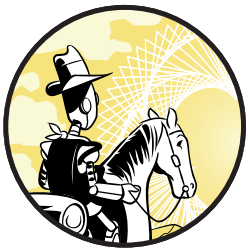
5

TRANSFORMING SHAPES WITH GEOMETRY

In the teahouse one day Nasrudin announced he was selling his house. When the other patrons asked him to describe it, he brought out a brick. "It's just a collection of these."
—Idries Shah



In geometry class, everything you learn about involves dimensions in space using shapes. You typically start by examining one-dimensional lines and two-dimensional circles, squares, or triangles, then move on to three-dimensional objects like spheres and cubes. These days, creating geometric shapes is easy with technology and free software, though manipulating and changing the shapes you create can be more of a challenge.

In this chapter, you'll learn how to manipulate and transform geometric shapes using the Processing graphics package. You'll start with basic shapes like circles and triangles, which will allow you to work with complicated shapes like fractals and cellular automata in later chapters. You will also learn how to break down some complicated-looking designs into simple components.

DRAWING A CIRCLE

Let's start with a simple one-dimensional circle. Open a new sketch in Processing and save it as *geometry.pyde*. Then enter the code in Listing 5-1 to create a circle on the screen.

geometry.pyde

```
def setup():
    size(600,600)

def draw():
    ellipse(200,100,20,20)
```

Listing 5-1: Drawing a circle

Before we draw the shape, we first define the size of our sketchbook, known as the *coordinate plane*. In this example, we use the `size()` function to say that our grid will be 600 pixels wide and 600 pixels tall.

With our coordinate plane set up, we then use the drawing function `ellipse()` to create our circle on this plane. The first two parameters, `200` and `100`, show where the center of the circle is located. Here, `200` is the x-coordinate and the second number, `100`, is the y-coordinate of this circle's center, which places it at `(200,100)` on the plane.

The last two parameters determine the width and height of the shape in pixels. In the example, the shape is 20 pixels wide and 20 pixels tall. Because the two parameters are the same, it means that the points on the circumference are equidistant from the center, forming a perfectly round circle.

Click the **Run** button (it looks like a play symbol), and a new window with a small circle should open, like in Figure 5-1.

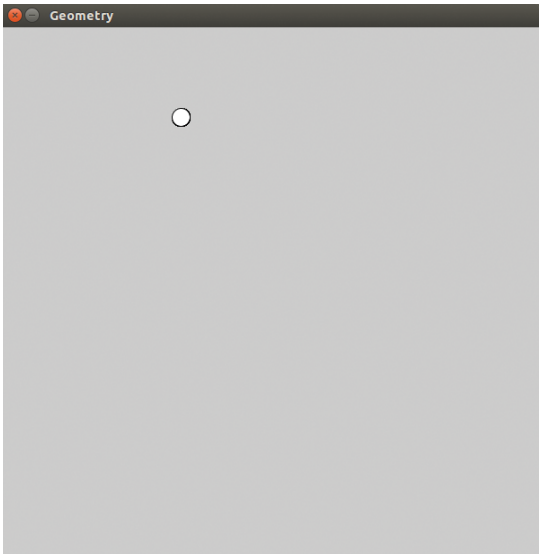


Figure 5-1: The output of Listing 5-1 showing a small circle

Processing has a number of functions you can use to draw shapes. Check out the full list at *https://processing.org/reference/* to explore other shape functions.

Now that you know how to draw a circle in Processing, you're almost ready to use these simple shapes to create dynamic, interactive graphics. In order to do that, you'll first need to learn about location and transformations. Let's start with location.

SPECIFYING LOCATION USING COORDINATES

In Listing 5-1, we used the first two parameters of the `ellipse()` function to specify our circle's location on the grid. Likewise, each shape we create using Processing needs a location that we specify with the coordinate system, where each point on the graph is represented by two numbers: (x,y). In traditional math graphs, the origin (where x=0 and y=0) is at the center of the graph, as shown in Figure 5-2.

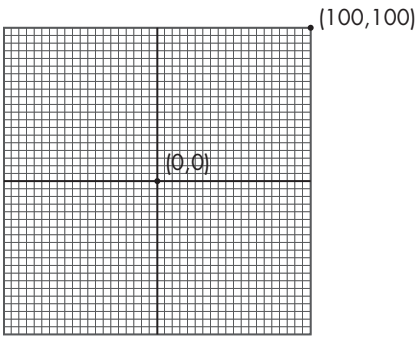


Figure 5-2: A traditional coordinate system with the origin in the center

In computer graphics, however, the coordinate system is a little different. Its origin is in the top-left corner of the screen so that x and y increase as you move right and down, respectively, as you can see in Figure 5-3.

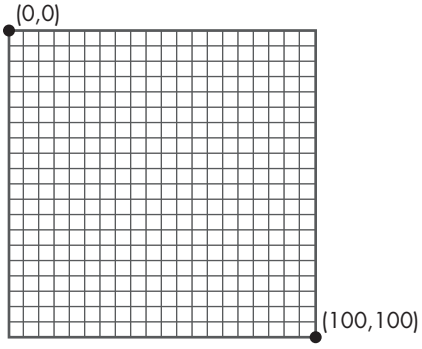


Figure 5-3: The coordinate system for computer graphics, with the origin in the top-left corner

Each coordinate on this plane represents a pixel on the screen. As you can see, this means you don't have to deal with negative coordinates. We'll use functions to transform and translate increasingly complex shapes around this coordinate system.

Drawing a single circle was fairly easy, but drawing multiple shapes can get complicated pretty quickly. For example, imagine drawing a design like the one shown in Figure 5-4.

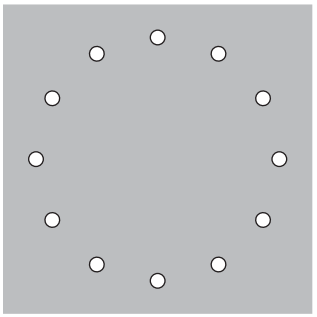


Figure 5-4: A circle made of circles

Specifying the size and location of each individual circle and spacing them out perfectly evenly would involve entering many lines of similar code. Fortunately, you don't really need to know the absolute x- and y-coordinates of each circle to do this. With Processing, you can easily place objects wherever you want on the grid.

Let's see how you can do this using a simple example to start.

TRANSFORMATION FUNCTIONS

You might remember doing transformations with pencil and paper in geometry class, which you performed on a collection of points to laboriously move a shape around. It's much more fun when you let a computer do the transforming. In fact, there wouldn't be any computer graphics worth looking at without transformations! Geometric transformations like translation and rotation let you change where and how your objects appear without altering the objects themselves. For example, you can use transformations to move a triangle to a different location or spin it around without changing its shape. Processing has a number of built-in transformation functions that make it easy to translate and rotate objects.

TRANSLATING OBJECTS WITH TRANSLATE()

To *translate* means to move a shape on a grid so that all points of the shape move in the same direction and the same distance. In other words, translations let you move a shape on a grid without changing the shape itself and without tilting it in the slightest.

Translating an object in math class involves manually changing the coordinates of all the points in the object. But in Processing, you translate an object by moving the *grid* itself, while the object's coordinates stay the same! For an example of this, let's put a rectangle on the screen. Revise your existing code in *geometry.pyde* with the code in Listing 5-2.

geometry.pyde

```
def setup():
    size(600,600)

def draw():
    rect(20,40,50,30)
```

Listing 5-2: Drawing a rectangle to translate

Here, we use the `rect()` function to draw the rectangle. The first two parameters are the x- and y-coordinates telling Processing where the top-left corner of the rectangle should be. The third and fourth parameters indicate its width and its height, respectively.

Run this code, and you should see the rectangle shown in Figure 5-5.

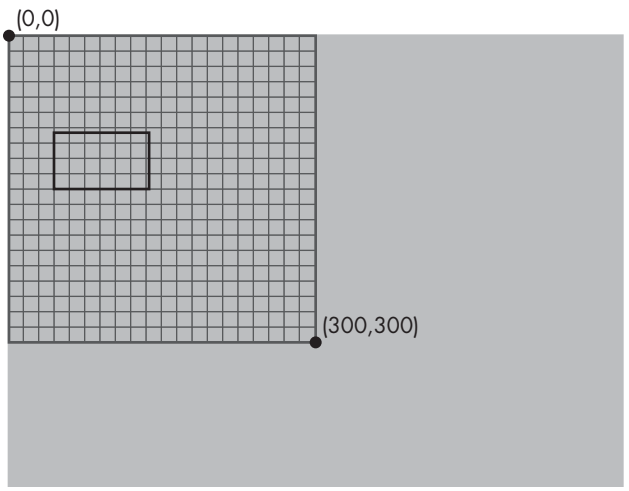


Figure 5-5: The default coordinate setup with the origin at the top left

NOTE *In these examples, I'm showing the grid for reference, but you won't see it on your screen.*

Now let's tell Processing to translate the rectangle using the code in Listing 5-3. Notice that we don't change the coordinates of the rectangle.

geometry.pyde

```
def setup():
    size(600,600)

def draw():
    translate(50,80);
    rect(50,100,100,60)
```

Listing 5-3: Translating the rectangle

Here, we use `translate()` to move the rectangle. We provide two parameters: the first tells Processing how far to move the grid in the horizontal (x) direction, and the second parameter is for how far to move the grid vertically, in the y-direction. So `translate(50,80)` should move the entire grid 50 pixels to the right and 80 pixels down, as shown in Figure 5-6.

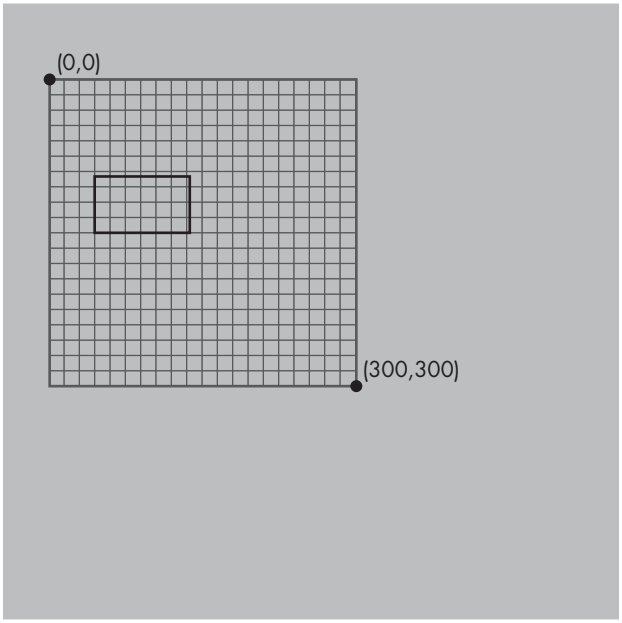


Figure 5-6: Translating a rectangle by moving the grid 50 pixels to the right and 80 pixels down

Very often it's useful (and easier!) to have the origin (0,0) in the center of the canvas. You can use `translate()` to easily move the origin to the center of your grid. You can also use it to change the width and height of your canvas if you want it bigger or smaller. Let's explore Processing's built-in `width` and `height` variables, which let you update the size of your canvas without having to change the numbers manually. To see this in action, update the existing code in Listing 5-3 so it looks like Listing 5-4.

geometry.pyde

```
def setup():
    size(600,600)

def draw():
    translate(width/2, height/2)
    rect(50,100,100,60)
```

Listing 5-4: Using the `width` and `height` variables to translate the rectangle

Whatever numbers you put in the `size` declaration in the `setup()` function will become the "width" and "height" of the canvas. In this case, because I used `size(600,600)`, they're both 600 pixels. When we change the `translate()` line to `translate(width/2, height/2)` using variables instead

of specific numbers, we tell Processing to move the location (0,0) to the center of the display window, no matter what the size is. This means that if you change the size of the window, Processing will automatically update `width` and `height`, and you won't have to go through all your code and change the numbers manually.

Run the updated code, and you should see something like Figure 5-7.

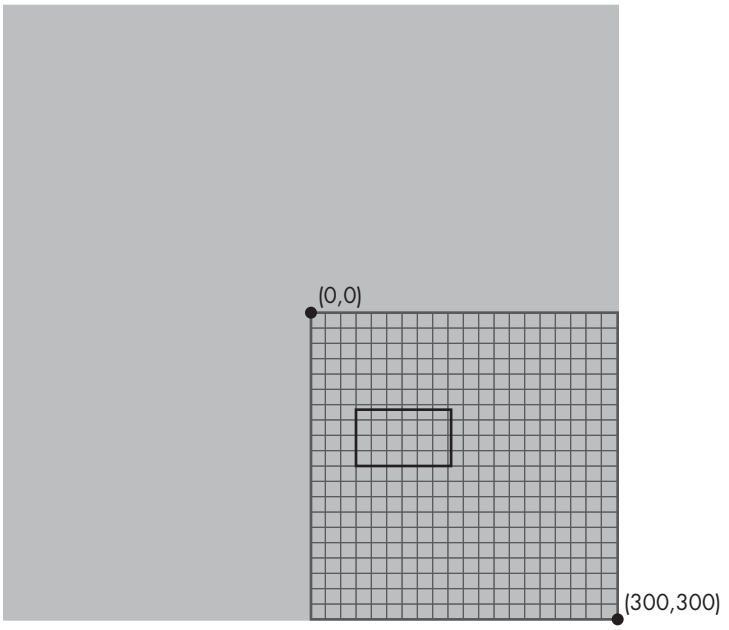


Figure 5-7: The grid is translated to the center of the screen.

Notice that the origin is still labeled as (0,0), which shows that we haven't actually moved the origin point but rather the entire coordinate plane itself so that the origin point falls in the middle of our canvas.

ROTATING OBJECTS WITH ROTATE()

In geometry, *rotation* is a kind of transformation that turns an object around a center point, as if it's turning on an axis. The `rotate()` function in Processing rotates the grid around the origin (0,0). It takes a single number as its argument to specify the angle at which you want to rotate the grid around the point (0,0). The units for the rotation angle are radians, which you learn about in precalculus class. Instead of using 360 degrees to do a full rotation, we can use 2π (around 6.28) radians. If you think in degrees, like I do, you can use the `radians()` function to easily convert your degrees to radians so you don't have to do the math yourself.

To see how the `rotate()` function works, enter the code shown in Figure 5-8 into your existing sketch by replacing the `translate()` code inside the `draw()` function with each of these examples, and then run them. Figure 5-8 shows the results.

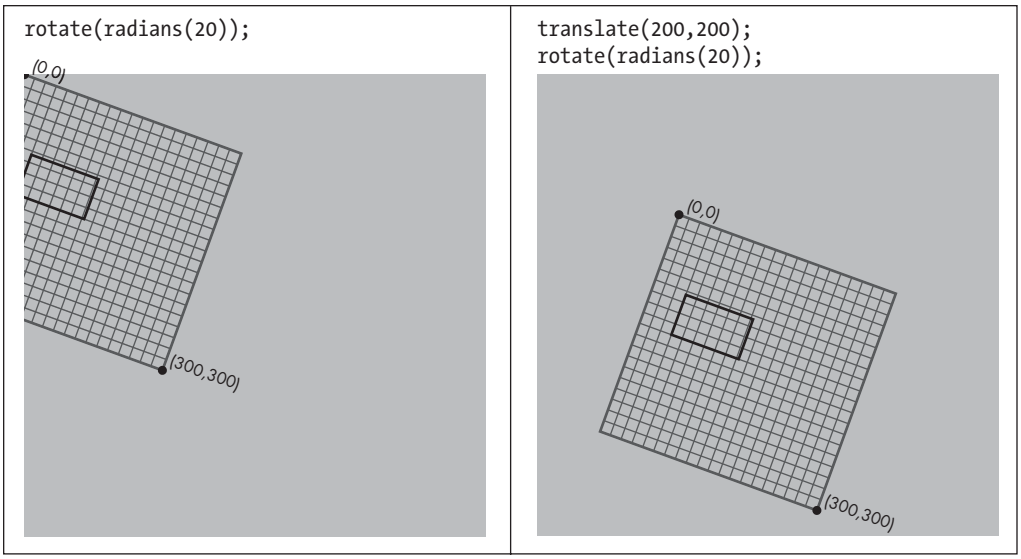


Figure 5-8: The grid always rotates around (0,0)

On the left side of Figure 5-8, the grid is rotated 20 degrees around (0,0), which is at the top-left corner of the screen. In the example on the right, the origin is first translated 200 units to the right and 200 units down and *then* the grid is rotated.

The `rotate()` function makes it easy to draw a circle of objects like the one in Figure 5-4 using the following steps:

1. Translate to where you want the center of the circle to be.
2. Rotate the grid and put the objects along the circumference of the circle.

Now that you know how to use transformation functions to manipulate the location of different objects on your canvas, let's actually re-create Figure 5-4 in Processing.

DRAWING A CIRCLE OF CIRCLES

To create the circles arranged in a circle in Figure 5-4, we'll use a `for i in range()` loop to repeat the circles and make sure the circles are evenly spaced. First, let's think about how many degrees should be between the circles to make a full circle, remembering that a circle is 360 degrees.

Enter the code shown in Listing 5-5 to create this design.

geometry.pyde

```
def setup():
    size(600,600)

def draw():
    translate(width/2,height/2)
    for i in range(12):
```

```
        ellipse(200,0,50,50)
        rotate(radians(360/12))
```

Listing 5-5: Drawing a circular design

Note that the `translate(width/2,height/2)` function inside the `draw()` function translates the grid to the center of the screen. Then, we start a `for` loop to create an ellipse at a point on the grid, starting at (200,0), as you can see from the first two parameters of the function. Then we set the size of each small circle by setting both the `width` and `height` of the ellipse to `50`. Finally, we rotate the grid by 360/12, or 30 degrees, before creating the next ellipse. Note that we use `radians()` to convert 30 degrees into radians inside the `rotate()` function. This means that each circle will be 30 degrees away from the next one.

When you run this, you should see what's shown in Figure 5-9.

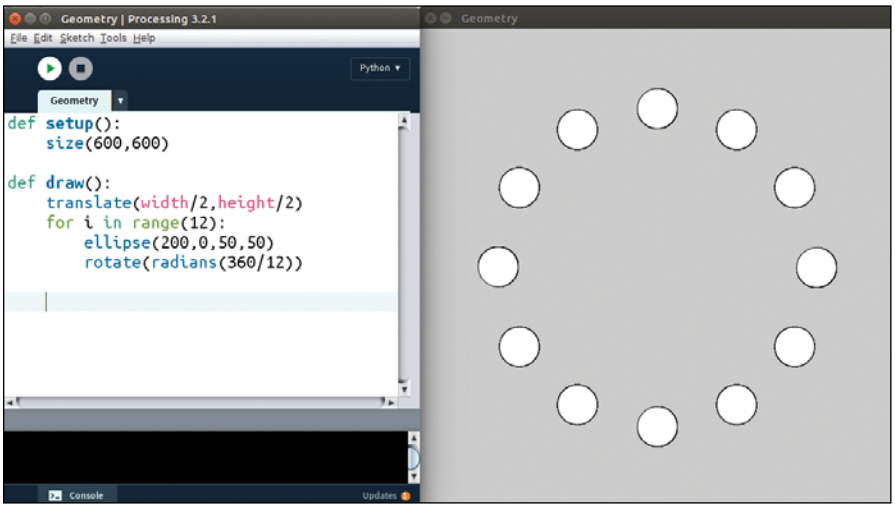


Figure 5-9: Using transformation to create a circular design

We have successfully arranged a bunch of circles into a circular shape!

DRAWING A CIRCLE OF SQUARES

Modify the program you wrote in Listing 5-5 and change the circles into squares. To do this, just change `ellipse` in the existing code to `rect` to make the circles into squares, as shown here:

geometry.pyde

```
def setup():
    size(600,600)

def draw():
    translate(width/2,height/2)
    for i in range(12):
        rect(200,0,50,50)
        rotate(radians(360/12))
```

That was easy!

ANIMATING OBJECTS

Processing is great for animating your objects to create dynamic graphics. For your first animation, you'll use the `rotate()` function. Normally, `rotate` happens instantly, so you don't get to see the action take place—only the result of the rotation. But this time, we'll use a time variable `t`, which allows us to see the rotation unfold in real time!

CREATING THE T VARIABLE

Let's use our circle of squares to write an animated program. To start, create the `t` variable and initialize it to 0 by adding **`t = 0`** before the `setup()` function. Then insert the code in Listing 5-6 before the `for` loop.

geometry.pyde

```
t = 0

def setup():
    size(600,600)

def draw():
    translate(width/2,height/2)
    rotate(radians(t))
    for i in range(12):
        rect(200,0,50,50)
        rotate(radians(360/12))
```

Listing 5-6: Adding the t variable

However, if you try to run this code, you'll get the following error message:

```
UnboundLocalError: local variable 't' referenced before assignment
```

This is because Python doesn't know whether we're creating a new local variable named `t` *inside* the function that doesn't have anything to do with the global variable `t` *outside* the function, or just calling the global variable. Because we want to use the global variable, add **`global t`** at the beginning of the `draw()` function so the program knows which one we're referring to.

Enter the complete code shown here:

geometry.pyde

```
t = 0

def setup():
    size(600,600)

def draw():
    global t
    #set background white
    background(255)
```

```
    translate(width/2,height/2)
    rotate(radians(t))
    for i in range(12):
        rect(200,0,50,50)
        rotate(radians(360/12))
    t += 1
```

This code starts `t` at 0, rotates the grid that number of degrees, increments `t` by 1, and then repeats. Run it, and you should see the squares start to rotate in a circular pattern, as in Figure 5-10.

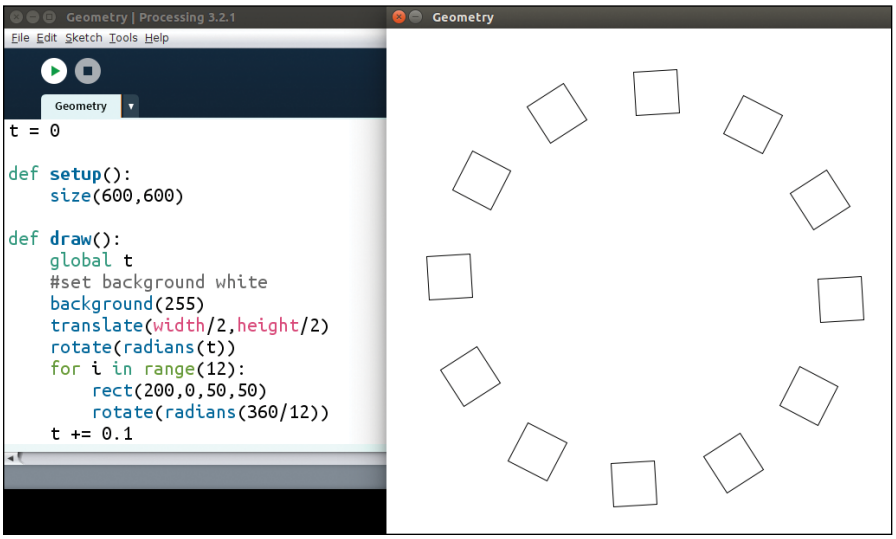


Figure 5-10: Making squares rotate in a circle

Pretty cool! Now let's try rotating each individual square.

ROTATING THE INDIVIDUAL SQUARES

Because rotating is done around (0,0) in Processing, inside the loop we first have to translate to where each square needs to be, then rotate, and finally draw the square. Change the loop in your code to look like Listing 5-7.

geometry.pyde

```
    for i in range(12):
        translate(200,0)
        rotate(radians(t))
        rect(0,0,50,50)
        rotate(radians(360/12))
```

Listing 5-7: Rotating each square

This translates the grid to where we want to place the square, rotates the grid so the square rotates, and then draws the square using the `rect()` function.

SAVING ORIENTATION WITH PUSHMATRIX() AND POPMATRIX()

When you run Listing 5-7, you should see that it creates some strange behavior. The squares don't rotate around the center, but keep moving around the screen instead, as shown in Figure 5-11.

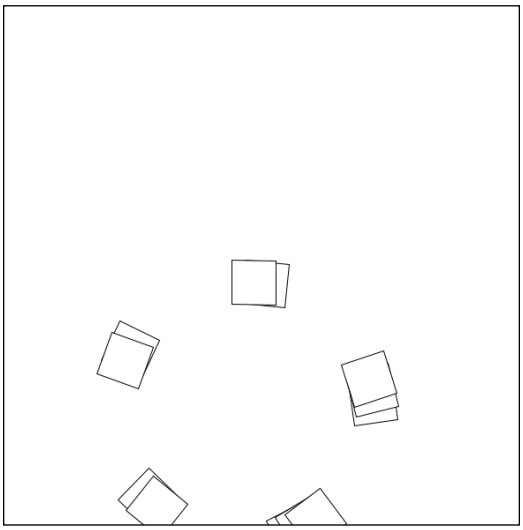


Figure 5-11: The squares are flying all over!

This is due to changing the center and changing the orientation of the grid so much. After translating to the location of the square, we need to rotate back to the center of the circle before translating to the next square. We could use another `translate()` function to undo the first one, but we might have to undo more transformations, and that could get confusing. Fortunately, there's an easier way.

Processing has two built-in functions that save the orientation of the grid at a certain point and then return to that orientation: `pushMatrix()` and `popMatrix()`. In this case, we want to save the orientation when we're in the center of the screen. To do this, revise the loop to look like Listing 5-8.

geometry.pyde

```
for i in range(12):
    pushMatrix() #save this orientation
    translate(200,0)
    rotate(radians(t))
    rect(0,0,50,50)
    popMatrix() #return to the saved orientation
    rotate(radians(360/12))
```

Listing 5-8: Using `pushMatrix()` and `popMatrix()`

The `pushMatrix()` function saves the position of the coordinate system at the center of the circle of squares. Then we translate to the location of the

square, rotate the grid so the square will spin, and then draw the square. Then we use `popMatrix()` to return instantly to the center of the circle of squares and repeat for all 12 squares.

ROTATING AROUND THE CENTER

The preceding code should work perfectly, but the rotation may look strange; that's because Processing by default locates a rectangle at its top-left corner and rotates it about its top-left corner. This makes the squares look like they're veering off the path of the larger circle. If you want your squares to rotate around their centers, add this line to your `setup()` function:

```
rectMode(CENTER)
```

Note that the all-uppercase `CENTER` in `rectMode()` matters. (You can also experiment with other types of `rectMode()`, like `CORNER`, `CORNERS`, and `RADIUS`.) Adding `rectMode(CENTER)` should make each square rotate around its center. If you want the squares to spin more quickly, change the `rotate()` line to increase the time in `t`, like so:

```
rotate(radians(5*t))
```

Here, `5` is the frequency of the rotation. This means the program multiplies the value of `t` by 5 and rotates by the product. Therefore, the square will rotate five times as far as before. Change it to see what happens! Comment out the `rotate()` line outside the loop (by adding a hashtag at the beginning) to make the squares rotate in place, as shown in Listing 5-9.

```
translate(width/2,height/2)
#rotate(radians(t))
for i in range(12):
    rect(200,0,50,50)
```

Listing 5-9: Commenting out a line instead of deleting it

Being able to use transformations like `translate()` and `rotate()` to create dynamic graphics is a very powerful technique, but it can produce unexpected results if you do things in the wrong order!

CREATING AN INTERACTIVE RAINBOW GRID

Now that you've learned how to create designs using loops and to rotate them in different ways, we'll create something pretty awesome: a grid of squares whose rainbow colors follow your mouse cursor! The first step is to make a grid.

DRAWING A GRID OF OBJECTS

Many tasks involved in math and in creating games like Minesweeper require a grid. Grids are necessary for some of the models and all the cellular automata we'll create in later chapters, so it's worth learning how to write code for making a grid that we can reuse. To begin with, we'll make a 12 × 12 grid of squares, evenly sized and spaced. Making a grid this size may seem like a time-consuming task, but in fact it's easy to do using a loop.

Open a new Processing sketch and save as *colorGrid.pyde*. Too bad we used the name "grid" previously. We'll make a 20 × 20 grid of squares on a white background. The squares need to be `rect`, and we need to use a `for` loop within a `for` loop to make sure they are all the same size and spaced equally. Also, we need our 25 × 25 pixel squares to be drawn every 30 pixels, using this line:

```
rect(30*x,30*y,25,25)
```

As the `x` and `y` variables go up by 1, squares are drawn at 50-pixel intervals in two dimensions. We'll start off, as usual, by writing our `setup()` and `draw()` functions, as in the previous sketch (see Listing 5-10).

colorGrid.pyde

```
def setup():
    size(600,600)

def draw():
    #set background white
    background(255)
```

Listing 5-10: The standard structure for a Processing sketch: `setup()` and `draw()`

This sets the size of the window at 600 by 600 pixels, and sets the background color to white. Next we'll create a nested loop, where two variables will both go from 0 to 19, for a total of 20 numbers, since we want 20 rows of 20 squares. Listing 5-11 shows the code that creates the grid.

```
def setup():
    size(600,600)

def draw():
    #set background white
    background(255)
    for x in range(20):
        for y in range(20):
            rect(30*x,30*y,25,25)
```

Listing 5-11: The code for a grid

This should create a 20 × 20 grid of squares, as you can see in Figure 5-12. Time to add some colors to our grid.

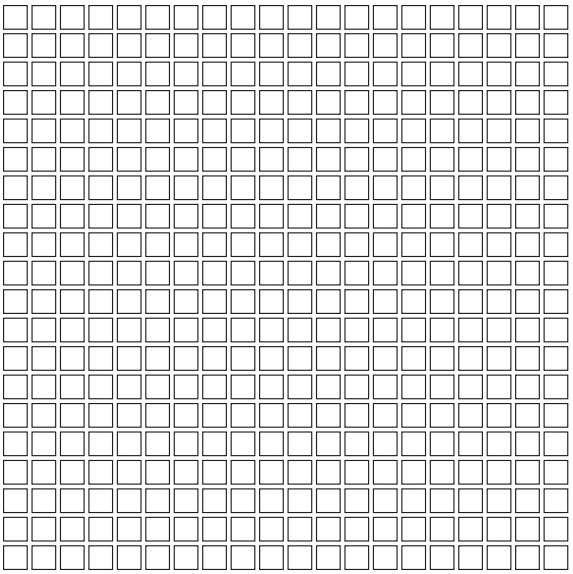


Figure 5-12: A 20 × 20 grid!

ADDING THE RAINBOW COLOR TO OBJECTS

Processing's `colorMode()` function helps us add some cool color to our sketches! It's used to switch between the RGB and HSB modes. Recall that RGB uses three numbers indicating amounts of red, green, and blue. In HSB, the three numbers represent levels of hue, saturation, and brightness. The only one we need to change here is the first number, which represents the hue. The other two numbers can be the maximum value, 255. Figure 5-13 shows how to make rainbow colors by changing only the first value, the hue. Here, the 10 squares have the hue values shown in the figure, with 255 for saturation and 255 for brightness.

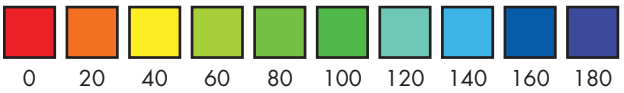


Figure 5-13: The colors of the rainbow using HSB mode and changing the hue value

Since we're locating the rectangles at `(30*x,30*y)` in Listing 5-11, we'll create a variable that measures the distance of the mouse from that location:

```
d = dist(30*x,30*y,mouseX,mouseY)
```

Processing has a `dist()` function that finds the distance between two points, and in this case it's the distance between the square and the mouse. It saves the distance to a variable called `d`, and we'll link the hue to that variable. Listing 5-12 shows the changes to the code.

colorGrid.pyde

```
def setup():
    size(600,600)
    rectMode(CENTER)
❶   colorMode(HSB)

def draw():
    #set background black
❷   background(0)
    translate(20,20)
    for x in range(30):
        for y in range(30):
❸           d = dist(30*x,30*y,mouseX,mouseY)
            fill(0.5*d,255,255)
            rect(30*x,30*y,25,25)
```

Listing 5-12: Using the `dist()` function

We insert the `colorMode()` function and pass `HSB` to it ❶. In the `draw()` function, we set the background to black first ❷. Then we calculate the distance from the mouse to the square, which is at `(30*x,30*y)` ❸. In the next line, we set the fill color using HSB numbers. The hue value is half the distance, while the saturation and brightness numbers are both 255, the maximum.

The hue is the only thing we change: we update the hue according to the distance the rectangle is from the mouse. We do this with the `dist()` function, which takes four arguments: the x- and y-coordinates of two points. It returns the distance between the points.

Run this code and you should see a very colorful design that changes colors according to the mouse's location, as shown in Figure 5-14.

Now that you've learned how to add colors to your objects, let's explore how we can create more complicated shapes.

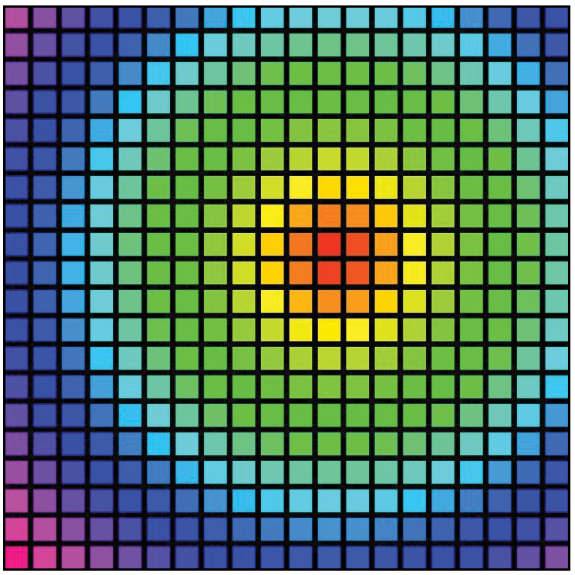


Figure 5-14: Adding colors to your grid

DRAWING COMPLEX PATTERNS USING TRIANGLES

In this section, we create more complicated, Spirograph-style patterns using triangles. For example, take a look at the sketch made up of rotating triangles in Figure 5-15, created by the University of Oslo's Roger Antonsen.

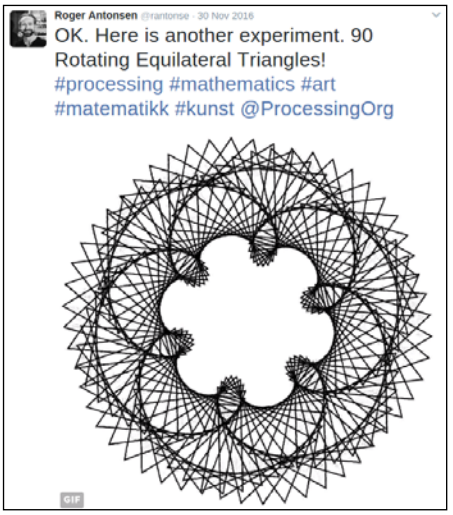


Figure 5-15: Sketch of 90 rotating equilateral triangles by Roger Antonsen. See it in motion at https://rantonse.no/en/art/2016-11-30.

The original design moves, but in this book you'll have to imagine all the triangles rotating. This sketch blew me away! Although this design looks very complicated, it's not that difficult to make. Remember Nasrudin's joke about the brick from the beginning of the chapter? Like Nasrudin's house, this complicated design is just a collection of identical shapes. But what shape? Antonsen gave us a helpful clue to creating this design when he named the sketch "90 Rotating Equilateral Triangles." It tells us that all we have to do is figure out how to draw an equilateral triangle, rotate it, and then repeat that for a total of 90 triangles. Let's first discuss how to draw an equilateral triangle using the `triangle()` function. To start, open a new Processing sketch and name it *triangles.pyde*. The code in Listing 5-13 shows one way to create a rotating triangle but not an equilateral one.

triangles.pyde

```
def setup():
    size(600,600)
    rectMode(CENTER)

t = 0

def draw():
    global t
    translate(width/2,height/2)
    rotate(radians(t))
    triangle(0,0,100,100,200,-200)
    t += 0.5
```

Listing 5-13: Drawing a rotating triangle, but not the right kind

Listing 5-13 uses the lessons you learned previously: it creates a `t` variable (for time), translates to where we want the triangle to be, rotates the grid, and then draws the triangle. Finally, it increments `t`. When you run this code, you should see something like Figure 5-16.

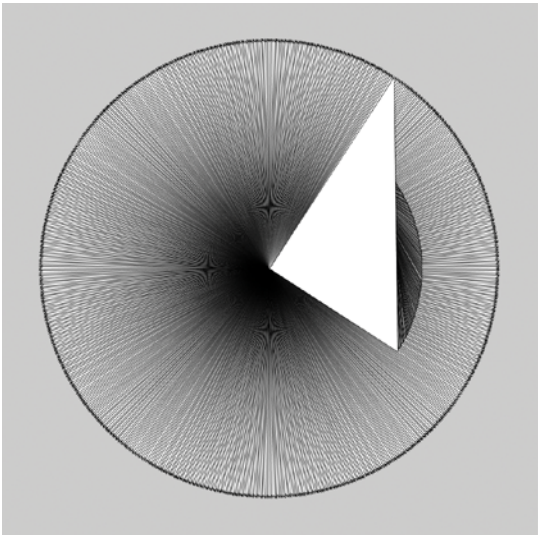


Figure 5-16: Rotating a triangle around one of its vertices

As you can see in Figure 5-16, the triangle rotates around one of its *vertices*, or points, and thus creates a circle with the outer point. You'll also notice that this is a right triangle (a triangle containing a 90-degree angle), not an equilateral one.

To re-create Antonsen's sketch, we need to draw an equilateral triangle, which is a triangle with equal sides. We also need to find the center of the equilateral triangle to be able to rotate it about its center. To do this, we need to find the location of the three vertices of the triangle. Let's discuss how to draw an equilateral triangle by locating it at its center and specifying the location of its vertices.

A 30-60-90 TRIANGLE

To find the location of the three vertices of our equilateral triangle, we'll review a particular type of triangle you've likely seen in geometry class: the *30-60-90 triangle*, which is a special *right triangle*. First, we need an equilateral triangle, as shown in Figure 5-17.

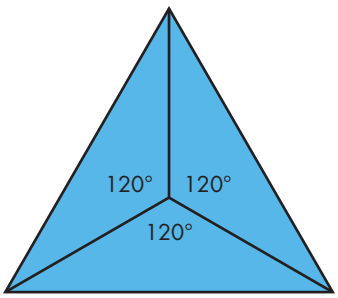


Figure 5-17: An equilateral triangle divided into three equal parts

This equilateral triangle is made up of three equal parts. The point in the middle is the center of the triangle, with the three dissecting lines meeting at 120 degree angles. To draw a triangle in Processing, we give the `triangle()` function six numbers: the x- and y-coordinates of all three vertices. To find the coordinates of the vertices of the equilateral triangle shown in Figure 5-17, let's cut the bottom triangle in half, as shown in Figure 5-18.

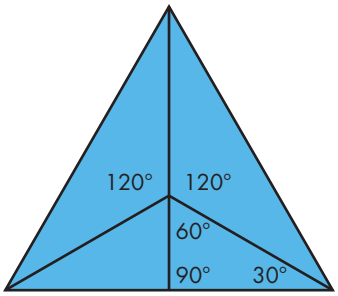


Figure 5-18: Dividing up the equilateral triangle into special triangles

Dividing the bottom triangle in half creates two right triangles, which are classic 30-60-90 triangles. As you might recall, the ratio between the sides of a 30-60-90 triangle can be expressed as shown in Figure 5-19.

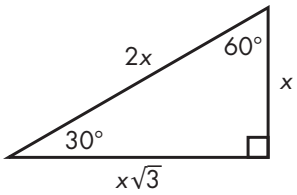


Figure 5-19: The ratios of the sides in a 30-60-90 triangle, from the legend on an SAT test

If we call the length of the smaller leg *x*, the hypotenuse is twice that length, or 2*x*, and the longer leg is *x* times the square root of 3, or approximately 1.732*x*. We're going to be creating our function using the length from the center of the big equilateral triangle in Figure 5-18 to one of its vertices, which happens to be the hypotenuse of the 30-60-90 triangle. That means we can measure everything in terms of that length. For example, if we call the hypotenuse `length`, then the smaller leg will be half that length, or `length/2`. Finally, the longer leg will be `length` divided by 2 times the square root of 3. Figure 5-20 zooms in on the 30-60-90 triangle.

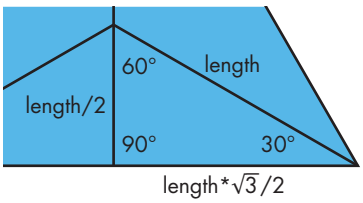


Figure 5-20: The 30-60-90 triangle up close and personal

As you can see, a 30-60-90 triangle has internal angles of 30, 60, and 90 degrees, and the lengths of the sides are in known proportions. You may

be familiar with this from the Pythagorean Theorem, which will come up again shortly.

We'll call the distance from the center of the larger equilateral triangle to its vertex the "length," which is also the *hypotenuse* of the 30-60-90 triangle. You'll need to know the ratios between the lengths of the sides of this special triangle in order to find the three vertices of the equilateral triangle with respect to the center—you can draw it (the big equilateral triangle we're trying to draw) by specifying where each point of the triangle should be.

The shorter leg of the right triangle opposite the 30 degree angle is always half the hypotenuse, and the longer leg is the measure of the shorter leg times the square root of 3. So if we use the center point for drawing the big equilateral triangle, the coordinates of the three vertices would be as shown in Figure 5-21.

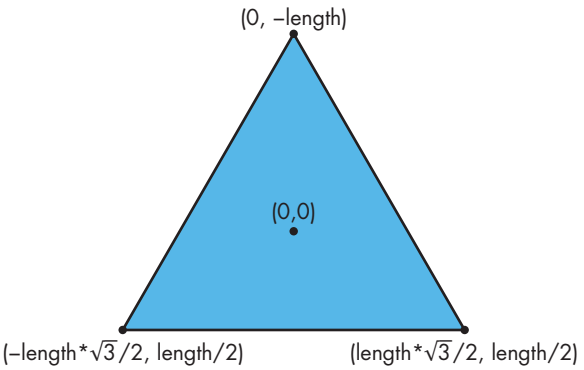


Figure 5-21: The vertices of the equilateral triangle

As you can see, because this triangle is made up of 30-60-90 triangles on all sides, we can use the special relation between them to figure out how far each vertex of the equilateral triangle should be from the origin.

DRAWING AN EQUILATERAL TRIANGLE

Now we can use the vertices we derived from the 30-60-90 triangle to create an equilateral triangle, using the code in Listing 5-14.

triangles.pyde

```
def setup():
    size(600,600)
    rectMode(CENTER)

t = 0

def draw():
    global t
    translate(width/2,height/2)
    rotate(radians(t))
    tri(200) #draw the equilateral triangle
    t += 0.5

❶ def tri(length):
```

```
    '''Draws an equilateral triangle
    around center of triangle'''
❷   triangle(0,-length,
             -length*sqrt(3)/2, length/2,
             length*sqrt(3)/2, length/2)
```

Listing 5-14: The complete code for making a rotating equilateral triangle

First, we write the `tri()` function to take the variable `length` ❶, which is the hypotenuse of the special 30-60-90 triangles we cut the equilateral triangle into. We then make a triangle using the three vertices we found. Inside the call to the `triangle()` function ❷, we specify the location of each of the three vertices of the triangle: `(0,-length)`, `(-length*sqrt(3)/2, length/2)`, and `(length*sqrt(3)/2, length/2)`.

When you run the code, you should see something like Figure 5-22.

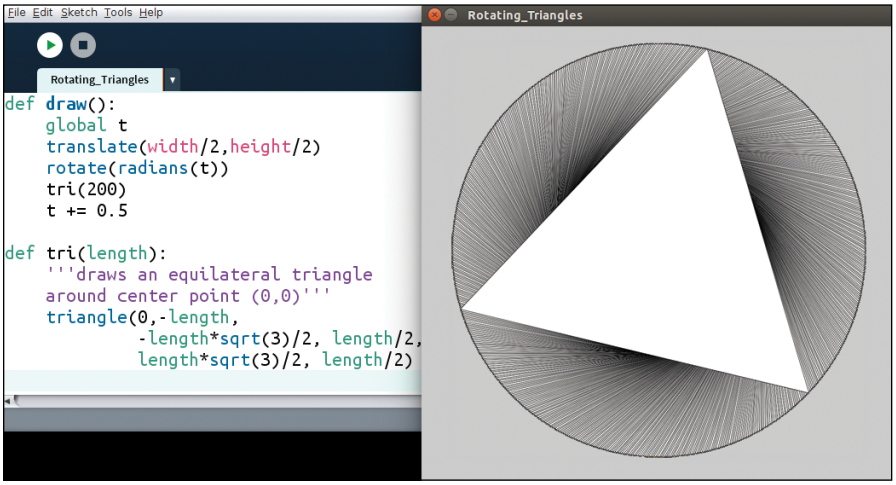


Figure 5-22: A rotating equilateral triangle!

Now we can cover up all the triangles created during rotation by adding this line to the beginning of the `draw()` function:

```
background(255) #white
```

This should erase all the rotating triangles except for one, so we just have a single equilateral triangle on the screen. All we have to do is put 90 of them in a circle, just like we did earlier in this chapter, using the `rotate()` function.

EXERCISE 5-1: SPIN CYCLE

Create a circle of equilateral triangles in a Processing sketch and rotate them using the `rotate()` function.

DRAWING MULTIPLE ROTATING TRIANGLES

Now that you've learned how to rotate a single equilateral triangle, we need to figure out how to arrange multiple equilateral triangles into a circle. This is similar to what you created while rotating squares, but this time we'll use our `tri()` function. Enter the code in Listing 5-15 in place of the `def draw()` section in Processing and then run it.

triangles.pyde

```
def setup():
    size(600,600)
    rectMode(CENTER)

t = 0

def draw():
    global t
    background(255)#white
    translate(width/2,height/2)
 ❶ for i in range(90):
        #space the triangles evenly
        #around the circle
        rotate(radians(360/90))
     ❷ pushMatrix() #save this orientation
        #go to circumference of circle
        translate(200,0)
        #spin each triangle
        rotate(radians(t))
        #draw the triangle
        tri(100)
        #return to saved orientation
     ❸ popMatrix()
    t += 0.5

def tri(length):

 ❹ noFill() #makes the triangle transparent

    triangle(0,-length,
             -length*sqrt(3)/2, length/2,
             length*sqrt(3)/2, length/2)
```

Listing 5-15: Creating 90 rotating triangles

At ❶, we use the `for` loop to arrange 90 triangles around the circle, making sure they're evenly spaced by dividing 360 by 90. Then at ❷ we use `pushMatrix()` to save this position before moving the grid around. At the end of the loop at ❸ we use `popMatrix()` to return to the saved position. In the `tri()` function at ❹, we add the `noFill()` line to make the triangles transparent.

Now we have 90 rotating transparent triangles, but they're all rotating in exactly the same way. It's kind of cool, but not as cool as Antonsen's sketch yet. Next, you'll learn how to make each triangle rotate a little differently from the adjacent ones to make the pattern more interesting.

PHASE-SHIFTING THE ROTATION

We can change the pattern in which the triangles rotate with a *phase shift*, which makes each triangle lag a little bit behind its neighbor, giving the sketch a "wave" or "cascade" effect. Each triangle has been assigned a number in the loop, represented by `i`. We need to add `i` to `t` in the `rotate(radians(t))` function, like this:

```
        rotate(radians(t+i))
```

When you run this, you should see something like Figure 5-23.

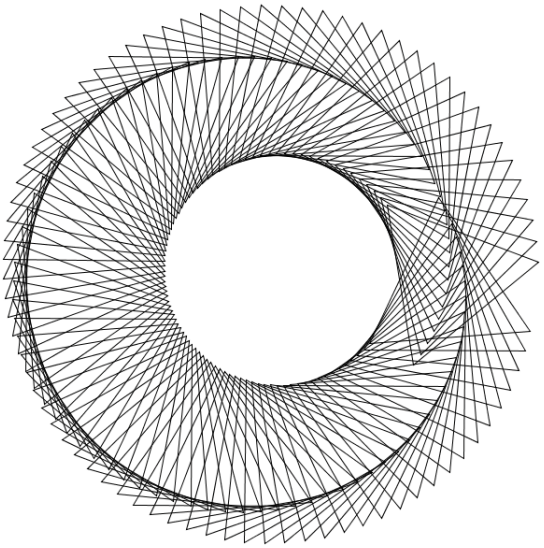


Figure 5-23: Rotating triangles with phase shift

Notice there's a break in the pattern on the right side of the screen. This break in the pattern is caused by the phase shifts not matching up from the beginning triangle to the last triangle. We want a nice, seamless pattern, so we have to make the phase shifts add up to a multiple of 360 degrees to complete the circle. Because there are 90 triangles in the design, we'll divide 360 by 90 and multiply that by `i`:

```
        rotate(radians(t+i*360/90))
```

It's easy enough to calculate 360/90, which is 4, and then use that number to plug into the code, but I'm leaving the expression in because we'll need it in case we want to change the number of triangles later. For now, this should create a nice seamless pattern, as shown in Figure 5-24.

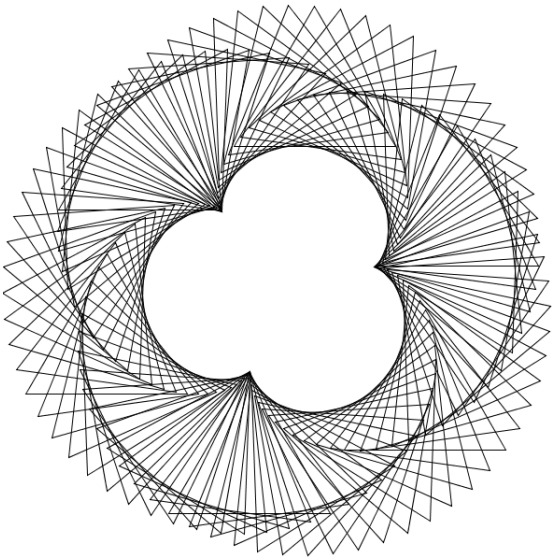


Figure 5-24: Seamlessly rotating triangles with phase shift

By making our phase shifts add up to a multiple of 360, we were able to remove the break in the pattern.

FINALIZING THE DESIGN

To make the design look more like the one in Figure 5-15, we need to change the phase shift a little. Play around with it yourself to see how you can change the look of the sketch!

Here, we're going to change the phase shift by multiplying `i` by 2, which will increase the shift between each triangle and its neighbor. Change the `rotate()` line in your code to the following:

```
    rotate(radians(t+2*i*360/90))
```

After making this change, run the code. As you can see in Figure 5-25, our design now looks very close to the design we were trying to re-create.

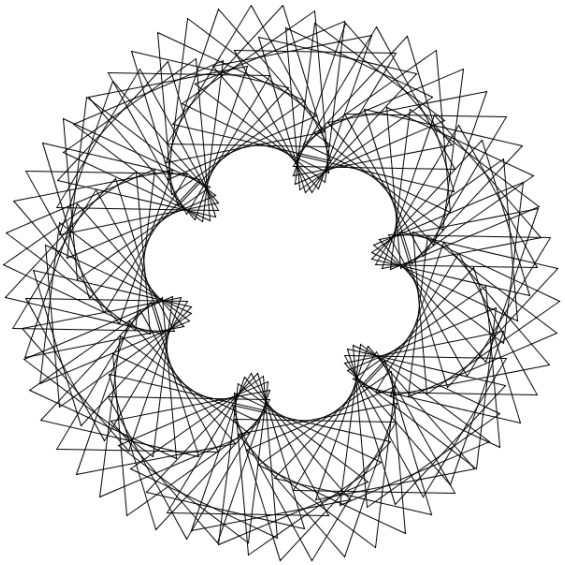
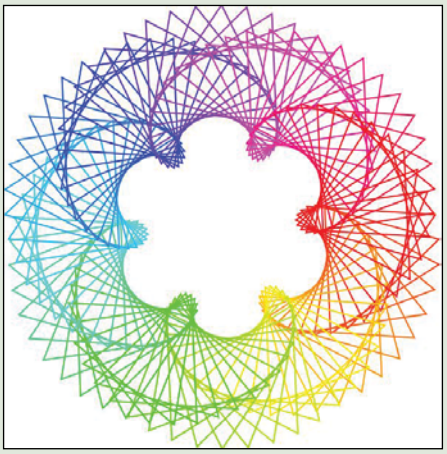


Figure 5-25: Re-creation of Antonsen's "90 Rotating Equilateral Triangles" from Figure 5-15

Now that you've learned how to re-create a complicated design like this, try the next exercise to test your transformation skills!

EXERCISE 5-2: RAINBOW TRIANGLES

Color each triangle of the rotating triangle sketch using `stroke()`. It should look like this.



SUMMARY

In this chapter, you learned how to draw shapes like circles, squares, and triangles and arrange them into different patterns using Processing's built-in transformation functions. You also learned how to make your shapes dynamic by animating your graphics and adding color. Just like how Nasrudin's house was just a collection of bricks, the complicated code examples in this chapter are just a collection of simpler shapes or functions.

In the next chapter, you'll build on what you learned in this chapter and expand your skills to using trigonometric functions like sine and cosine. You'll draw even cooler designs and write new functions to create even more complicated behaviors, like leaving a trail and creating any shape from a bunch of vertices.