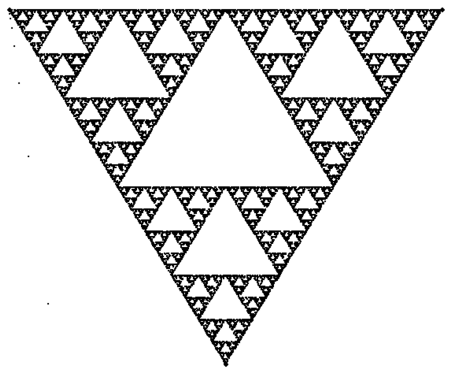
Chaos Project

David Jiang

May 2023

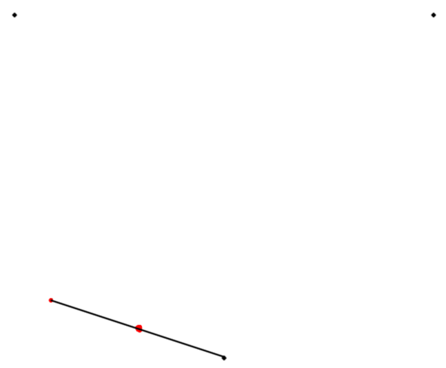
1 Introduction

The Chaos Game is a curious method in the generation of certain fractals. The one that I will be talking about will generate the Sierpinski triangle/gasket.

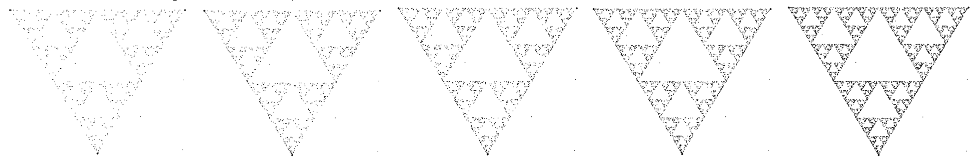


The game starts off with the three vertices of an equilateral triangle as well as a random point anywhere on the plane. The image is kind of hard to see:

From the red point, we select a vertex at random and draw a new point in the middle of our red point and the vertex we picked.



We will continue from this new point continuing our process. You will notice that once you do thousands of points, it will start to look like the Sierpinski triangle.
Iterations every 500 or 1,000 times:

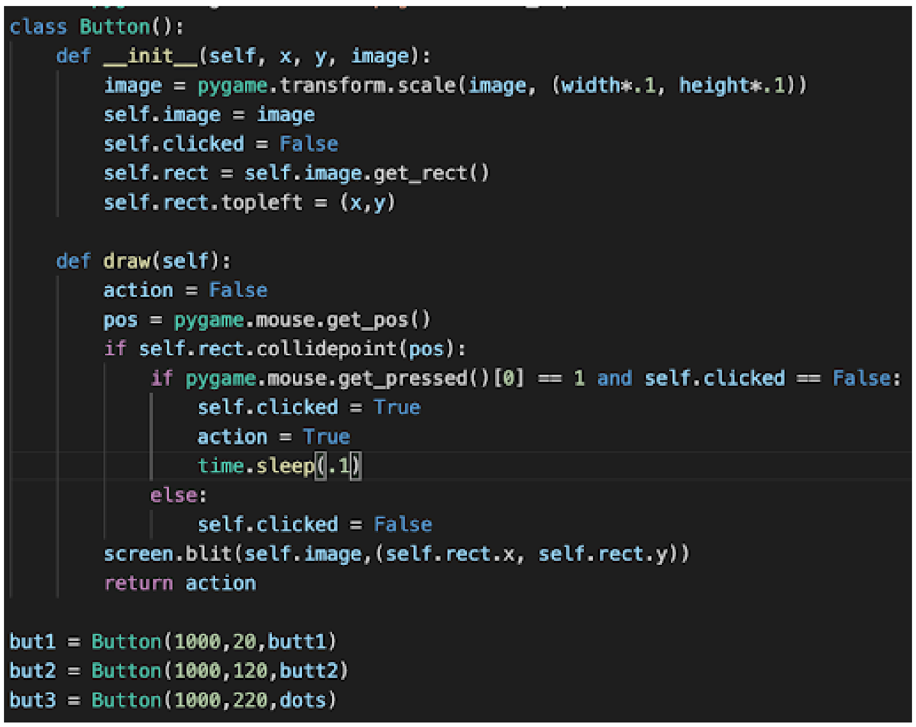


2 Code

I will briefly document and discuss how I programmed my own version of the game. Although I believe my method is probably not very good, it was just the first thing I thought of. My program is created using Python and particularly the interface is created using PyGame. I'm not extremely familiar with either, but I knew and was able to learn enough to complete this program.

I used some general boilerplate code I found online for animating a stage for the game. Then I created an array which contained the coorindates of our original 3 vertices. Using Python random, I generate a random point within the confines of the canvas. As I've mentioned, this can be anywhere in the plane, but for simplicity I bounded it. This point is saved in a seperate array, this array is responsible for storing all of the coordinates of points that we've selected since PyGame requires us to redraw each frame as we go on.

I added some artistic flare by creating a bunch of hue's of green that the points shuffle between. I just thought it made the program look a bit more interesting rather than a uniform color. There is an option to remove this as well. I added a few buttons that denote how many dots it will add. I stole some code online that added functionality for the buttons. There was one bug that I did need to fix. Since the code I found online made it so that the button triggered anytime the mouse button was pressed and it was hovering above the confines of the button, it would click multiple times per click. That defeated the purpose of adding one dot at a time. So I decided to change it so that each button has latency of 0.1 seconds. This disallows you from spam clicking it super fast, but fixes the issue of the button clicking multiple times which I found to be a more annoying problem.

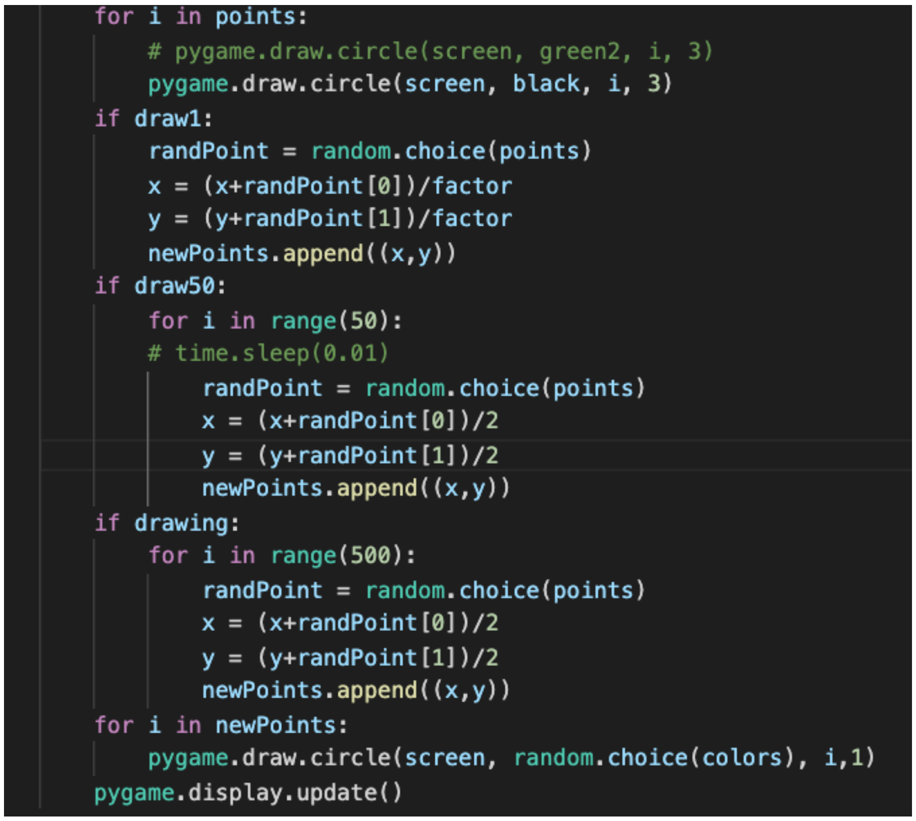


```python
class Button():
    def __init__(self, x, y, image):
        image = pygame.transform.scale(image, (width*.1, height*.1))
        self.image = image
        self.clicked = False
        self.rect = self.image.get_rect()
        self.rect.topleft = (x,y)

    def draw(self):
        action = False
        pos = pygame.mouse.get_pos()
        if self.rect.collidepoint(pos):
            if pygame.mouse.get_pressed()[0] == 1 and self.clicked == False:
                self.clicked = True
                action = True
                time.sleep(.1)
            else:
                self.clicked = False
        screen.blit(self.image,(self.rect.x, self.rect.y))
        return action

but1 = Button(1000,20,butt1)
but2 = Button(1000,120,butt2)
but3 = Button(1000,220,dots)
```

I wanted to create a method to draw circles. But there was something finicky going on with PyGame disallowing me to do so, so instead I just copy and pasted the code responsible for picking and drawing points in the function for all 3 of the buttons. This clutters the code a decent amount, and if I were programming it seriously I would figure this issue out. But all it does essentially is pick a random point in our vertices array and then find the midpoint of the previous point and the vertex. After that it would add this new point to our points array and then reassign the previous points coordinate.



```
    for i in points:
        # pygame.draw.circle(screen, green2, i, 3)
        pygame.draw.circle(screen, black, i, 3)
    if draw1:
        randPoint = random.choice(points)
        x = (x+randPoint[0])/factor
        y = (y+randPoint[1])/factor
        newPoints.append((x,y))
    if draw50:
        for i in range(50):
        # time.sleep(0.01)
            randPoint = random.choice(points)
            x = (x+randPoint[0])/2
            y = (y+randPoint[1])/2
            newPoints.append((x,y))
    if drawing:
        for i in range(500):
            randPoint = random.choice(points)
            x = (x+randPoint[0])/2
            y = (y+randPoint[1])/2
            newPoints.append((x,y))
    for i in newPoints:
        pygame.draw.circle(screen, random.choice(colors), i,1)
    pygame.display.update()
```

The code for the game itself is not too difficult logically speaking. There are just some fun things you can do with the graphic interface that I found interesting. There were other things that I intended on adding. In my version you can choose your own scale factor, number of points, and where the points are. I wanted to add the ability to select what point you wanted by drawing it on the board, but I never got around to adding this. There are also other rule sets that can be used. Such as the previously used vertex can't be used again which generate certain fractals for 4 corners, but I also didn't add this functionality.

Here is a link to watch a video of the program running: Link!

3 Understanding the game

Here we will prove that this game does converge to the Sierpinski triangle. This proof is adapted from a version of Arun Chagantys's proof on his website . We will do this by proving 3 statements that will prove the algorithm as a whole.

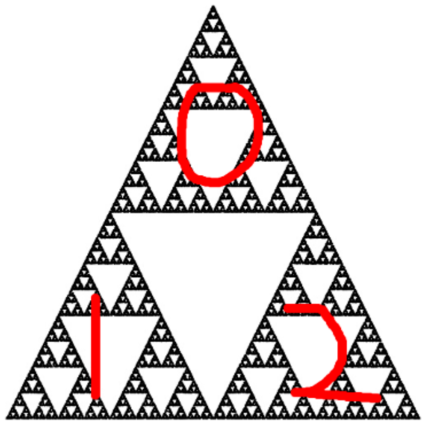
1. Any point on the Sierpinski triangle will stay on the Sierpinski triangle after iterating.

2. Any point not on the Sierpinski triangle will converge to a point on the Sierpinski triangle after iterating.

3. All points on the Sierpinski triangle will be arbitrarily close a point generated in finite time by our game.

3.1 Statement 1

Each sub-triangle in our fractal can be defined using a ternary expansion. We denote the top triangle as having an address of 0, left triangle as 1, and right triangle as 2.

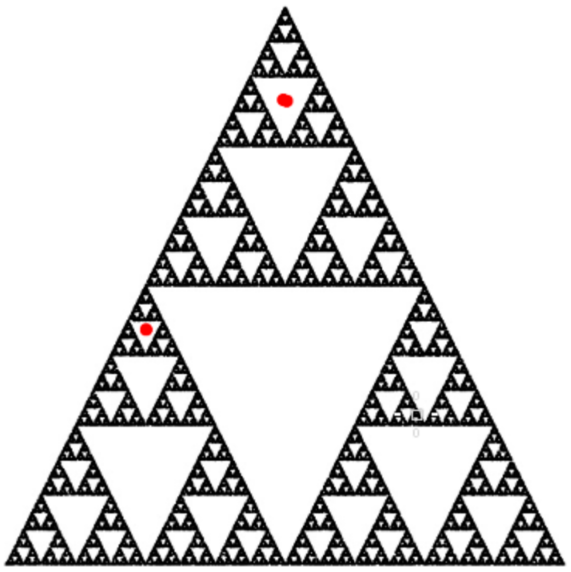


This is very similar to our construction of our Cantor set addresses. Each triangle can also be uniquely defined by their circumcenter. I will show now that for any circumcenters you select, for an of the 3 vertices you select, you will always end up being on another circumcenter, which will define another subtriangle.

We will first conjecture the following claim:

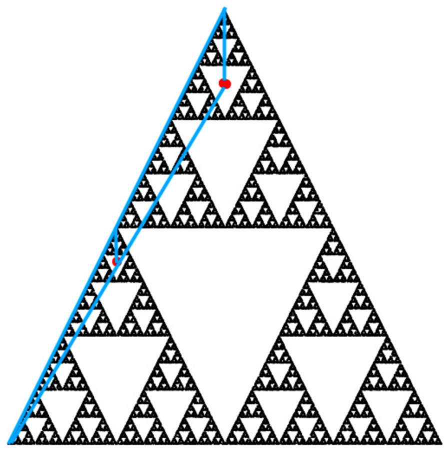
Claim: Given a ternary expansion for the center of a subtriangle to be x, the points on the triangle will all get mapped to the triangles $0x$, $1x$, or $2x$ depending on which vertex is selected.

Example:



This sends the subtriangle from 00_3 to 100_3. From a cursory view, it looks approximately like the midpoint between 00_3 and the left vertex. We will show that it is more rigorously.

Proof: Consider the triangles that I've highlighted:



We will show that these two triangles are similar and that their scale factor is $\frac{1}{2}$. We know that their long side is $\frac{1}{2}$ the size of each other since we're partitioning them by the equilateral triangle. By vertical angles we know that they share 2 angles with each other. By angle-angle similarity, that means these two triangles are similar and differe by a scale factor of $\frac{1}{2}$. In any construction of points, this will remain true because they will always share those angles and differ by a scale of $\frac{1}{2}$. Since the points around the circumcenter are uniquely defined by the circumcenter, that means that the entire triangle will be mapped in a similar manner. Therefore, any point on the Sierpinski triangle will remain on the Sierpinski triangle after iterating through our game.

3.2 Statement 2

Next we need to show that any point that is not on the Sierpinski triangle will eventually converge to a point on the Sierpinski triangle after we iterate enough times.

Given a point p_0 that is not on our Sierpinski triangle, let's say that q_0 is the point that is closest to p_0 that is on the Sierpinski triangle. We will show that as we continue our game, these two points will converge to the same point.

Let's let $\varepsilon_0 = |p_0 - q_0|$. Each iteration we will be halving the distance between p_n and q_n again because of similar triangles. This means that

$$|p_n - q_n| = \frac{\varepsilon_0}{2^n}$$

We can bound $\frac{\varepsilon_0}{2^n}$ by all $\varepsilon > 0$ by selecting a large enough n, which shows that the two points will eventually converge to the same point. Alternatively, here is the proof given by Arun:

For each step of the game, the points p_0 and q_0 move half way towards the vertex v_0. Such that $p_1 = \frac{p_0+v_0}{2}$ and $p_n = \frac{p_{n-1}+v}{2}$ and q_i work analogously. Then let's look at

$$|p_n - q_n| = \left|\frac{p_{n-1}+v}{2} - \frac{q_{n-1}+v}{2}\right| = \left|\frac{q_{n-1}+p_{n-1}}{2}\right|$$

Since we also have a way of defining each q_{n-1} and p_{n-1} we can unroll this recursive sequence. We notice that each time everything cancels out except for the next term in the sequence and an additional factor of $\frac{1}{2}$. Therefore this shows us that the distance must be $\frac{\varepsilon_0}{2^n}$.

3.3 Statement 3

I've spent a decent amount of time trying to understand the proof laid out by Arun. While I understand bits and pieces of it, I don't think I would have a good time trying to explain it myself or summarizing it in my own words. For this reason I will just refer you to his website for the proof of this statement.

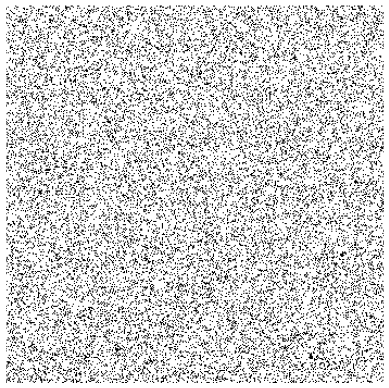
It is interesting to note that this proof is not specific to equilateral triangles and it is applicable to just triangles in general. This means that you can start off with any 3 points on a plane and applying the Chaos game to it, you will end up with a triangular gasket.

4 Other Fractals

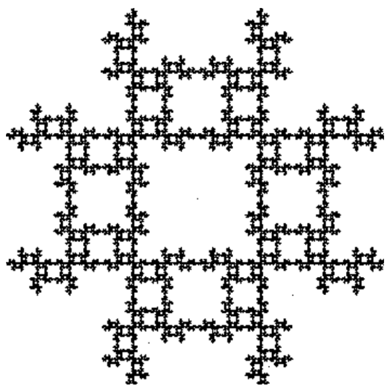
I mentioned in an earlier section that you can mess around with the number of points, the scale factor, or the rules in which you pick vertices. I will quickly talk about which interesting cases there are, but I won't be diving into the math in a similar way that I did for the basic Chaos game.

4.1 4 Vertices

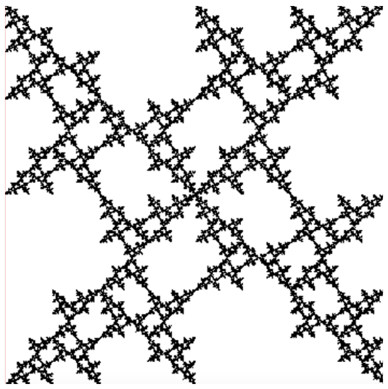
The first thing you might be wondering is what happens if we increase the amount of vertices. Applying the game to 4, we notice that nothing comes out of it. It approximately just fills in the square/rectangle itself.



This doesn't mean that you can't generate fractals with 4 vertices. If you add an additional rule where the current vertex that you selected can't be used for the next iteration, you end up with the following fractal:



You can even play with this rule set. If the current vertex cannot be one place away from the previous vertex, then we will achieve this fractal:



There are a lot more rules and things you can accomplish. I recommend looking at the wiki if you want to see more. All the examples and photos that I am providing are pulled from there.

4.2 More Cool Shit

I just wanted to share this one because it was super cool. If you have 5 points in a regular pentagon, then changing the scale factor to be $\frac{1}{\varphi}$ will give you the following fractal: