Recursive Graphics: Recursive Fractals Using L-Systems

Ilanna Langton, Karina Larochelle,
Lily Nguyen, Jenny You

December 4, 2023

Table of Contents

1 Introduction

1.1 Topic

Recursive graphics are composed of a simple shape being drawn repeatedly to form complex and captivating images. Through the iterative process of recursion, fractals are formed and repeated, showcasing self-similarity at various scales to create a larger image. Many of these graphics mesmerizing patterns are well known among math and computer science as understanding how recursion works is important to problem solving and realizing how code can be reused to create elegant solutions.

For many years, people have been studying recursive graphs to understand how they work. Scientists have learned a lot about their properties and behavior. In computer science, we use these graphs to make models of things like trees and lists of information. They're also helpful for showing how complicated systems, like fractals and networks, have repeating patterns. These graphs are used in different areas like machine learning, understanding languages with computers, and more. They're like a tool that helps us study how computer programs work and how hard problems are to solve, giving us useful information in many different areas.

1.2 Project

In our project, we explore the three recursive fractals: Sierpinski triangle, Koch Snowflake, and the Hilbert Curve by utilizing Python Turtle to visualize our C++ program.
We chose recursive graphics because we wanted to explore recursion more in depth. We also enjoyed the visualization aspect of the project where we could creatively implement code. We utilized Python Turtle to visualize our fractals and implemented the recursive algorithms to generate the patterns for the L-system in C++.

2 Methods

2.1 Python Turtle & L-Systems

An L-System, also known as a Lindenmayer system, was named after the Hungarian botanist Aristid Lindenmayer. The Lindenmayer system was created in 1968 as a grammar based system used to model plant growth. More recently, L-systems have been used in computer graphics for the generation of fractals and the realistic modeling of plants. For example, L-systems can be used to provide a set of instructions to Python Turtle. Turtle graphics is an implementation of a geometric drawing tool seen in Python. The "turtle" represents what is essentially a robot with a pen drawing on a sheet of paper.

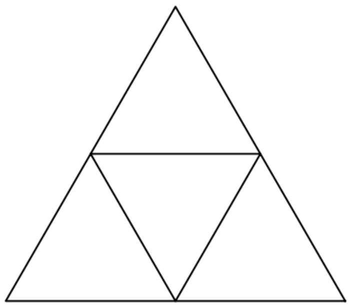
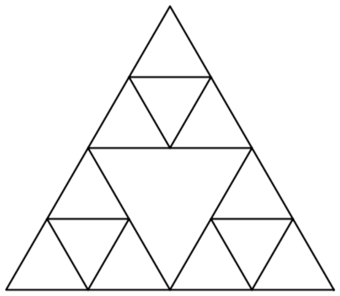
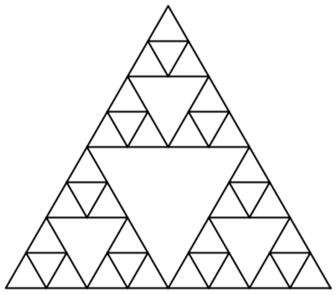
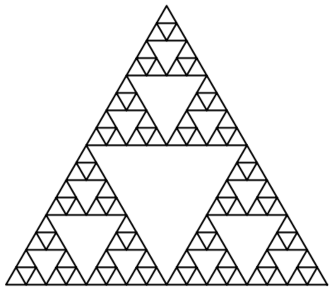
The way L-systems are used to draw fractals is simple: code is used to generate a set of instructions recursively based on the number of generations and then passed into Python turtle in order to be drawn. To be more precise, these are a the set of instructions generated by L-systems that we used in our Python graphics drawing:

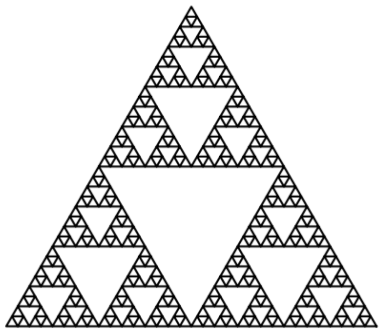
F: Draw a line and move forward
+: Turn right
-: Turn left

2.2 Sierpinski Triangle

The Sierpinski Triangle is a famous fractal pattern named after the Polish mathematician Wacław Sierpiński. It is constructed by recursively dividing an equilateral triangle into smaller equilateral triangles and removing the central triangle at each iteration. The result is a self-replicating pattern with intricate triangular voids.

The Sierpinski Triangle begins with an equilateral triangle, which serves as the starting shape for the fractal process. At each iteration, the triangle is divided into four smaller equilateral triangles. The central triangle of these four is then removed, leaving three equilateral triangles. This process creates the 'voids' or gaps that are characteristic of the Sierpinski Triangle. Each of the remaining triangles is then subjected to the same process of division and removal of the central triangle. This self-similarity at different scales is a hallmark of fractals.

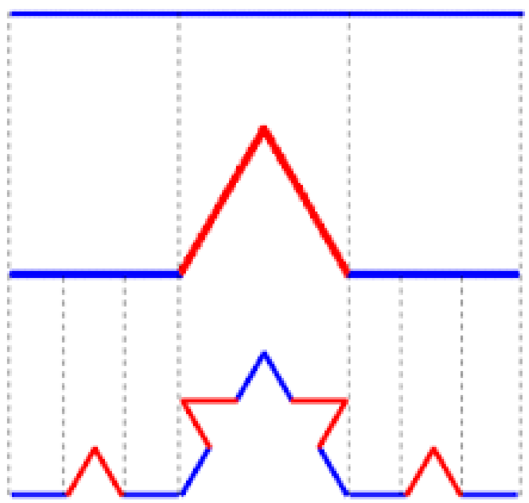
Level 1	Level 2
	
Level 3	**Level 4**
	

Level 5


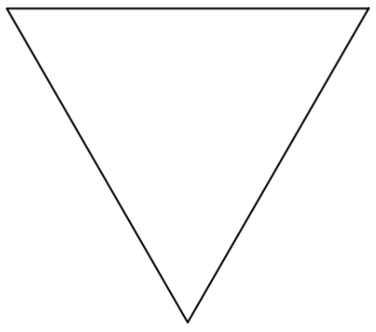
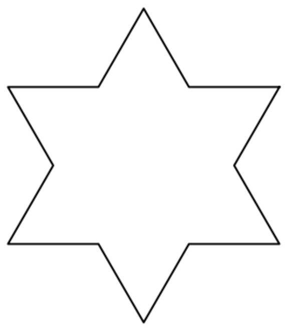
2.3 Koch Snowflake

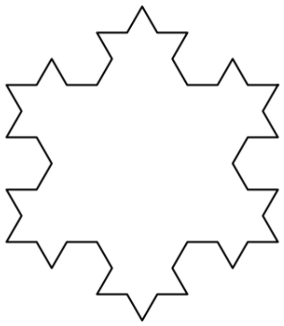
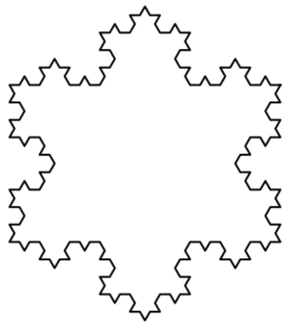
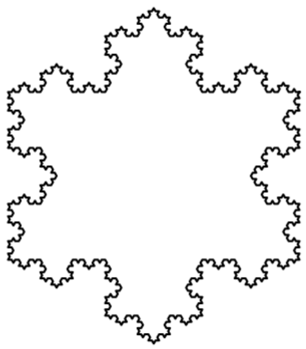
The Koch snowflake is one of the earliest fractals to be described. It was created and named after mathematician Niels Fabian Helge von Koch in 1904. The shape begins as an equilateral triangle and gets more complex as smaller equilateral triangles are added to the edges of the original. The snowflake is created by drawing a smaller shape known as the Koch curve three times at 120 angle from the last until they connect.

Each edge of the shape serves as a base for the Koch curve. The transformation into the Koch curve begins with the line segment being broken up into three equal parts. The middle segment then becomes the base of an outward facing equilateral triangle and the base is then removed. This process is repeated to each line segment during every interaction.



After each edge of the original equilateral triangle has been transformed into a Koch curve to whatever level is specified the shape begins to take on the Koch snowflake shape. The more iterations the more it looks like a snowflake as shown below.

Level 1	Level 2
	

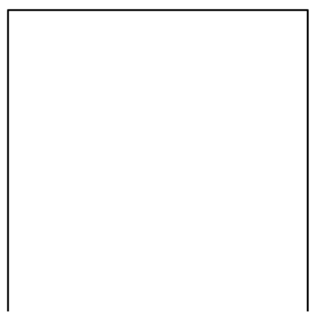
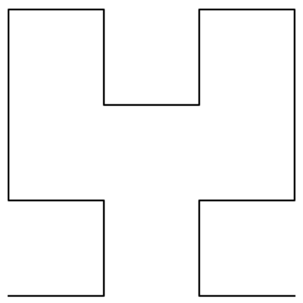
Level 3	Level 4
	
Level 5	
	

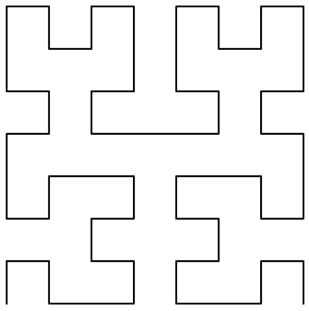
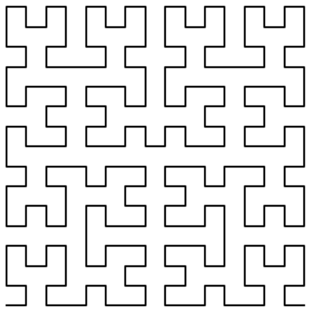
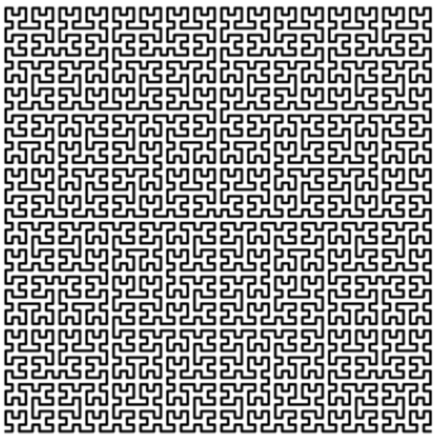
2.4 Hilbert’s Curve

Hilbert's curve, originally proposed by mathematician David Hilbert in 1891, is a space-filling fractal known for its composition of straight lines and 90-degree turns. The fractal fills a 2 x 2 square, forming a single U curve. The initial step involves drawing a line connecting two neighboring quadrants. This process repeats until only one line is missing to complete a full square, resulting in a U shape—the fundamental pattern for all subsequent levels.

To progress further, the basic U curve is turned in different directions and connected with lines. At Level 2, a 4 x 4 matrix is formed, with each quadrant divided into four sub-quadrants. The top two curves have their middle sides facing each other, while the bottom two curves have their middle sides facing downward. Completing the entire curve involves connecting the ends of all four sub-curves. This recursive process continues at each level until all sub-curves merge, creating the full Hilbert Curve and covering all cells in the grid.

The size of each matrix is determined by the desired number of levels, represented by 'n,' resulting in an n x n matrix. Each quadrant contains the basic U curve, splitting each curve into four quadrants and yielding an m x m matrix, where the matrix size 'm' is determined by the formula $m = 2^n$.

Level 1	Level 2
	

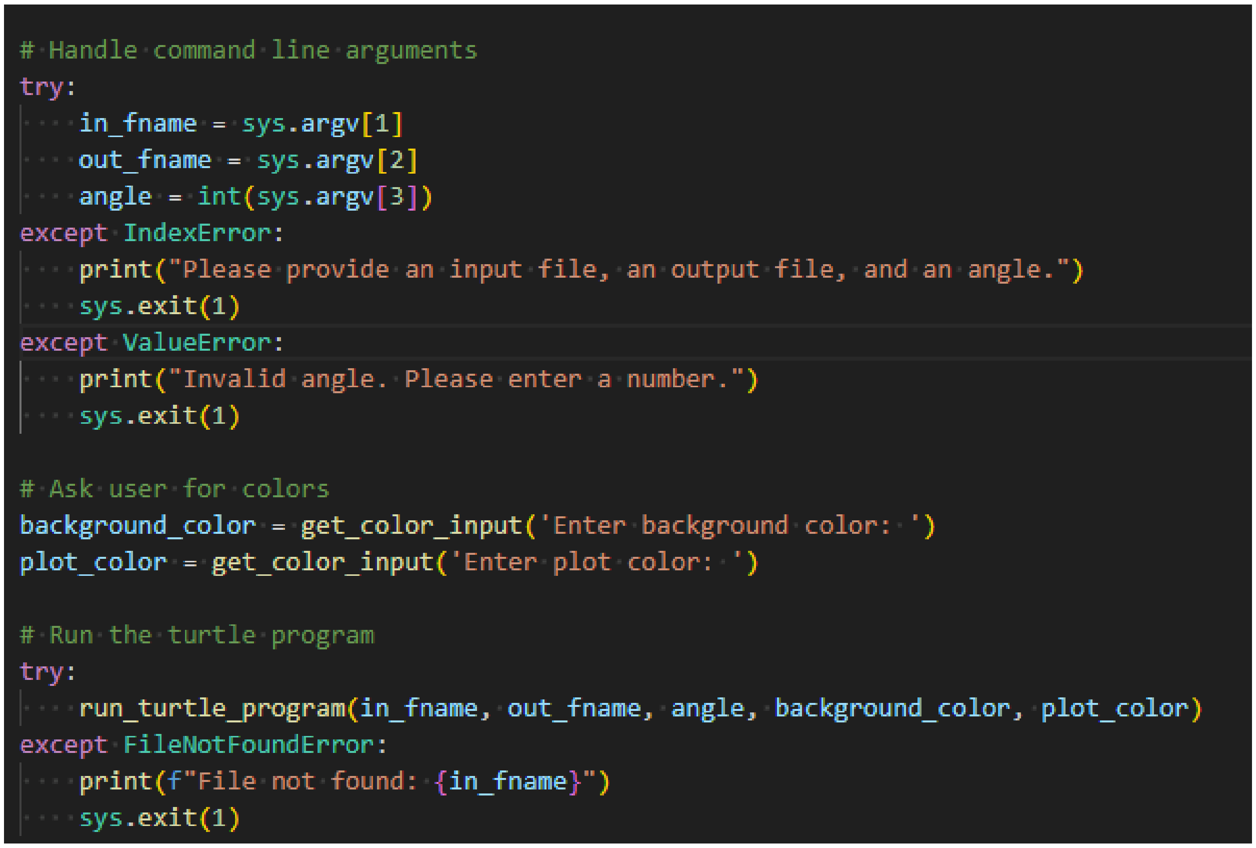
Level 3	Level 4
	
Level 6	
	

3 Implementation

3.1 Python Graphic

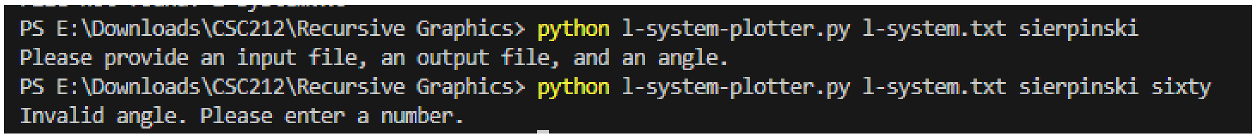
The Python implementation is a very simple program that reads the L-System files generated by the other programs and uses turtle graphics to plot the fractals. The Python code is split into 5 functions: plot_coords, print_coords turtle_to_coords, run_turtle_program, and get_color_input.

During runtime, the python file takes in 3 arguments: the input file name, the name of the output file, and the angle used for the turtle graphics. I added error handling to this part of the code, which ensures that the user provides the 3 arguments at the command line. After validation, these arguments are then passed into the run_turtle_program function, which further validates the input. This part of the code ensures that the user enters a valid file name and angle when running the program.



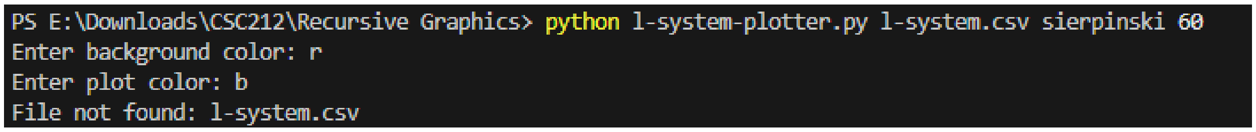
```python
# Handle command line arguments
try:
    in_fname = sys.argv[1]
    out_fname = sys.argv[2]
    angle = int(sys.argv[3])
except IndexError:
    print("Please provide an input file, an output file, and an angle.")
    sys.exit(1)
except ValueError:
    print("Invalid angle. Please enter a number.")
    sys.exit(1)

# Ask user for colors
background_color = get_color_input('Enter background color: ')
plot_color = get_color_input('Enter plot color: ')

# Run the turtle program
try:
    run_turtle_program(in_fname, out_fname, angle, background_color, plot_color)
except FileNotFoundError:
    print(f"File not found: {in_fname}")
    sys.exit(1)
```

This prevents the user from inputting too few arguments, or inputting an angle that isn't an integer. An example of this error handling can be seen below:



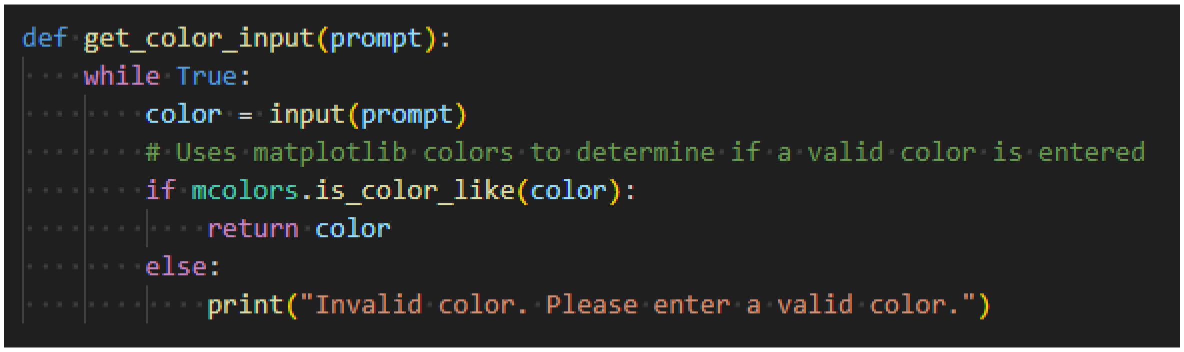
```
PS E:\Downloads\CSC212\Recursive Graphics> python l-system-plotter.py l-system.txt sierpinski
Please provide an input file, an output file, and an angle.
PS E:\Downloads\CSC212\Recursive Graphics> python l-system-plotter.py l-system.txt sierpinski sixty
Invalid angle. Please enter a number.
```

The second part of the error handling also ensures that the user does not attempt to enter an input file that does not exist. For example:



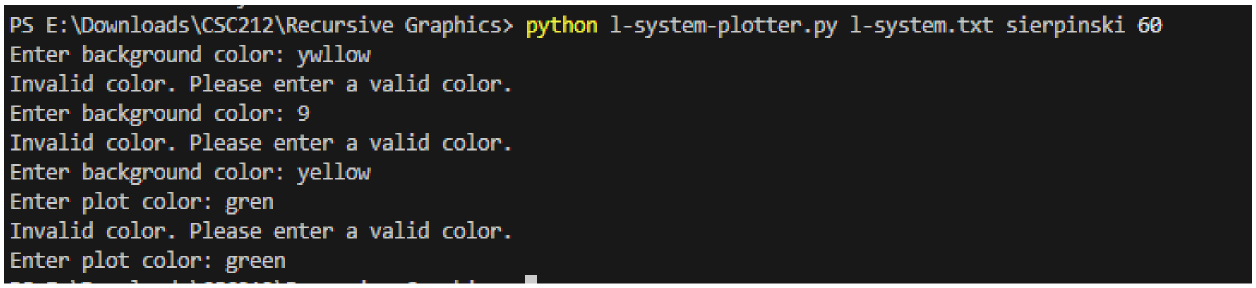
```
PS E:\Downloads\CSC212\Recursive Graphics> python l-system-plotter.py l-system.csv sierpinski 60
Enter background color: r
Enter plot color: b
File not found: l-system.csv
```

Another functionality that I added was the option to pick the color of the graph output, with the get_color_input function working to ensure that a valid color is entered.



```python
def get_color_input(prompt):
    while True:
        color = input(prompt)
        # Uses matplotlib colors to determine if a valid color is entered
        if mcolors.is_color_like(color):
            return color
        else:
            print("Invalid color. Please enter a valid color.")
```

As can be seen below, this prevents the user from entering misspelled colors or numbers that would also cause the program to crash.



```
PS E:\Downloads\CSC212\Recursive Graphics> python l-system-plotter.py l-system.txt sierpinski 60
Enter background color: ywllow
Invalid color. Please enter a valid color.
Enter background color: 9
Invalid color. Please enter a valid color.
Enter background color: yellow
Enter plot color: gren
Invalid color. Please enter a valid color.
Enter plot color: green
```

After the input is validated, the input file name, output file name, angle, background color, and plot color are all passed into a function called run_turtle_program, which opens the file and uses plot_coords and turtle_to_coords to write to an output file.

The plot_coords() function is used to plot a list of coordinates on a matplotlib graph. It first checks if the bare_plot parameter is `True`. If it is, it turns off the axis markers using plt.axis('off'). Then, it sets the background color of the plot to the value specified by the background_color parameter using `plt.gcf().set_facecolor(background_color)`. After that, it ensures that the aspect ratio of the plot is equal using `plt.axes().set_aspect('equal', 'datalim')`.

After the initial graph setup is complete, it converts the list of coordinates into separate lists of X and Y values. These values are then plotted on the graph using `plt.plot(X, Y, plot_color)`, where plot_color is a parameter that specifies the color of the plot. Finally, it saves the plot to a file specified by the fname parameter using `plt.savefig(fname)`.

The print_coords() function prints a list of coordinates to the console. It iterates over the list of coordinates and checks if the x-value of each coordinate is a number. If it's not a number (i.e. it's NaN), it prints '<gap>'. Otherwise, it prints the coordinate in the format '(x, y)', with each value rounded to two decimal places.

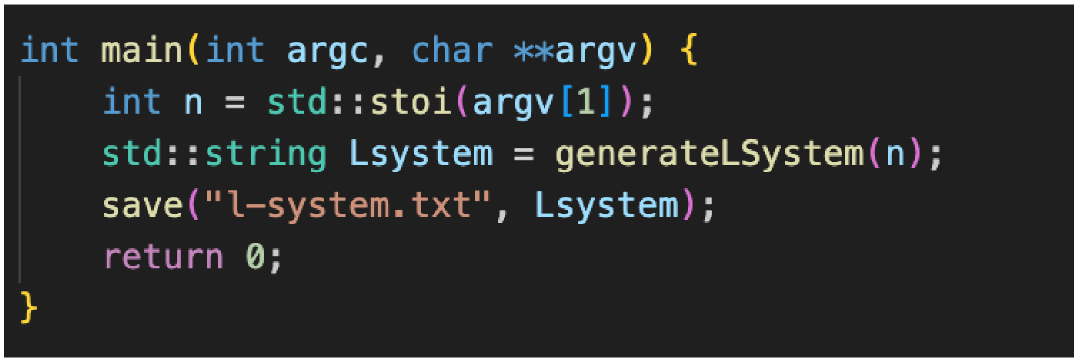
The turtle_to_coords() function implements a turtle graphics program. It takes as input a string of turtle commands and a turn amount. It starts with an initial state of (0.0, 0.0, 0.0) representing the turtle's x-coordinate, y-coordinate, and angle. It then iterates over the turtle program one character at a time, updating the state and yielding the turtle's location based on the command:

- If the command is 'F' or 'B', it moves the turtle forward or backward by adjusting the x and y coordinates. The direction is -1 for 'F' and 1 for 'B'. It then yields the new location.
- If the command is '+', it turns the turtle clockwise by adjusting the angle.
- If the command is '-', it turns the turtle counter-clockwise by adjusting the angle

3.2 Sierpinski

The C++ implementation breaks down into 4 main functions: save, drawTriangle, generateLSystem and main.

1) Main function:



```cpp
int main(int argc, char **argv) {
    int n = std::stoi(argv[1]);
    std::string Lsystem = generateLSystem(n);
    save("l-system.txt", Lsystem);
    return 0;
}
```

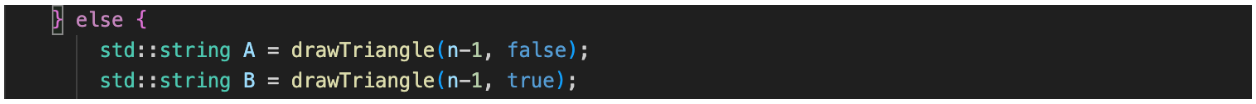
The main function takes in the depth from the user as command-line arguments and stores it in a variable n of typed int. ‘n’ defines the depth of recursion for generating the Sierpinski triangle using an L-system. Specifically, n dictates how many times the recursive function drawTriangle is called within itself, ultimately influencing the complexity and detail of the generated pattern.

It then calls the generateLsystem function that takes in ‘n’ as the argument. This function activates the recursive function drawTriangle and generates a string that represents the L-system pattern for the Sierpinski triangle at the specified recursion depth ‘n’.

The generated L-system string is then saved to a file named “l-system.txt.”

2) DrawTriangle

The function returns the string “F “ at the base case if (n == 0). ‘F’ represents the forward movement in the L-system. The function then calls 2 recursive calls and stores the resulting pattern in string A and B.

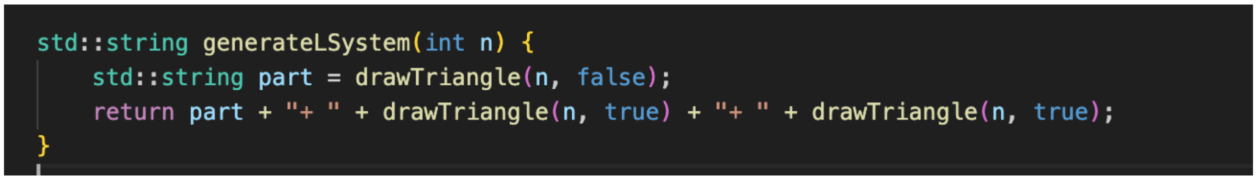


```
} else {
    std::string A = drawTriangle(n-1, false);
    std::string B = drawTriangle(n-1, true);
```

The function calls itself twice, each time decreasing the value of ‘n’ by 1. The gradual decrementing of ‘n’ ensures that the recursion reaches the base case. The function returns two sub-patterns, ‘A’ and ‘B.’

Depending on the value of ‘isSecondPattern,’ the function constructs and returns the different concatenations of ‘A’, ‘B’, ‘+’ and ‘-’ which are the patterns of smaller triangles and organizes them in a way that forms the larger triangle.

3) generateLSystems



```
std::string generateLSystem(int n) {
    std::string part = drawTriangle(n, false);
    return part + "+ " + drawTriangle(n, true) + "+ " + drawTriangle(n, true);
}
```

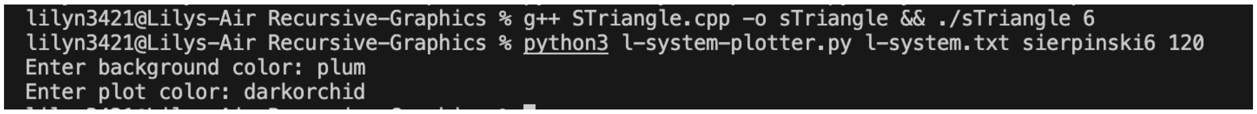
The function takes an integer ‘n’ as its parameter.

The function calls the ‘drawTriangle’ function with the current recursion depth ‘n’ and ‘isSecondPatter’ and sets it to ‘false.’ This generates the base pattern. It then returns the combinations of patterns to generate the final L-system pattern for the Sierpinski triangle.

4) Save

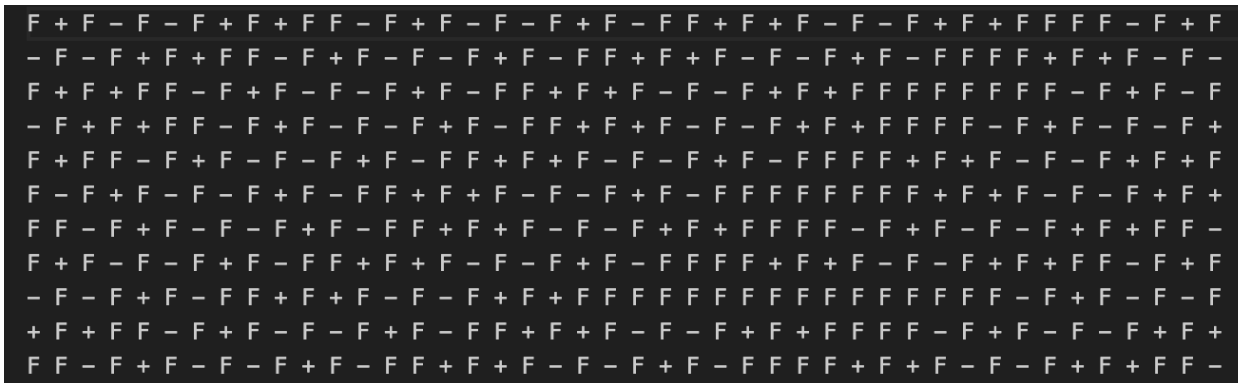
The ‘save’ function saves data into a file and takes in two parameters: ‘filename’ and ‘Lsystem’. ‘Filename’ is a string that specifies the name of the file where the data will be saved. ‘Lsystem’ is the string that will be written to the file, which represents the generated Lindenmayer System for the Sierpinski triangle.

An example of the compile line, input and output for the C++ and python files can be found below.:



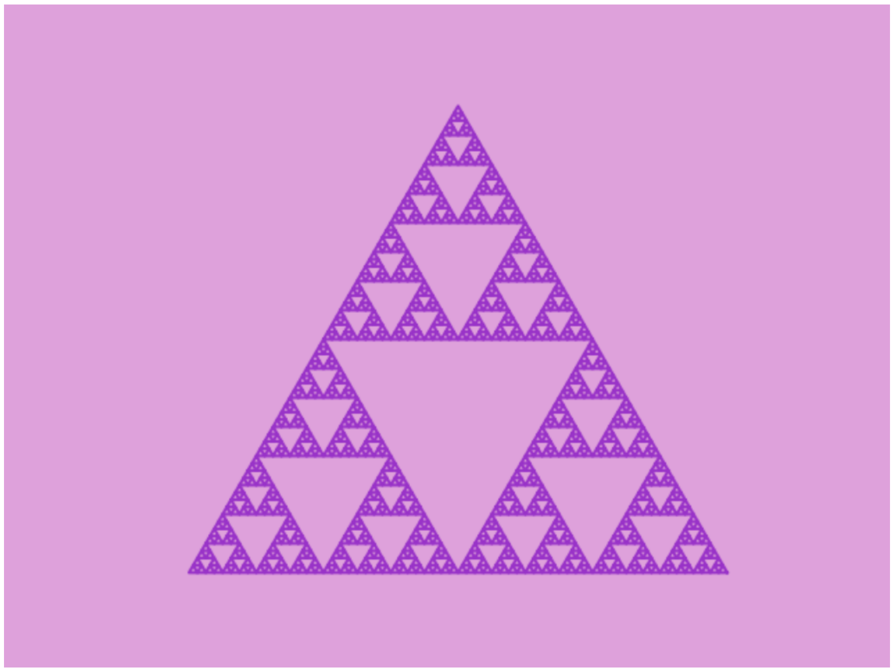
```
lilyn3421@Lilys-Air Recursive-Graphics % g++ STriangle.cpp -o sTriangle && ./sTriangle 6
lilyn3421@Lilys-Air Recursive-Graphics % python3 l-system-plotter.py l-system.txt sierpinski6 120
Enter background color: plum
Enter plot color: darkorchid
```

C++ compile line and C++ and Python command lines to draw triangle with level 6 complexity, “plum” background, and “darkorchid” lines



```
F + F - F - F + F + F F - F + F - F - F + F - F F + F + F - F - F + F + F F F F - F + F
- F - F + F + F F - F + F - F - F + F - F F + F + F - F - F + F - F F F F + F + F - F -
F + F + F F - F + F - F - F + F - F F + F + F - F - F + F + F F F F F F F F - F + F - F
- F + F + F F - F + F - F - F + F - F F + F + F - F - F + F + F F F F - F + F - F - F +
F + F F - F + F - F - F + F - F F + F + F - F - F + F - F F F F + F + F - F - F + F + F
F - F + F - F - F + F - F F + F + F - F - F + F - F F F F F F F F + F + F - F - F + F +
F F - F + F - F - F + F - F F + F + F - F - F + F + F F F F - F + F - F - F + F + F F -
F + F - F - F + F - F F + F + F - F - F + F - F F F F + F + F - F - F + F + F F - F + F
- F - F + F - F F + F + F - F - F + F + F F F F F F F F F F F F F F F - F + F - F - F
+ F + F F - F + F - F - F + F - F F + F + F - F - F + F + F F F F - F + F - F - F + F +
F F - F + F - F - F + F - F F + F + F - F - F + F - F F F F + F + F - F - F + F + F F -
```

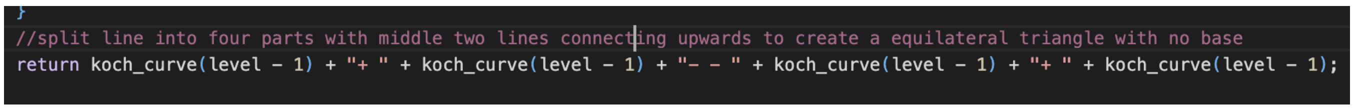
First 60 characters of the l-systems.txt file used to command the python turtle



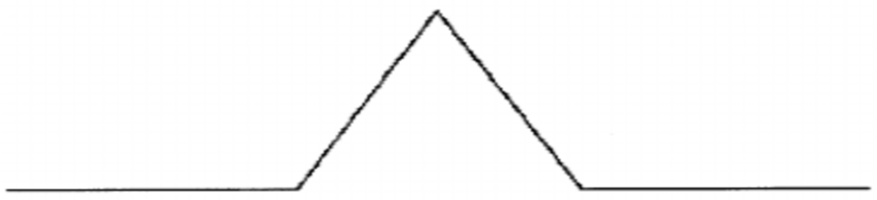
Python turtle output generated Sierpinski Triangle with level 5 complexity

3.2 Koch Snowflake

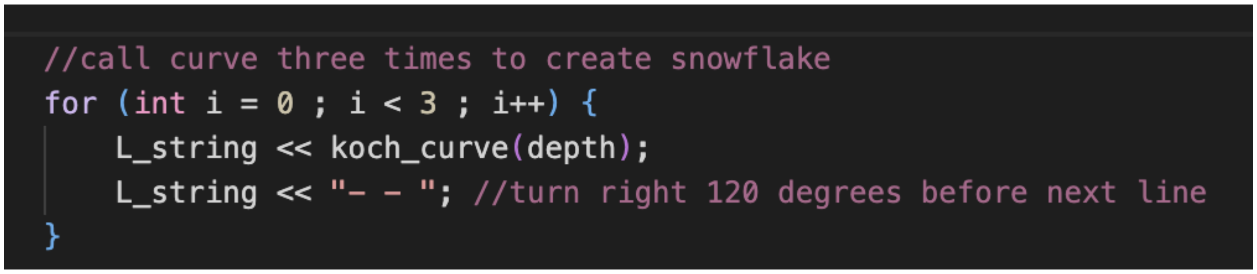
The implementation of Koch snowflake can be broken down into 2 simple functions: koch_curve and the main. The koch_curve is a recursive function of string data type that returns a single Koch curve. The function takes in an integer, “level”, that represents the level of complexity/number of iterations that will be applied to the segment. The base case (level == 1) returns the string "F ", which in the lindenmeier system represents moving the python turtle forward creating a line segment. For all levels imputed that are larger than one, the function returns a recursive call to itself that concatenates forward movements and left and right turns using “+” and “-” respectively. This sequence of the recursive calls and turns applies the Koch curve pattern to each segment.



```
}
//split line into four parts with middle two lines connecting upwards to create a equilateral triangle with no base
return koch_curve(level - 1) + "+ " + koch_curve(level - 1) + "- - " + koch_curve(level - 1) + "+ " + koch_curve(level - 1);
```

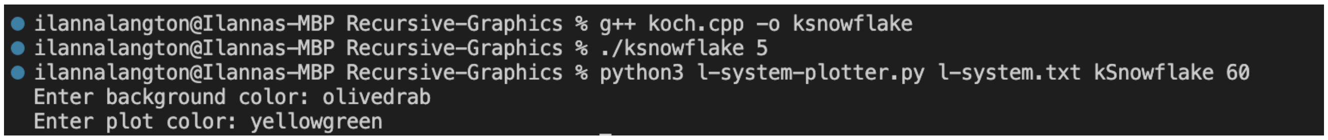


The call to the koch_curve function is contained in the main. It is within a for loop that will run three times, once for each edge of the original equilateral triangle. The string that is returned from the koch_curve function is written to the “l-system.txt” output file using the fstream library. In addition to the function call the loop will also output “- - ” to the text file. This accounts for the 120 degree right turn in between each curve in order to maintain the triangle shape.



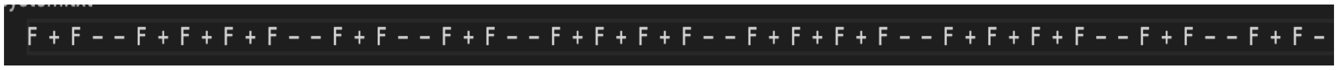
```
//call curve three times to create snowflake
for (int i = 0 ; i < 3 ; i++) {
    L_string << koch_curve(depth);
    L_string << "- - "; //turn right 120 degrees before next line
}
```

In addition to the for loop the main function takes in the level of recursion as a command-line argument (argv[1]), which, as stated earlier, determines the complexity of the generated Koch curve. The code also includes exception handling that prevents a level less than one from being imputed into the function. An example of the compile line, input and output for the C++ and python files can be found below.



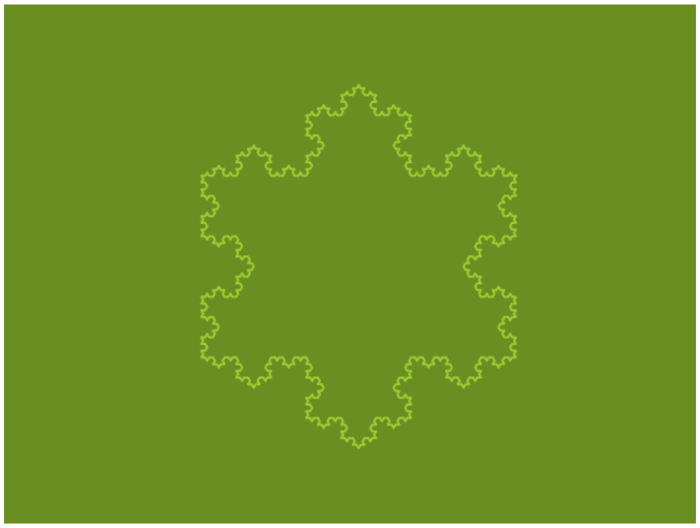
```
● ilannalangton@Ilannas-MBP Recursive-Graphics % g++ koch.cpp -o ksnowflake
● ilannalangton@Ilannas-MBP Recursive-Graphics % ./ksnowflake 5
● ilannalangton@Ilannas-MBP Recursive-Graphics % python3 l-system-plotter.py l-system.txt kSnowflake 60
  Enter background color: olivedrab
  Enter plot color: yellowgreen
```

C++ compile line and C++ and Python command lines to draw snowflake with level 5 complexity, “olivedrab” background, and “yellowgreen” lines



```
F + F - - F + F + F + F - - F + F - - F + F - - F + F + F + F - - F + F + F + F - - F + F + F + F - - F + F - - F + F -
```

First 60 characters of the l-systems.txt file used to command the python turtle

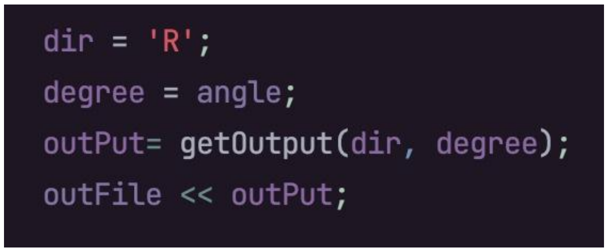


Python turtle generated Koch Snowflake with level 5 complexity, “olivedrab” background, and “yellowgreen” lines

3.3 Hilbert Curve

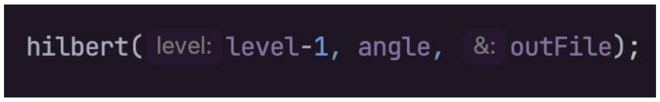
To implement Hilbert's Curve, there are three key functions.

The first, "hilbert," is the engine that generates the curve using recursion. It takes the level of recursion, angle, and an output file reference. If the recursion level is 0, it does nothing. Inside, it uses variables to determine direction, angle, and the character for the L-system, passed to the "getOutput" function.



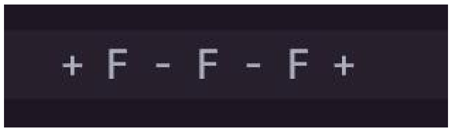
```
dir = 'R';
degree = angle;
outPut= getOutput(dir, degree);
outFile << outPut;
```

Example of what it looks like to set and push the variables into the getOutput function.



```
hilbert( level: level-1, angle,  &: outFile);
```

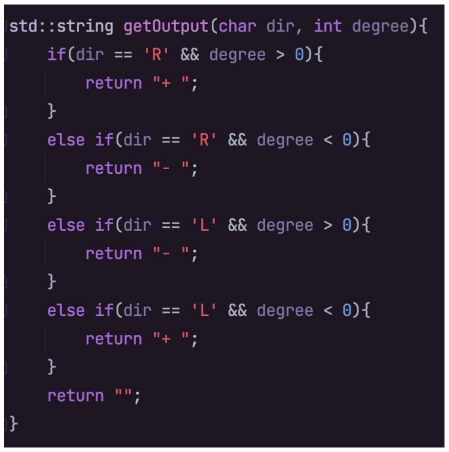
Example of recursive call in the hilbert function. All of them take in level-1, either positive angle or negative, and the same outFile that was taken in as a parameter.



```
+ F - F - F +
```

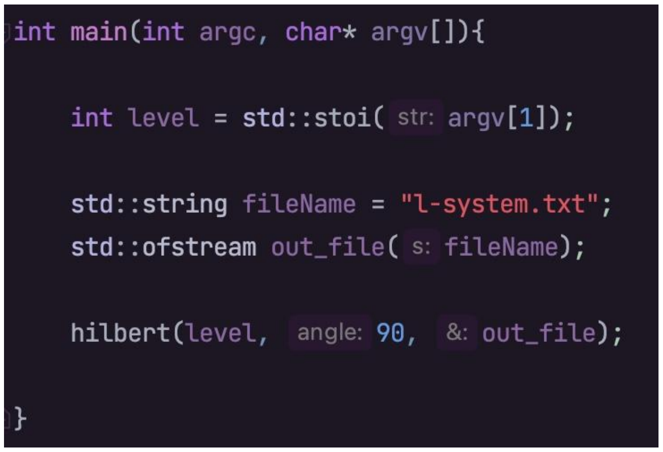
L-system.txt when the level = 1

The "getOutput" function translates direction and angle into L-system code. It checks for 'R' or 'L' and whether the angle is negative, then returns the corresponding L-system letter.


```
std::string getOutput(char dir, int degree){
    if(dir == 'R' && degree > 0){
        return "+ ";
    }
    else if(dir == 'R' && degree < 0){
        return "- ";
    }
    else if(dir == 'L' && degree > 0){
        return "- ";
    }
    else if(dir == 'L' && degree < 0){
        return "+ ";
    }
    return "";
}
```

getOutput function

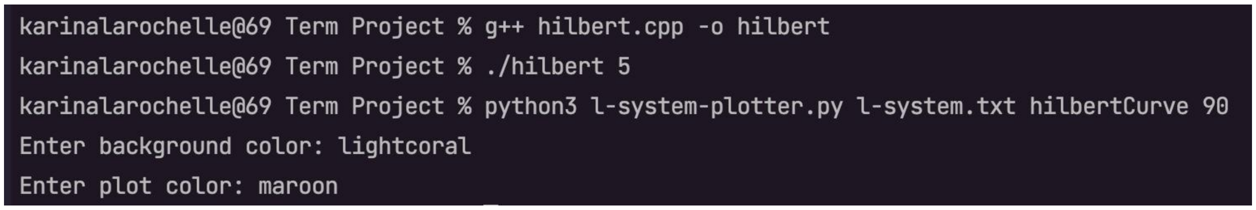
In the "main" function the rules are set for generating Hilbert's Curve. It takes a command-line argument for recursion level, writes the curve to "l-system.txt," and uses a hardcoded angle of 90 degrees in the "hilbert" function.


```
int main(int argc, char* argv[]){

    int level = std::stoi( str: argv[1]);

    std::string fileName = "l-system.txt";
    std::ofstream out_file( s: fileName);

    hilbert(level,  angle: 90,  &: out_file);

}
```

Looking at the code structure, iostream is included for basic interaction and fstream for file output. The "generateHilbert" function progressively builds the curve, creating the basic U shape at level 1. The "getOutput" function assists in translating directions and angles into L-system code.

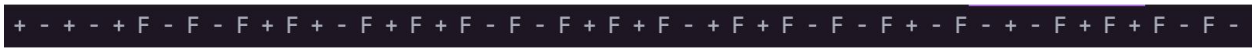
In simpler terms, the code uses these functions to draw Hilbert's Curve by breaking it into smaller pieces and connecting them. The main function sets the rules, and the output is saved in a file. The angle and direction play a crucial role in shaping the curve.

Calling this cpp file looks like and returns the following:



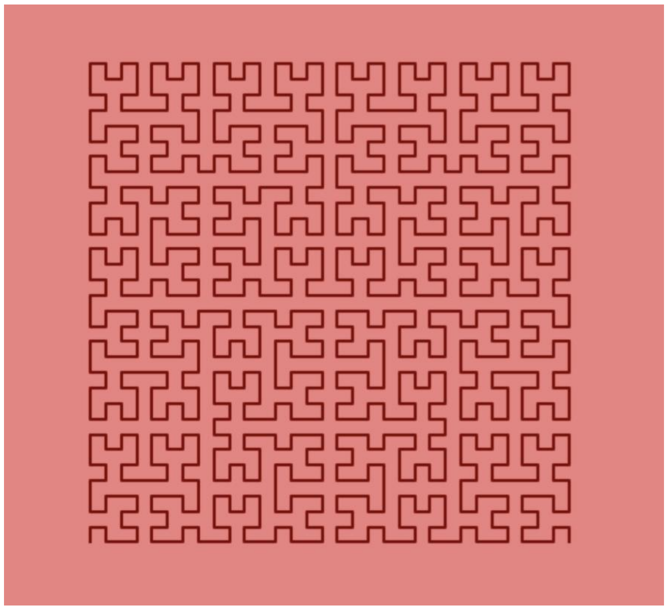
```
karinalarochelle@69 Term Project % g++ hilbert.cpp -o hilbert
karinalarochelle@69 Term Project % ./hilbert 5
karinalarochelle@69 Term Project % python3 l-system-plotter.py l-system.txt hilbertCurve 90
Enter background color: lightcoral
Enter plot color: maroon
```

C++ compile line and C++ and Python command lines to draw curve with level 5 complexity, “lightcoral” background, and “maroon” lines



```
+ - + - + F - F - F + F + - F + F + F - F - F + F + F - + F + F - F - F + - F - + - F + F + F - F -
```

First 50 characters of the l-system.txt when level = 5



Python turtle generated Koch Snowflake with level 5 complexity, “olivedrab” background, and “yellowgreen” lines

4 Conclusions

As we wrap up our project on recursive fractals, it's clear how combining simple rules and shapes can create incredibly intricate and beautiful patterns. By focusing on the Sierpinski Triangle, Koch Snowflake, and Hilbert Curve, we've seen firsthand how a basic process like repeating a shape over and over can lead to complex and stunning designs. Our use of Python Turtle and C++ to draw these fractals has been a great way to show the power of recursion in a visually engaging way. This project has been about more than just making pretty patterns; it's been a journey into the heart of recursion, revealing how simple steps can build up to something cool. It's a reminder of how combining different areas like math, computer science, and art can lead to fascinating discoveries and creations.

We can see how recursive graphics principles can be used in more advanced computer graphics and animation. Recursive fractals could be used in creating new kinds of visual effects in movies or video games, for example. Additionally, the visual and engaging nature of fractals, combined with their mathematical foundation, makes them excellent educational tools. They can be used to teach concepts in mathematics, computer science, and art, making learning more interactive and intuitive.

5 Contributions

Name	Contribution %	Work Completed
Lily Nguyen	25%	● Sierpinski cpp ● Sierpinski Images ● Sierpinski Method section of Paper ● Sierpinski Implementation of Paper ● Sierpinski Part of ReadMe ● Wrote Sierpinski Bullet Points for Presentation ● Conclusion
Ilanna Langton	25%	● Koch cpp ● Koch Images ● Koch Method section of Paper ● Koch Implementation of Paper ● Koch Part of ReadMe ● Wrote Koch Bullet Points for Presentation ● Wrote project introduction
Karina Larochelle	25%	● Hilbert cpp ● Hilbert Images ● Hilbert Method Section of Paper ● Hilbert Implementation of Paper ● Hilbert Part of ReadMe ● Wrote Hilbert Bullet Points for Presentation ● Wrote topic introduction
Jenny You	25%	● L-System-Plotter.py ○ Error handling ○ Color choice ● L-System/Python Method of Paper ● L-System/Python Implementation of Paper ● L-System/Python part of ReadMe ● Put together majority of slide show and graphics

6 Sources

Bibliography

“Hilbert Curves.” *Hilbert Curves*, datagenetics.com/blog/march22013/index.html.

“An Introduction to Lindenmayer Systems.” *An Introduction to Lindenmayer Systems*, www1.biologie.uni-hamburg.de/b-online/e28_3/lsys.html.

“Koch Snowflake.” *Koch Snowflake - Go Figure Math*, gofiguremath.org/fractals/koch-snowflake/.

“List of Named Colors#.” *List of Named Colors - Matplotlib 3.8.2 Documentation*, matplotlib.org/stable/gallery/color/named_colors.html#sphx-glr-gallery-color-named-colors-py. Accessed 04 Dec. 2023.

“The Nature of Code.” *The Nature of Code*, natureofcode.com/book/chapter-8-fractals/#chapter08_section6.

“The Sierpinski Triangle – Fractals.” *Mathigon*, mathigon.org/course/fractals/sierpinski.

“Turtle - Turtle Graphics.” *Python Documentation*, docs.python.org/3/library/turtle.html.