Recursive Graphics

Chace Carey, Kerem Erkmen, Connor Montague, Nancy Radwan

April 25, 2023

Contents

1 Introduction

1.1 Topic

Recursive graphs are a fascinating area of study in mathematics and computer science. A recursive graph is a type of graph that contains one or more vertices that are connected to themselves through one or more edges, forming a cycle or loop. This means that a path starting from one of these vertices can circle back to the same vertex after traversing a certain number of edges.

One of the key characteristics of recursive graphs is that they exhibit self-similarity, meaning that they have a similar structure at different scales. For example, if we zoom in on a portion of a recursive graph, we may see a similar substructure to that of the entire graph. This property has led to the use of recursive graphs in many fields, including computer science, mathematics, physics, and even art.

The study of recursive graphs has been ongoing for several decades, and researchers have made significant contributions to our understanding of their properties and behavior.
In computer science, recursive graphs are often used to model recursive data structures such as trees or linked lists. They are also useful in modeling complex systems that exhibit self-similarity, such as fractals or networks. Recursive graphs have applications in machine learning, natural language processing, and computational linguistics, among others.
Recursive graphs have also been used to study the behavior of algorithms and the complexity of computational problems.

1.2 Project

For our project, we were tasked with focusing on one of these aspects of our recursive graphics topic, creating the code to recursively draw 3 different fractals. These three fractals consist of the Sierpinski Triangle, Koch Snowflake, and the Hilbert Curve. After looking into each of the topics, our group decided on recursive graphics due to the fact that we felt we could be more creative with this topic and that the results would by far be the most pleasing to achieve. Our plan would be to implement each of the recursive drawing algorithms in C++ since that's the language of this course, however, C++ and our IDEs had no simple way to implement an actual drawing module, so, we decided that it would be best to use Python's Turtle graphics module. This way, we would be able to implement a C++ algorithm to write out all of the commands for our Python code to read and tell the turtle what to do.

2 Methods

2.1 Python Graphics

Before discussing the recursion used to generate the shapes themselves, it's important to understand how the shapes are getting drawn. To achieve this, Python's Turtle graphics module is used. The Python script follows a simple linear algorithm which reads the output generated by the shape algorithms written in C++, and translates them into commands that are fed to Python's Turtle module which allows the shapes to be drawn for the user. The Python script utilizes a small amount of user input to determine which shape to draw as well as what color(s) to use, and handles edge cases where the user gives an invalid input.

2.2 Sierpinski Triangle

A Sierpinski Triangle of depth n is achieved by drawing an equilateral triangle, which in itself contains three additional equilateral triangles whose dimensions are half the outer triangle. This pattern is repeated for every triangle drawn using recursion, where the depth decreases and the side length is halved for every sub-triangle until the depth reaches a base value. When the base value is reached, a simple equilateral triangle is drawn. As the base triangles are drawn the outer triangles will begin to take shape until the outermost triangle is completely drawn, resulting in the final output shape.

2.3 Koch Snowflake

A Koch Snowflake is made by drawing a smaller shape, known as the "Koch Curve", and repeating said curve three times on offset angles until the ends of all three curves meet, resulting in the final shape. In order to draw the Koch Curve, a straight line is drawn and divided into three equal parts. The middle third is removed, and is instead replaced by two line segments of equal length. This results in a curve made of four straight lines, whose lengths are all a third of the original line. To draw a Koch Curve of greater depth, this process is repeated for every individual line segment. This results in a curve in which every line segment is reduced to a third of its original length for every layer of depth, yet the length of every curve is $\frac{4}{3}$ the length of the original line segment when transformed. This overall process is to be repeated three times, offsetting the angle of the original line by 120 degrees, resulting in the Koch Snowflake: a closed shape made up of three identical Koch Curves.

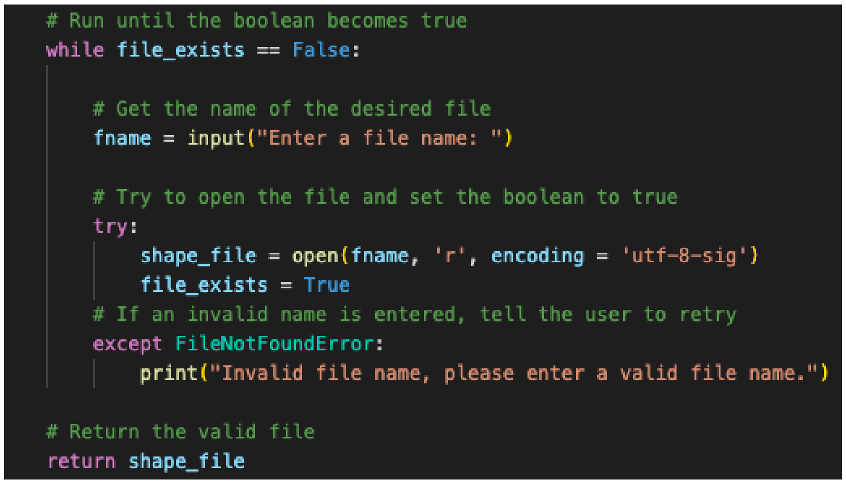
2.4 Hilbert Curve

The Hilbert Curve is the most advanced of the three recursive fractals. To create the most basic Hilbert Curve, first divide a square into an even 2x2 grid. Next, draw a line segment connecting the center of any quadrant to an adjacent quadrant. This process can be repeated, until only one connection remains to be made. At this point the curve resembles a horseshoe, or "U" shape (though it may be rotated depending on which quadrant was selected as the starter point. This is the Hilbert Curve at the most basic depth of 1. To draw a Hilbert Curve of deeper depth, divide all quadrants into 4 additional sub quadrants. Starting from an unaltered square, perform this step n times, where n is the desired depth of the curve. This will result in a $m \times m$ grid, where $m = 2^n$. For a grid of depth 2, the Hilbert Curve can be created by drawing the basic Hilbert Curve between 4 cells of the grid, 4 times. The top two curves are oriented so that the middle sides on the curve are facing inwards, and the bottom two curves are oriented so that the middle sides are facing downwards. To complete the curve, the adjacent ends of all four sub-curves are connected. For any depth, this pattern can be repeated until all sub-curves expand into the final Hilbert Curve, connecting all the cells of the grid.

3 Implementation

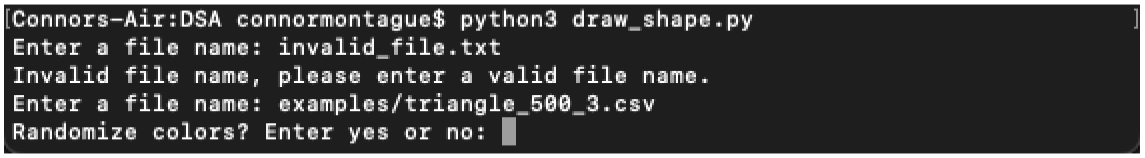
3.1 Python Graphics

On the implementation side, the Python graphics script contains two functions which are both called during runtime: open_file and draw_shape, the latter of which the script itself is named after. The first function, open_file, is called during draw_shape and prompts the user to input the name of the given file that they would like the script to draw. If/when the user enters a valid file name, the function opens said file and returns it to the draw_shape function. The open_file function also contains exception handling for the case where an invalid file is given to the script during runtime:



```python
# Run until the boolean becomes true
while file_exists == False:

    # Get the name of the desired file
    fname = input("Enter a file name: ")

    # Try to open the file and set the boolean to true
    try:
        shape_file = open(fname, 'r', encoding = 'utf-8-sig')
        file_exists = True
    # If an invalid name is entered, tell the user to retry
    except FileNotFoundError:
        print("Invalid file name, please enter a valid file name.")

# Return the valid file
return shape_file
```

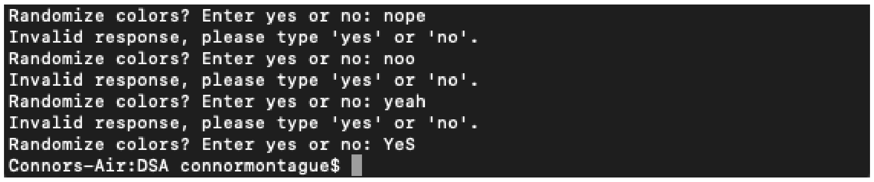
This prevents the user from inputting a file which doesn't exist, which would cause the script to crash. An example of this error handling is shown below below:



```
[Connors-Air:DSA connormontague$ python3 draw_shape.py
Enter a file name: invalid_file.txt
Invalid file name, please enter a valid file name.
Enter a file name: examples/triangle_500_3.csv
Randomize colors? Enter yes or no:
```

As can be seen, when an invalid file name was given, the script informed the user that the file they tried to enter could not be opened, and re-prompted them to enter a valid file name. When a valid file name *was* entered, the script moved on to the next step of the drawing process, contained in the draw_shape function.

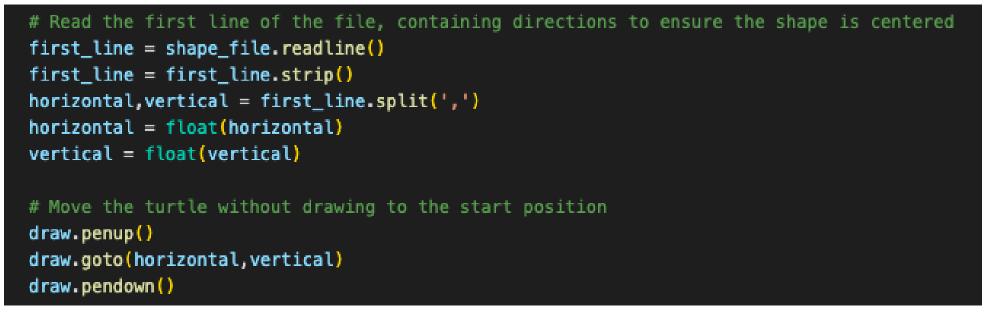
The draw_shape function begins by calling open_file and storing the input file to a variable so that it can be read through. As drawing the desired fractals strictly in black & white would be rather boring, the user is prompted if they'd like the colors of the shapes to be randomized or not (as shown by the previous example image). This input follows the same error handling as open_file in the case where the user is unable to answer a yes or no question with a simple “yes” or “no”. Additionally, this input is not case sensitive, as illustrated by the example below.



```
Randomize colors? Enter yes or no: nope
Invalid response, please type 'yes' or 'no'.
Randomize colors? Enter yes or no: noo
Invalid response, please type 'yes' or 'no'.
Randomize colors? Enter yes or no: yeah
Invalid response, please type 'yes' or 'no'.
Randomize colors? Enter yes or no: YeS
Connors-Air:DSA connormontague$ 
```

Entering “no” causes the shape to be drawn in standard black, whereas entering “yes” will cause the shape to be drawn with randomly changing colors. Adding additional options to offer more color variety is possible, but was left for the time being as a randomized set of colors, as the script to draw the shapes is a simple algorithm which didn’t need to be over-complicated.

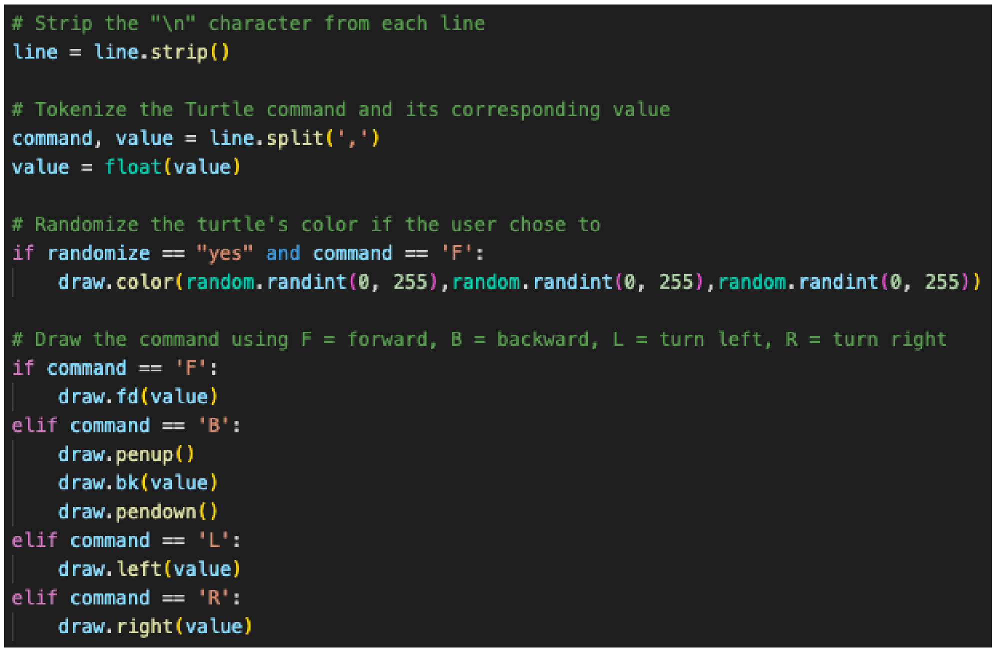
In the rest of the algorithm, the script reads the input file and acts on various commands to draw the shape input by the user. In this script’s implementation, the first line of the input file is always a coordinate pair of the form (x,y) which tells the drawing cursor where to start, where (0,0) is the center of the drawing window.



```
# Read the first line of the file, containing directions to ensure the shape is centered
first_line = shape_file.readline()
first_line = first_line.strip()
horizontal,vertical = first_line.split(',')
horizontal = float(horizontal)
vertical = float(vertical)

# Move the turtle without drawing to the start position
draw.penup()
draw.goto(horizontal,vertical)
draw.pendown()
```

This feature is important, as the coordinate pair is calculated by each shape C++ file to ensure that the shape is centered within the drawing window.

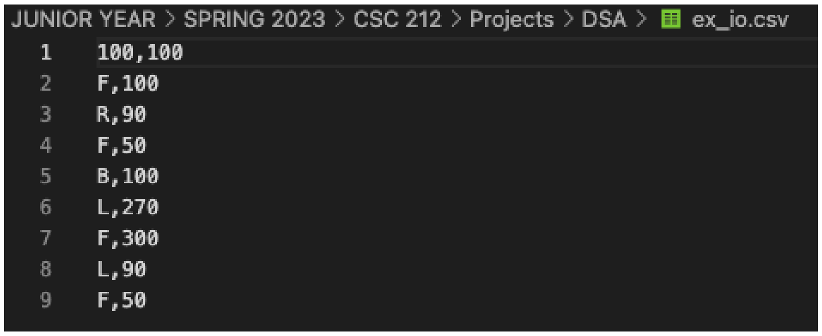
Once the cursor is in position, the script linearly reads the rest of the file line by line. Each line contains two distinct values separated by a comma, in the form “command, value”. The “command” value is a single character which tells the turtle whether to move forward or backward, or to turn left or right, while "value" is how many pixels/degrees the cursor is to move/turn by.



```python
# Strip the "\n" character from each line
line = line.strip()

# Tokenize the Turtle command and its corresponding value
command, value = line.split(',')
value = float(value)

# Randomize the turtle's color if the user chose to
if randomize == "yes" and command == 'F':
    draw.color(random.randint(0, 255),random.randint(0, 255),random.randint(0, 255))

# Draw the command using F = forward, B = backward, L = turn left, R = turn right
if command == 'F':
    draw.fd(value)
elif command == 'B':
    draw.penup()
    draw.bk(value)
    draw.pendown()
elif command == 'L':
    draw.left(value)
elif command == 'R':
    draw.right(value)
```

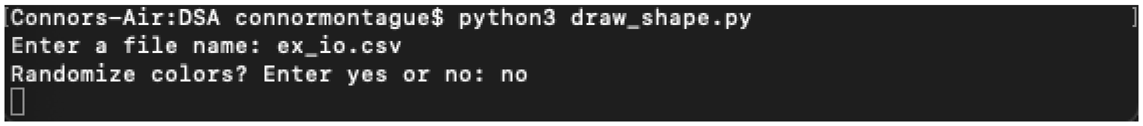
From the image above, it is seen that the cursor will perform various actions based on the contents of each line. Additionally, the cursor will change colors every time it moves forward to draw should the user have chosen it to. When the current command is equal to 'B' for "backwards", the cursor will move backwards while lifting the pen, so that unnecessary overlap does not occur during the drawing process. This is because the Sierpinski Triangle requires that the cursor move backwards at times.

An example of a generic input and output using the basic drawing commands is shown below:

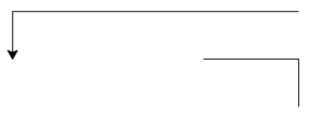


JUNIOR YEAR > SPRING 2023 > CSC 212 > Projects > DSA > ex_io.csv

```
100,100
F,100
R,90
F,50
B,100
L,270
F,300
L,90
F,50
```



```
Connors-Air:DSA connormontague$ python3 draw_shape.py
Enter a file name: ex_io.csv
Randomize colors? Enter yes or no: no
```



From the basic input file "ex_io.csv", the Python script parsed through the file and drew each command to the graphics window.

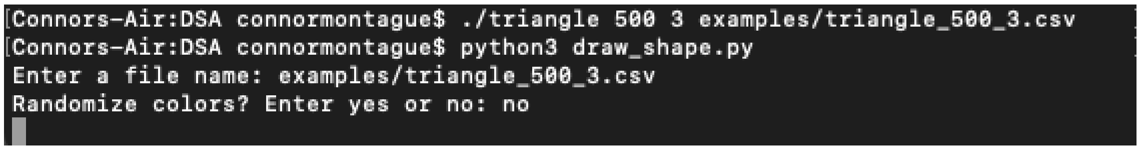
3.2 Sierpinski Triangle

In order to implement the algorithm described above to generate a Sierpinski Triangle, the executable name along with three other command line arguments are entered into the. The command line arguments, in order, are the desired outermost length and depth of the triangle, and the name of the output file which the algorithm will write Turtle commands to.

Before the recursive algorithm begins running, the length and depth are stored into float type variables, while the file name is used to open an output filestream. The length and depth variables are stored as floats as they need to be divided several times, wherein some cases the quotient may contain a remainder. Since the Python graphics window is only 700x700 pixels, the program will correct any lengths that are too large down to 700, and likewise lengths below 100 up to 100, as it is difficult to visualize larger depths when the base shape is already small. Lastly, before the recursive algorithm runs, mathematical operations based on geometry are performed in order to find the coordinates which the Python graphics cursor should start at, ensuring that the shape is centered in the graphics window. All steps and practices mentioned thus far, which take place in main, are also performed in the programs which generate the Koch Snowflake, as well as the Hilbert Curve. As such, these details will be omitted in those sections in order to prevent redundancy.

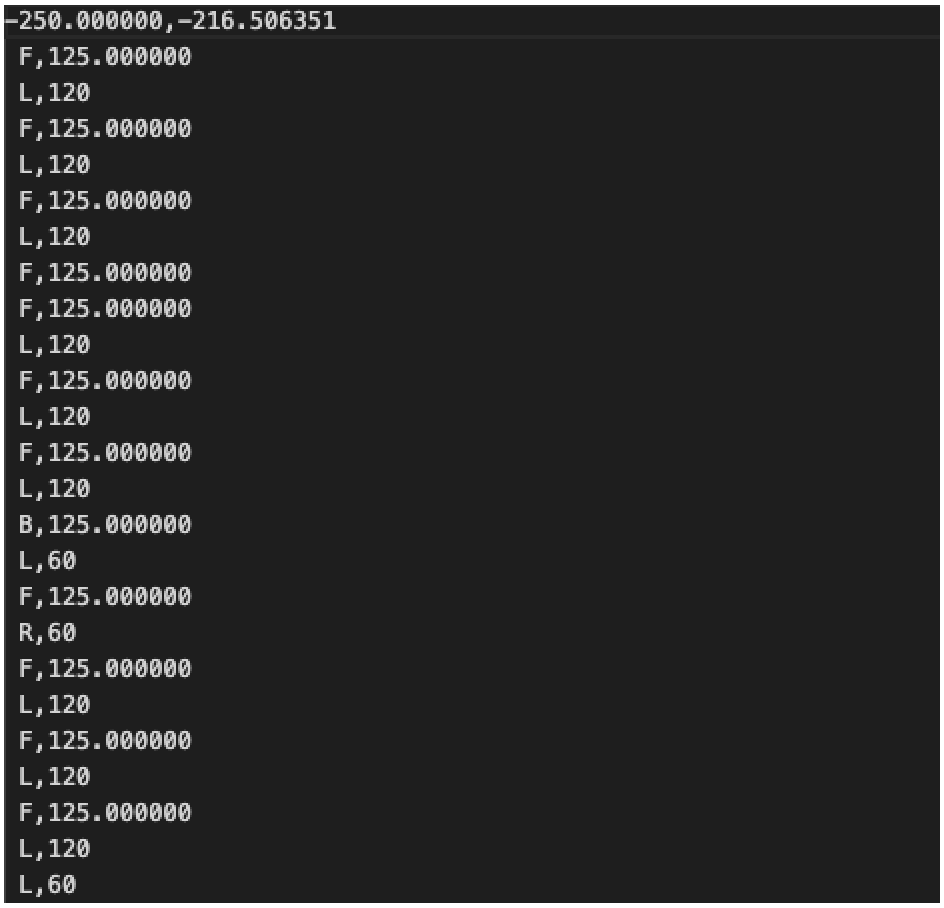
The void type function which generates the commands to draw the Sierpinski Triangle, titled “draw_triangle”, accepts the depth, length, and output filestream as arguments when it is called. The base case for the recursive function is when depth is equal to 1. When the base case is reached, the function will write to the output file the commands to draw a simple equilateral triangle. Outside the base case, the function first calls itself, dividing the length by 2 each time. Once this call is returned, the function will write a command to move the cursor forwards by the current length, where it will then call itself again. Once the second recursive call returns, there will be two side by side equilateral sub-triangles with the half current side length. The function then writes to the output file a sequence of commands to position the cursor at the upper vertex of the left sub-triangle, where it will then call itself for a third and final time in order to draw the last sub-triangle in the upper middle position. Once the third sub-triangle has been drawn, the function will return back up the call stack, a final sequence of commands is written to return the cursor to the original starting position of the first sub-triangle. After these commands are written, the function returns back up the call stack. where more triangles may be drawn depending on the current and desired depths.

An example input and output of the C++ program and Python script to generate and draw a Sierpinski Triangle with length 500 and depth 5 is shown below.



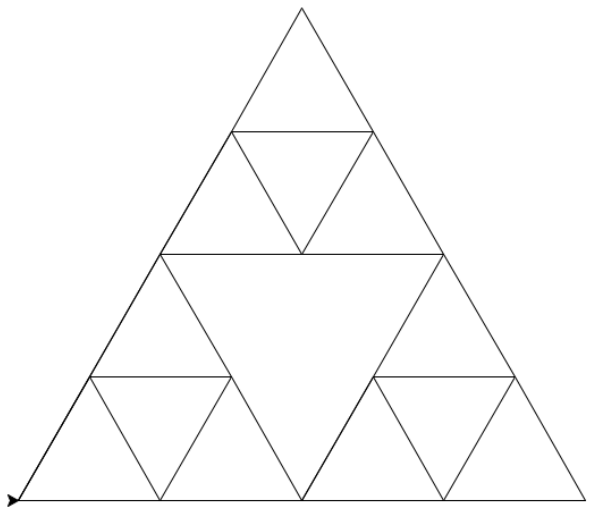
```
[Connors-Air:DSA connormontague$ ./triangle 500 3 examples/triangle_500_3.csv
[Connors-Air:DSA connormontague$ python3 draw_shape.py
Enter a file name: examples/triangle_500_3.csv
Randomize colors? Enter yes or no: no
```

Command Line inputs to generate and draw a triangle with length 500 and depth 3



```
-250.000000,-216.506351
 F,125.000000
 L,120
 F,125.000000
 L,120
 F,125.000000
 L,120
 F,125.000000
 F,125.000000
 L,120
 F,125.000000
 L,120
 F,125.000000
 L,120
 B,125.000000
 L,60
 F,125.000000
 R,60
 F,125.000000
 L,120
 F,125.000000
 L,120
 F,125.000000
 L,120
 L,60
```

The first 25 lines of the output file for a Sierpinski Triangle with length 500 and depth 3 (88 lines total)



Python generated output for the given input commands

3.3 Koch Snowflake

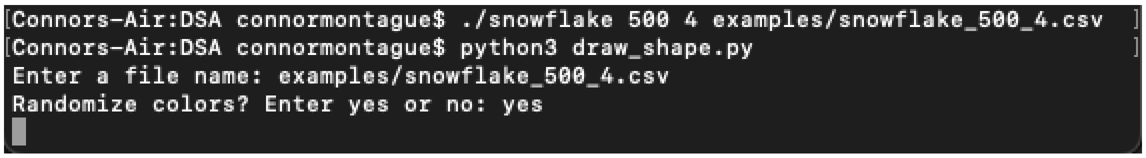
As described in Section 2.3, the Koch Snowflake is drawn not by generating commands for the entire snowflake in one go, but rather generating the commands to draw the Koch Curve three times, each offset by 120 degrees. Similar to the Sierpinski Triangle, the type void recursive function to generate the commands which draw the Koch Curve (“draw_curve”) accepts the length, depth, and output file stream as arguments.

The base case for this recursive function is if the current depth is equal to 0. Outside of the base case, the function first calls itself where it divides the length of the current call by 3. Once this call returns, a command to turn the cursor left by 60 degrees is written to the output file. The function then calls itself again, dividing the current length by 3 as before. When the second recursive call returns, a command to turn the cursor 120 degrees to the right is called. The function is called a third time in the same fashion, and after a command to turn the cursor left by

60 degrees again is written to the output file. Lastly, the function calls itself a fourth and final time. This results in the curve being procedurally drawn with side lengths continuously dividing by 3 until the desired depth is reached.

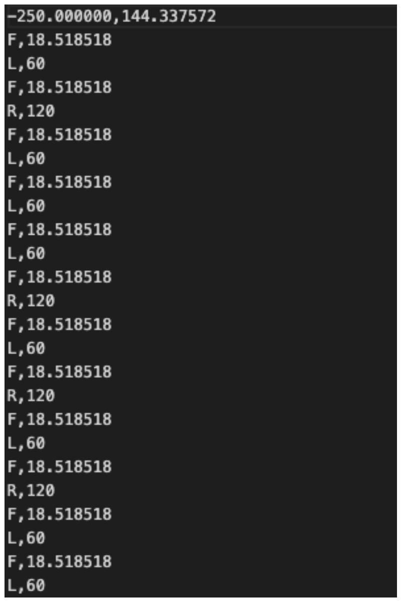
When the original call of draw_curve returns, a command is written to the output file to turn the cursor right by 120 degrees. The original function call, as well as the additional turn right command, are contained in main in a for-loop which runs 3 times , resulting in 3 connected Koch Curves which combine to make the Koch Snowflake.

An example input and output C++ program and Python script to generate and draw a Koch Snowflake of length 500 and depth 4, with randomized colors, is shown below.



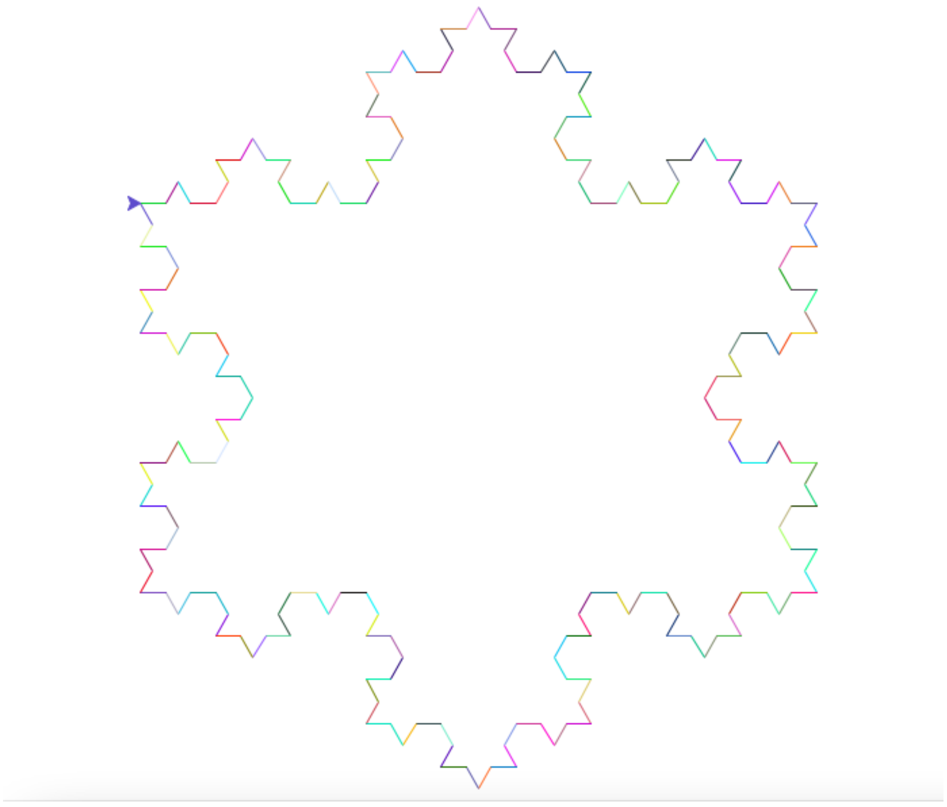
```
[Connors-Air:DSA connormontague$ ./snowflake 500 4 examples/snowflake_500_4.csv  ]
[Connors-Air:DSA connormontague$ python3 draw_shape.py                            ]
 Enter a file name: examples/snowflake_500_4.csv
 Randomize colors? Enter yes or no: yes
```

Command Line inputs to generate and draw a snowflake with length 500 and depth 4



```
-250.000000,144.337572
F,18.518518
L,60
F,18.518518
R,120
F,18.518518
L,60
F,18.518518
L,60
F,18.518518
L,60
F,18.518518
R,120
F,18.518518
L,60
F,18.518518
R,120
F,18.518518
L,60
F,18.518518
R,120
F,18.518518
L,60
F,18.518518
L,60
```

The first 25 lines of Turtle commands to draw the Koch Snowflake with base length 500 and depth 4 (385 lines total)

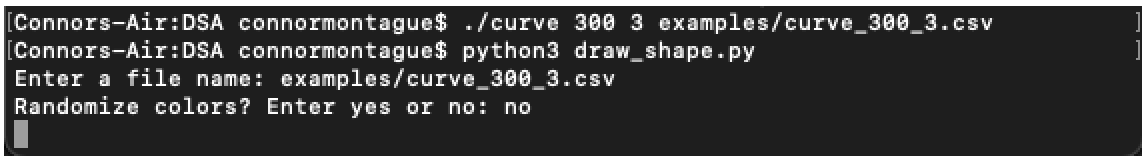

Python drawn Koch Snowflake with base length of 500, depth of 4, and randomized colors

3.4 Hilbert Curve

The commands to draw the Hilbert Curve are generated by the void type function "hilbert". The function takes the desired length, depth, turn angle (always 90 degrees), and output file stream as arguments. Since the curve is made by connecting the centers of cells in a grid, the length isn't divided recursively, but instead is input to the original function call divided by 2^{n-1}, where n is the desired depth of the Hilbert Curve.

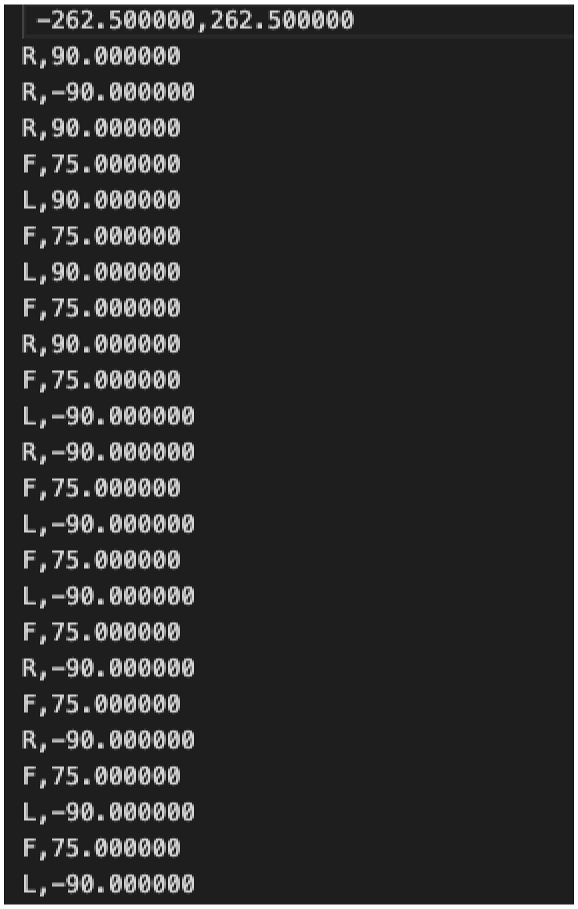
The base case for the function hilbert is when the depth is equal to 0. In the base case no commands are written to the output file, the function simply returns up the call stack. The function starts by writing to the output a turn 90 degrees to the right, then it calls itself, decreasing the depth by 1 and changing the sign of the turn angle. The sign on the turn angle is flipped because the sub Hilbert-Curves are drawn in continuously different rotations, and the sub-curves are drawn starting from the two different legs of the curves. Once the first call returns, commands are written to the output to move the cursor forwards and turn left by the current angle (or right when the angle is negative). After this, the function calls itself two more times, this time only decreasing the depth. In between the second and third calls, is an output command to move the cursor forwards by the length. Once the third call returns, commands to move the cursor forwards and turn left are written to the output, and the function calls itself one final time again changing the sign of the turn angle. When the fourth recursive call returns, a final command is written to the output to turn the cursor right by the current angle, which returns the cursor to its original orientation from before the sub-curve was drawn.

An example input and output of the C++ program and Python script to generate and draw a Hilbert Curve of base length 300 and depth of 3 is shown below.



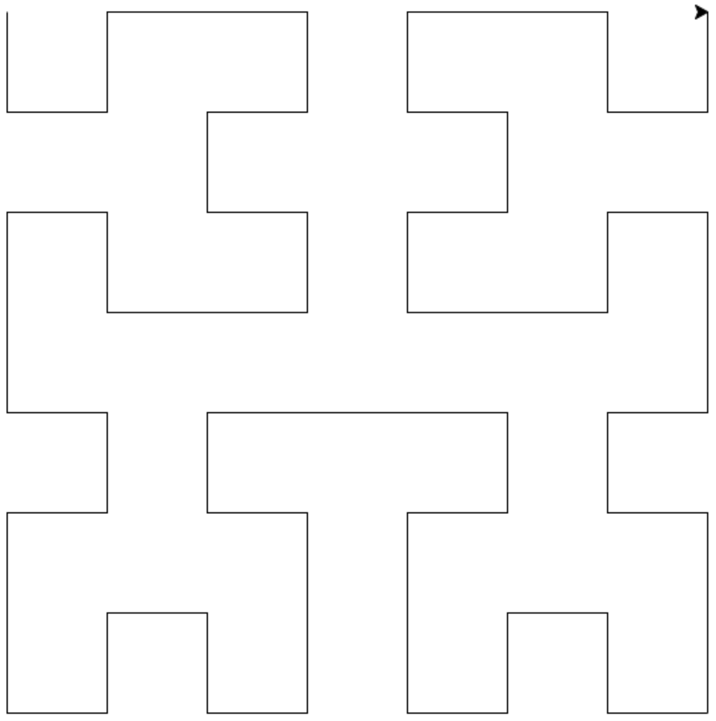
```
[Connors-Air:DSA connormontague$ ./curve 300 3 examples/curve_300_3.csv          ]
[Connors-Air:DSA connormontague$ python3 draw_shape.py                           ]
 Enter a file name: examples/curve_300_3.csv
 Randomize colors? Enter yes or no: no
```

Command Line inputs to generate and draw a curve of base length 300 and depth 3



```
-262.500000,262.500000
R,90.000000
R,-90.000000
R,90.000000
F,75.000000
L,90.000000
F,75.000000
L,90.000000
F,75.000000
R,90.000000
F,75.000000
L,-90.000000
R,-90.000000
F,75.000000
L,-90.000000
F,75.000000
L,-90.000000
F,75.000000
R,-90.000000
F,75.000000
R,-90.000000
F,75.000000
L,-90.000000
F,75.000000
L,-90.000000
```

First 25 lines of Turtle commands for a Hilbert Curve of base length 300 and depth 3 (148 lines total)


Python generated graphic of a Hilbert Curve with base length 300 and depth 3

4 Contributions

Name	Contribution %
Chace Carey	25%
Kerem Erkmen	25%
Connor Montague	25%
Nancy Radwan	25%

5 Conclusions

Along the way, as we were programming and testing, there were many small issues that we would stumble upon where we realized we could optimize our algorithm to make it run faster or take up less memory. One of the issues we ran into which was unexpected was actually making sure that when the fractals are drawn, the turtle stays within the screen’s boundaries. We discovered that the Python Turtle is set to always begin at the center of the screen, which causes the fractals of larger lengths and depths to be pushed out of bounds. To fix this, we deduced that the center of the screen should instead match up with the geometric center of the fractal. This took some math to figure out but when we had the correct equation for each shape and set a max length, the problem was solved. As for runtime and memory optimization, initially we had our C++ code writing all of the turtle commands, separated by a space, to a string. Then we’d take that string and loop through it to write each command and its value, separated by a comma, to an individual line in a csv file. Instead of this, we optimized it to skip the string entirely and write each command directly into the outfile. This cut down on the program’s overall runtime and the memory used during runtime, as for shapes of greater depth the strings would become very long. This is what the topic of recursive graphics is all about, taking a lengthy and monotonous process and optimizing it, turning it into smaller processes in order to cut down on runtime and runtime memory usage.

6 References

Editors of Encyclopædia Britannica. “Fractal.” *Encyclopædia Britannica*, Encyclopædia Britannica, Inc., 2004, https://www.britannica.com/science/fractal.

“Introduction to Recursion - Data Structure and Algorithm Tutorials.” *GeeksforGeeks*, GeeksforGeeks, 31 Mar. 2023, https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/.

Matsumoto, Saburo. “7.4: Fractals.” *Mathematics LibreTexts*, Libretexts, 12 Sept. 2020, https://math.libretexts.org/Courses/College_of_the_Canyons/Math_100%3A_Liberal_Arts_Mathematics_(Saburo_Matsumoto)/07%3A_Mathematics_and_the_Arts/7.04%3A_Fractals.