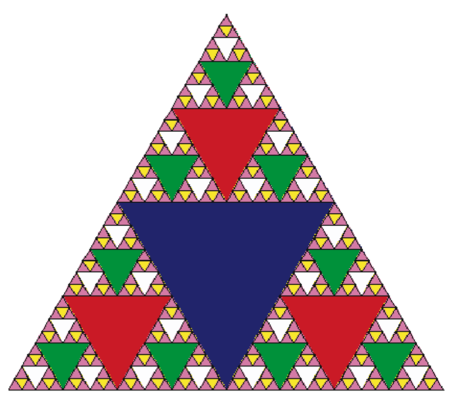
Sierpinski Code

(<http://interactivepython.org/courselib/static/pythonds/Recursion/pythondsSierpinskiTriangle.html>)

Another fractal that exhibits the property of self-similarity is the Sierpinski triangle. An example is shown in [Figure 3](http://interactivepython.org/courselib/static/pythonds/Recursion/pythondsSierpinskiTriangle.html#fig-sierpinski). The Sierpinski triangle illustrates a three-way recursive algorithm. The procedure for drawing a Sierpinski triangle by hand is simple. Start with a single large triangle. Divide this large triangle into four new triangles by connecting the midpoint of each side. Ignoring the middle triangle that you just created, apply the same procedure to each of the three corner triangles. Each time you create a new set of triangles, you recursively apply this procedure to the three smaller corner triangles. You can continue to apply this procedure indefinitely if you have a sharp enough pencil. Before you continue reading, you may want to try drawing the Sierpinski triangle yourself, using the method described.



Since we can continue to apply the algorithm indefinitely, what is the base case? We will see that the base case is set arbitrarily as the number of times we want to divide the triangle into pieces. Sometimes we call this number the “degree” of the fractal. Each time we make a recursive call, we subtract 1 from the degree until we reach 0. When we reach a degree of 0, we stop making recursive calls. The code that generated the Sierpinski Triangle in [Figure 3](http://interactivepython.org/courselib/static/pythonds/Recursion/pythondsSierpinskiTriangle.html#fig-sierpinski) is shown in [ActiveCode 1](http://interactivepython.org/courselib/static/pythonds/Recursion/pythondsSierpinskiTriangle.html#lst-st).

import turtle

def drawTriangle(points,color,myTurtle):

myTurtle.fillcolor(color)

myTurtle.up()

myTurtle.goto(points[0][0],points[0][1])

myTurtle.down()

myTurtle.begin\_fill()

myTurtle.goto(points[1][0],points[1][1])

myTurtle.goto(points[2][0],points[2][1])

myTurtle.goto(points[0][0],points[0][1])

myTurtle.end\_fill()

def getMid(p1,p2):

return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

def sierpinski(points,degree,myTurtle):

colormap = ['blue','red','green','white','yellow',

'violet','orange']

drawTriangle(points,colormap[degree],myTurtle)

if degree > 0:

sierpinski([points[0],

getMid(points[0], points[1]),

getMid(points[0], points[2])],

degree-1, myTurtle)

sierpinski([points[1],

getMid(points[0], points[1]),

getMid(points[1], points[2])],

degree-1, myTurtle)

sierpinski([points[2],

getMid(points[2], points[1]),

getMid(points[0], points[2])],

degree-1, myTurtle)

def main():

myTurtle = turtle.Turtle()

myWin = turtle.Screen()

myPoints = [[-100,-50],[0,100],[100,-50]]

sierpinski(myPoints,3,myTurtle)

myWin.exitonclick()

main()

(<https://en.wikipedia.org/wiki/Sierpi%C5%84ski_arrowhead_curve>)

void sierpinski\_arrowhead\_curve( unsigned order, double length)

{

*// If order is even we can just draw the curve.*

**if** ( 0 == (order & 1) ) {

curve( order, length, +60);

}

**else** */\* order is odd \*/* {

turn( +60);

curve( order, length, -60);

}

}

void curve( unsigned order, double length, int angle)

{

**if** ( 0 == order ) {

draw\_line( length);

} **else** {

curve( order - 1, length / 2, - angle);

turn( + angle);

curve( order - 1, length / 2, + angle);

turn( + angle);

curve( order - 1, length / 2, - angle);

}

}

(<https://courses.cs.washington.edu/courses/cse143/16wi/lectures/Sierpinski.java>)

// Program that draws the Sierpinski fractal.

import java.awt.\*;

import java.util.\*;

public class Sierpinski {

public static final int SIZE = 512; // height/width of DrawingPanel

public static void main(String[] args) {

// prompt for level

Scanner console = new Scanner(System.in);

System.out.print("What level do you want? ");

int level = console.nextInt();

// initialize drawing panel

DrawingPanel p = new DrawingPanel(SIZE, SIZE);

p.setBackground(Color.CYAN);

Graphics g = p.getGraphics();

// compute triangle endpoints and begin recursion

int triangleHeight = (int) Math.round(SIZE \* Math.sqrt(3.0) / 2.0);

Point p1 = new Point(0, triangleHeight);

Point p2 = new Point(SIZE / 2, 0);

Point p3 = new Point(SIZE, triangleHeight);

drawFigure(level, g, p1, p2, p3);

}

// Draws a Sierpinski fractal to the given level inside the triangle

// whose vertices are (p1, p2, p3).

public static void drawFigure(int level, Graphics g,

Point p1, Point p2, Point p3) {

if (level == 1) {

// base case: simple triangle

Polygon p = new Polygon();

p.addPoint(p1.x, p1.y);

p.addPoint(p2.x, p2.y);

p.addPoint(p3.x, p3.y);

g.fillPolygon(p);

} else {

// recursive case, split into 3 triangles

Point p4 = midpoint(p1, p2);

Point p5 = midpoint(p2, p3);

Point p6 = midpoint(p1, p3);

// recurse on 3 triangular areas

drawFigure(level - 1, g, p1, p4, p6);

drawFigure(level - 1, g, p4, p2, p5);

drawFigure(level - 1, g, p6, p5, p3);

}

}

// returns the midpoint of p1 and p2

public static Point midpoint(Point p1, Point p2) {

return new Point((p1.x + p2.x) / 2, (p1.y + p2.y) / 2);

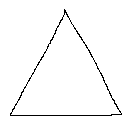
}

}

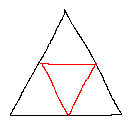
(<http://lodev.org/cgtutor/sierpinski.html>)

The Sierpinski Triangle, also called Sierpinski Gasket and Sierpinski Sieve, can be drawn by hand as follows:

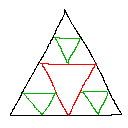
Start with a single triangle. This is the only triangle in this direction, all the others will be upside down:



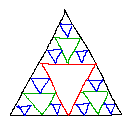
Inside this triangle, draw a smaller upside down triangle. Its corners should be exactly in the centers of the sides of the large triangle:



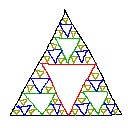
Now, draw 3 smaller triangles in each of the 3 triangles that are pointing upwards, again with the corners in the centers of the sides of the triangles that point upwards:



Now there are 9 triangles pointing upwards. In each of these 9, draw again smaller upside down triangles:



In the 27 triangles pointing upwards, again draw 27 triangles pointing downwards:



Rinse, repeat.

After infinite steps, and if all triangles pointing upwards would be filled, you have the Sierpinski Sieve.

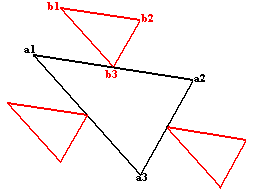
Every step, more triangles have to be drawn. This is a recursive process, and it can be drawn the same way with a computer.

**With Recursion**

We'll now program that what was drawn by hand on the computer, by making a triangle drawing function that'll call itself again 3 times, until n steps of recursions are reached. This program is made so that it works for any initial triangle, it doesn't have to be symmetrical, the only condition is that the corners lie inside the screen.

The main function sets up the screen and calls the drawSierpinski function. The drawSierpinski function itself will draw only one triangle: the initial one that points upwards. Then it'll call the function subTriangle, and that is the actual recursive function, that'll draw all the upside down triangles.

The function subTriangle draws a single upside down triangle, with the 3 corners you give it with its parameters. Then it'll call itself 3 times again, to draw 3 smaller triangles. For these 3 triangles, new corners are used of course, and these have to be calculated. In the following image, if the black triangle is the big triangle the subTriangle function has drawn, then the 3 red triangles are the new ones that have to be calculated:



The corners of the big triangle are a1, a2 and a3. The corners of one of the smaller triangles are b1, b2 and b3 as you can see on the image. If we see all these points as vectors, the the formulas for the points b (with the points a being known) are:

b3 = (a1 + a2) / 2, because b3 lies in the center between a1 and a2, this point is the average of a1 and a2!

b1 = b3 + (a1 - a3) / 2: if you examine well, you'll see that the point b1 is the sum of the point b3 and the vector (a1 - a3) / 2, it's divided through 2 because the corresponding side of the smaller triangle is half as big.

And finally,

b2 = b3 + (a2 - a3) / 2: this is very similar to the other formula, but with the other side.

For the other 2 small triangles, something similar is done.

In the code, we aren't using a vector class, so x and y are separate variables, and because of the way vector additions work, we simply have to do the same thing twice, once for x, and once for y, with the same formulas.

Another totally different way to draw an approximation of the Sierpinski Triangle works as follows:

1) define 3 points with coordinates: a (ax, ay), b (bx, by) and c (cx, cy). These will become the corners of the large outer triangle.

2) define another point, p (px, py) and place it in a corner of the triangle (for example px = ax and py = ay).

3) draw a point at location (px, py) with a pencil

4) roll a theoretical die with 3 sides (side 0, side 1 and side 2), or just make a random choice between "0", "1" and "2".

5) now change the coordinates of point p, depending on what number you rolled:

• if you rolled 0, p = (p+a) / 2

• if you rolled 1, p = (p+b) / 2

• if you rolled 2, p = (p+c) / 2

6) go back to step 2, there place the point at the new position of p, and keep looping through this until you get tired

Even though the whole process is randomized, after enough steps, the result will look more and more like a Sierpinski Triangle!

This isn't too hard to code, the comments explain how everything works:

//How much pixels should be randomly chosen and drawn

#define numSteps 10000

int main(int argc, char \*argv[])

{

//create the screen and make it white

screen(256, 256, 0, "Sierpinski Triangle");

cls(RGB\_White);

//the 3 corners of the outer triangle

float ax = 10;

float ay = h - 10;

float bx = w - 10;

float by = h - 10;

float cx = w / 2;

float cy = 10;

//initial coordinates for the point px

float px = ax;

float py = ay;

//do the process numSteps times

for(int n = 0; n < numSteps; n++)

{

//draw the pixel

pset(int(px), int(py), RGB\_Black);

//pick a random number 0, 1 or 2

switch(abs(rand() % 3))

{

//depending on the number, choose another new coordinate for point p

case 0:

px = (px + ax) / 2.0;

py = (py + ay) / 2.0;

break;

case 1:

px = (px + bx) / 2.0;

py = (py + by) / 2.0;

break;

case 2:

px = (px + cx) / 2.0;

py = (py + cy) / 2.0;

break;

}

}

//redraw, sleep and end the program

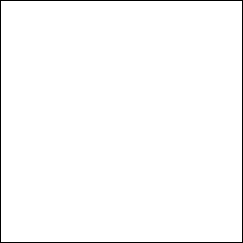
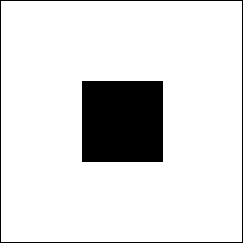
redraw();

sleep();

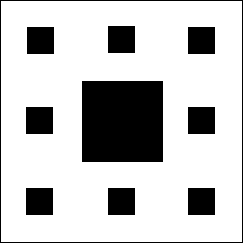
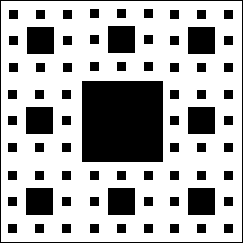
return(0);

}

A different fractal is the Sierpinski Carpet. To draw one by hand, start with a white square, and then draw a black square in the center with each side 1/3th of the size of the original square:

Now, around the black square, are 8 white squares. In each of these 8, draw again a black square that is 1/8th smaller, and in the 8\*8=64 new white squares, do it yet again:

Keep doing this until infinity, and you get a sierpinski carpet!

To draw it, we can use a recursive function that's pretty similar to the one used for the sierpinski triangle with rectangles, but now the function has to call itself 8 times instead of only 3, and use different coordinates. The coordinates x1,y1-x2,y2 get divided in 9 sections, in the center one the rectangle is drawn with rect, and the other 8 ones are used as parameters for the drawCarpet calls of the next recursion step.

//the coordinates are 2 corners of the rectangle, n is the current recursion step

void drawCarpet(int n, float x1, float y1, float x2, float y2);

//how much recursions maximum

#define maxRecursions 6

int main(int argc, char \*argv[])

{

//create the screen and make it white, use powers of 3 for the screen size for best result

screen(243, 243, 0, "Sierpinski Carpet");

cls(RGB\_White);

//start the recursive function

drawCarpet(1, 0, 0, w - 1, h - 1);

//redraw, sleep, etc...

redraw();

sleep();

return(0);

}

void drawCarpet(int n, float x1, float y1, float x2, float y2)

{

//draw black rectangle with 1/3th the size in the center of the given coordinates

drawRect(int((2 \* x1 + x2) / 3.0), int((2 \* y1 + y2) / 3.0), int((x1 + 2 \* x2) / 3.0) - 1, int((y1 + 2 \* y2) / 3.0) - 1, RGB\_Black);

//call itself 8 times again, now for the 8 new rectangles around the one that was just drawn

if(n < maxRecursions)

{

drawCarpet(n + 1, x1 , y1 , (2 \* x1 + x2) / 3.0, (2 \* y1 + y2) / 3.0);

drawCarpet(n + 1, (2 \* x1 + x2) / 3.0, y1 , (x1 + 2 \* x2) / 3.0, (2 \* y1 + y2) / 3.0);

drawCarpet(n + 1, (x1 + 2 \* x2) / 3.0, y1 , x2 , (2 \* y1 + y2) / 3.0);

drawCarpet(n + 1, x1 , (2 \* y1 + y2) / 3.0, (2 \* x1 + x2) / 3.0, (y1 + 2 \* y2) / 3.0);

drawCarpet(n + 1, (x1 + 2 \* x2) / 3.0, (2 \* y1 + y2) / 3.0, x2 , (y1 + 2 \* y2) / 3.0);

drawCarpet(n + 1, x1 , (y1 + 2 \* y2) / 3.0, (2 \* x1 + x2) / 3.0, y2 );

drawCarpet(n + 1, (2 \* x1 + x2) / 3.0, (y1 + 2 \* y2) / 3.0, (x1 + 2 \* x2) / 3.0, y2 );

drawCarpet(n + 1, (x1 + 2 \* x2) / 3.0, (y1 + 2 \* y2) / 3.0, x2 , y2 );

}

}