative numbers along one side and positive numbers on the opposing side, with zeros in the middle. For all kernels, the sum of the values must be 1 for the brightness to

remain the same when the image is convolved. If the sum is a smaller or larger number, the image will become darker or lighter in value than the original.

The following example demonstrates how to use a 3 × 3 kernel matrix to transform an image. Modify the values in the `kernel[][]` array to try different filter techniques. There are a few samples on the adjacent page. The `createImage()` function creates an empty pixel buffer. The function requires three parameters that assign the width, height, and format of the image. The format can be RGB (full color) or ARGB (full color with alpha). It is not necessary to use `loadPixels()` immediately after `createImage()`.

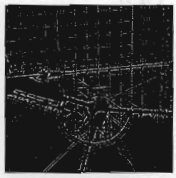
40-13

```
float[][] kernel = { { -1, 0, 1 },
                     { -2, 0, 2 },
                     { -1, 0, 1 } };
size(100, 100);

PImage img = loadImage("arch.jpg");  // Load the original image
img.loadPixels();
// Create an opaque image of the same size as the original
PImage edgeImg = createImage(img.width, img.height, RGB);

// Loop through every pixel in the image.
for (int y = 1; y < img.height-1; y++) {  // Skip top and bottom edges
  for (int x = 1; x < img.width-1; x++) { // Skip left and right edges
    float sum = 0;  // Kernel sum for this pixel
    for (int ky = -1; ky <= 1; ky++) {
      for (int kx = -1; kx <= 1; kx++) {
        // Calculate the adjacent pixel for this kernel point
        int pos = (y + ky)*width + (x + kx);
        // Image is grayscale, red/green/blue are identical
        float val = red(img.pixels[pos]);
        // Multiply adjacent pixels based on the kernel values
        sum += kernel[ky+1][kx+1] * val;
      }
    }
    // For this pixel in the new image, set the gray value
    // based on the sum from the kernel
    edgeImg.pixels[y*img.width + x] = color(sum);
  }
}
// State that there are changes to edgeImg.pixels[]
edgeImg.updatePixels();
image(edgeImg, 0, 0);  // Draw the new image
```



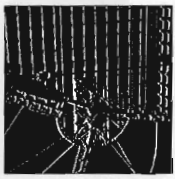
```
.11  .11  .11
.11  .11  .11
.11  .11  .11
```

```
.11  .11  .11
.11  .66  .11
.11  .11  .11
```

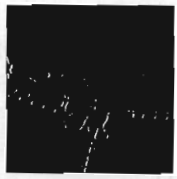


```
-1  -1  -1
-1   8  -1
-1  -1  -1
```

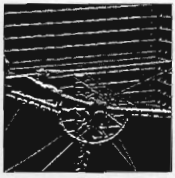
```
-1  -1  -1
-1  12  -1
-1  -1  -1
```



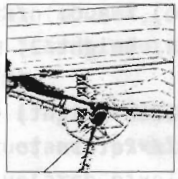
```
-1   0   1
-2   0   2
-1   0   1
```



```
-2   0   1
-3   0   2
-2   0   1
```

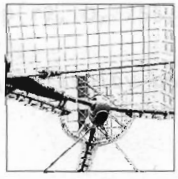


```
-1  -2  -1
 0   0   0
 1   2   1
```



```
-1  -2  -1
 0   0   0
 2   3   2
```

```
 0  -1   0
-1   4  -1
 0  -1   0
```



```
 0  -1   0
-1   6  -1
 0  -1   0
```

Convolving an image

Eight common 3 × 3 convolution kernels and their effects. A kernel is normalized if the sum of the values is 1. If the sum is above 1, the image becomes lighter, and if it's below 1, the image becomes darker. These numbers can be inserted into code 40-13.

Image as data

This unit has introduced digital images as one-dimensional sequences of numbers that define colors. This numerical data, however, need not be viewed as colors—it can be used to generate motion or define the vertices of a shape. The following examples use the data from the `pixels[]` array of an image to generate alternative representations.

40-14

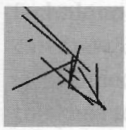
```
// Convert pixel values into a circle's diameter

PImage arch;
int index;

void setup() {
  size(100, 100);
  smooth();
  fill(0);
  arch = loadImage("arch.jpg");
  arch.loadPixels();
}

void draw() {
  background(204);
  color c = arch.pixels[index];  // Get a pixel
  float r = red(c) / 3.0;  // Get the red value
  ellipse(width/2, height/2, r, r);
  index++;
  if (index == width*height) {
    index = 0;  // Return to the first pixel
  }
}
```

40-15

```
// Convert the red values of pixels to line lengths

PImage arch;

void setup() {
  size(100, 100);
  arch = loadImage("arch.jpg");
  arch.loadPixels();
}

void draw() {
  background(204);
```

```
    int my = constrain(mouseY, 0, 99);
    for (int i = 0; i < arch.height; i++) {
      color c = arch.pixels[my*width + i];  // Get a pixel
      float r = red(c);  // Get the red value
      line(i, 0, i, height/2 + r/6);
    }
  }
```

40-15 cont.



```
// Convert the blue values from one row of the image
// to the coordinates for a series of lines

PImage arch;

void setup() {
  size(100, 100);
  smooth();
  arch = loadImage("arch.jpg");
  arch.loadPixels();
}

void draw() {
  background(204);
  int mx = constrain(mouseX, 0, arch.width-1);
  int offset = mx * arch.width;
  beginShape(LINES);
  for (int i = 0; i < arch.width; i += 2) {
    float r1 = blue(arch.pixels[offset + i]);
    float r2 = blue(arch.pixels[offset + i + 1]);
    float vx = map(r1, 0, 255, 0, height);
    float vy = map(r2, 0, 255, 0, height);
    vertex(vx, vy);
  }
  endShape();
}
```

40-16

Exercises

1. *Write your own image filter by modifying the values of* `pixels[]`.
2. *Explore different kernels to convolve an image and write a program to display your most interesting discovery.*
3. *Load an image and use its data to generate an animation that reflects the original image.*

Drawing 2: Kinetic Forms

This unit focuses on developing kinetic drawing tools and elements unique to software.

The experimental animation techniques of drawing, painting, and scratching directly onto film are all predecessors to software-based kinetic drawings. The immediacy and freshness of short films such as Norman McLaren's *Hen Hop* (1942), Len Lye's *Free Radicals* (1957), and Stan Brakhage's *The Garden of Earthly Delights* (1981) is due to the extraordinary qualities of physical gesture which software later made more accessible. In his 1948 essay "Animated Films," McLaren wrote, "In one operation, which is drawing directly onto the 35mm clear machine leader with an ordinary pen nib and India ink, a clean jump was made from the ideas in my head to the images on what would normally be called a developed negative." He further explains, "The equivalents of Scripting, Drawing, Animating, Shooting, Developing the Negative, Positive Cutting, and Negative Cutting were all done in one operation."[1] Like working directly on film, programming provides the ability to produce kinetic forms with immediate feedback.

Software animation tools further extend film techniques by allowing the artist to edit and animate elements continuously after they have been drawn. In 1991, Scott Snibbe's *Motion Sketch* extended to software the techniques explored by McLaren, Lye, and Brakhage. The application translates hand motion to visual elements on the screen. Each gesture creates a shape that moves in a one-second loop. The resulting animations can be layered to create a work of spatial and temporal complexity reminiscent of Oskar Fischinger's style. Snibbe extended this concept further with *Motion Phone* (1995), which enabled people to work simultaneously in a shared drawing space via the Internet.

Many artists have developed their own software in pursuit of creative animation. Since 1996, Bob Sabiston has developed Rotoshop, a set of tools for drawing and positioning graphics on top of video frames. He refined the software to make the ambitious animated feature *Waking Life* (p. 383). Ed Burton's *MOOVL* software extends ideas from the *Sodaconstructor* (p. 263) to a drawing program in which visual elements are aware of their relation to their environment and other elements. In *MOOVL*, shapes can be drawn, connected, and trained to move. The behavior can be mediated via changes in the gravity and other aspects of the simulation. The *Mobility Agents* software created by John F. Simon, Jr. (1989–2005) augments lines drawn by hand with additional lines drawn by the software. Drawn lines are augmented by or replaced with lines that correspond to the angle and speed at which the initial lines are drawn. Zach Lieberman's *Drawn* software (2005) explores a hybrid space of physical materials and software animation. Marks made on paper with a brush and ink are brought to life through the clever use of a video camera and computer vision techniques. The camera takes an image and the software calculates a mark's location and shape, at which point the mark can respond like any other reactive software form.

Artists explore software as a medium for pushing drawing in new directions. Drawing with software provides the ability to integrate time, response, and behavior with drawn marks. Information from the mouse (introduced in Input 5, p. 245) can be combined with techniques of motion (introduced in Motion 2, p. 291) to produce animated drawings that capture the kinetic gestures of the hand and reinterpret them as intricate motion. Other unique inputs, such as voice captured through a microphone and body gestures captured through a camera, can be used to control drawings.

Active tools

Software drawing instruments can change their form in response to gestures made by the hand. Comparison of `mouseX` and `mouseY` variables with previous mouse values can determine the direction and speed of motion. In the following example, the change in the mouse position between the last frame and current frame sets the size of the ellipse drawn to the screen. If the ellipse does not move, the size reverts to a single pixel.

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

void draw() {
  float s = dist(mouseX, mouseY, pmouseX, pmouseY) + 1;
  noStroke();
  fill(0, 102);
  ellipse(mouseX, mouseY, s, s);
  stroke(255);
  point(mouseX, mouseY);
}
```

Software drawing instruments can follow a rhythm or abide by rules independent of drawn gestures. This is a form of collaborative drawing in which the draftsperson controls some aspects of the image and the software controls others. In the examples that follow, the drawing elements obey their own rules, but the draftsperson controls each element's origin. In the next example, the drawing tool pulses from a small to a large size, supplementing the motion of the hand.





```
int angle = 0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  fill(0, 102);
}

void draw() {
  // Draw only when mouse is pressed
  if (mousePressed == true) {
    angle += 10;
    float val = cos(radians(angle)) * 6.0;
    for (int a = 0; a < 360; a += 75) {
      float xoff = cos(radians(a)) * val;
      float yoff = sin(radians(a)) * val;
      fill(0);
      ellipse(mouseX + xoff, mouseY + yoff, val/2, val/2);
    }
    fill(255);
    ellipse(mouseX, mouseY, 2, 2);
  }
}
```

44-02

In the next example, the `Blade` class defines a drawing tool that creates a growing diagonal line when the mouse is not moving and resets the line to a new position when the mouse moves.



```
Blade diagonal;

void setup() {
  size(100, 100);
  diagonal = new Blade(30, 80);
}

void draw() {
  diagonal.grow();
}

void mouseMoved() {
  diagonal.seed(mouseX, mouseY);
}
```

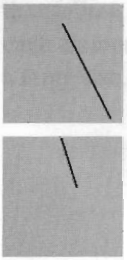
44-03

```
class Blade {
  float x, y;

  Blade(int xpos, int ypos) {
    x = xpos;
    y = ypos;
  }

  void seed(int xpos, int ypos) {
    x = xpos;
    y = ypos;
  }

  void grow() {
    x += 0.5;
    y -= 1.0;
    point(x, y);
  }
}
```

Active drawings

Individual drawing elements with their own behavior can produce drawings with or without input from a person. These active drawings are a bit like what would result from a raccoon stumbling into a paint tray and then running across pavement. Though created by a series of predetermined rules and actions, the drawings are partially or totally autonomous.

The code for the next example is presented in steps because it's longer than most in the book. Before writing the longer program, we first wrote a small program to test the desired effect. This code displays a line that changes position very slightly with each frame. Over a long period of time, the line's position changes significantly. This is similar to code 32-05 (p. 296), but is more subtle.



```
float x1, y1, x2, y2;

void setup() {
  size(100, 100);
  smooth();
  x1 = width / 4.0;
  y1 = x1;
  x2 = width - x1;
  y2 = x2;
}
```

```
void draw() {
  background(204);
  x1 += random(-0.5, 0.5);
  y1 += random(-0.5, 0.5);
  x2 += random(-0.5, 0.5);
  y2 += random(-0.5, 0.5);
  line(x1, y1, x2, y2);
}
```

44-04 cont.

If several such lines are drawn, the drawing will degrade over time as each line continues to wander from its original position. In the next example, the code from above was modified to create the `MovingLine` class. Five hundred of these `MovingLine` objects populate the display window. When the lines are first drawn, they vibrate but maintain their form. Over time, the image degrades into chaos as each line wanders across the surface of the window.

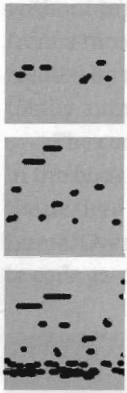
44-05

```
int numLines = 500;
MovingLine[] lines = new MovingLine[numLines];
int currentLine = 0;

void setup() {
  size(100, 100);
  smooth();
  frameRate(30);
  for (int i = 0; i < numLines; i++) {
    lines[i] = new MovingLine();
  }
}

void draw() {
  background(204);
  for (int i = 0; i < currentLine; i++) {
    lines[i].display();
  }
}

void mouseDragged() {
  lines[currentLine].setPosition(mouseX, mouseY,
                                 pmouseX, pmouseY);
  if (currentLine < numLines - 1) {
    currentLine++;
  }
}
```

44-05 cont.

```
class MovingLine {
  float x1, y1, x2, y2;

  void setPosition(int x, int y, int px, int py) {
    x1 = x;
    y1 = y;
    x2 = px;
    y2 = py;
  }

  void display() {
    x1 += random(-0.1, 0.1);
    y1 += random(-0.1, 0.1);
    x2 += random(-0.1, 0.1);
    y2 += random(-0.1, 0.1);
    line(x1, y1, x2, y2);
  }
}
```

The next example shows a simple animation tool that displays a continuous cycle of twelve images. Each image is displayed for 100 milliseconds (one tenth of a second) to create animation. While each image is displayed, it's possible to draw directly into it by pressing the mouse and moving the cursor.



44-06

```
int currentFrame = 0;
PImage[] frames = new PImage[12];
int lastTime = 0;

void setup() {
  size(100, 100);
  strokeWeight(4);
  smooth();
  background(204);
  for (int i = 0; i < frames.length; i++) {
    frames[i] = get();  // Create a blank frame
  }
}

void draw() {
  int currentTime = millis();
  if (currentTime > lastTime+100) {
    nextFrame();
    lastTime = currentTime;
  }
```

44-06 cont.

```
    if (mousePressed == true) {
      line(pmouseX, pmouseY, mouseX, mouseY);
    }
  }

  void nextFrame() {
    frames[currentFrame] = get();  // Get the display window
    currentFrame++;  // Increment to next frame
    if (currentFrame >= frames.length) {
      currentFrame = 0;
    }
    image(frames[currentFrame], 0, 0);
  }
```

Exercises

1. *Design and program your own active drawing instrument.*
2. *Design and program visual elements that change after they have been drawn to the display window.*
3. *Extend code 44-06 into a more complete animation program.*

Notes

1. Norman McLaren, "Animated Films," in *Experimental Animation*, edited by Robert Russett and Cecile Starr (Da Capo Press, 1976), p. 122.