6 [ChapterStart]

I've got an oscillating fan at my house. The fan goes back and forth. It looks like the fan is saying "No". So I like to ask it questions that a fan would say "no" to. "Do you keep my hair in place? Do you keep my documents in order? Do you have 3 settings? Liar!" My fan lied to me. - Mitch Hedberg

Trigonometry

Trigonometry is the study of triangles. It starts with right triangles and the three ratios between the sides: sine, cosine and tangent, given in Figure

Figure : The ratios of the sides of a right triangle

Students in Trigonometry class (or “Pre-calculus”) are assigned innumerable problems on finding missing sides or angles in triangles. But this is seldom how trig functions are used in reality. In this chapter you'll see sine and cosine applied to oscillating motion, and you'll make some interesting, dynamic, interactive sketches. You'll apply the tangent function (and its inverse) to a problem of making a field of objects orient themselves to the location of your mouse.

Rotations

First of all, sines and cosines make rotations a cinch. In Figure 6-2, you can imagine r, the hypotenuse of the triangle, and imagine it rotating around (0,0), becoming the radius of a circle.

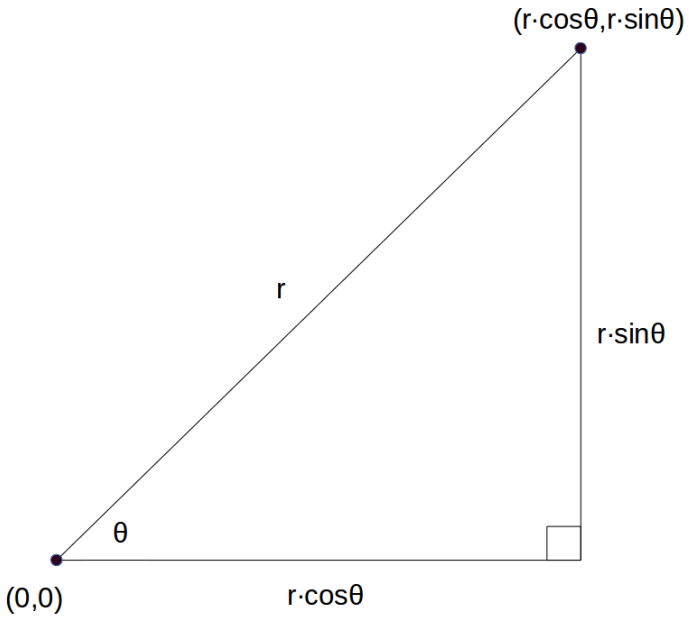


Figure 6-2

Polygons

This makes creating polygons very easy. Remember how much geometry we needed to know to draw an equilateral triangle in the Geometry chapter? With trigonometry functions helping us with rotations, all we have to do is use Figure 6-? to create a “polygon” function. The Processing functions “beginShape” and “endShape” define any shape we want using the “vertex” function to say where the points of the shape go. We can use as many vertices as we want. Here's a simpler way to make an equilateral triangle:

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

def draw():
    translate(width/2,height/2)
    polygon(3,100) #3 sides, vertices 100 units from the center

def polygon(sides,sz):
    '''draws a polygon given the number
    of sides and length from the center'''
    beginShape()
    for i in range(sides):
        vertex(sz*cos(i*TWO_PI/sides),sz*sin(i*TWO_PI/sides))
    endShape()
```

You get what's in Figure 6-?:

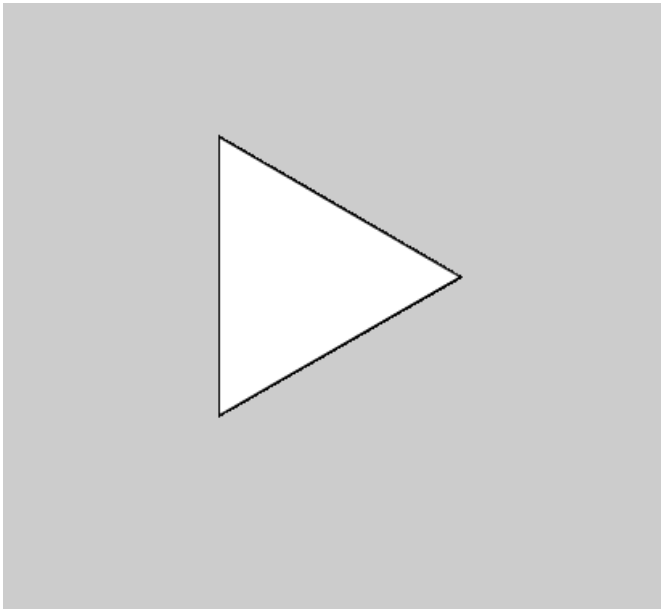


Figure 6.: A much simpler Triangle!

Now making polygons of any number of sides (“n-gons”) a breeze. No square roots necessary!

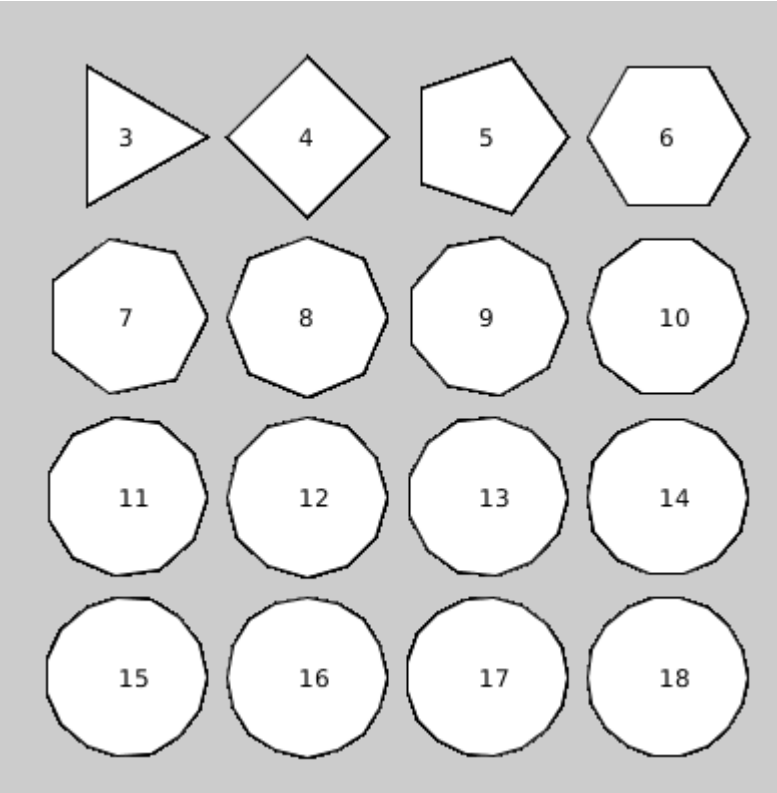


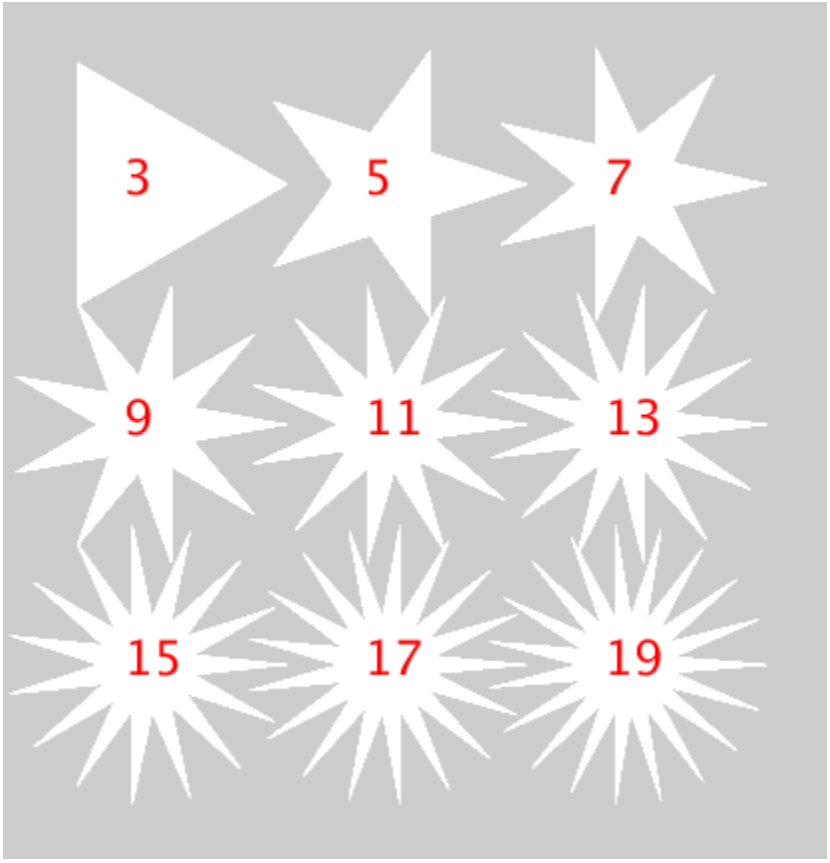
Figure 6. : All the Polygons you want!

Stars

Why stop with Polygons? Copy the “polygon” function and change a few lines so it looks like this:

```
def star(x,y,sides,sz):
    '''draws a star given the number
    of prongs and size at location (x,y)'''
    turns = int(sides/2) #number of turns
    beginShape()
    for i in range(sides):
        vertex(x+sz*cos(i*TWO_PI*turns/sides),y+sz*sin(i*TWO_PI*turns/sides))
    endShape(CLOSE)
```

It'll allow you to make stars of any (odd) number of prongs. Figure 6.? contains a sample:



Trigonometry's True Purpose: Oscillation

Like Mitch Hedberg's fan at the beginning of the chapter, Sines and Cosines are for rotating and oscillating. Sines and cosines make waves when the height of a point on a circle is measured over time. Let's make a circle. Then we'll put a smaller circle on the circumference and as it travels around the circle, its height will draw out a sine wave.

Start a new Processing sketch and create a big circle on the left side of the screen like in Figure 6-1. Try it yourself before looking at the code below:

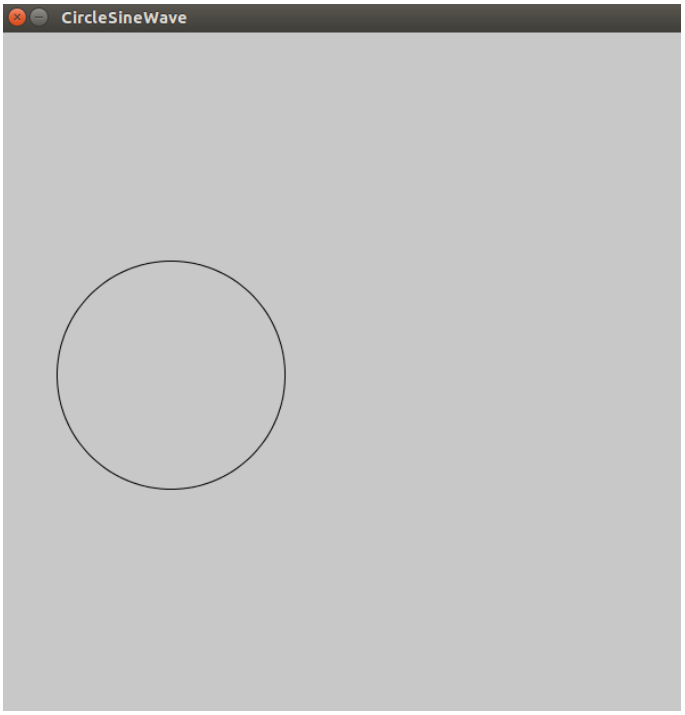


Figure 6-1: The start of the Sine Wave sketch

Here's the code:

```
r1 = 100 #radius of big circle
r2 = 10  #radius of small circle
t = 0 #time variable

def setup():
    size(600,600)

def draw():
    background(200)
    #move to left-center of screen
    translate(width/4,height/2)
    noFill() #don't color in the circle
    stroke(0) #black outline
    ellipse(0,0,2*r1,2*r1)
```

Now how do we make another circle spin around the big circle? Finding out the x- and y-coordinates of every spot on the circle isn't obvious, but it is obvious what the radius of the circle is, and all we have to do is rotate around the center. Using r as the radius of the circle and the Greek letter

theta as the measure of the angle and using trig ratios, we can define a point on the circle as seen in Figure 6.2.

It might look more complicated than (x,y) but we only have to code it once and it'll work perfectly! Here's how to use sine and cosine to define the position of the ellipse that will rotate around the first circle.

```
#circling ellipse:
fill(255,0,0) #red
y = r1*sin(t)
x = r1*cos(t)
ellipse(x,y,r2,r2)
```

At the end of the draw function, make the time variable go up by a little bit:

```
t += 0.05
```

If you try to run this right now, you'll get an error message about “local variable 't' referenced before assignment.” Python functions have local variables but we want the draw function to use the global time variable t. So we have to add this line to the beginning of the draw function:

```
global t
```

Now you'll see a red ellipse traveling along the circumference of the big circle, as in Figure 6-3:

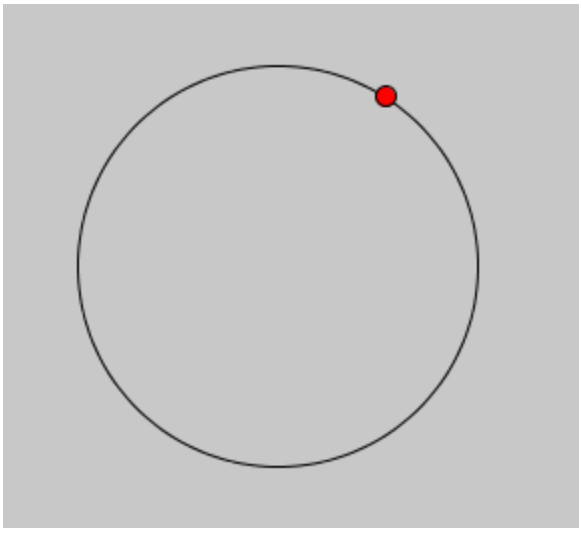


Figure 6-3: The red circle travels along the circumference of the big circle.

Now we'll chose a place over to the right of the screen to start drawing the wave. We'll extend a green line from the red circle to, say, x = 200. Add these lines to your draw function right before “t += 0.05.”

```
    stroke(0,255,0) #green for the line
    line(x,y,200,y)
    fill(0,255,0) #green for the ellipse
    ellipse(200,y,10,10)
```

Run your program and you'll see we've added a green circle that only measures the height of the red circle, as in Figure 6-4:

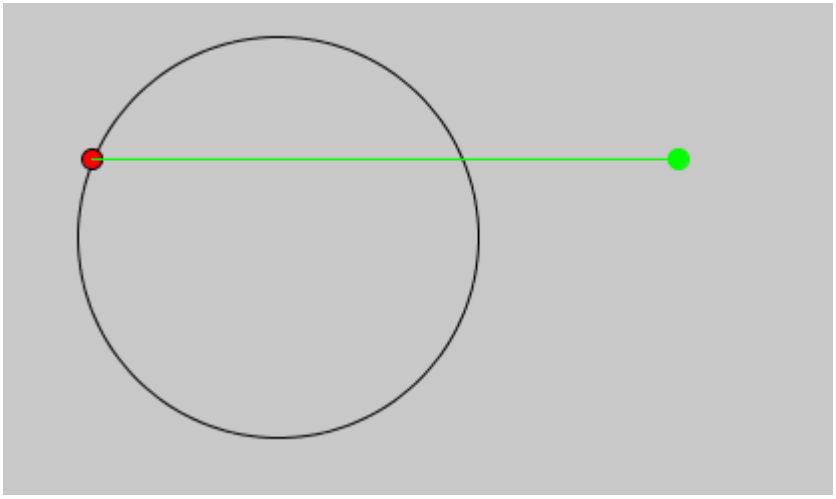


Figure 6-4

Now we want the green ellipse to leave a trail to show its height over time. Leaving a trail really means we save all the heights and display them all, every loop. How do we save a bunch of things? We need a list. Add this line to the variables we declared at the beginning of the program, before the setup function:

```
circleList = []
```

And add it to the “global” line in the draw function:

```
global t, circleList
```

After we calculate x and y in the draw function, save the y-coordinate to the circleList:

```
#add to list:
circleList.insert(0,y)
```

At the end of the draw function (before incrementing t) we'll put in a loop. It will loop through all the elements of the circleList and draw a new ellipse, to look like the green ellipse is leaving a trail.

```
#loop over circleList to leave a trail:
    for i in range(len(circleList)):
        #small circle for trail:
```

```
        ellipse(200+i,circleList[i],5,5)
```

A more “Pythonic” way of doing that is to use Python's built-in “enumerate” function. It’s a handy way of keeping track of the index and value of the items in a list. You’ll notice there are two “iterators” instead of just one, like this:

```
>>> for index, value in enumerate(["I","love","using","Python"]):
print(index,value)

0 I
1 love
2 using
3 Python
```

When you do this for your circle list, you can keep track of the “i” and the circle “c.”

```
#loop over circleList to leave a trail:
for i,c in enumerate(circleList):
        #small circle for trail:
        ellipse(200+i,c,5,5)
```

This is the start of your Trigonometry class of the future! You made a sine wave, as you see in Figure 6-5:

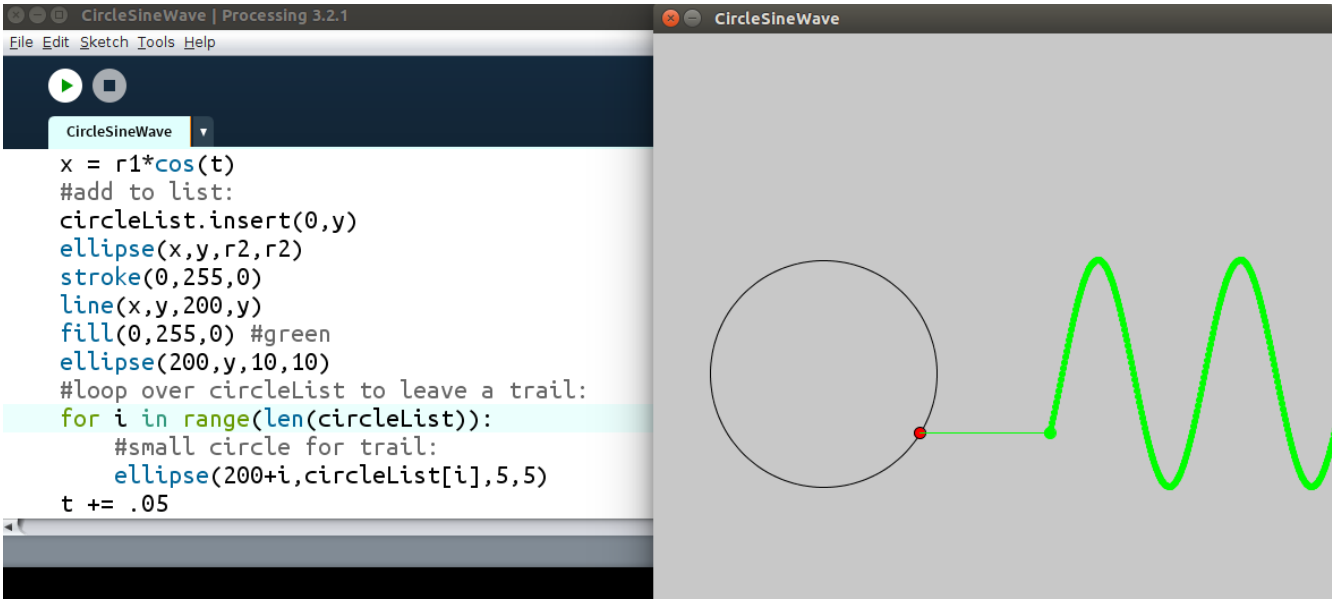


Figure 6-5: A sine wave

If you leave the program going for a while, you'll notice it starts to slow down. That's because the list of points is getting long, into the tens of thousands long! We can put a few lines of code in the draw function (right before the loop) to delete the first points if the list gets long enough:

```
if len(circleList)>300:
        circleList.remove(circleList[-1])
```

Spirograph

Now that we can rotate circles and leave trails, it's time to make a Spirograph model!

First we'll start our sketch, put a big circle in the middle and create variables for the big circle and the small one:

```
from __future__ import division
'''Spirograph Model'''

r1 = 300 #radius of big circle
r2 = 175 #radius of circle 2
#location of big circle:
x1 = 0
y1 = 0
t = 0 #time variable
points = [] #empty list to put points in
```

The first line is necessary so Python won't round off when it divides integers.

Now we can put the big circle on the screen and put a smaller circle on its circumference. But where to put the smaller circle?

Exercise 6-1: Our Ellipses Are so Close

Place the smaller circle on the circumference of the big circle, as in Figure 6-X:

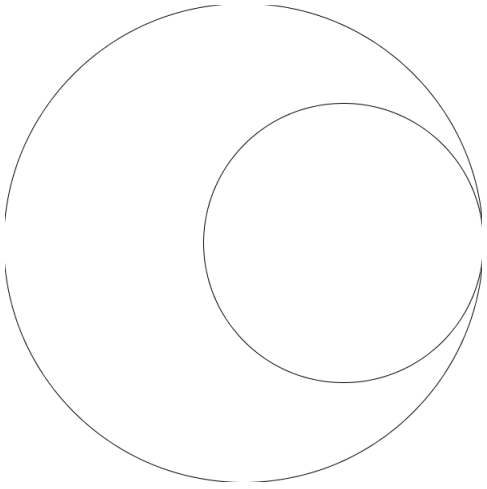


Figure 6-X

Solution:

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

def draw():
    global r1,r2,x1,y1
    translate(width/2,height/2)
    background(255)
    noFill()
    #big circle
    stroke(0)
    ellipse(x1,y1,2*r1,2*r1)
    #circle 2
    x2 = (r1 - r2)
    y2 = (r1 - r2)
    ellipse(x2,y2,2*r2,2*r2)
```

Exercise 6-2: I Can See Your Ellipse Moving

Make the smaller circle rotate around “inside” the bigger circle, as if it's a Spirograph gear.

Solution

We'll need to declare our global variable t at the beginning of the draw function, and add the “sin” and “cos” parts to the location of circle 2:

```
x2 = (r1 - r2)*cos(t)
y2 = (r1 – r2)*sin(t)
```

Finally at the very end of the draw function, we have to increment our time variable:

```
t += 0.05
```

Now it should spin nicely! But how about that hole on the gear where the pen sits and draws the trail? We'll create a third ellipse to represent that point. Its location will be the second circle's center plus the difference of r

```
#drawing dot
x3 = x2+(r2 - r3)*cos(t)
y3 = y2+(r2 – r3)*sin(t)
fill(255,0,0)
ellipse(x3,y3,r3,r3)
```

Run this and you'll see the drawing dot right on the edge of circle 2, rotating as if circle 2 were sliding along circle 1's circumference. Circle 3 has to be a certain proportion between the center of circle 2 and its circumference so we'll introduce a “prop” variable:

```
prop = 0.9
…
global r1,r2,x1,y1,t,prop
…
x3 = x2+prop*(r2 - r3)*cos(t)
y3 = y2+prop*(r2 – r3)*sin(t)
```

Now we have to figure out how fast it rotates. It only takes a little Algebra (or common sense?) to prove its angular velocity (how fast it spins around) is the ratio of the size of the big circle to the little circle. And the negative sign means it spins in the opposite direction:

```
x3 = x2+prop*(r2 - r3)*cos(-((r1-r2)/r2)*t)
y3 = y2+prop*(r2 - r3)*sin(-((r1-r2)/r2)*t)
```

All that's left is to save the point (x3,y3) to a “points” list and draw lines between the points, just like we did in the wave sketch. Create a points list and put that with the variables before the setup function:

```
points = []
```

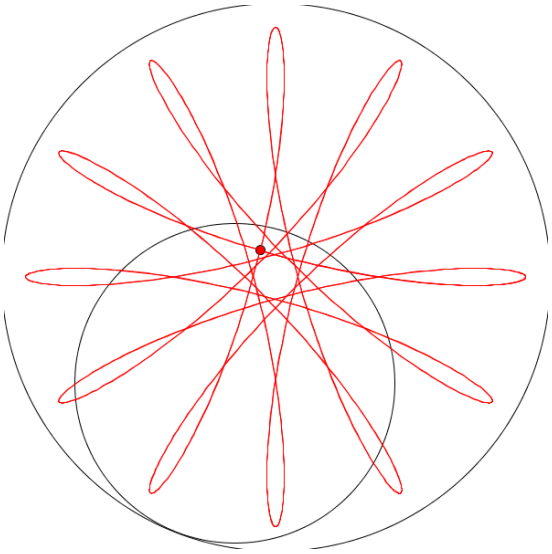
Then add the points list to the global line:

```
global r1,r2,r3,x1,y1,t,prop,points
```

After drawing the third ellipse, put the points into a list. Finally go through the list and draw lines between the points:

```
    points.append([x3,y3])
    if len(points) > 2000: #if the list gets too long
        points.pop() #remove the first point
    for i,p in enumerate(points): #go through the points list
        if i < len(points)-1: #up to the next to last point
            stroke(255,0,0)# draw red lines between the points
            line(p[0],p[1],points[i+1][0],points[i+1][1])
```

Run this and watch the program draw a Spirograph!



Our First Slice of Pi

What kind of math book makes you wait until Chapter 6 to get to pi? Pi is the ratio of the circumference of a circle to its diameter. The Greeks and Babylonians noticed the ratio was just over 3. Three and a half? Not even close. Three and a quarter? Closer, but still a bit off. Three and an eighth? Very close. In fact, 3 1/7 stood in for pi for centuries in math classes before the calculator. How can we get closer? Just a little trigonometry. We're going to follow in the footsteps of Archimedes, Ancient Greece's greatest scientist, who used polygons inscribed in a circle to approximate pi.

(rotatng triangles sketch moved to Geometry)

Adding Sliders

At the top of the sketch, before the setup function, type this line:

```
from slider import Slider
```

This means “From the slider.py file, import all the functionality of the Slider class.” Next, before the setup function, create an instance of the Slider object. Of course, you can create more than one by giving them different names:

```
slider1 = Slider(1,120,90)
```

This means “Create a slider with a range of values from 1 to 120, and set it initially to 90. Call this 'slider1.'”

Inside the setup function, set the slider position to an x-y coordinate, in this case (20,20):

```
slider1.position(20,20)
```

The last thing is to assign the value of the slider to a variable and keep updating the value in the draw function. For example, I'm going to create a variable called “num” which will be the number of triangles I put in this sketch. Every loop, the num value will be set to whatever value the slider shows. Add this line to the draw function.

```
num = int(slider1.value())
```

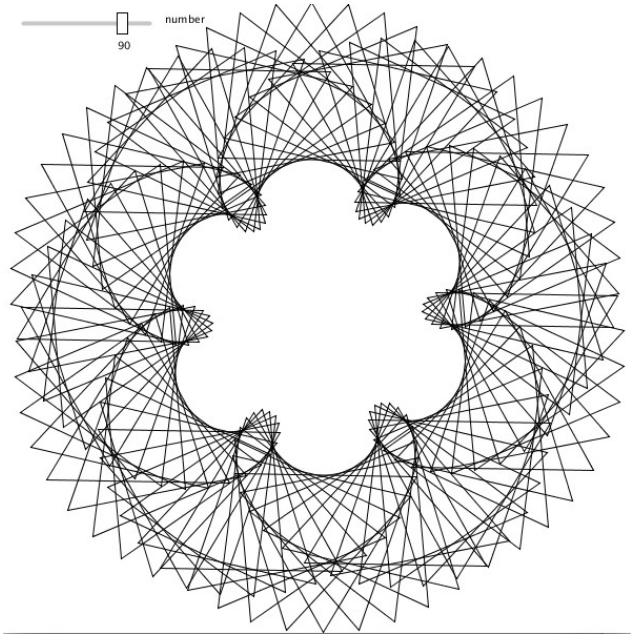
The “int” part makes sure the value assigned to “num” is an integer. That's important if you're using “num” in a loop, as we are in the Rotating Triangles sketch. “num” has to be an integer:

```
for i in range(num):
```

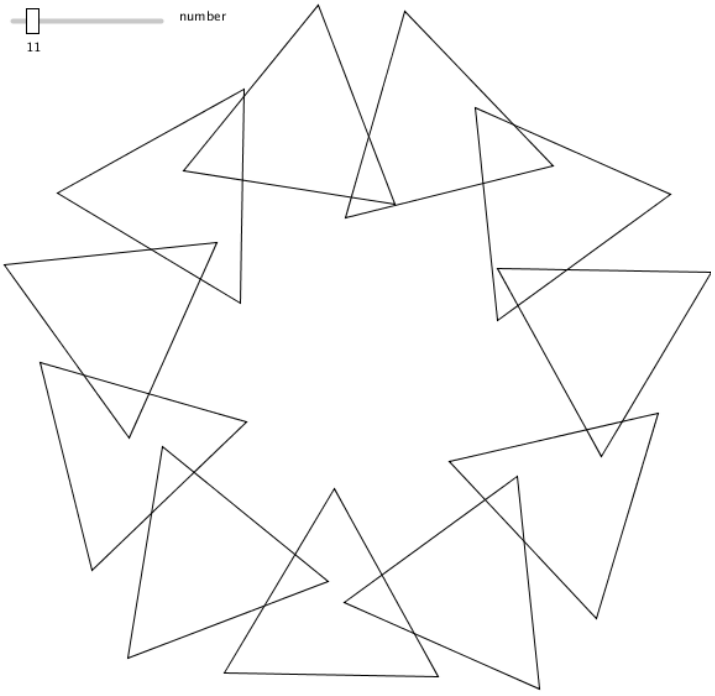
Optionally, you can add a label to your slider, to let your user know what they're changing. After the “position” line, add this line:

```
slider1.label = “number”
```

In the code, you have to replace all the places where you specified the number of triangles (it was 90) with the variable “num.” Now when you run the sketch, you'll have a bunch of rotating triangles. Initially, you'll have the “default” number, as in the figure below:



But when you move the slider you'll see the number update in real time:



Oscillating with Trigonometry

Another sketch that uses trig functions to oscillate back and forth is one by Dave “Bees and Bombs” Whyte. The interesting thing about this oscillation is that unlike our up-and-down sine wave in Figure 6.5, the wave oscillates in and out from the center. Start with the circle of circles template from the Geometry chapter in Figure 6.19:

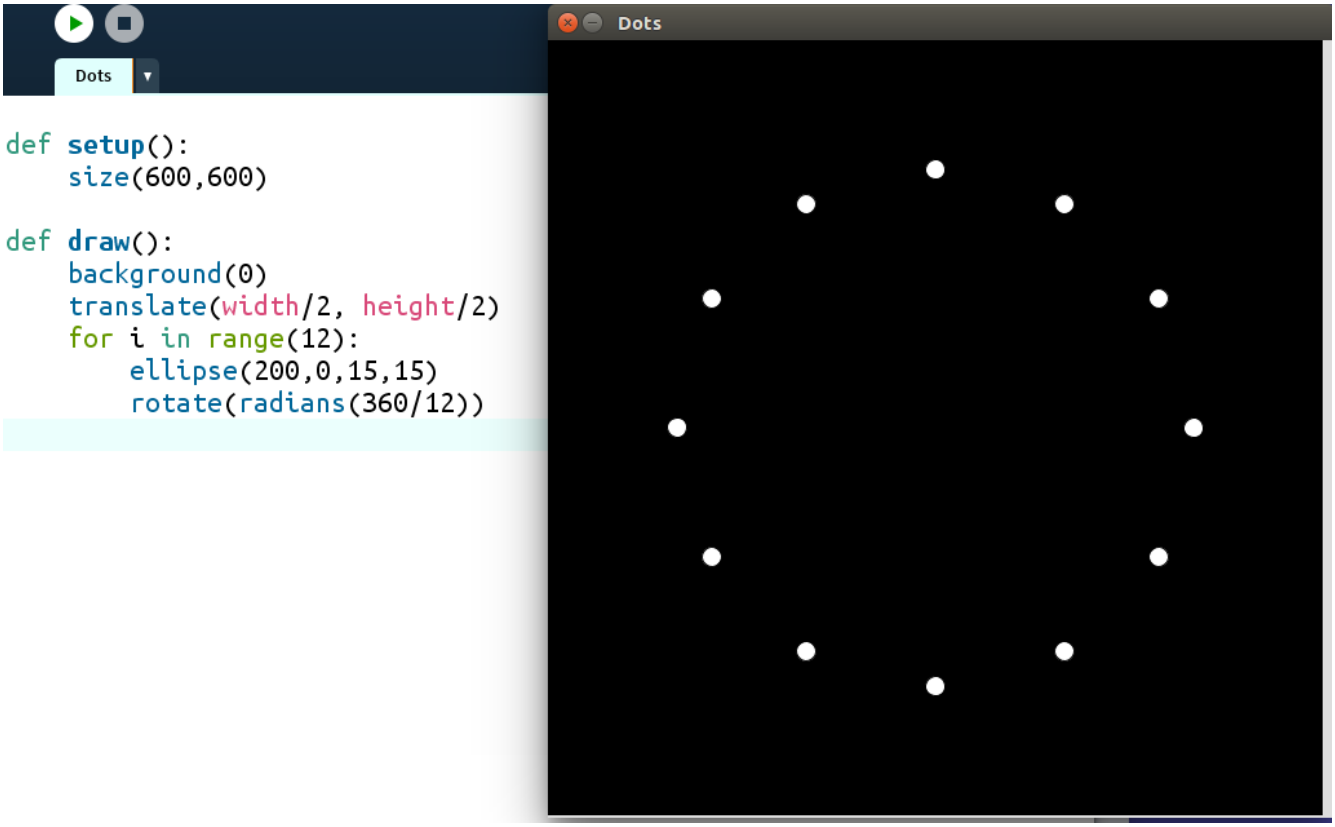


Figure 6.19

Now to oscillate the dots. Create a global time variable, use “sin(t)” in the ellipse location and increment time at the end of the draw function. The code now looks like this:

```
t = 0

def setup():
    size(600,600)

def draw():
    global t
    background(0)
    translate(width/2, height/2)
    for i in range(12):
        ellipse(120 + 90*sin(t),0,15,15)
        rotate(radians(360/12))
    t += 0.1
```

The dots are now oscillating in and out from the center. But they're moving in exactly the same way. Let's make them vary by their “i” value. Change the ellipse line to this:

```
ellipse(120 + 90*sin(t + i),0,15,15)
```

This makes each dot oscillate differently from its neighbors. You can change the look of the design by multiplying the “i” by a number. Play around with different numbers. To copy the sketch I saw, I used 0.75, and I changed the number of dots to 24:

```
for i in range(24):
    ellipse(120 + 90*sin(t+.75*i),0,15,15)
    rotate(radians(360/24))
```

To make some pretty rainbow colors, we’ll use the “Hue, Saturation, Brightness” color mode instead of “Red, Green, Blue.” Add this to your setup function:

```
colorMode(HSB) #hue, saturation, brightness
```

and just before drawing the ellipses in your draw function, add this:

```
#here’s the color; Vary the hue for rainbow
fill(15*i) % 255, 300, 300)
```

Harmonographs

Oscillating in one direction is one thing, but what about two directions? In the 1800s there was an invention called the harmonograph which was a table attached to two pendulums. When the pendulums swung the attached pen would draw on a piece of paper. When the pendulums died down (decayed), the patterns would change in interesting ways:

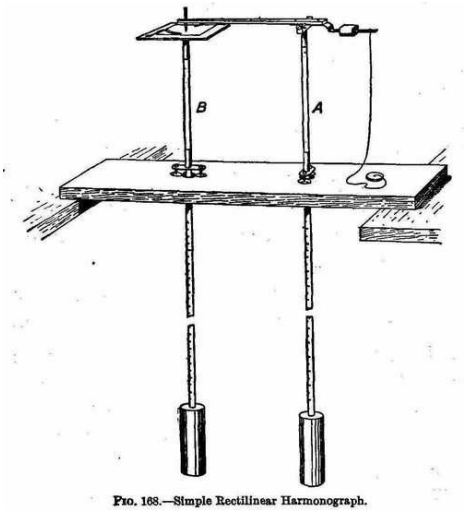
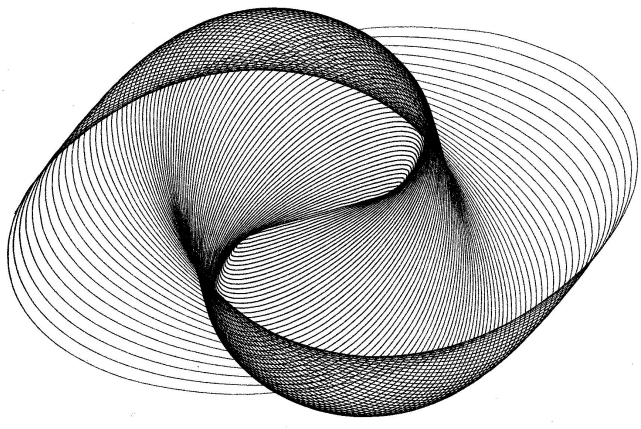


FIG. 168.—Simple Rectilinear Harmonograph.



Harmonograph machine and image

Using programming and a few equations, we can model how a harmonograph drew its patterns. The equation to model the oscillation of one pendulum is

$$y = \sin(ft + p)e^{-dt}$$

where y is the vertical displacement (up and down distance), f is the frequency of the pendulum, t is the elapsed time, p is the phase shift, e is the base of the natural logarithms (it's a constant, around 2.7), and d is a decay factor (how fast the pendulum slows down).