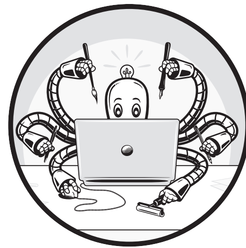
3

2D GRAPHICS AND ANIMATION



Sketch 27: Saving an Image and Adjusting Transparency

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

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

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

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

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

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

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

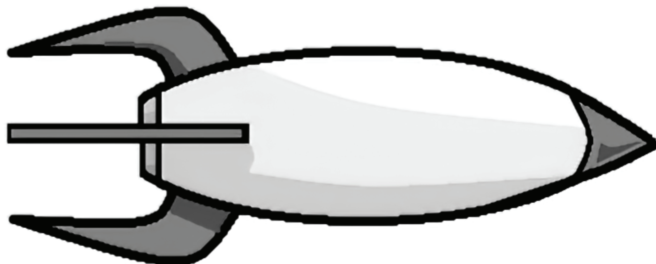
This creates an empty image of the correct size, and now we must copy all of the pixels from `img1` into `img2`. When we do so, the pixels in `img2` have the alpha component, because it was specified in the `createImage()` call. When a mouse click specifies a background color, all pixels of that color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

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

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

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

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



Sketch 28: Bouncing an Object in a Window

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

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

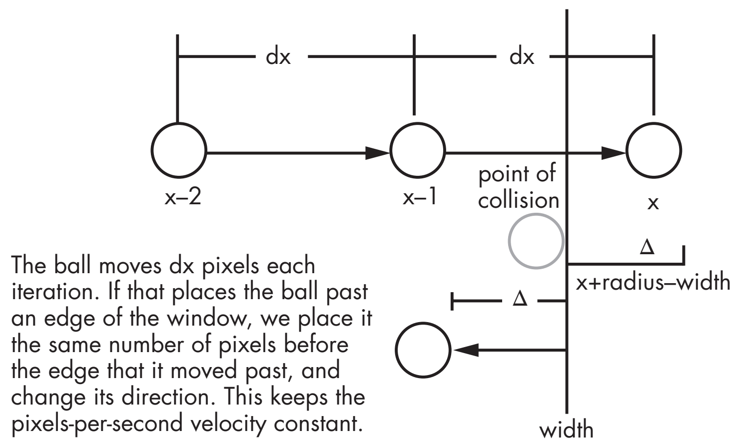


Figure 28-1: A fast-moving ball might overshoot a boundary before you can tell it to bounce back.

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

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

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

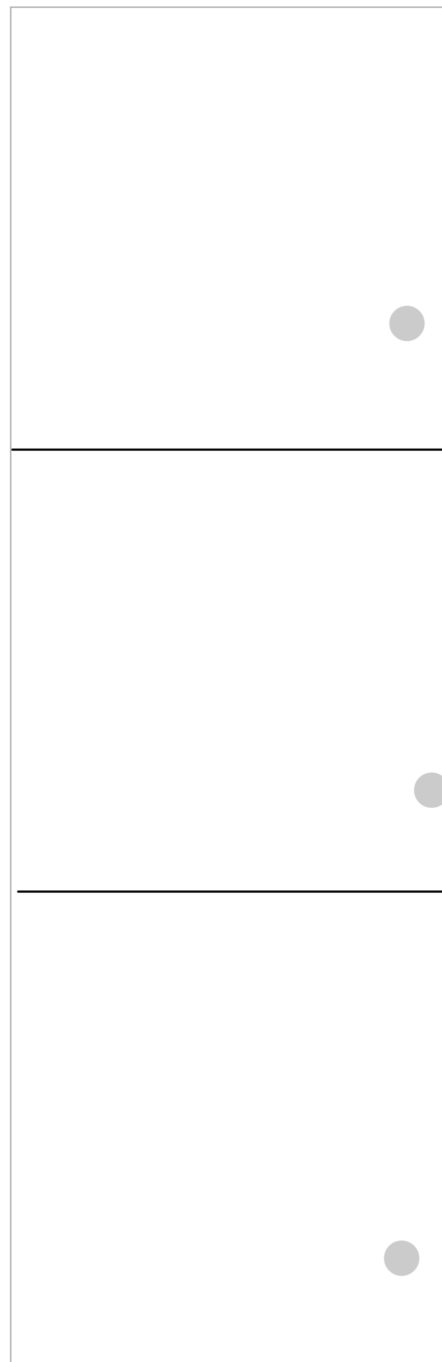
NOTE *Most objects are not circular but can have a (virtual, invisible) circle drawn around them, and we can use this circle to detect collisions against the boundary.*

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

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

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

void xbounce ()
{
  int delta = 0;
❶ delta = (x+radius) - width;
❷ if (x+radius > width)  // right side
  {
  ❸ x = width-delta-radius;
    dx = -dx;
❹ } else if (x < radius) // left side
  {
  ❺ x = (2*radius)-x;
    dx = -dx;            // Reverse x-direction
  }
}
❻ void ybounce ()
{
  int delta = 0;
  delta = (y+radius) - height;
  if (y < radius)               // top side
  {
    y = (2*radius)-y;
    dy = -dy;
  } else if (y+radius > height) // bottom side
  {
    y = height-delta-radius;
    dy = -dy;                   // Reverse y-direction
  }
}
```



An Artist's Guide to Programming (Sample Chapter) © 01/28/2022 by Jim Parker

Sketch 29: Basic Sprite Graphics

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

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

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

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

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

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

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

The checks are symmetrical for the y-coordinate ❸.

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

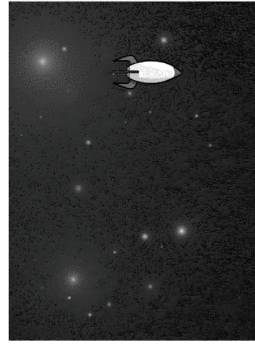
```
void keyPressed()
{
  if (key == 'w') py = py - 1;
  if (key == 's') py = py + 1;
  if (key == 'd') px = px + 1;
  if (key == 'a') px = px - 1;
}
```

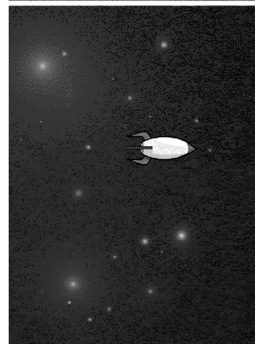
```
PImage img1, sprite;
color c=color(0,0,0);
int px=100, py=100, dx=2, dy=1;

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

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

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





Sketch 30: Detecting Sprite-Sprite Collisions

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

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

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

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

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

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

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

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

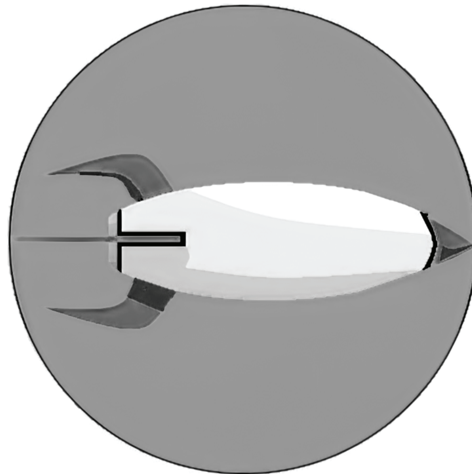
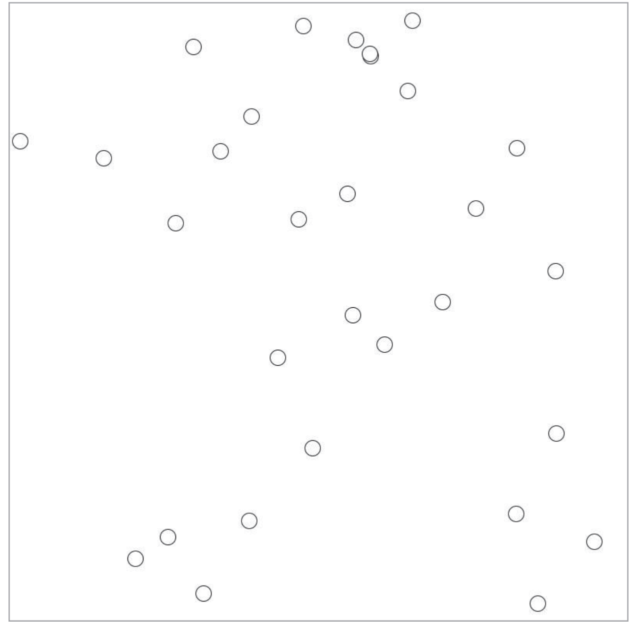


Figure 30-1: The enclosing circle for a rectangular object

An Artist's Guide to Programming (Sample Chapter) © 01/28/2022 by Jim Parker



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

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

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

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

An Artist's Guide to Programming (Sample Chapter) © 01/28/2022 by Jim Parker

Sketch 31: Animation—Generating TV Static

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

- Nature uses unpredictable forms and shapes. Placing trees in a forest in a two-dimensional grid is a giveaway that there was a mind at work in the planting. This does not happen in nature. Instead, trees in a forest have an average distance from each other and seem otherwise to form a random collection.
- Intelligent creatures do not behave predictably. Cars on a freeway that all behave in the same manner look very odd. Cars have random distances from each other, random speeds, and random behaviors within a possible range.
- When playing poker or craps, the cards and dice ought to display random values, or the game is simply no fun.

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

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

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

To make it appear as though a channel were poorly tuned in, we could display an image faintly over the static by setting the alpha for the image to a low value, perhaps 30 or so. The static would be visible through the image. The `tint()` function changes the color and transparency of whatever is drawn from then on, so we could use it to change the transparency of the channel image, as follows:

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

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

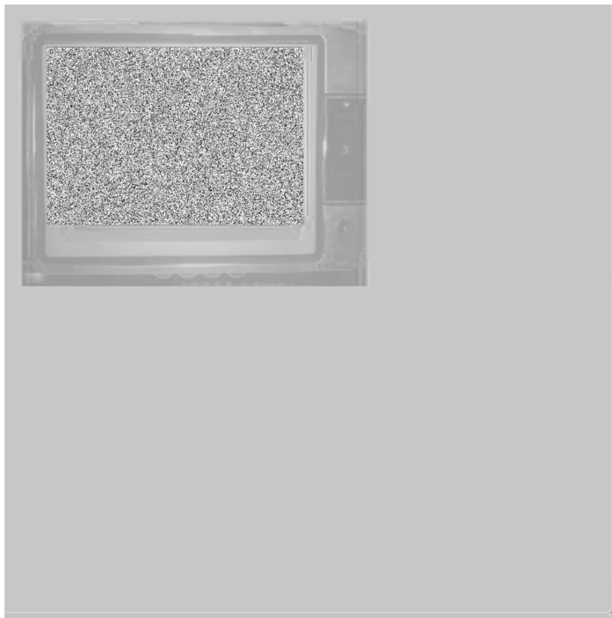
In the code for this sketch, the tint and TV image are commented out. To see the image, remove the comment characters from those two lines ❷.

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

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

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

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



Sketch 32: Frame Animation

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

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

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

Example A

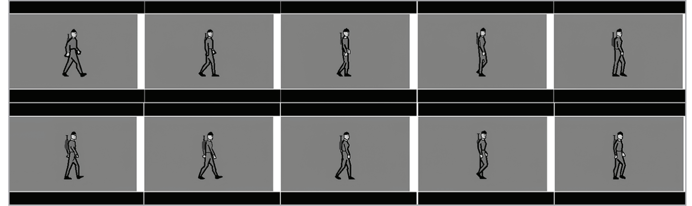
Eleven images, named *a000.bmp* through *a010.bmp*, represent the animation. The program reads the images into consecutive elements of the `frames` array ❶. The `draw()` function displays the next image in sequence each time it's called, increasing an index variable `n` from 0 to 10 and decreasing it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

Example B

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

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

Example A



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

Example B

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

Sketch 33: Flood Fill—Filling in Complex Shapes

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

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

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

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

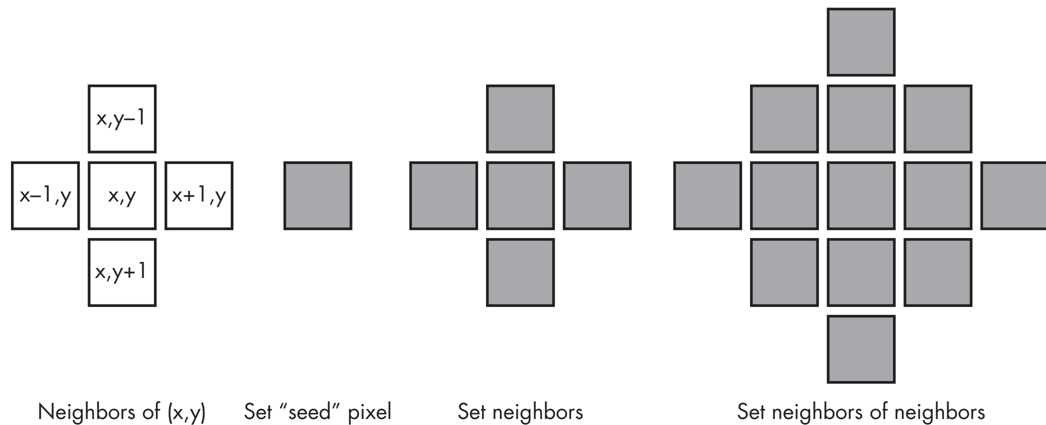


Figure 33-1: Filling in neighboring pixels

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

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

```
PImage inputImage;
color bgColor, fillColor;

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

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

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

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

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


}
```