Processing

Creative Coding and Computational Art

Ira Greenberg

friendsof
DESIGNER TO DESIGNER™
an Apress® company

Processing: Creative Coding and Computational Art

ISBN-13: 978-1-59059-617-3

ISBN-10: 1-59059-617-X

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Credits

Lead Editor
Chris Mills

Technical Editor
Charles E. Brown

Technical Reviewers
Carole Katz, Mark Napier

Editorial Board
Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Jeff Pepper, Dominic Shakeshaft, Matt Wade

Project Manager
Sofia Marchant

Copy Edit Manager
Nicole Flores

Copy Editor
Damon Larson

Assistant Production Director
Kari Brooks-Copony

Production Editor
Ellie Fountain

Compositor
Dina Quan

Artist
Milne Design Services, LLC

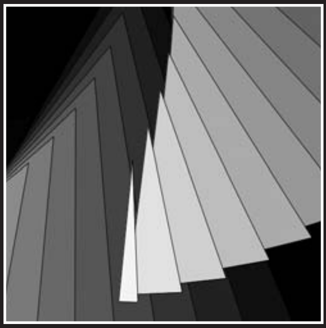
Proofreaders
Linda Seifert and Nancy Sixsmith

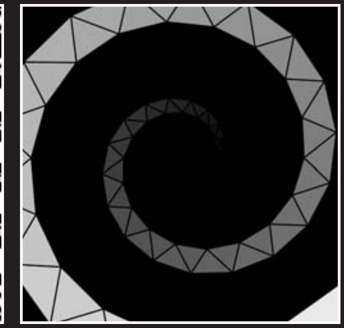
Indexer
John Collin

Interior and Cover Designer
Kurt Krames

Manufacturing Director
Tom Debolski

9 SHAPES





Shapes are a natural extension of lines and curves. In the simplest sense, a shape is just the enclosed space formed between three or more lines. Of course, a single continuous curve can enclose space as well (e.g., an ellipse). Shapes can be generally categorized. For example, an enclosed three-sided shape is referred to as a triangle, even if its size or configuration can't be determined. Fundamental 2D shapes (e.g., rectangles, circles, and triangles) are commonly referred to as **primitives**. Primitives are essential building-block shapes, which when combined allow you to (efficiently) construct most other shapes. Processing includes both 2D and 3D primitive functions. In this chapter, you'll just be dealing with 2D shape functions.

I'll begin by creating some simple sketches based on Processing's primitive 2D shape functions. These sketches will also provide a quick review of some of the core concepts covered in earlier chapters. Using Processing's drawing commands, I'll then show you how to create your own custom shapes, and eventually you'll work your way to implementing an OOP approach for shape creation—putting to good use some of the concepts you looked at in the last chapter. Before you dive in, first a word of encouragement.

Patterns and principles (some encouragement)

Hopefully by this point in the book you're beginning to recognize some common coding and implementation patterns, duplicated in many of the examples. From my experiences in the classroom, I find most students are able to grasp these coding patterns and principles much more quickly than they can remember the specific language commands/syntax. You can always look up a command—that's what the language reference (API) is for. Conceptualization and design are far more important factors than language retention. Thus, if you're beginning to get the bigger picture, but are still struggling with implementation details, you're doing well. If on the other hand, fundamental coding concepts such as variables, loops, conditional statements, and arrays still seem unclear, you might want to review Chapter 3 again.

Processing's shape functions

Processing comes with pre-made shape creation commands, some you've looked at already. However, I'll discuss each again with some simple examples. As you review these functions, I recommend trying to deduce the underlying algorithms operating within the functions, rather than simply trying to memorize how to apply them. For example, if you were to create your own rectangle-drawing function, how would you do it? This algorithm-centered approach will lead you to a deeper understanding of coding and Processing, and ultimately allow you to code any shape, rather than make you helplessly reliant on the few functions included in the API.

The first shape I'll cover is the rectangle. Using Processing, creating a rectangle couldn't be easier (and something you're probably very familiar with by now). The following code creates the rectangle shown in Figure 9-1:

```
void setup(){
  size(400, 400);
  background(255);
  strokeWeight(10);
  fill(127);
  rect(100, 50, 200, 300);
}
```

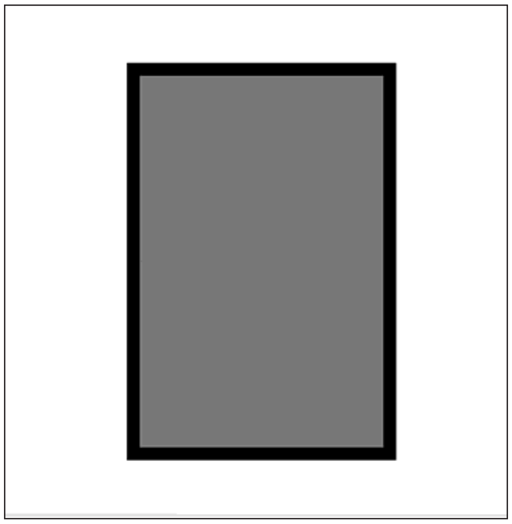


Figure 9-1. Processing's rect() function

The shape functions in Processing rely on a drawing mode variable for defining their origin—the point from which the shapes are drawn. The default origin point for a rectangle is the top-left corner, while for an ellipse it's the center. However, you can easily change these by calling `rectMode()` or `ellipseMode()` before calling the `rect()` or `ellipse()` function, respectively. This next example (shown in Figure 9-2) draws three rectangles utilizing three different drawing modes: `rectMode(CORNER)`, `rectMode(CENTER)`, and `rectMode(CORNERS)`.

```
void setup(){
  size(400, 400);
  background(255);
  strokeWeight(10);
  fill(127);
  rect(103, 120, 130, 100);
  rectMode(CENTER);
  rect(103, 120, 130, 100);
  rectMode(CORNERS);
  rect(233, 220, 363, 320);
}
```

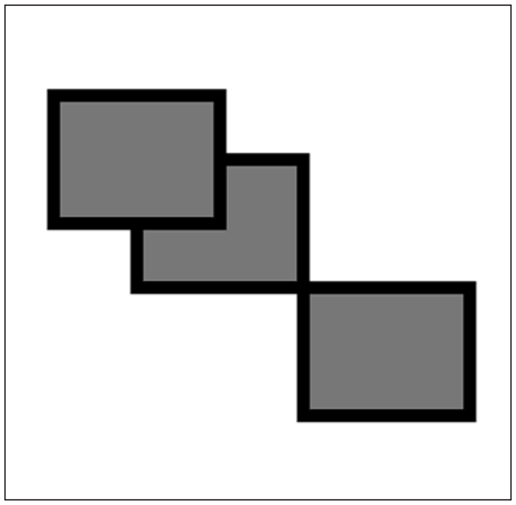


Figure 9-2. rectMode() example

The default mode is `rectMode(CORNER)`, which draws the rectangle from the top-left corner. Since this is the default, I didn't need to explicitly call it in the sketch. The `CENTER` and `CORNER` modes treat the four `rect()` arguments as x, y, width, and height. However, the `CORNERS` mode treats the four arguments as x1, y1, x2, y2, specifying the top-left and bottom-right corner points of the rectangle. If I add one more `rect()` call to the bottom of the program, without specifying another drawing mode call, what mode do you think will be used? This question is a little tricky, as many people will assume the default state (`CORNER`) will be used, but the answer is actually (`CORNERS`). Every time you issue one of these drawing mode commands in a program, you change the current state of the drawing mode. This type of programming change is referred to as a **state change**, as you are changing the drawing state of the program, at least in terms of drawing rectangles.

Processing's `ellipse()` function, which has the same four parameters as `rect(int x, int y, int w, int h)`, works similarly. However, as mentioned earlier, the default drawing mode for `ellipse()` is `ellipseMode(CENTER)`. `ellipse()` also has access to one additional mode, `ellipseMode(CENTER_RADIUS)`. This mode does the same thing as `CENTER`, only it treats the `width` and `height` arguments as radii rather than diameters. One somewhat useful thing to do with these two modes together is to quickly generate an ellipse within an ellipse (see Figure 9-3):

```
size(400, 400);
background(255);
strokeWeight(10);
fill(127);
ellipseMode(CENTER_RADIUS);
ellipse(200, 200, 170, 170);
fill(255);
ellipseMode(CENTER);
ellipse(200, 200, 170, 170);
```

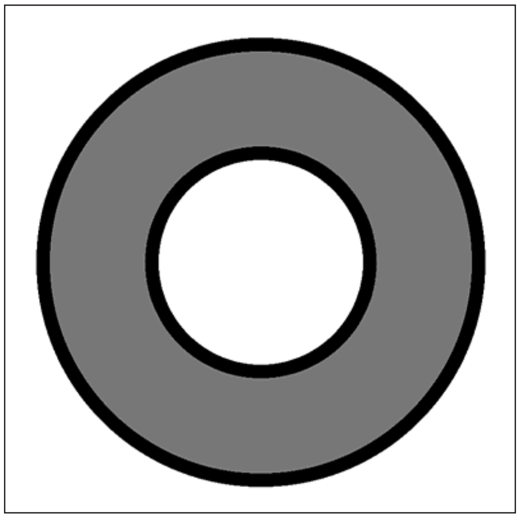


Figure 9-3. ellipseMode() example

This next example is the earlier `rectMode` example converted to draw ellipses (see Figure 9-4). I left in the `rect()` calls, treating them as bounding boxes around the ellipses.

```
//Ellipses with Bounding Boxes
void setup(){
  size(400, 400);
  background(255);
  strokeWeight(10);
  fill(127);
  ellipse(103, 120, 130, 100);
  ellipseMode(CORNER);
  ellipse(103, 120, 130, 100);
  ellipseMode(CORNERS);
  ellipse(233, 220, 363, 320);

  // bounding boxes
  strokeWeight(1);
  noFill();
  rect(103, 120, 130, 100);
  rectMode(CENTER);
  rect(103, 120, 130, 100);
  rectMode(CORNERS);
  rect(233, 220, 363, 320);
}
```

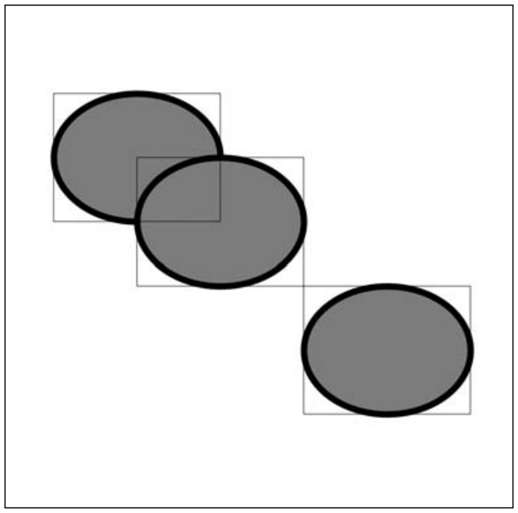


Figure 9-4. Ellipses with Bounding Boxes sketch

A bounding box is the smallest rectangular region that encloses a shape. Bounding boxes are important in computer graphics, as it's simpler and less computationally demanding for the computer to calculate the area of a rectangle than an ellipse or some other irregular shape. Thus, in a game, for example, collision detection can be calculated with regard to the bounding box of a shape, instead of the shape's actual perimeter. This will allow the game to perform better (but there may be some loss of accuracy with regard to how the actual detection looks on the screen).

In addition to the `rect()` and `ellipse()` shape functions, Processing includes `point()`, `arc()`, `triangle()`, and `quad()` functions. I discussed `point()` exhaustively in Chapters 6 and 7, and technically a point is not a shape (mathematically speaking), so I'll skip it. I also covered `arc()` in the context of curves—but here's one more example, illustrating the `arc()` function's pie shape feature (see Figure 9-5):

```
//Arcs with Bounding Boxes
void setup(){
  size(400, 400);
  background(255);
  strokeWeight(10);
  fill(127);
  arc(103, 120, 130, 100, 0, PI);
  ellipseMode(CORNER);
  arc(103, 120, 130, 100, 0, HALF_PI);
  ellipseMode(CORNERS);
  arc(233, 220, 363, 320, 0, TWO_PI-HALF_PI);

  // bounding boxes
  strokeWeight(1);
  noFill();
```

```
    rect(103, 120, 130, 100);
    rectMode(CENTER);
    rect(103, 120, 130, 100);
    rectMode(CORNERS);
    rect(233, 220, 363, 320);
  }
```

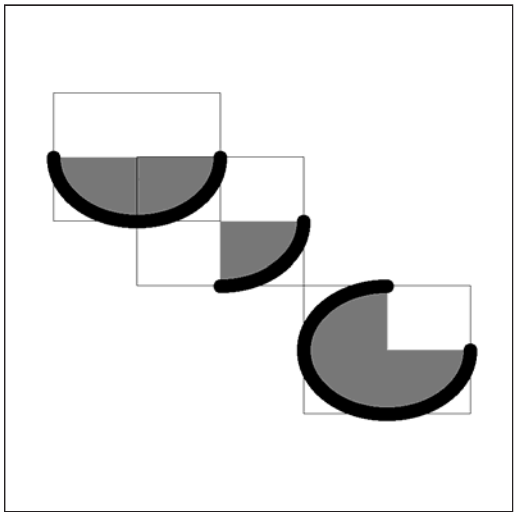


Figure 9-5. Arcs with Bounding Boxes sketch

In this last sketch, I used the `ellipse()` example code and simply switched the `ellipse` keyword with `arc`, adding the two additional required start and end angle arguments. (The angle arguments need to be specified in radians, and the value of `PI` in radians is equivalent to 180 degrees, or 1/2 rotation around an ellipse.) I was able to use the `ellipse()` example source code, as `arc()` shares the same four initial arguments as `ellipse()` and also uses the `ellipseMode()` function to specify its drawing mode. Really, an arc and an ellipse are related, as both are internally implemented the same way using cosine and sine functions. Later in the chapter, I'll show you how to create an ellipse in Processing using some trig functions. As I demonstrated in Chapter 7, if you use `0` and `TWO_PI` as your two angle arguments when calling `arc()`, you'll get an ellipse. I leave that for you to try on your own, if you missed it earlier. Next is an example that uses Processing's `triangle()` function (see Figure 9-6):

```
  //Triangle
  Point[]p = new Point[3];
  void setup(){
    size(400, 400);
    background(190);
    p[0] = new Point(2, height-2);
    p[1] = new Point(width-2, height-2);
    p[2] = new Point(width/2, 2);
    stroke(0);
    strokeWeight(2);
    fill(225);
    triangle(p[0].x, p[0].y, p[1].x, p[1].y, p[2].x, p[2].y);
  }
```

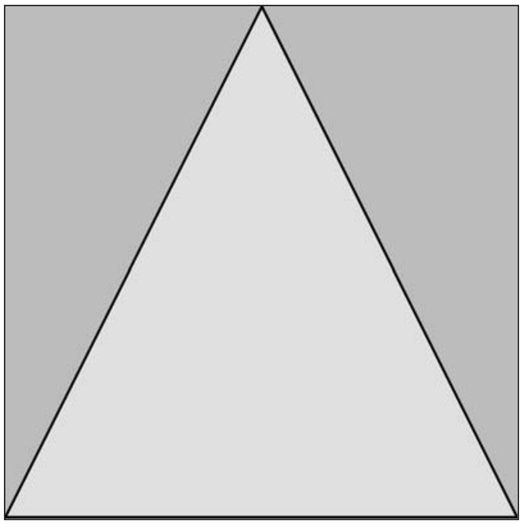


Figure 9-6. Triangle sketch

The examples thus far in the chapter have been pretty dull, so we'll do something a little more interesting with the `triangle()` function. Also note that I used Java's `Point` class again, as I find it very convenient when plotting stuff. This class is not included in Processing, so you won't find it in the Processing reference. However, it's in the Java API (see `http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Point.html`). Arrays in Processing and Java can be declared of any data type. (Remember from Chapter 8 that a class is also a data type.) `Point` is a class in Java and thus a valid data type. This allowed me to create a `Point` array and to refer to each `Point` object in the array by a single index value (e.g., `p[0]` or `p[1]`). Since each `Point` object has an x and y property, this is a convenient data type for keeping track of both components (x and y) at the same time. If I had just used arrays of type `float` instead of the `Point` class, I'd need to use either two separate parallel arrays—one for x and one for y—or I'd need to use a multidimensional array (`p[][]`). I think using an array of `Points` is the simplest solution. Here's a little variation on the Triangle sketch (see Figure 9-7):

```
//Triangle Zoom
Point[]p = new Point[3];
float shift = 10;
void setup(){
  size(400, 400);
  background(190);
  smooth();
  p[0] = new Point(1, height-1);
  p[1] = new Point(width-1, height-1);
  p[2] = new Point(width/2, 1);
  stroke(0);
  strokeWeight(1);
  fill(225);
  triangle(p[0].x, p[0].y, p[1].x, p[1].y, p[2].x, p[2].y);
  triBlur();
}
void triBlur(){
  triangle(p[0].x+=shift, p[0].y-=shift/2, p[1].x-=shift, ➥
         p[1].y-=shift/2, p[2].x, p[2].y+=shift);
  if(p[0].x<width/2){
    // recursive call
    triBlur();
  }
}
```

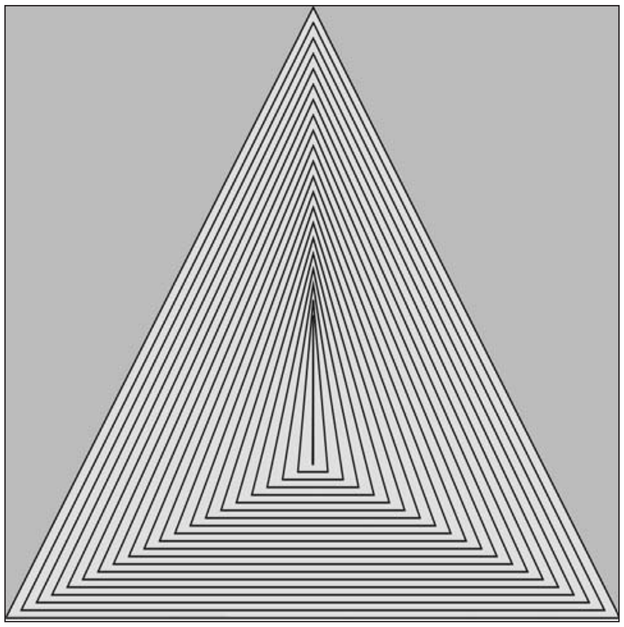


Figure 9-7. Triangle Zoom sketch

This sketch introduces a new advanced concept called recursion. **Recursion** is a process in which a function calls itself, as happened in `triBlur()` in the last example. I didn't need to use recursion in this last example—I could have also handled the multiple calls to `triangle()` iteratively using a `while` or `for` loop, but it was a good excuse to demonstrate how recursion works. One danger in using recursion is the increased possibility of generating an infinite loop, which is a lot easier to do than you might assume. The notion of something calling itself can be a little confusing, and therefore it's not that hard to make a simple logic mistake. I avoided an endless loop by wrapping the recursive call in a conditional statement. Since I increased the value of p[0].x each time the function executed, I used the conditional to ensure that p[0].x would never exceed the midpoint of the display window. Next, I'll add a gradient blur effect to the sketch (see Figure 9-8):

```
//Triangle Blur
Point[]p = new Point[3];
float shift = 2;
float fade = 0;
float fillCol = 0;
void setup(){
  size(400, 400);
  background(0);
  smooth();
  fade = 255.0/(width/2.0/shift);
  p[0] = new Point(1, height-1);
  p[1] = new Point(width-1, height-1);
  p[2] = new Point(width/2, 1);
  noStroke();
  triBlur();
}
void triBlur(){
  fill(fillCol);
  fillCol+=fade;
  triangle(p[0].x+=shift, p[0].y-=shift/2, p[1].x-=shift, ➥
           p[1].y-=shift/2, p[2].x, p[2].y+=shift);
  if(p[0].x<width/2){
    // recursive call
    triBlur();
  }
}
```

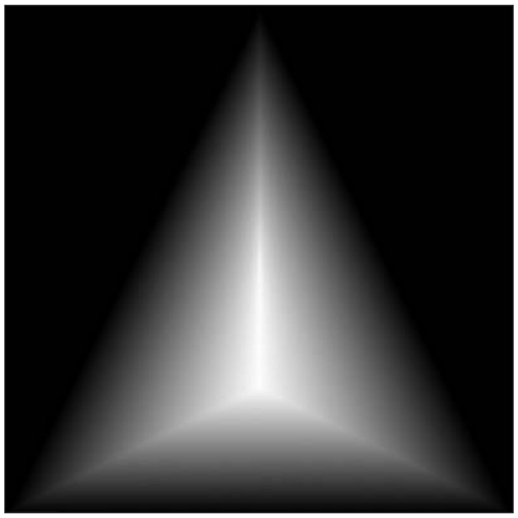


Figure 9-8. Triangle Blur sketch

The blur was generated by incrementing the fill color from black to white each loop iteration, based on the fade factor. I also turned off the stroke and decreased the value of the `shift` variable to make the transition seamless. Hopefully you've been able to follow all this. With the exception of the recursion, there's nothing new here. In the next modification, I'll introduce a new concept. I'll add some rotation to the triangles, which in itself isn't very complicated. However, the first attempt won't pan out as you might expect. I'll then discuss some new concepts to get the sketch working. Here's the flawed initial attempt (see Figure 9-9):

```
//Triangle Spin
Point[]p = new Point[3];
float shift = 2;
float fade = 0;
float fillCol = 0;
float spin = 0;

void setup(){
  size(400, 400);
  background(0);
  smooth();
  fade = 255.0/(width/2.0/shift);
  spin = 360.0/(width/2.0/shift);
  p[0] = new Point(1, height-1);
  p[1] = new Point(width-1, height-1);
  p[2] = new Point(width/2, 1);
  noStroke();
  triBlur();
}
void triBlur(){
  fill(fillCol);
  fillCol+=fade;
  rotate(spin);
  triangle(p[0].x+=shift, p[0].y-=shift/2, p[1].x-=shift, ➥
           p[1].y-=shift/2, p[2].x, p[2].y+=shift);
  if(p[0].x<width/2){
    // recursive call
    triBlur();
  }
}
```

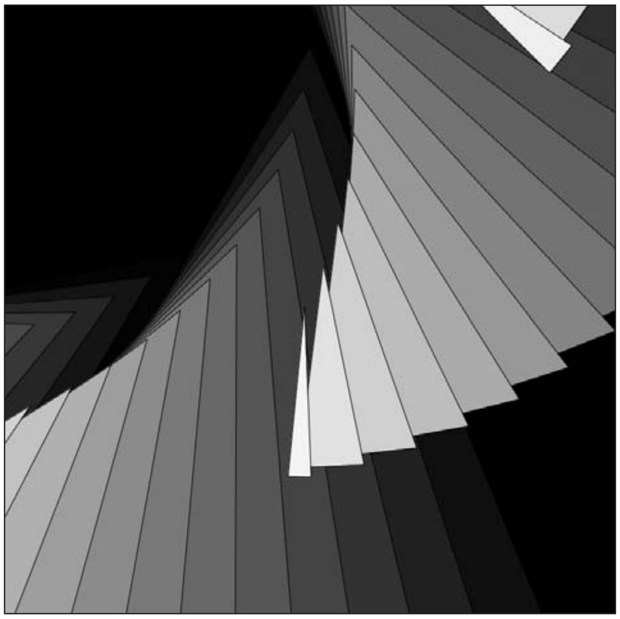


Figure 9-9. Triangle Spin sketch

Transforming shapes

In the last example, I wanted to create a nice, even rotation around the center point of the triangles. Instead, what I got was the triangles spinning out of the display window. What happened? The computer didn't actually do anything wrong (although I suspect you realized that). I asked it to rotate 3.6 degrees each iteration of the loop, and it did just that. When you call the `rotate()` function, Processing rotates around the origin, which is (0, 0), or the top-left corner of the display window. The triangles, however, are centered in the middle of window. So Processing did exactly what I asked of it—it rotated around the origin, not the center point of the triangles or the display window.

To fix this issue, I either need to write my own custom triangle and rotate functions using some trig functions, or (the much simpler way) I can draw my triangles centered around the origin (0,0), and then use Processing's `translate()` function to move them to the center of the display window. In reality, this shifts the graphics context of the entire display window, not just the triangles. The graphic context is the virtual drawing space of the display window. Here's the last example corrected using Processing's `translate()` function (see Figures 9-10 and 9-11):

```
//Triangle Flower
Point[]p = new Point[3];
float shift = 1.0;
float fade = 0;
float fillCol = 0;
float rot = 0;
float spin = 0;
void setup(){
  size(400, 400);
  background(0);
  smooth();
  fade = 255.0/(width/2.0/shift);
  spin = 360.0/(width/2.0/shift);
  p[0] = new Point(-width/2, height/2);
  p[1] = new Point(width/2, height/2);
  p[2] = new Point(0, -height/2);
  noStroke();
  translate(width/2, height/2);
  triBlur();
}
void triBlur(){
  fill(fillCol);
  fillCol+=fade;
  rotate(spin);
  // another interesting variation: uncomment the line below
  // rotate(rot+=radians(spin));
  triangle(p[0].x+=shift, p[0].y-=shift/2, p[1].x-=shift, ➥
         p[1].y-=shift/2, p[2].x, p[2].y+=shift);
  if(p[0].x<0){
    // recursive call
    triBlur();
  }
}
```

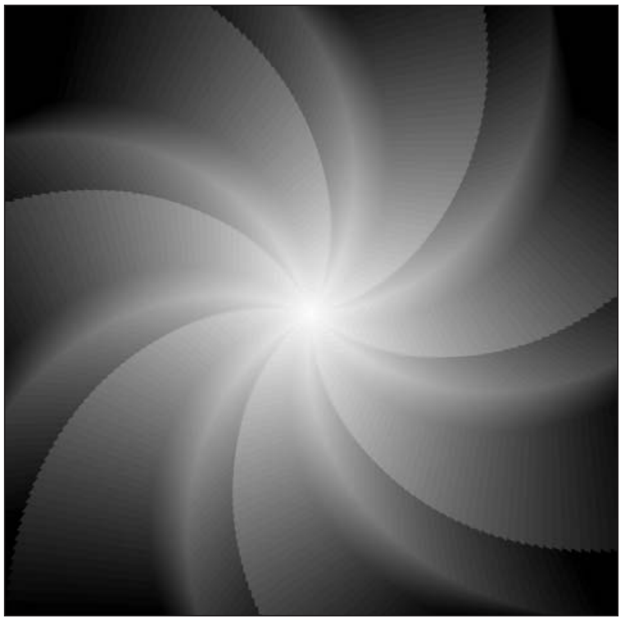


Figure 9-10. Triangle Flower sketch, variation 1

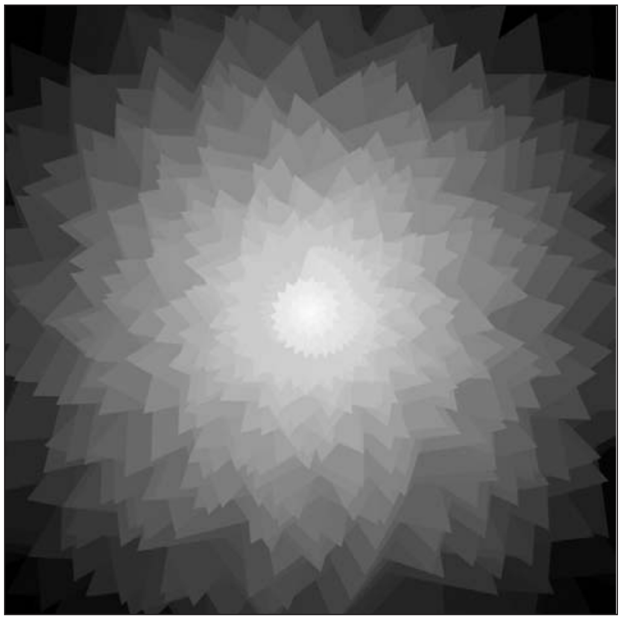


Figure 9-11. Triangle Flower variation sketch, variation 2

Look at this sketch carefully, as the concepts are important. Also notice that I included a second `rotate()` option. Simply uncomment the `// rotate(rot+=radians(spin));` statement to see the effect.

If you want to create any multi-shape object, you usually need to deal with translation. In Java, the class that encapsulates this sort of thing is called `AffineTransform`. **Affine transform** is a term used to describe a special type of transformation involving matrices. You can simply think of a matrix (singular of matrices) as a table structure composed of rows and columns. In computer graphics, the use of matrices is an efficient way to represent coordinate geometry. It is possible to use a matrix to store the position, scale, and rotation of point coordinates. It is also possible to transform the data in the matrix, and ultimately the coordinate geometry, using simple operations (e.g., addition and multiplication). Matrices are too complex an issue to cover in depth here, but here's a good link with information about them: `http://en.wikipedia.org/wiki/Matrix_%28mathematics%29`.

An affine transformation allows shapes to be moved, scaled, rotated, and even sheared without distortion by ensuring that lines that are parallel and/or straight prior to the transformation remain so afterwards. Affine transformations in Processing affect the graphics context and are cumulative. In other words, If I rotate with the function call `rotate(PI)`, and then again with `rotate(HALF_PI)`, the graphics context will be rotated 1 1/2 pi, or 270 degrees. What has already been drawn on the screen won't be effected, but any new data added to the screen will now be initially rotated 1 1/2 pi. This same cumulative effect occurs when using the `rotate()`, `translate()`, and `scale()` functions. Transformations can be confusing when you first start using them. Here's a simple example that uses a series of transformations to construct a little truck (see Figure 9-12):

```
//Toy Truck
int truckW = 300;
int truckH = 100;
int truckX = -truckW/2;
int truckY = -truckH/2;

void setup(){
  size(500, 200);
  background(0);
  smooth();

  //body1
  translate(width-truckW/2-50, height/2);
  rect(truckX, truckY, truckW, truckH);

  //body2
  translate(-width+316, 17);
  scale(.2, .65);
  rect(truckX, truckY, truckW, truckH);

  //body3
  translate(-width+250, 20);
  scale(.7, .59);
  noStroke();
  rect(truckX, truckY, truckW, truckH);

  //tires
  stroke(255);
  strokeWeight(7);
  scale(.70, .75);
  translate(160, 172);
  ellipse(truckX, truckY, truckW, truckH);
  translate(1300, 0);
  ellipse(truckX, truckY, truckW, truckH);
  translate(1800, 0);
  ellipse(truckX, truckY, truckW, truckH);
  translate(400, 0);
  ellipse(truckX, truckY, truckW, truckH);

  //window mask
  translate(-3355, -284);
  scale(.7, .9);
  noStroke();
  fill(0);
  triangle(truckX, truckY + truckH, truckX+truckW, ➥
           truckY, truckX, truckY);
}
```

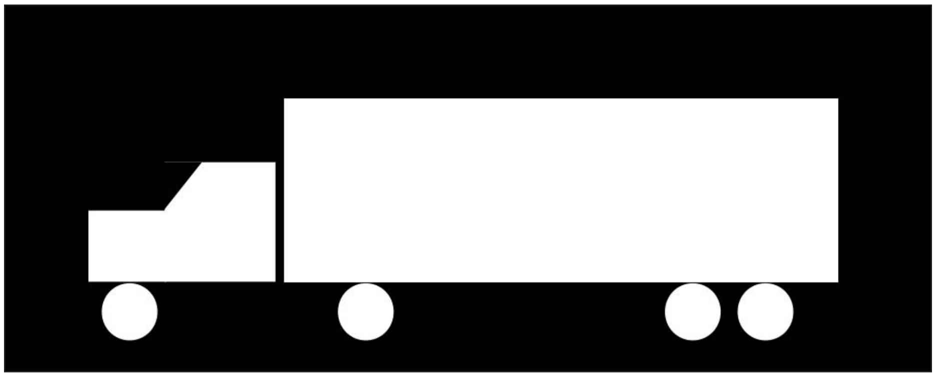


Figure 9-12. Toy Truck sketch

When looking at the code in the last example, I hope some of you protested, exclaiming "Hey, you can't use all those magic numbers!" It's true, it isn't good practice to use magic numbers. And this really is a pretty lousy implementation of a truck (although it's kind of cute, don't you think?). The problem I faced was dealing with all the accumulating matrix transformations and having to manually plug in and try different values until I got my truck. Because of the scaling effect, some of the translation values got pretty wacky, with a few values far exceeding the display window dimensions. Fortunately, there is a much better way to handle resetting the matrix values, which I'll discuss shortly.

The last basic shape function I'll cover is `quad()`. This function works just like the `triangle()` function, but uses four vertices instead of three. The function requires eight arguments, defining the `x` and `y` components of each of the four vertices. Unlike the `rect()` function, in which each corner is 90 degrees, `quad()` can create a four-point polygon with different angles for each corner. In the following simple `quad()` example, I illustrate this using the `random()` function for each vertex (see Figure 9-13):

```
// Simple Quad
size(300, 300);
background(0);
noStroke();
smooth();
// quad(p1.x, p1.y, p2.x, p2.y, p3.x, p3.y, p4.x, p4.y)
quad(50+random(-25, 25), 50+random(-25, 25), ➥
    250+random(-25, 25), 50+random(-25, 25), ➥
    250+random(-25, 25), 250+random(-25, 25), ➥
    50+random(-25, 25), 250+random(-25, 25));
```

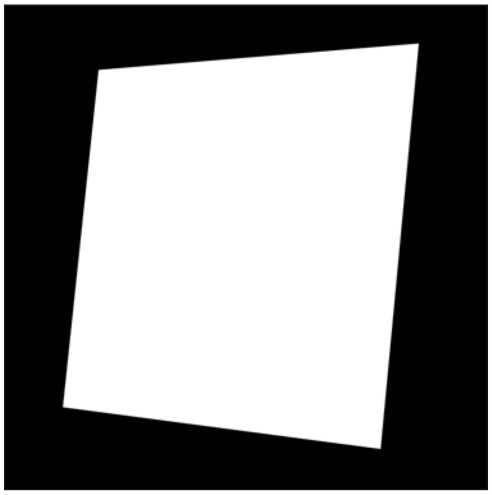


Figure 9-13. Simple Quad sketch

Next is a more interesting quad() example that also includes Processing's resetMatrix() function, used to reset the affine transformation matrix discussed earlier (see Figure 9-14):

```
// Disintegrating Quad Wall

float randShift = .2;
int quadW = 15;
int quadH = quadW;
float[]q = { -quadW/2, -quadH/2, quadW, quadH };

void setup() {
  size(600, 600);
  background(255);
  smooth();
  noStroke();
  /* generate a table structure of
   quads progressivley adding more
   randomization to each quad */
  for (int i=0, k=1; i<height-quadH; i+=quadH, k++){
    /* resetting the transformation matrix
     keeps the translations from continually
     accumulating. Try commenting out the
     resetMatrix() call to see the effect. */
    resetMatrix();
    translate(0, quadH*k);
    for (int j=0; j<width-quadW; j+=quadW){
      translate(quadW, 0);
      fill(random(0, 255));
      // r(k) is a function call
      quad(q[0]+r(k), q[1]+r(k), ➥
      q[0]+q[2]+r(k), q[1]+r(k), ➥
      q[0]+q[2]+r(k), q[1]+q[3]+r(k), ➥
      q[0]+r(k), q[1]+q[3]+r(k));
    }
  }
}

float r(int i){
  return random(-i*randShift, i*randShift);
}
```

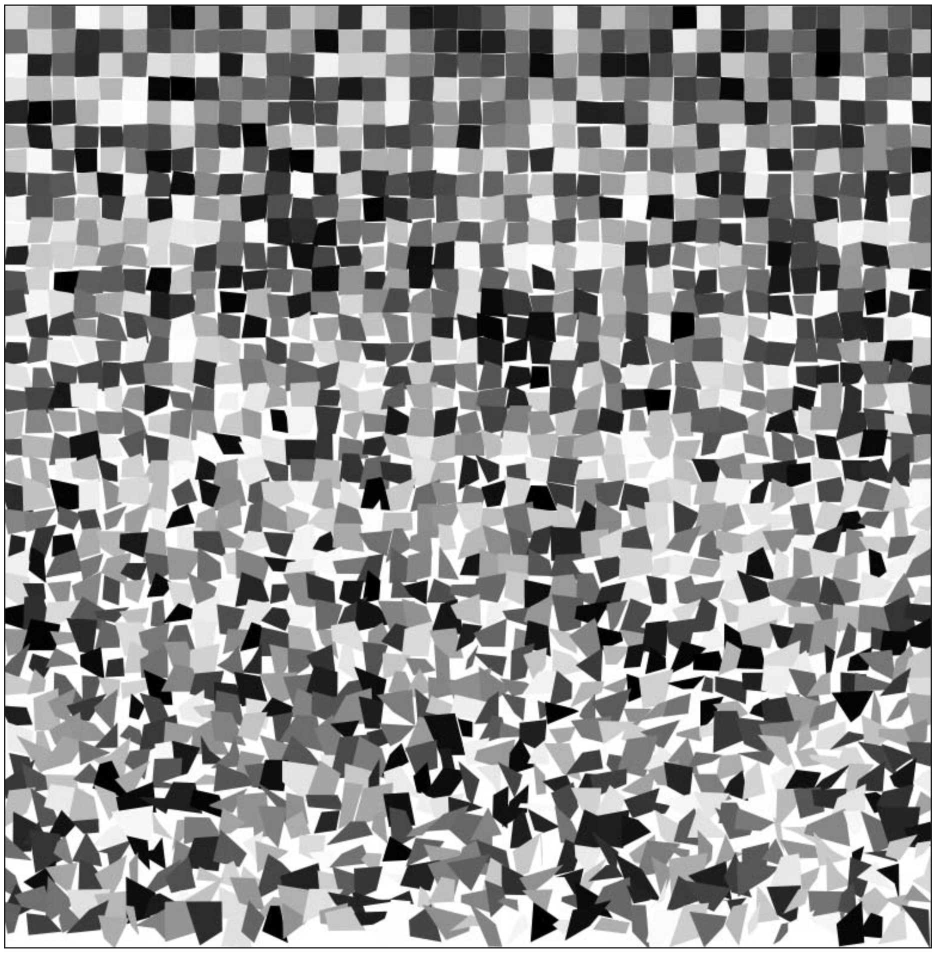


Figure 9-14. Disintegrating Quad Wall sketch

This is a dense little sketch, so it may not be apparent at first glance what's happening. The nested for loop generates a table structure of quadrilaterals. In the outer for loop head, I initialized two variables (i and k) since I needed one variable to be incremented by quadH (i) and one to be incremented by 1 (k). The first line in the loop, after the comment, is the function call resetMatrix(). This simple step clears the transformation matrix at the beginning of each iteration of the loop. Since the matrix was getting reset each time, I couldn't use it to step down the rows of the table. I solved this by multiplying the second argument (which controls the y position) in the outer loop translate() call with k, as in translate(0, quadH*k). However, by resetting the matrix, I was able to repeatedly call translate(quadW, 0) within the inner loop, which took care of stepping the quadrangles across the display window. Finally, I set up a convenient little function, r(val), to add a random offset to the quadrangle points. Since I used the outer loop variable k to specify the random range, randomization increased as the loop ran. Try changing the randShift variable at the top of the sketch to see how it affects the output.

Before moving on to some other drawing functions, I want to provide one more example that will hopefully help clarify the matrix transformations. The next sketch uses a convenient Processing function, printMatrix(), which actually prints to the screen the current contents of the transformation matrix. As I mentioned earlier, transformations affect the virtual drawing, or graphics context, so in this next example, all I include are the actual matrix transformations—which is all I need to see how the individual transformation function calls affect the overall matrix. You don't need to spend too much time on this, but just look at which part of the matrix is affected by each call (see Figure 9-15):

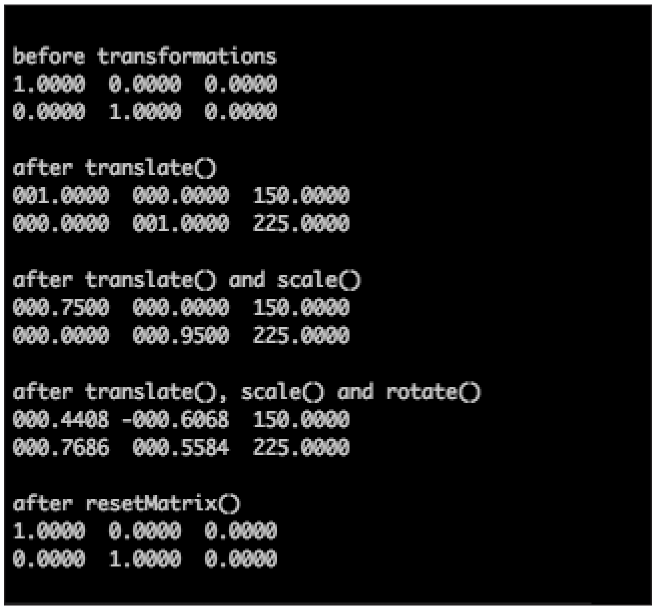
```
// printMatrix()
//initial state
println(" before transformations");
printMatrix();

//translate
translate(150, 225);
println(" after translate()");
printMatrix();

//scale
scale(.75, .95);
println(" after translate() and scale()");
printMatrix();

//rotate
rotate(PI*.3);
println(" after translate(), scale() and rotate()");
printMatrix();

//reset
resetMatrix();
println(" after resetMatrix()");
printMatrix();
```



```
before transformations
1.0000  0.0000  0.0000
0.0000  1.0000  0.0000

after translate()
001.0000  000.0000  150.0000
000.0000  001.0000  225.0000

after translate() and scale()
000.7500  000.0000  150.0000
000.0000  000.9500  225.0000

after translate(), scale() and rotate()
000.4408 -000.6068  150.0000
000.7686  000.5584  225.0000

after resetMatrix()
1.0000  0.0000  0.0000
0.0000  1.0000  0.0000
```

Figure 9-15. printMatrix() sketch

Plotting shapes

Although the basic shape functions `rect()`, `ellipse()`, `triangle()`, and `quad()` offer some convenience, they are also pretty limited. Processing has a more general and versatile approach to shape creation that you've looked at before. Utilizing `beginShape()`, `endShape()`, and a series of `vertex()` commands, any shape can be created using Processing. These shapes can include combinations of both straight and curved sections. `beginShape()` also offers some advanced modes, allowing more complex polygonal structures to be created. You've used these functions before, so some of this will likely be review. I'll begin by examining the default closed mode, which will generate a polygon (see Figure 9-16):

```
//Octagon
size(400, 400);
background(255);
smooth();
int margin = 50;
fill(0);
stroke(127);
strokeWeight(6);
beginShape();
vertex(3, height/2);
vertex(margin, margin);
vertex(width/2, 3);
vertex(width-margin, margin);
vertex(width-3, height/2);
vertex(width-margin, height-margin);
vertex(width/2, height-3);
vertex(margin, height-margin);
endShape(CLOSE);
```

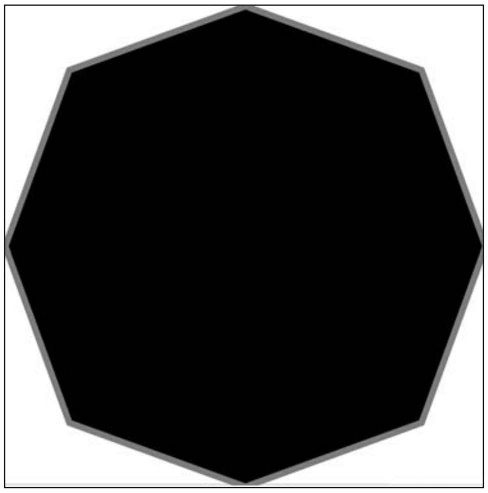


Figure 9-16. Octagon sketch

The default beginShape() mode is a filled but open shape. By adding the CLOSE argument to the endShape() call, the shape is closed (by connecting the last vertex to the initial vertex), as illustrated in the Octagon example. Plotting the octagon wasn't too difficult because an 8-sided regular shape on a 4-sided window boils down to plotting vertices at the corners and midpoints of the display window—but what about switching to a pentagon or heptagon (7 sides), or even an enneakaidecagon (a 19-sided polygon)? Using a little trig, you can very easily create a general-purpose polygon creator. In the following example, I've parameterized lots of details to make the makePoly() function useful. I've also called this function a bunch of times in the sketch to showcase the range of images it can create (see Figure 9-17).

```
/*
 Polygon Creator
 Ira Greenberg, December 26, 2005
 */
void setup(){
  size(600, 600);
  background(127);
  smooth();
  /*
```

```
   //complete parameter list
   makePoly(x, y, pts, radius 1, radius 2, initial rotation, ➥
   stroke Color, stroke Weight, fill Color, endcap, stroke join)
   */
  // makePoly function calls
  makePoly(width/2, height/2, 72, 420, 270, 45, color(0, 0, 0), ➥
      16, color(255, 255, 255));
  makePoly(width/2, height/2, 16, 300, 250, 45, color(200, 200, 200),➥
      10, color(20, 20, 20));
  makePoly(width/2, height/2, 60, 210, 210, 45, color(255,255,255), ➥
      8, color(0,0,0), PROJECT, ROUND);
  makePoly(width/2, height/2, 60, 200, 155, 45, color(120, 120, 120),➥
      6, color(255, 255, 255), PROJECT, ROUND);
  makePoly(width/2, height/2, 50, 280, -200, 45, ➥
      color(200, 200, 200), 6, color(50, 50, 50), PROJECT, ROUND);
  makePoly(width/2, height/2, 8, 139, 139, 68, color(255, 255, 255), ➥
      5, color(0,0,0));
  makePoly(width/2, height/2, 24, 125, 60, 90, color(50, 50, 50), 12,➥
      color(200, 200, 200), ROUND, BEVEL);
  makePoly(width/2, height/2, 4, 60, 60, 90, color(0,0,0), 5, ➥
      color(200,200,200), ROUND, BEVEL);
  makePoly(width/2, height/2, 4, 60, 60, 45, color(255, 255, 255), 5,➥
      color(20, 20, 20), ROUND, BEVEL);
  makePoly(width/2, height/2, 30, 30, 30, 90, color(75, 75, 75), 10, ➥
      color(60,60,60), ROUND, BEVEL);
  makePoly(width/2, height/2, 30, 28, 28, 90, color(255, 255,255), 2,➥
      color(60,60,60), ROUND, BEVEL);
  makePoly(width/2, height/2, 24, 10, -25, 45, #000000, .75, ➥
      color(255, 255, 255), SQUARE, MITER);
}

//default - if no args passed
void makePoly(){
  // call main makePoly function
  makePoly(width/2, height/2,  4, width/4, width/4, ➥
    45, #777777, 4, #AAAAAA, SQUARE, MITER);
}

// x, y, pts args
void makePoly(float x, float y, int pts){
 // call main makePoly function
  makePoly(x, y, pts, width/4, width/4, ➥
    45, #777777, 4, #AAAAAA, SQUARE, MITER);
}
```

```
// x, y, pts, rad1, rad2 args
void makePoly(float x, float y, int pts, float rad1, float rad2){
 // call main makePoly function
  makePoly(x, y, pts, rad1, rad2, 45, #777777, 4, #AAAAAA, ➥
    SQUARE, MITER);
 }

// x, y, pts, rad1, rad2,, initRot, strokeCol, strokeWt, fillCol args
void makePoly(float x, float y, int pts, float rad1, float rad2, ➥
  float initRot, color strokeCol, float strokeWt, color fillCol){
// call main makePoly function
  makePoly(x, y, pts, rad1, rad2, initRot, strokeCol, strokeWt, ➥
    fillCol, SQUARE, MITER);
}

// main function - called by other overloaded functions/methods
void makePoly(float x, float y, int pts, float rad1, float rad2, ➥
    float initRot, color strokeCol, float strokeWt, ➥
    color fillCol, int endCap, int endJoin){

  float px = 0, py = 0, angle = initRot;
  stroke(strokeCol);
  strokeWeight(strokeWt);
  strokeCap(endCap);
  strokeJoin(endJoin);
  fill(fillCol);

  beginShape();
  for (int i = 0; i< pts; i++){
    if (i%2 == 0){
      px = x+cos(radians(angle))*rad1;
      py = y+sin(radians(angle))*rad1;
    }
    else {
      px = x+cos(radians(angle))*rad2;
      py = y+sin(radians(angle))*rad2;
    }
    vertex(px, py);
    angle+=360/pts;
  }
  endShape(CLOSE);
}
```

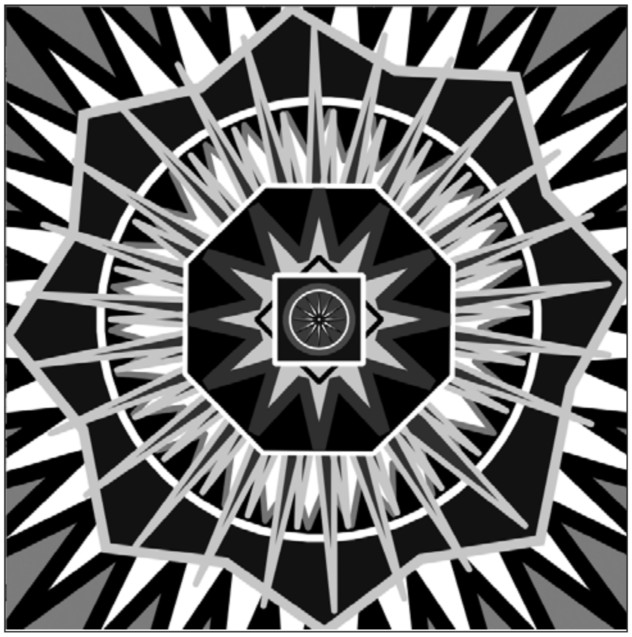


Figure 9-17. Polygon Creator sketch

The trig functions, based on unit circle relationships, do most of the real work in this sketch. I've included `sin()` and `cos()` expressions in numerous examples earlier in the book, and a general discussion about trig and the unit circle can be found in Chapter 4 and also in Appendix B. In addition, I owe some debt to Ric Ewing, who a couple years ago published a series of wonderful drawing methods for ActionScript that have informed my own approach (see `www.macromedia.com/devnet/flash/articles/adv_draw_methods.html`).

I nested the trig functions in a conditional `if...else` structure as a simple way to generate two radii. I've used this technique in other places in the book as well. The modulus operator, `%`, returns the remainder of the division between the two operands (e.g., **9 % 5 = 4**, as 5 goes into 9 once, leaving a remainder of 4). I used `(i%2 == 0)` in the head of the conditional so that points would alternate between radius 1 and radius 2. If you don't see it yet, this works because only even numbers will return 0, but odds won't.

The other possibly surprising aspect of this sketch is the use of five `makePoly()` function definitions, in which—in all but one of them—another `makePoly()` function is called. This approach of creating multiple functions (or methods) with the same name is a common technique in OOP, and is referred to as **method overloading**. As long as the method signatures are different, the compiler sees them as unique structures. The signature, in Java and Processing, is a combination of the method name and parameter list (including the number and type of parameters). If the number and/or type of parameters is different in the signatures, then the compiler sees the methods as unique, regardless of whether the methods share the same name. This is a convenient implementation, as it gives users a choice of how to call/use the method.

I created five versions of the makePoly() function. It is possible to create a lot more, although it's probably not necessary. Only one of the makePoly() functions includes the full parameter list; the others use a partial list, and one of them includes no parameter. Each of the four makePoly() functions (without the full parameter list) internally calls the makePoly() function with the full list. I did this so that the actual plotting algorithm could be put in one place—inside the makePoly() function with the full parameter list.

When one of the functions without the full parameter list is called, it internally passes the arguments it received—adding default values for the remaining ones—to the makePoly() function with the full parameter list and plotting implementation. I recommend trying to add your own version of a makePoly() function to the sketch to get a better sense of how this all works.

Regular and star polygons are fine, but there are some other useful shapes that you can build by modifying the last algorithm a bit. Here's a sprocket creator sketch, in which I call the makeSprocket()function a bunch of times to demonstrate the range of shapes the function is capable of generating (see Figure 9-18).

```
/*
 Sprocket Creator
 Ira Greenberg, December 27, 2005
 */

void setup(){
  size(600, 600);
  background(65);
  smooth();

  makeSprocket(width/2, height/2, 20, 280, 440, 45, color(0, 0, 0), ➥
      20, color(255, 255, 255), SQUARE, MITER);
  makeSprocket(width/2, height/2, 120.0, 275, -230, 45, ➥
      color(200, 200, 200), 2, color(20, 20, 20), SQUARE, ROUND);
  makeSprocket(width/2, height/2, 20.0, 250, 120, 45, color(0, 0, 0),➥
      12, color(255, 255, 255), PROJECT, MITER);
  makeSprocket(width/2, height/2, 8.0, 120, 190, 45, ➥
      color(20, 20, 20), 14, color(200, 200, 200), PROJECT, MITER);
  makeSprocket(width/2, height/2, 8.0, 120, 170, 22.5, ➥
       color(245, 245, 245), 20, color(10, 10, 10), PROJECT, MITER);
  makeSprocket(width/2, height/2, 25.0, 90, 35, 45, ➥
       color(255, 255, 255), 2, color(0, 0, 0), PROJECT, MITER);
  makeSprocket(width/2, height/2, 8.0, 25, 10, 45, ➥
      color(127, 127, 127), 4, color(255, 255, 255), PROJECT, MITER);
}

void makeSprocket(float x, float y, float spokes, float rad1, ➥
     float rad2, float initRot, color strokeCol, float strokeWt, ➥
     color fillCol, int endCap, int endJoin){
```

```
  float px = 0, py = 0, angle = initRot;
  float ang = (360.0/spokes)/2.0;
  float ang2 = (360.0/spokes)/4.0;
  stroke(strokeCol);
  strokeWeight(strokeWt);
  strokeCap(endCap);
  strokeJoin(endJoin);
  fill(fillCol);

  beginShape();
  for (int i = 0; i<spokes; i++){
    px = x+cos(radians(angle))*rad1;
    py = y+sin(radians(angle))*rad1;
    vertex(px, py);
    angle+=ang;
    px = x+cos(radians(angle))*rad1;
    py = y+sin(radians(angle))*rad1;
    vertex(px, py);
    angle+=ang2;
    px = x+cos(radians(angle))*rad2;
    py = y+sin(radians(angle))*rad2;
    vertex(px, py);
    angle+=ang2;
  }
  endShape(CLOSE);
}
```

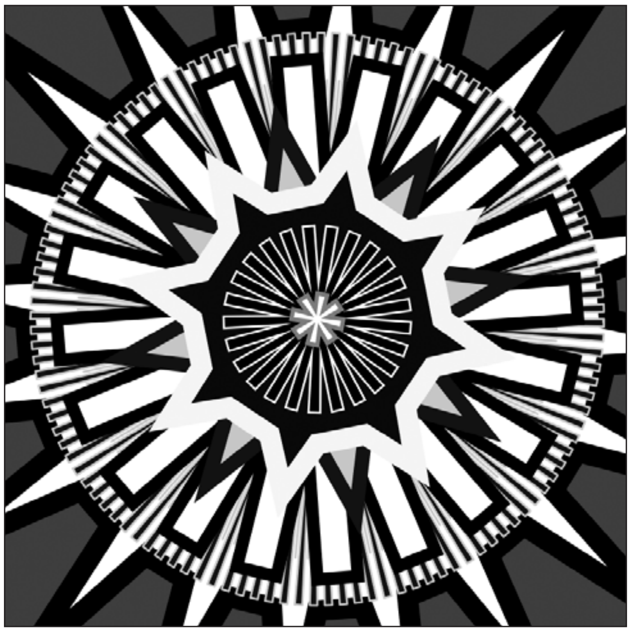


Figure 9-18. Sprocket Creator sketch

The Sprocket Creator sketch is similar to the Polygon Creator sketch. The main difference is the few extra trig expressions in the makeSprocket() function. Instead of plotting vertices at equal angle rotations, each spoke of the sprocket is composed of three vertices. The first and second share the same radius and are the outer edge of the spoke. The third vertex is an inner radius, and is at the base between each spoke. Notice also that I used two different angle values. I somewhat arbitrarily defined the angle between the two spoke vertices (within the spoke) as twice as large as the angle between the spokes. The final angle incrementation (angle+=ang2) ensures that there is symmetry between each spoke.

As reading descriptions like this can be confusing, I suggest playing with the sketch and creating some of your own sprockets. You might also want to try to spin your own shape by modifying the existing drawSprocket() method.

Creating hybrid shapes

The beginShape() and endShape() functions allow you to combine straight and curved lines to form hybrid shapes. Once you introduce curves into to your shapes, calculating coordinates can get a little tricky. It often helps to break the problem down into more manageable sections—in the next two example sketches, I do just that. In the first sketch, I hack out a list of coordinates (using magic numbers) and individual drawing function calls that create a hybrid shape. In the second sketch, I develop a more general plotting algorithm, removing all the magic numbers. Here's the initial code in all its magic number glory (see Figure 9-19):

```
// Hybrid Shape
size(600, 600);
curveTightness(-.4);
smooth();

beginShape();
curveVertex(200, -300);
curveVertex(200, 100);
curveVertex(400, 100);
curveVertex(400, -300);

vertex(500, 100);
vertex(500, 200);

curveVertex(800, 200);
curveVertex(500, 200);
curveVertex(500, 400);
curveVertex(800, 400);

vertex(500, 500);
vertex(400, 500);
```

```
curveVertex(400, 800);
curveVertex(400, 500);
curveVertex(200, 500);
curveVertex(200, 800);

vertex(100, 500);
vertex(100, 400);

curveVertex(-200, 400);
curveVertex(100, 400);
curveVertex(100, 200);
curveVertex(-200, 200);

vertex(100, 100);
endShape(CLOSE);
```

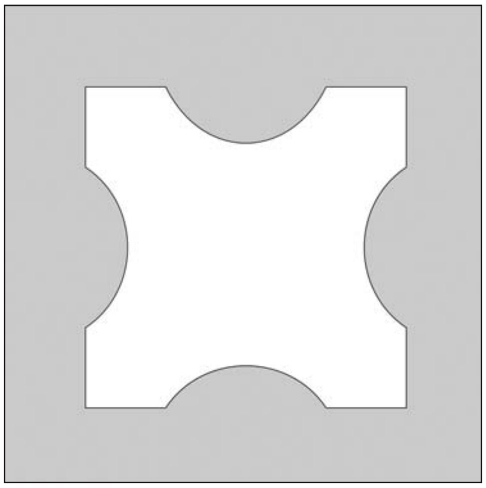


Figure 9-19. Hybrid Shape sketch

The next step was looking at the ugly code I had written and seeing if I could figure out an algorithm to generalize the form. This backward approach—making before thinking—seems to be a general pattern I use when I get stuck. I guess if you're better at math than me, you might be able to lay down clean algorithms by just thinking about problems, but I usually have to get my hands dirty first. Because the shape created in the sketch is symmetrical, and there is a recurring pattern of function calls in the code, I knew it would be possible, without too much hair pulling, to figure out the algorithm. I also knew, because of the radial symmetry, that using trig functions would somehow be the easiest solution. Here's what I got (see Figure 9-20):

```
// Hybrid Shape 2
size(600, 600);
curveTightness(-.4);
smooth();
strokeWeight(10);
float sides = 8;
float angle = 360.0/sides/2;
float px = 0, py = 0;
float cx = 0, cy = 0;
float ang = 360.0/(sides+sides/2.0);
float ang2 = ang/2.0;
float rad1 = 250.0;
float rad2 = rad1*2.0;
int x = width/2;
int y = height/2;

beginShape();
for (int i=0; i<sides; i++){
  cx = x+cos(radians(angle))*rad2;
  cy = y+sin(radians(angle))*rad2;
  curveVertex(cx, cy);
  px = x+cos(radians(angle))*rad1;
  py = y+sin(radians(angle))*rad1;
  curveVertex(px, py);
  angle+=ang;
  px = x+cos(radians(angle))*rad1;
  py = y+sin(radians(angle))*rad1;
  curveVertex(px, py);
  cx = x+cos(radians(angle))*rad2;
  cy = y+sin(radians(angle))*rad2;
  curveVertex(cx, cy);
  px = x+cos(radians(angle))*rad1;
  py = y+sin(radians(angle))*rad1;
  vertex(px, py);
  angle+=ang2;
  px = x+cos(radians(angle))*rad1;
  py = y+sin(radians(angle))*rad1;
  vertex(px, py);
}
endShape(CLOSE);
```

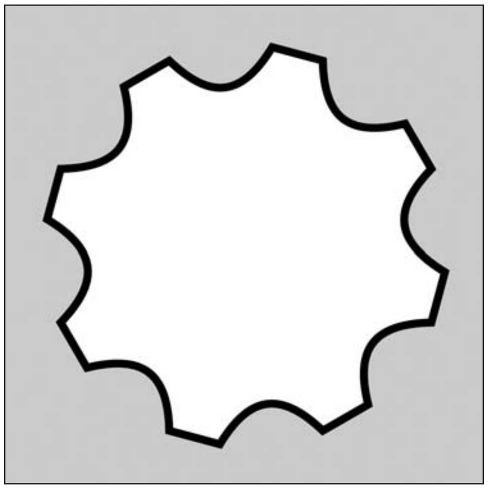


Figure 9-20. Hybrid Shape 2 sketch

This sketch makes a good gear shape. If you change the value of the sides variable in the example, the shape scales nicely. I think it would be a good exercise to try to "functionalize" this last example. In other words, try sticking the main drawing routine in a function with a bunch of parameters, as I did with the polygon and sprocket examples. Besides curveVertex(), you can also use bezierVertex() in conjunction with regular vertex() calls, but I'll also leave that for you to try on your own.

The other shape modes

There are a number of additional shape modes. Many of these are more applicable to 3D than 2D, for creating a skin or polygonal mesh around a 3D form. In the 3D chapters, you'll learn how to plot some 3D forms. For now, I'm just going to give a brief overview of what the other modes do. The first two you'll look at, TRIANGLES and QUADS, don't form a contiguous mesh, but rather simply create individual triangles and quadrangles from lists of coordinates. In the next example, I'll generate 90 triangles using a for loop and the TRIANGLES mode (see Figure 9-21):

```
//Random Triangles
size (500, 500);
background(255);
smooth();
beginShape(TRIANGLES);
for (int i=0; i<90; i++){
  stroke(random(0, 200));
  fill(random(225, 255), 150);
  strokeWeight(random(.5, 5));
  vertex(random(width), random(height));
}
endShape();
```

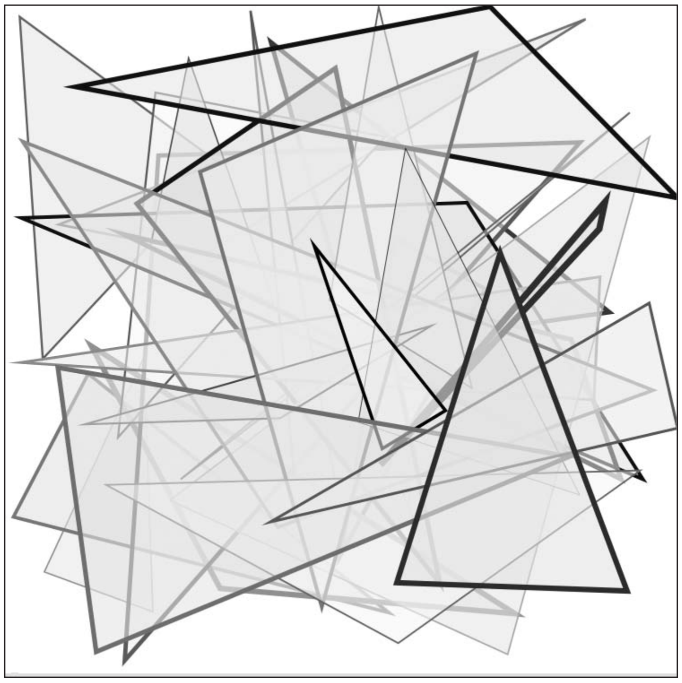


Figure 9-21. Random Triangles sketch (TRIANGLES mode)

Although you could relatively easily write your own function to pull this sort of thing off, it is still convenient having this option. It would be a good exercise to try to re-create the functionality of the beginShape(TRIANGLES) mode using the triangle() function or the no-argument version of beginShape().

Notice the extra argument in the command fill(random(225, 255), 150);, after the closed parentheses of the random call. This optional second argument allows you to control the alpha, or transparency, of the grayscale fill. The value range is from 0 to 255 (0 being transparent and 255 being completely opaque). If you don't specify an argument, the default value is 255. The stroke() and fill() commands each allow you to specify one, two, three, or four arguments. In the next chapter, I'll discuss color and imaging, and concepts like alpha in greater detail.

beginShape(QUADS) mode works similarly to TRIANGLE mode, except that shapes are closed in groups of four vertex() calls instead of three. In this next example, I generate random quads, bounded within a rectangular region, forming a sort of shape texture (see Figure 9-22).

```
//Random Bounded Quads
size (250, 250);
background(255);
smooth();
int[]boundinBox = {50, 50, 150, 150};
fill(0);
```

```
rect(boundinBox[0], boundinBox[1], boundinBox[2], boundinBox[3]);
strokeJoin(BEVEL);
beginShape(QUADS);
for (int i=0; i<1000; i++){
  stroke(random(50, 255));
  fill(0, 0);
  strokeWeight(random(.5, 2));
  vertex(random(boundinBox[0], boundinBox[0]+boundinBox[2]), ➥
        random(boundinBox[1], boundinBox[1]+boundinBox[3]));
}
endShape();
```

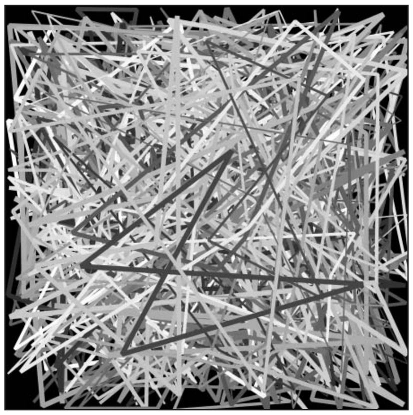


Figure 9-22. Random Bounded Quads sketch (QUADS mode)

You'll notice in the output that the shapes don't form a pattern of little rectangles, even though they were generated by using QUADS mode. This is because a quadrilateral is not necessarily a rectangular structure. It is possible to create a quadrilateral in which no two sides are parallel; this shape is sometimes referred to as a trapezium. If any two of the sides are parallel, it's usually called a trapezoid, which I assume you remember from some math class. There is a simple and complex classification for polygons, which affects quadrilaterals as well. Simple polygons don't intersect with themselves, while complex ones do. Here's an example of each, using QUADS mode (see Figure 9-23):

```
//Simple/Complex Quads
size (300, 150);
smooth();
beginShape(QUADS);
//simple quad
vertex(25, 40);
vertex(125, 30);
vertex(120, 120);
vertex(20, 118);
//complex quad
vertex(175, 40);
vertex(275, 30);
vertex(170, 118);
vertex(270, 120);
endShape();
```

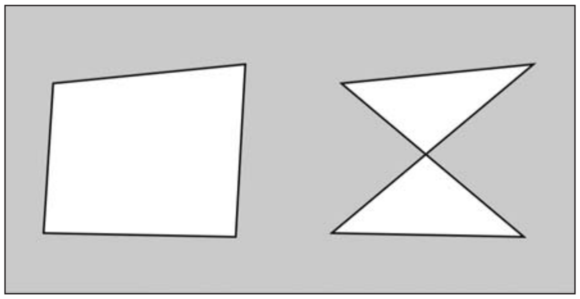


Figure 9-23. Simple/Complex Quads sketch

Simple vs. complex polygons is a big issue in 3D, where quadrilateral surfaces can become non-planar and cause anomalies when rendering. Triangles, on the other hand, are always planar. For our current purposes, in 2D, complex quads just form an interesting pattern. If you want to learn more about quads in general, check out `http://en.wikipedia.org/wiki/Quadrilateral` (I especially like the taxonomic classification chart). One other minor but significant point about using either `TRIANGLE` or `QUADS` mode is that you'll want to make sure that you use the right number of `vertex()` commands. The number of commands should be divisible by 3 for `TRIANGLES` and 4 for `QUADS`. Additional `vertex()` lines that aren't grouped in three or four lines, respectively, will be disregarded. For example, as shown in Figure 9-24, the fourth and fifth lines in the following sequence will be disregarded:

```
size (120, 120);
beginShape(TRIANGLES);
vertex(20, 20);
vertex(100, 100);
vertex(40, 100);
vertex(30, 60);
vertex(20, 50);
endShape();
```

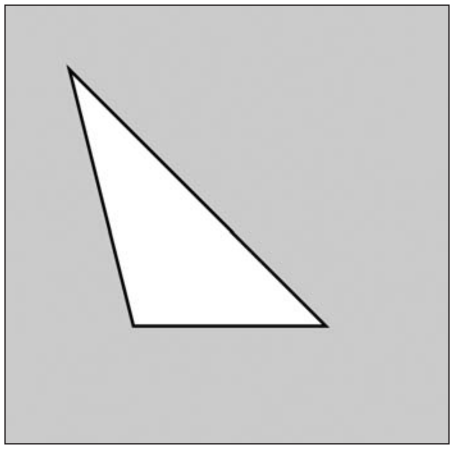


Figure 9-24. Disregarded extra vertex() calls

The last three modes I'll discuss briefly are TRIANGLE_STRIP, TRIANGLE_FAN, and QUAD_STRIP. These modes build contiguous meshes of polygons, which is very useful in 3D. They work sort of similarly to the last two modes discussed, except that the multiple forms created are attached. Also, extra vertex() lines are not disregarded. However, a minimum number of vertex() commands need to be issued—three for TRIANGLE_STRIP and TRIANGLE_FAN, and four for QUAD_STRIP—before any shapes are rendered. Here's an example using TRIANGLE_STRIP mode (see Figure 9-25):

```
// TRIANGLE_STRIP Mode
size(400, 400);
smooth();
int x = width/2;
int y = height/2;
int outerRad = 150;
int innerRad = 200;
float px = 0, py = 0, angle = 0;
float pts = 36;
float rot = 360.0/pts;

beginShape(TRIANGLE_STRIP);
for (int i=0; i<pts; i++) {
  px = x+cos(radians(angle))*outerRad;
  py = y+sin(radians(angle))*outerRad;
  angle+=rot;
  vertex(px, py);
  px = x+cos(radians(angle))*innerRad;
  py = y+sin(radians(angle))*innerRad;
  vertex(px, py);
  angle+=rot;
}
endShape();
```

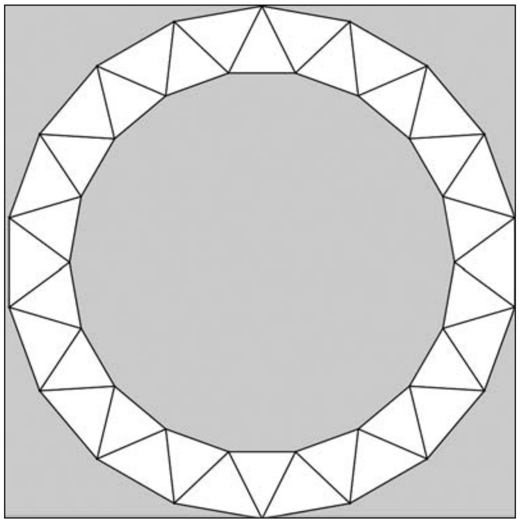


Figure 9-25. TRIANGLE_STRIP Mode sketch

I reused the basic trig functions I've been using throughout the book for generating an ellipse. By setting an outer and inner radius, this shape could be useful as an end cap for a hollow 3D cylinder. Next, I simply decreased the radii and fill color value (from white to black) each loop iteration, creating a spiral (see Figure 9-26):

```
// TRIANGLE_STRIP Spiral
size(400, 400);
background(0);
smooth();
int x = width/2;
int y = height/2;
int outerRad = 160;
int innerRad = 200;
float px = 0, py = 0, angle = 0;
int pts = 36;
float rot = 360.0/pts;
int fillCol = 255;
int fillfade = fillCol/pts;

beginShape(TRIANGLE_STRIP);
for (int i=0; i<pts; i++) {
  px = x+cos(radians(angle))*outerRad;
  py = y+sin(radians(angle))*outerRad;
  angle+=rot;
  vertex(px, py);
  px = x+cos(radians(angle))*innerRad;
  py = y+sin(radians(angle))*innerRad;
  vertex(px, py);
  outerRad-=4;
  innerRad-=4.25;
  fill(fillCol);
  fillCol-=fillfade;
  angle+=rot;
}
endShape();
```

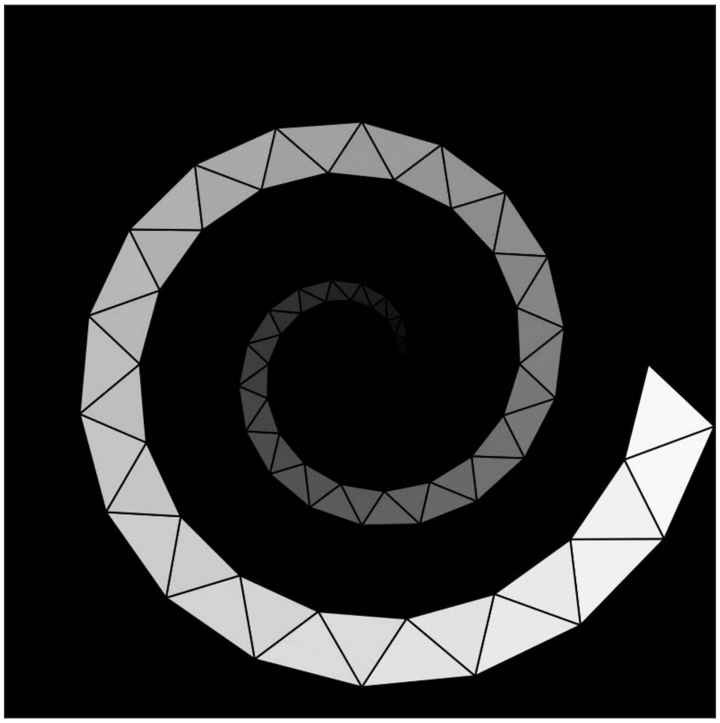


Figure 9-26. TRIANGLE_STRIP Spiral sketch

Tessellation

Another interesting thing you can do with these functions is create a tessellation. A **tessellation** is a pattern of shapes that fit together, without any gaps, covering a surface. For more of a definition, have a look at what Wikipedia has to say about tessellations, at `http://en.wikipedia.org/wiki/Tessellation`. `TRIANGLE_FAN` mode is a little tricky, as it needs to rotate clockwise, and the initial point should define the center part of the fan. In 3D, forms are commonly converted to and/or rendered as triangle meshes. `TRIANGLE_FAN` mode is a somewhat convenient function for triangulating planar geometry. Triangulating means converting a larger polygon into triangles, which is a form of tessellation. Since triangles are always planar and simple (with regard to polygons—not a commentary on their intelligence), they are easier for the computer to render than quadrilaterals. This mode works by defining the center point of the fan and then selecting the vertices in a clockwise fashion along the perimeter of the shape you want to triangulate. Once again, the trig functions come in handy. Here's an example sketch that triangulates any regular polygon, using TRIANGLE_FAN mode (see Figure 9-27):

```
//TRIANGLE_FAN
size(400, 400);
smooth();
strokeWeight(1.5);
float px = 0, py = 0;
float angle = 0;
float radius = 150;
int pts = 8;
int x = width/2;
int y = height/2;

// needs to rotate clockwise
beginShape(TRIANGLE_FAN);
vertex(x, y);
for (int i=0; i<=pts; i++){
  px = x+cos(radians(angle))*radius;
  py = y+sin(radians(angle))*radius;
  vertex(px, py);
  angle+=360/pts;
}
```

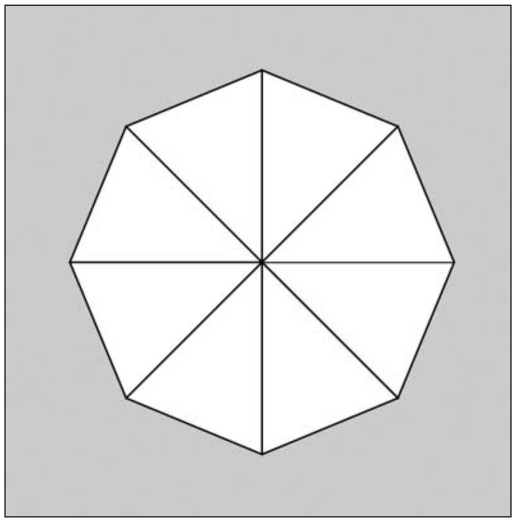


Figure 9-27. TRIANGLE_FAN sketch

Next, I'll use the last solution as part of a sketch for a tessellated plane (see Figure 9-28):

```
/*
 Tessellated Plane
 Ira Greenberg, December 30, 2005
 */
void setup(){
  size(400, 400);
  smooth();
  tesselate(6, 20);
}

void tesselate(int points, int radius){
  /* catch and handle out of
   range point count */
  if (points<=5){
    points = 4;
  }
  else {
    points  = 6;
  }

  // eventually add some more patterns
  switch(points){
  case 4:
    for (int i = 0, k = 0; i<=width+radius; i+=radius*2, k++){
      for (int j = 0; j<=height+radius; j+=radius*2){
        drawPoly(i-radius, j-radius, points, 0, radius);
        drawPoly(i, j, points, 0, radius);
      }
    }
    break;
  case 6:
    for (float i = 0, k = 0; i<=width+radius; i+=radius*1.5, k++){
      for (float j = 0; j<=height+radius*2; j+=(cos(radians(30))* ➥
           radius)*2){
        if (k%2==0){
          drawPoly(i, j-cos(radians(30))*radius, points, 0, radius);
        }
        else{
          drawPoly(i, j, points, 0, radius);
        }
      }
    }
    break;
  }
}
```

```
// draw triangle fan
void drawPoly(float x, float y, int pts, float initAngle, float rad){
  strokeWeight(1.5);
  float px = 0, py = 0;
  float angle = initAngle;
  // needs to rotate clockwise
  beginShape(TRIANGLE_FAN);
  vertex(x, y);
  for (int i=0; i<=pts; i++){
    fill(255/pts*i);
    px = x+cos(radians(angle))*rad;
    py = y+sin(radians(angle))*rad;
    vertex(px, py);
    angle+= 360/pts;
  }
}
```

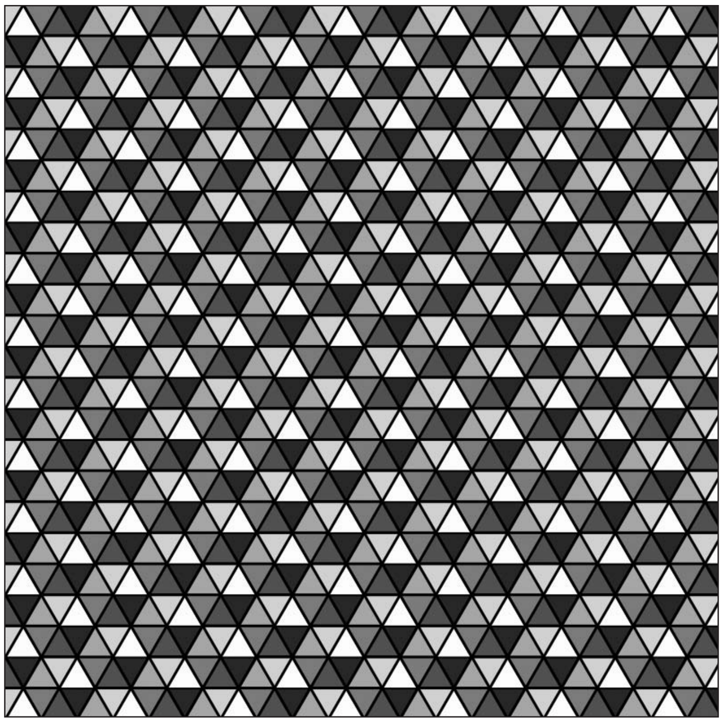


Figure 9-28. Tesselated Plane sketch

This last example is fairly complex; though not much is new. It took some finessing to get the geometry to tile properly and a bit of trig for the hexagon tiling. There are two things that I think are significant about this sketch. The first involves encapsulation and the latter involves exception handling. Both custom functions `tessellate()` and `drawPoly()` involve somewhat complicated implementations. However, to use both functions in the last example, all that needed to be called was `tessellate(6, 20);`. When you encapsulate procedural complexity into self-contained modular units such as functions, it becomes relatively simple to use even the most complexly implemented procedures. This illustrates a general notion of the concept of encapsulation. There is also the more specific but related definition of encapsulation (discussed in Chapter 8) that applies specifically to OOP.

The second point, exception handling, is illustrated by the conditional statement at the top of the `tessellate()` function. The conditional syntax you've seen before, but the concept is new. I only wrote tessellation routines for four- and six-sided polygons. However, someone could conceivably use the function at some point and put in a value other than 4 or 6. Rather than generate an unexpected error, I wrote code to catch the out-of-range value and correct it. Eventually, additional tessellation routines could be added to the program, at which time you would simply update the conditional. The concept of building in code structures to catch and deal with input errors is generally referred to as **exception handling**. In a language like Java, exception handling is formalized, with numerous class structures. Processing doesn't have its own native exception handling structures (but you're free to use Java's structures). Generally speaking, though, a formalized exception handling process is beyond what most people will want to do with Processing. In addition, the places where exception handling is required in Processing (e.g., input/output) are already encapsulated and invisible to the user. Remember that this is kind of the whole point of Processing—to minimize the annoying lower-level stuff, allowing people to more freely express themselves. So there may be times, as in my last example, that you might want to build in some simple safety checks. If, however, you do end up building something that you feel requires a more elaborate and formal error-checking system, you'll want to look into Java's exception handling capabilities. Here's some info from Sun: `http://java.sun.com/docs/books/tutorial/essential/exceptions/`.

Applying OOP to shape creation

The last section of this chapter utilizes an OOP approach. Chapter 8 dealt exclusively with OOP, but I'll review some of the fundamental concepts here as well. OOP is not easy for new coders to get their heads around, so try not to be too hard on yourself if you find some of the material difficult—it really is.

The "object-oriented" part of OOP is an approach to programming that treats concepts (both things and processes) as self-contained modular units. These units, called classes, are like blueprints describing a concept, which includes attributes (properties) and functions (methods) contained within the class definition. For example, if you create a class called `Rectangle`, the properties of this class might include its x and y position, its width and height, its stroke and fill color, and so on. Its methods might include getting and setting its width or height, and drawing itself. To use the `Rectangle` class, you create (instantiate) objects from it. Since the class is like a blueprint, when you make an object, you get a copy of all the properties and methods defined in the class; and each object remains independent from all the other objects made from the same class. This allows you to have ten rectangle objects, each with different postion, size, color, and so on. Since I've gone this far with the rectangle metaphor, let's actually constuct a simple rectangle class and implement it in Processing.

```
void setup(){
  // create new rectangle object
  new Rectangle();
}

// class description
class Rectangle {
}
```

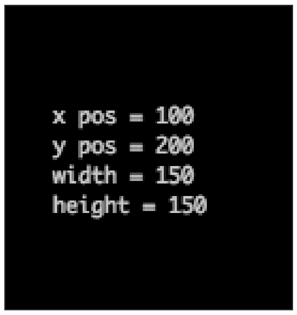
This is all it takes to create a class in Processing. Of course, this class won't do anything, but it's a start. To create a class, you use the `class` keyword followed by the name of the class. You capitalize the name of your classes. To create an object from the `Rectangle` class, you use the `new` keyword. Typically, you also assign the new object to a variable so that you can refer to it later in your program. In the last example, even though I created a `Rectangle` object, I can't communincate with it, as there is no variable assigned a reference to the object. Figure 9-29 shows an improved (and more useful) version of the last sketch:

```
// object (reference) variable
Rectangle r1;

void setup(){
  // create new rectangle object
  r1 = new Rectangle(100, 200, 150, 150);
}

// class description
class Rectangle {

  //class constructor
  Rectangle(int x, int y, int w, int h) {
    println("x pos = "+ x+"\ny pos = " + y);
    println("width = "+ w+"\nheight = " + h);
  }
}
```



```
x pos = 100
y pos = 200
width = 150
height = 150
```

Figure 9-29. OOP println() output example

This wasn't much of an improvement, but I got some output at least. However, there are a bunch of new concepts to discuss in this sketch. At the top of the program, I declared a variable of type Rectangle. Remember that variables in Processing and Java need to be declared of a data type. The data type controls things like how much memory is allocated to the variable and what type of data the variable can be assigned. As I discussed earlier in the book, there are two types of variables in Processing: primitive and reference.

Primitive variables, which were discussed extensively in Chapter 3, are assigned actual values. int, float, and char are examples of primitive variable types. Primitive variables are relativley simple for the computer to handle, as each evaluates to a single value.

Reference variables are more complex, as they each often need to refer to multiple values. To solve this problem, reference variables, rather than directly being assigned a value, hold a reference or link to the data in memory. Thus, the Rectangle r1 variable will ultimately hold a reference to the Rectangle object data, not the data directly. There are some important issues due to this arrangement, but nothing you need to worry about right now. The important point to remember is that the data type of the variable dictates what type of data can be assigned to it; and each class you create is a unique data type. Rectangle r1 can be assigned object references of the Rectangle type (or a subclass of the Rectangle type), but it can't be assigned an int or a String value or reference of another data type.

Looking at the class definition, you'll notice that I added a constructor. Remember that the constructor begins with the same name as the class and also utilizes parentheses for a parameter list. When an object is created from a class, a constructor inside the class is always run. You can think of a constructor as an object initializer function. If you don't explicitly include a constructor, a default (internal) class constructor is called that has no parameters. By creating my own constructor, new Rectangle objects will now use my constructor, instead of the default one. Therefore, if a Rectangle object is now created without four int arguments, an errror will occur. Try replacing r1 = new Rectangle(100, 200, 150, 150); with r1 = new Rectangle(); to see what happens. When the constructor is called successfully, the code within its block (between the curly braces) is executed.

In this revised version of the sketch, the constructor outputs some sensational information about the Rectangle object's properties. (Don't worry, you'll make some images eventually.) Also notice the syntax I used in the println() commands. I generated four lines of output with just two println() calls by using the newline escape sequence, \n. Escape sequences are little character shortcuts you can include in strings to do things like add a new line or include special characters. For example, if you need to include a quote character in output, you'd do it like this: println("\"Help! I'm surrounded by quotes\"");. For a list of escape character sequences, check out http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html. Finally, the following code gets the Rectangle class to actually draw something (see Figure 9-30):

```
// Drawing with OOP
// object (reference) variable
Rectangle r1;

void setup(){
  size(400, 400);
  background(255);
  // create new rectangle object
  r1 = new Rectangle(100, 200, 150, 150);
}

// class description
class Rectangle {

  //class constructor
  Rectangle(int x, int y, int w, int h) {
   rect(x, y, w, h);
  }
}
```

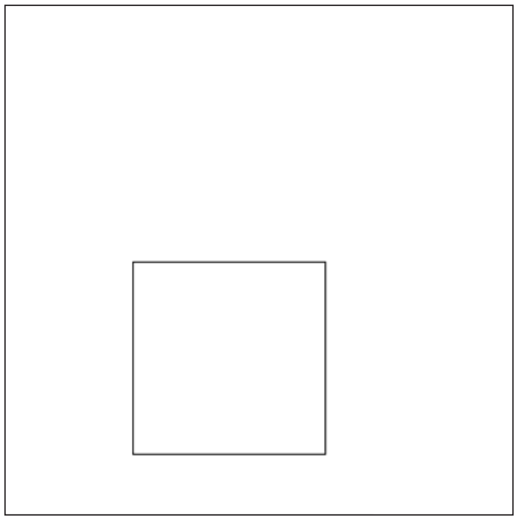


Figure 9-30. Drawing with OOP sketch

Creating a neighborhood

Hopefully, this last sketch makes some sense. Don't worry if it doesn't yet. Also, you'd be correct in thinking that it doesn't really make much sense to go through the bother of creating a class just to call `rect()`, which you could call from within `setup()`. This is certainly true. So let's add some stuff to make all this class business worth it. Let's build a simple house, with a door, some windows, and a roof, and then let's put a couple houses together to form a little neighborhood. I'm going to make a class for each of the following elements: door, window, roof, and house. In this example, I'll keep things simple by not adding decorative features like paint, shingles, cornices, and so on. And I'll keep all the classes within the same PDE file (which isn't really necessary). Later in the book, I'll show you how to begin working with multiple files in separate Processing tabs.

Door class

In designing these really simple classes, it helps to think a little bit about how the house will fit together. I'll begin with the door—what properties will the door have? I want to keep everything really simple for now. The door will have an x position, a y position, width and height, and a doorknob. What methods will the door have? Working in pure Java, I would set up a set and get method for each of the properties. For example, for the x property, I'd have a setX() and a getX() method. In addition, in pure Java, I'd make the properties private, meaning that users couldn't call the properties directly (e.g., d1.x), but would be forced to use the respective set and get methods for each property (e.g., d1.getX()). Although this is the classic OOP approach, it is not really the Processing approach. So I won't make the properties private, nor will I create set and get methods for all the properties, which will actually save me a ton of work. The only methods I will add to the Door class are setknob() and drawDoor(). The setknob() method will control the side of the door on which the doorknob is drawn, and the drawDoor() method will draw the door. Here's the Door class:

```
class Door{
  // instance properties
  int x;
  int y;
  int w;
  int h;

  // for knob
  int knobLoc = 1;

  // constants - class properties
  final static int RT = 0;
  final static int LFT = 1;

  // constructor
  Door(int w, int h){
    this.w = w;
    this.h = h;
  }

  // draw the door
  void drawDoor(int x, int y) {
    rect(x, y, w, h);
    int knobsize = w/10;
    if (knobLoc == 0){
      //right side
      ellipse(x+w-knobsize, y+h/2, knobsize, knobsize);
    }
    else {
      //left side
      ellipse(x+knobsize, y+h/2, knobsize, knobsize);
    }
  }
```

```
    // set knob position
    void setKnob(int knobLoc){
      this. knobLoc = knobLoc;
    }
  }
```

At the top of the class, I declared some variables. The first five variables are properties that each Door object will have access to; these are called, in OOP terminology, instance variables or object properties. Each Door object will have its own unique set of these variables, so an object can assign its own values to the variables without affecting any other Door objects. For the doorknob, I needed some variables to specify which side of the door the knob should be on. I created two variables, LFT and RT, which I declared as constants. I used the Java keyword final to specify that the value of the two constants can't be changed—which is precisely what a constant is. Secondly, I used the Java keyword static to specify that these constants are static variables, also sometimes referred to as class variables.

Static variables, unlike instance variables, are not unique to each object created from the class. Instead they each hold a single value, accessible from any class, using the syntax *class name.property*. For example, if I wanted to use the Door class's LFT property, I would simply write Door.LFT. I don't need to create an object to do this, and I can do it from any class. So theoretically, if you ever needed the value 1, you could use Door.LFT. However, I don't recommend doing that unless using the constant name makes sense in the context of the program, as in setKnob(Door.LFT);. You'll see more how these constants are used shortly, when an actual door is created.

The next code block is the constructor, in which I included two parameters for the door's width and height (which is perhaps the measurements you'd think about if you were to actually purchase a door). OOP is modeled after the physical world, so it's not a bad idea when designing classes to think about how you would accomplish the same task in the real world. Within the constructor block are two possibly odd looking assignments (although they're discussed in Chapter 8):

```
this.w = w;
this.h = h;
```

Since the instance variables declared at the top of the class and the parameters specified between the parentheses of the constructor have the same identifiers (names), I need a way to identify which is which. I could have simply given the parameters different names from the instance variables. However, it is pretty common practice to use the same names for instance variables and parameters in the constructor, and I tend to like doing it.

The parameters represent the values that will be passed into the constructor when an object is created. These values will need to be seen throughout the class. However, parameters only have local scope (meaning they can only be "seen" within the method or function they're declared in). This is a problem if another method in the class wants to be able to access these values. By assigning the value of the parameters to the instance variables declared at the top of the class—giving them global scope—the values can now be seen throughout the class. Since the parameters have this special relationship to their partner instance variables, I find it convenient to give them the same name.

Outside of the constructor, w and h always refer to the values of the instance variables by those names. But in the constructor, w and h refer to the values of the parameters. That's because local scope takes precedence over global scope. By using the special keyword this preceding the instance variable names (connected with a dot), this.w and this.h are now seen as the instance variables, not the parameters. The w and the h, when not preceded with this, are still the parameters.

Let's skip the drawDoor() method for a moment and look at the setKnob() method. This method contains another assignment, similar to the ones in the constructor. In the drawDoor() method, which I'll discuss next, you'll need to know which side to draw the doorknob on. By assigning a value to the knobLoc instance variable in the setKnob() method, the drawDoor() method will now be able to assess the side on which to attach the knob. Notice also that at the top of the class I gave the knobLoc variable an initial value when I declared it. I did this just in case a position wasn't eventually specified using the setKnob() method.

Finally, in the drawDoor() method, I simply draw the door, using Processing's rect() function. Using a conditional statement and the knobLoc property, the method also figures out which side to draw the knob on, which it then does using Processing's ellipse() function. The next sketch uses the Door class to actually draw a door (see Figure 9-31):

```
// Drawing a Door
// object (reference) variable
Door door1;

void setup(){
  size(200, 350);
  background(200);
  smooth();

  // create new Door object
  door1 = new Door(100, 250);
  door1.setKnob(Door.LFT);
  door1.drawDoor(50, 50);
}

class Door{
  //door properties
  int x;
  int y;
  int w;
  int h;

  // for knob
  int knobLoc = 1;
  //constants
  final static int RT = 0;
  final static int LFT = 1;
```

```
    // constructor
    Door(int w, int h){
      this.w = w;
      this.h = h;
    }

    // draw the door
    void drawDoor(int x, int y) {
      rect(x, y, w, h);
      int knobsize = w/10;
      if (knobLoc == 0){
        //right side
        ellipse(x+w-knobsize, y+h/2, knobsize, knobsize);
      }
      else {
        //left side
        ellipse(x+knobsize, y+h/2, knobsize, knobsize);
      }
    }

    // set knob position
    void setKnob(int knobLoc){
      this. knobLoc = knobLoc;
    }
  }
```

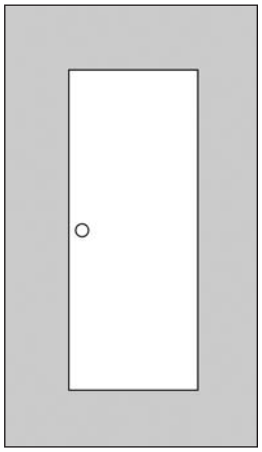


Figure 9-31. Drawing a Door sketch

The three lines at the top of the sketch create the door.

```
door1 = new Door(100, 250);
door1.setKnob(Door.LFT);
door1.drawDoor(50, 50);
```

First, I need to create the Door object; in OOP this is called **instantiation**. Then I optionally set its knob position. Notice how I used the static variable LFT, preceded by the class name and a dot. Finally, the door is drawn. Look carefully at the syntax—the door object door1 is attached directly to its method with a dot. If you eventually wanted to change the value of one of the properties, you'd target it the same way. For example, to change the x property to 194, you'd write: door1.x = 194. Of course, this command would have to come after the Door object has been instantiated: door1 = new Door(100, 250);. Before you move on to the Window class, I strongly recommend you review the Door class a few times, as well as my description. I also suggest trying to make some more doors on your own. This little sketch covers a lot of important issues that will be built upon.

Window class

A window shares certain characteristics with a door. Both have an x and y position, as well as width and height. If this example were more detailed, both objects might also have color, material, price, and inventory number characteristics. There is a way of minimizing this redundancy between classes in OOP, called inheritance (covered extensively in Chapter 8), in which classes can inherit properties and methods from other classes. However, inheritance adds another level of abstraction and complexity, so for this example I'll redundantly create the same properties in the different classes. Later on in the book, when you're an advanced object-oriented programmer, you'll apply inheritance. Here's a sketch that draws three windows using the new Window class (see Figure 9-32):

```
// Drawing Some Windows
void setup(){
  size(450, 250);
  background(200);
  smooth();

  Window window1 = new Window(100, 150);
  window1.drawWindow(50, 50);
  Window window2 = new Window(100, 150, true, Window.DOUBLE);
  window2.drawWindow(175, 50);
  Window window3 = new Window(100, 150, true, Window.QUAD);
  window3.drawWindow(300, 50);
}

class Window{
  //window properties
  int x;
  int y;
  int w;
  int h;

  // customized features
  boolean hasSash = false;
```

```
// single, double, quad pane
int style = 0;
//constants
final static int SINGLE = 0;
final static int DOUBLE = 1;
final static int QUAD = 2;


// constructor 1
Window(int w, int h){
  this.w = w;
  this.h = h;
}
// constructor 2
Window(int w, int h, int style){
  this.w = w;
  this.h = h;
  this.style = style;
}
 // constructor 3
Window(int w, int h, boolean hasSash, int style){
  this.w = w;
  this.h = h;
  this.hasSash = hasSash;
  this.style = style;
}

// draw the window
void drawWindow(int x, int y) {
  //local variables
  int margin = 0;
  int winHt = 0;
  int winWdth = 0;

  if (hasSash){
    margin = w/15;
  }

  switch(style){
  case 0:
    //outer window (sash)
    rect(x, y, w, h);
    //inner window
    rect(x+margin, y+margin, w-margin*2, h-margin*2);
    break;
  case 1:
    winHt = (h-margin*3)/2;
    //outer window (sash)
    rect(x, y, w, h);
```

```
        //inner window (top)
        rect(x+margin, y+margin, w-margin*2, winHt);
        //inner windows (bottom)
        rect(x+margin, y+winHt+margin*2, w-margin*2, winHt);
        break;
      case 2:
        winWdth = (w-margin*3)/2;
        winHt = (h-margin*3)/2;
        //outer window (sash)
        rect(x, y, w, h);
        //inner window (top-left)
        rect(x+margin, y+margin, winWdth, winHt);
        //inner window (top-right)
        rect(x+winWdth+margin*2,  y+margin, winWdth, winHt);
        //inner windows (bottom-left)
        rect(x+margin, y+winHt+margin*2, winWdth, winHt);
         //inner windows (bottom-right)
        rect(x+winWdth+margin*2,  y+winHt+margin*2, winWdth, winHt);
        break;
      }
    }

    // set window style (number of panes)
    void setStyle(int style){
      this.style = style;
    }
  }
```

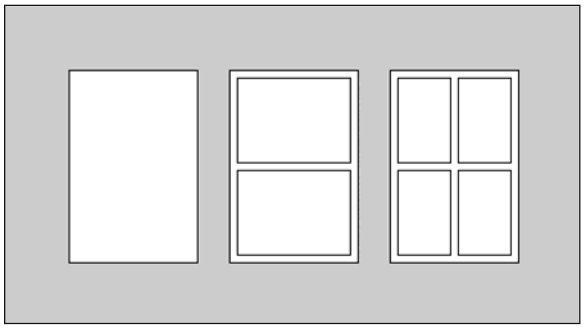


Figure 9-32. Drawing Some Windows sketch

Although the `Window` class is longer than the `Door` class, it works very similarly. In the sketch, I created three windows, each with different features. The major difference between the `Door` and `Window` classes is the use of multiple constructors in the `Window` class. Remember that in Processing, you're allowed to name multiple methods and functions with the same identifier, as long as their signatures are different. In OOP speak, this is called method overloading. It is quite common practice to do this with constructors, as it provides an easy way for users to instantiate objects with various initial configurations and values, as I did in this last example. One other minor point is that I created a couple local variables in the `drawWindow()` method:

```
//local variables
int margin = 0;
int winHt = 0;
int winWdth = 0;
```

Like parameters, these variables are only scoped within the method that they are defined within. In general, it's a good rule to only give variables the scope they require. These variables are only used within the `drawWindow()` method, so it made sense to make them local. Take a few minutes to look at the switch statement in the `drawWindow()` method, as well as the procedure I used to draw the different windows. Again, I relied on Processing's `rect()` function to take care of all the drawing.

Roof class

The last piece of our house I need is the roof. Next is a sketch that draws a roof using the new `Roof` class (see Figure 9-33):

```
// Drawing Some Roofs
void setup(){
  size(400, 150);
  background(200);
  smooth();
  Roof roof1 = new Roof();
  roof1.drawRoof(25, 100, 100, 150);
  Roof roof2 = new Roof(Roof.GAMBREL);
  roof2.drawRoof(150, 100, 100, 150);
  Roof roof3 = new Roof(Roof.DOME);
  roof3.drawRoof(275, 100, 100, 100);
}

class Roof{
  //roof properties
  int x;
  int y;
  int w;
  int h;

  // roof style
  int style = 0;
  //constants
  final static int CATHEDRAL = 0;
  final static int GAMBREL = 1;
  final static int DOME = 2;
  // default constructor
  Roof(){
  }

   // constructor 2
   Roof(int style){
    this.style = style;
  }

  // draw the roof
  void drawRoof(int x, int y, int w, int h) {
    switch(style){
    case 0:
      beginShape();
      vertex(x, y);
      vertex(x+w/2, y-h/3);
      vertex(x+w, y);
      endShape(CLOSE);
      break;
    case 1:
     beginShape();
      vertex(x, y);
      vertex(x+w/7, y-h/5);
      vertex(x+w/2, y-h/2.75);
      vertex(x+(w-w/7), y-h/5);
      vertex(x+w, y);
      endShape(CLOSE);
      break;
    case 2:
      ellipseMode(CORNER);
      arc(x, y-h/2, w, h, PI, TWO_PI);
      line(x, y, x+w, y);
      break;
    }

  }

  // set roof style
  void setStyle(int style){
    this.style = style;
  }
}
```

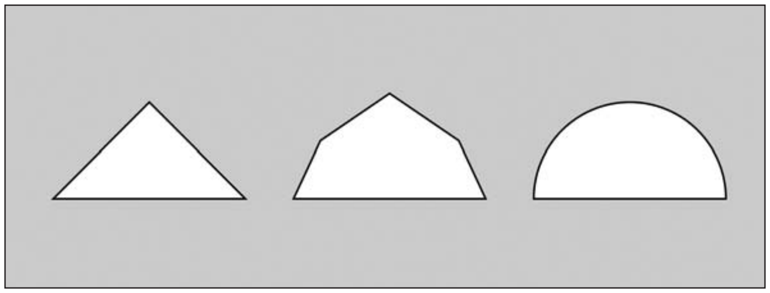


Figure 9-33. Drawing Some Roofs sketch

There is nothing new in this sketch, and I now have all the House component classes built. The next thing to do is put them all together. Again, it is worthwhile considering this problem in the context of the real world. If you were really building a house, you would be at the stage at which you have identified your door, window, and roof suppliers, and have established an agreed upon set of properties and methods. In a sense, you have developed a contractual agreement with your suppliers. This is precisely how OOP works.

House class

The public properties and methods of a class are referred to as the **public interface** to the class. If you know the public interface to a class, you don't even need to know what happens within the class. The interface, in a sense, enforces the contract. Classes should be as modular and self-contained as possible. Thus, each of the classes takes care of calculating and drawing itself. The House class is no exception. The House class, besides drawing some type of external structure, will need to orchestrate how the Door, Window, and Roof component classes are used in its own creation. To make this happen, you need to somehow include these classes within the House class. You also want to be able to customize these component classes. There is a classic OOP way to solve this problem, which is called **composition**. I'll actually include variables of the respective component types within the House class. In other words, I'll include reference variables of the Door, Window, and Roof data types within the House class definition. When I instantiate the House class, I'll actually pass in Door, Window, and Roof object references as arguments, which, you'll remember, will need to be received by parameters of the same data type within a House constructor.

Here's the House class. Please note that the following code is not a finished sketch yet, but only the House class. Shortly, I'll put all the classes (Window, Door, Roof, and House) together into a sketch to draw a little neighborhood.

```
class House{
  //house properties
  int x;
  int y;
  int w;
  int h;
```

```
//component reference variables
Door door;
Window window;
Roof roof;

//optional autosize variable
boolean AutoSizeComponents = false;

//door placement
int doorLoc = 0;
//constants
final static int MIDDLE_DOOR = 0;
final static int LEFT_DOOR = 1;
final static int RIGHT_DOOR = 2;

//constructor
House(int w, int h, Door door, Window window, Roof roof, ➥
       int doorLoc) {
  this.w = w;
  this.h = h;
  this.door = door;
  this.window = window;
  this.roof = roof;
  this.doorLoc = doorLoc;
}

void drawHouse(int x, int y, boolean AutoSizeComponents) {
  this.x = x;
  this.y =y;
  this.AutoSizeComponents = AutoSizeComponents;

  //automatically sizes doors and windows
  if(AutoSizeComponents){
    //autosize door
    door.h = h/4;
    door.w = door.h/2;

    //autosize windows
    window.h = h/3;
    window.w = window.h/2;

  }
  // draw bldg block
  rect(x, y, w, h);

  // draw door
  switch(doorLoc){
  case 0:
        door.drawDoor(x+w/2-door.w/2, y+h-door.h);
        break;
      case 1:
        door.drawDoor(x+w/8, y+h-door.h);
        break;
      case 2:
        door.drawDoor(x+w-w/8-door.w,  y+h-door.h);
        break;
      }

      // draw windows
      int windowMargin = (w-window.w*2)/3;
      window.drawWindow(x+windowMargin, y+h/6);
      window.drawWindow(x+windowMargin*2+window.w, y+h/6);

      // draw roof
      roof.drawRoof(x, y, w, h);
    }

    // catch drawHouse method without boolean argument
    void drawHouse(int x, int y){
      // recall with required 3rd argument
      drawHouse(x, y, false);
    }
  }
```

The House class is implemented similarly to the Door, Window, and Roof classes. It contains a constructor; it has x, y, w, and h instance variables; and it has some constants and a drawHouse() method. Again, the redundant elements and structures in these classes could be handled more efficiently using inheritance, which I covered in Chapter 8 and will also use in later chapters. What's markedly different between these classes, however, is the compositional relationship between the House class and its component classes. The House class contains reference variables of the Door, Window, and Roof classes. When I instantiate the House class, I'll include Door, Window, and Roof objects as arguments that will be sent to the House constructor.

Within the House constructor, the Door, Window, and Roof instance properties declared at the top of the House class will each be assigned the values (object references) of their matching parameters. Remember, the parameters declared within the constructor head (between the parentheses) are local to the constructor, so it's necessary to do these assignments in the constructor to create global visibility for these passed-in values so that they may be seen throughout the class. Because the House class will now contain object references to these component classes (Door, Window, and Roof), the properties and methods of each of these classes can be utilized from within the House class with the normal syntax *object.property* (e.g., roof1.x) or *object.method* (e.g., window1.drawWindow()). Without the object references being passed into the House constructor, the House class would have no way of communicating with its Door, Window, or Roof components.

The abstraction of OOP takes some time to get your head around. So if your head is spinning, you're definitely not alone. It's helpful to remember that each class in OOP should be a self-contained modular unit. Someone using the class shouldn't need to look inside the class to see how it's implemented. Instead, they just need to know the class's pubic interface (its public properties and methods, including any constructors). When you use a class, the class should, in a sense, take care of itself. So when you create a window and then ask the window to draw itself, you don't need to be privy to how it actually implements the drawing; you just need to make sure you ask properly—using the correct method and any required arguments.

When the `House` class creates itself through its `drawHouse()` method, it will need to compose its components—position and size its door, window, and roof. For the `House` class to be able to do this, it needs the object references to these components, and when requested to do so, each of the components will be responsible for drawing itself. Since the position of each of the house's components is ultimately dependent on the size and position of the house, it makes sense for the house to make these calls after its own position and size have been determined.

One last point is that it's also possible (albeit not very desirable) to totally encapsulate the creative (customization) process from the user. In the upcoming neighborhood sketch, I'll provide the user the capability to customize the individual `House` components (`Door`, `Window`, and `Roof`) as each component object is instantiated. The user will then simply pass these customized component objects as arguments when instantiating a `House` object. Instead of giving the user this customization ability—to select a roof style, for example—you could just let the `House` class set all the style properties for the individual components. The `House` class could even take care of instantiating the component `Door`, `Window`, and `Roof` objects from within its own class, hiding this process entirely from the user and avoiding the need to pass in any object reference arguments. This greatly simplifies the process of using the `House` class for the user, but also removes any customization options. Thus, balance needs to be considered carefully when you design a class—on one hand, you should think about ease of use, but on the other, you should think about providing users with enough creative freedom to make the class ultimately worth using.

I'm going to end this chapter with a completed Neighborhood sketch (see Figure 9-34), including the `Door`, `Window`, `Roof`, and `House` classes. However, the following example code only includes the final sketch's `setup()` function, since the four finished classes are printed in their entirety earlier in the chapter. To run the sketch, of course, you'll need to add the `Door`, `Window`, `Roof`, and `House` classes to the sketch. (Note that when adding the class code, you should not include the `setup()` functions from the earlier individual class sketch examples—just the code beginning with `class`, down to the final closing curly brace at the bottom of the entire class.) You can either type/paste the class code directly into Processing, below the `setup()` function at the top of the Neighborhood sketch, or better yet, download the completed Neighborhood sketch code from the book's Download section on the friends of ED site (`www.friendsofed.com/`).

The Neighborhood sketch's `setup()` function primarily includes instantiation statements. Each of the three houses in the Neighborhood sketch has its own `Door`, `Window`, and `Roof` objects passed into its constructor when instantiated. Notice how I instantiate the component classes first and then pass the reference variables for each of the component objects as arguments when I instantiate the House.

In laying out the neighborhood, I used the previous house's x and w positions to ensure that the houses lined up properly, but also didn't overlap.

In spite of the sketch's length (when the four classes are added), the object-oriented structure adds organization and (reusable) modularity to the code, as compared to a long series of function calls. In addition, the OOP structure provides a logical framework in which to add new features to the entire neighborhood, the House class, or any of the component classes. Really take your time going through (and playing) with this example. I also suggest trying to reimplement some of the internal drawing methods. For example, the cathedral roof could be drawn with Processing's triangle() function, and the dome roof could be created with the bezier() or curve() function. Changing the internal implementation of a class, if done properly, should have no effect on how you use the class. In addition (if you're feeling ambitious), you could even try to apply a tessellated texture as a skin to some of the houses.

```
/*
 Neighborhood
 Ira Greenberg, January 1, 2006
 Happy New Year!
 */
void setup(){
  size(600, 400);
  background(190);
  smooth();
  //ground plane
  int groundHeight = 10;
  fill(0);
  rect(0, height-groundHeight, width, groundHeight);
  fill(255);

  // Houses
  Door door1 = new Door(40, 80);
  Window window1 = new Window(100, 125, false, Window.DOUBLE);
  Roof roof1 = new Roof(Roof.DOME);
  House house1 = new House(150, 150, door1, window1, roof1, ➥
        House.MIDDLE_DOOR);
  house1.drawHouse(50, height-groundHeight-house1.h, true);

  Door door2 = new Door(40, 80);
  Window window2 = new Window(100, 125, true, Window.QUAD);
  Roof roof2 = new Roof(Roof.GAMBREL);
  House house2 = new House(200, 120, door2, window2, roof2, ➥
         House.LEFT_DOOR);
  house2.drawHouse(50+house1.w, height-groundHeight-house2.h, true);

  Door door3 = new Door(40, 80);
  door3.setKnob(Door.RT);
  Window window3 = new Window(125, 125, true, Window.SINGLE);
  Roof roof3 = new Roof(Roof.CATHEDRAL);
  House house3 = new House(150, 200, door3, window3, roof3, ➥
         House.RIGHT_DOOR);
  house3.drawHouse(house2.x+house2.w, height-groundHeight-house3.h, ➥
         true);
}
// add Door class
// add Window class
// add Roof class
// add House class
```

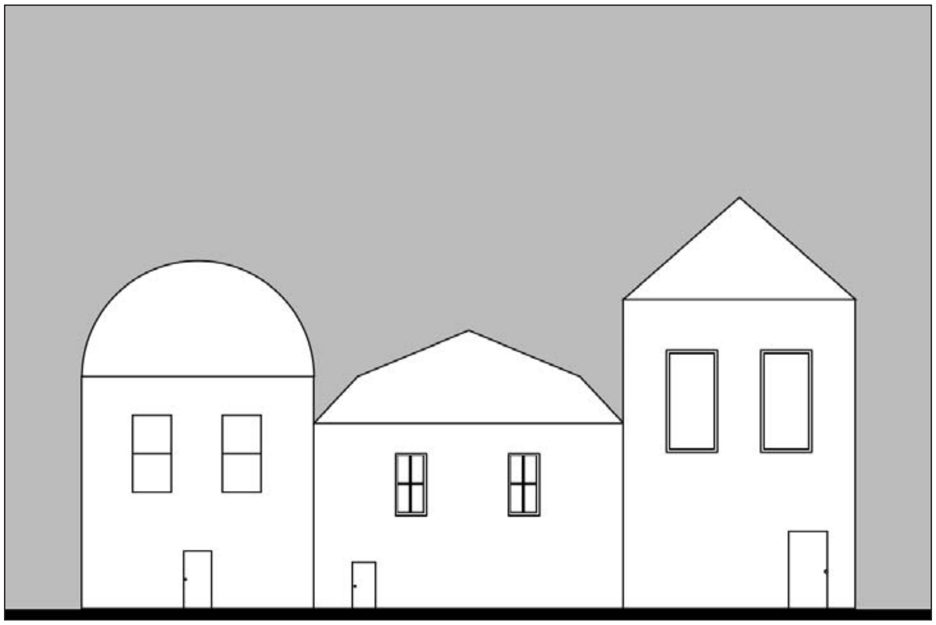


Figure 9-34. Neighborhood sketch

Summary

Beginning with Processing's simple shape commands, `rect()`, `ellipse()`, `triangle()`, `arc()`, and `quad()`, this chapter introduced some of the underlying computer graphics principals involved in generating and working with shapes, including the definition of an origin point, the use of bounding boxes, and the concept of state changes. Moving beyond simple shape functions, you learned about a more general and powerful approach for shape creation, using Processing's `vertex()` function and companion `beginShape()` and `endShape()` record functions. Utilizing a powerful programming technique called recursion, you learned about an alternative approach to iterating, including the risk of generating infinite loops with recursion. You looked under the hood at Processing's matrix functions and how drawing in Processing happens internally in a graphics context, which has its own local coordinate system. Finally, you looked at applying an object-oriented approach to generating shapes, developed a simple neighborhood example, and used an advanced OOP concept called composition. Next chapter, I'll introduce color/imaging techniques using Processing, and also expand the discussion of OOP to incorporate some more advanced concepts.