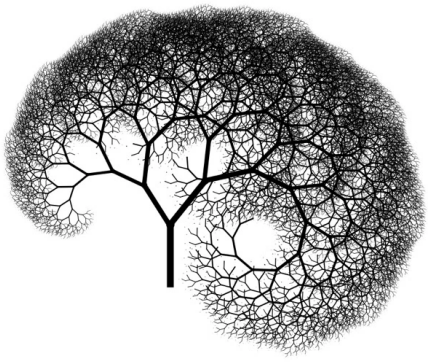
return 1;

University of Rhode Island

College of Arts and Science

Recursive Graphics



By: Zach Albanese, Alex Bergeron, Stephen Brown and Kyle Nadeau

Submission: 24 April, 2023

Table Of Contents

Introduction

The following document will act as a report, documenting the topic chosen for our DSA project completed during the Spring semester of 2023. This document will include a range of information including all topics, methodologies and reasonings behind the implementation of the project.

Topic

Our chosen topic was Recursive Graphics, a sect of computer graphics that uses recurring function calls to generate imagery. These fractal patterns are defined by two main properties, a base case and a recursive step. The base case of a recursive function is the place at which a “final condition” is met, thus ending recursive calls and beginning the unwinding of the function. Base cases for the recursive algorithms we studied either returned empty strings or a small set of plotting instructions to the previous call. Generally the base case is better to construct first, as it is the point at which your recursion begins to unwind, returning values until some final output is gathered. Recursive steps in terms of graphics is where a function’s base case is not met, continuing to the body and passing a set of values to itself until the base case is satisfied. Another term we may use throughout this paper is Lindenmayer or L-Systems, which are commonly used for fractal-like art, and consist of recursive string re-writing frameworks, similarly to the one we utilize later on.

Context

When deciding upon the topic for this project we wanted to choose one that the class and CSC212 Staff would be interested in- not only computationally but visually. Speaking for the group, recursion can be challenging, with complex rules, however there is something innately beautiful about the patterns in which they follow. This is illustrated perfectly by recursive graphics, seeing how the magnitude of the function can alter the output, where small patterns constitute a larger tapestry. For our project we decided to make a User

Interface in C++, similar to the one we made for our Review Project (that was quite unbreakable,) with a few different rules. First, we wanted the user to be able to run the program and choose which algorithm they would like to create a graphic for. There are eight total algorithms we implemented, including Sierpinski's Triangle, Snowflake Fractals, Hilbert I & II Curve, Dragon Curve, Square Curve, Peano Curve and a custom algorithm we made with a special twist called Schrader's Symmetry. After choosing the number that correlates with a specific algorithm you are prompted to input the degree of magnitude of the function. In many cases these have a concrete upper and lower limit, as some functions may grow exponentially, and we do not have infinite bandwidth or memory. After submitting an input the program will run the function, and output a file that we can then plot using a python script, ending with a graph.

Project

The project section will outline the planning surrounding our project after the topic was initially chosen, this will include the general structure and software utilized.

Planning

After deciding upon Recursive Graphics as our topic we met to discuss the most suitable way to carry out the project. During our first meeting we decided that the program was the first order of business, and that we should divide the required algorithms among us, to read up on before starting any actual programming. The successive meetings were our whole group (or whoever did not have class,) meeting up to work on the GUI and implementing the functions. One highly important decision we had to make was whether or not we should use classes for each algorithm. Overall we decided it would be inefficient; due to the design of our script it would only run once, and if there is only one instantiation of an object it defeats the purpose of having classes, in which multiple objects of a user specified type are created. There are also no member attributes or methods that we would

be using on these objects, as the only function each algorithm needs to complete is generating a text file to be used in our Python plotting script.

Software

Due to the nature of this topic we will be utilizing multiple languages in order to properly compute as well as graph the different algorithms that are implemented within our program. The heart of our project is 'main.cpp,' a C++ script that implements all of the algorithms, in an easily accessible User Interface. Upon running the user is prompted to enter which graphic they wish to use, and the order for the graphic. After this, a file is created within your working directory, called "file", which contains the directions for the Python Plotting program to graph your fractal. The python program, plotter.py can then be run, which will complete with a png image being downloaded to the same location in your working directory, if you open the image you will see your specified fractal. In addition, we created a coloringbook.py application that utilizes Python's Pillow Graphics library, as well as tkinter, a popular package used for designing GUI's. If you choose to run this, you are able to upload your fractal and color it in (to relieve all the stress of teaching... or learning CSC212!)

Compilation

If you are using *CLION*:

For main.cpp, add to cmakelists in your current working directory, and run the code. There are no needed CLA's, the only input will be within the terminal by the user of the program, the output will consist of a file within the users working directory named "file," This contains the instructions the python program will use to map / plot the functions.

For the plotter.py file, we will create a new run configuration, with the working directory set to the file path, for example mine is C:\Users\kylen\CLionProjects\DSA\ for the working directory and C:\Users\kylen\CLionProjects\DSA\plotter.py as the script path.

```
return 6;
```

Next, specify the file paths as arguments for your input and output file, as well as a degree at which the algorithm will plot at. It will consist of the input file, the output png image and degrees as seen below. (Apologies, small font so it is on one line.)

```
C:\Users\kylen\CLionProjects\DSA\cmake-build-debug\file  C:\Users\kylen\CLionProjects\DSA\cmake-build-debug\image.png 90
```

After this, run the program and view the png output.

If you are using *G++:*
For the main.cpp, simply open your g++ compiler and use:

```
g++ main.cpp -o main && ./main
```

This will result in an output file named "file" within the same directory as the main.cpp file. For the python file navigate to the local file using a terminal and use:

```
python3 plotter.py file.txt image.png <degree for turns>
```

If you are using *GCC:*
For the main.cpp, simply open your gcc compiler and use:

```
gcc main.cpp -o main
```

This will result in an output file named "file" within the same directory as the main.cpp file. Run the same python script as above via terminal to execute the plotter.py

Methods

The following section will outline each graphic chosen to be represented within our project, describing why utilizing recursive functions are used for each case, as well as a bit of history behind each algorithm. All pictures are generated via our code.

Python Plotter

The python plotter we are utilizing within our project was provided during Week 6's lab on recursion. The program uses matplotlib as its main graphing library, taking three program arguments. The first argument is the text file that is outputted from running our main program, which consists of a string representing rules for printing a graphic using L-Systems. The second argument is the output file name, for example "hilbert.png," and the last is the degrees at which the turns will abide by. An example of the full input may be "file.txt image.png 90" as seen mirroring the compilation instructions above. It is important to note that the whole file path may be required depending where you save your original text file to, as well as what your current working directory is. The actual functionality is described using the turtle analogy, where he follows a set of instructions in order to create your graphic. For example, if he was passed the input "F + F - F F," he would start at (0,0) on the graph facing right, move forward 1 unit, then turn right some degree amount, forward again, left by some degree amount, and finally forward twice. The output is then graphed using matplot, and output as a png image.

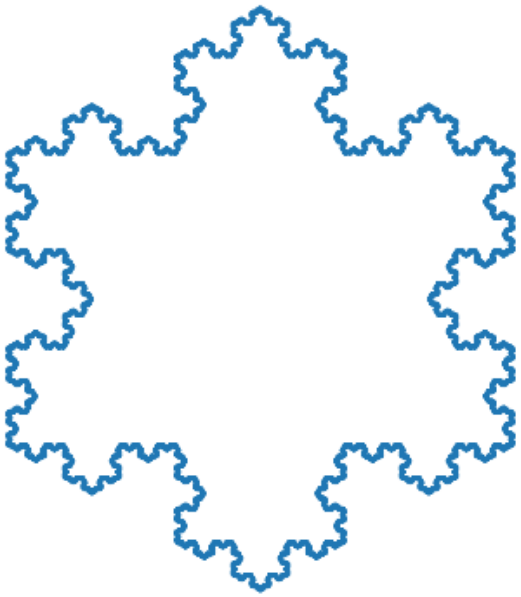
Koch's Snowflake

Koch's snowflake is a fractal that involves a series of triangles nested within one another to create a fractal that can appear to be similar to a snowflake. The premise involved in generating Koch's snowflake is that you take a straight line, break it into three parts, and then get rid of the straight line in the center part and replace it with two sides of a triangle of equal length, creating an arch. This is done multiple times to eventually create a shape that has multiple similar points that gets more complex as the snowflake goes through iterations. As such, every iteration of the snowflake has not only three identical sides, but also has the general pattern of the previous iteration of the snowflake contained within it,

return 8;

up to the base case iteration (which is just a triangle). Naturally, this means that recursion is the method of choice when attempting to generate a Koch snowflake, as recursion allows for a fairly consistent pattern to generate a consistent result. As for how the recursive function itself should generate one side of the snowflake by creating directions, it should have a base case that returns a forward movement, as well as three recursive calls outside of it, all of which should decrease by some amount that will lead to the base case, and have an angle change after each one; which should be a 60 degree turn left, a 120 degree turn right, and another 60 degree turn left. Finally, there should be one more recursive call that decreases by an amount that will eventually lead to the base case, before the program has a return at the end. .

Fig 1. 8th order Koch Snowflake



Sierpinski Triangle

The Sierpinski Triangle is a fractal pattern that is named after the Polish mathematician Waclaw Sierpinski. To create this shape, you start with a single equilateral triangle. Then, you divide it into four smaller equilateral triangles by connecting the midpoints of each side. Next, you remove the middle triangle, leaving three triangles. You repeat this process recursively with each of the remaining triangles, dividing them into smaller triangles, and removing the middle triangle. You can continue this process infinitely, creating an increasingly complex pattern of triangles. An interesting fact about the Sierpinski Triangle is that it has an infinite perimeter but a finite area. This fractal pattern, like many others, can be used in the study of “chaos theory” by demonstrating the butterfly effect - a small change in the initial conditions can lead to drastic changes over time as the pattern grows.

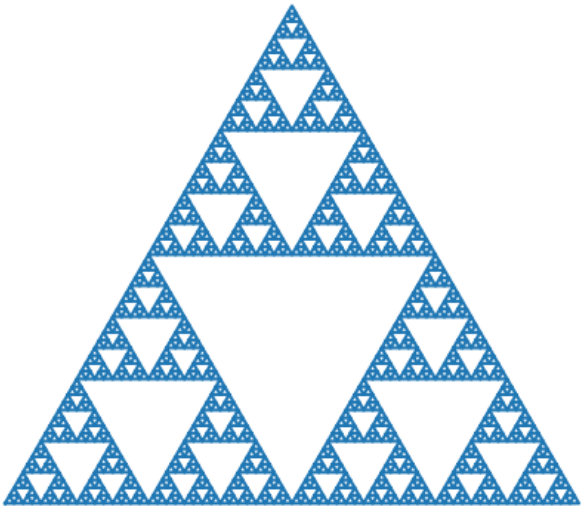


Fig 2. 6th order Sierpinski Triangle

Hilbert Curve

The Hilbert Curve is a continuous fractal space filling curve first described by the German mathematician David Hilbert during the late 19th Century, and is a variant of a later described algorithm the Peano Curve. The geometry of the Hilbert curve begins with its first order, represented by three congruent lines, intersecting at perpendicular angles- basically, it looks like a square that is missing a singular side. The hilbert curve, like other algorithms described within this section grows exponentially with order, with the length being equal to 2^n - $\frac{1}{2^n}$. As the magnitude of the Hilbert curve increases, it may be noted that four of the previously used order's structures constitute the current, being either the inverse or rotated version, connected by a "bridge." Many common image and video graphics programs utilize the hilbert curve for ray tracing and rendering. It also has applications within image compression and dithering where the state of adjacent pixels is important.

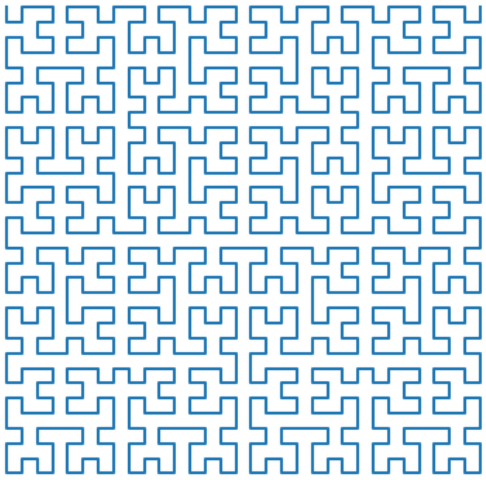


Fig 3. 5th order Hilbert Curve

Dragon Curve

A dragon curve is any member of a family of fractal curves, such as the Heighway dragon. It self-replicates, meaning that it can be scaled and rotated to create larger or smaller copies of itself.

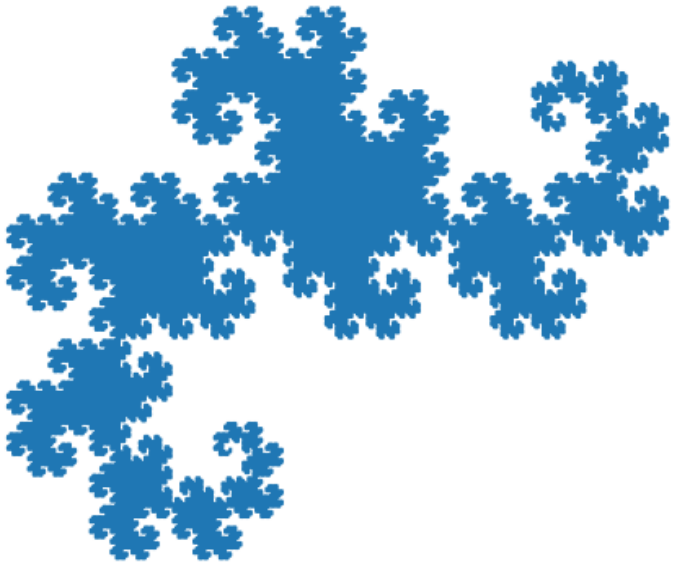
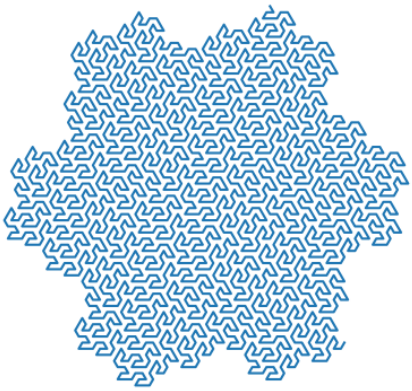


Fig 4. 15th order Dragon Curve

Peano-Gosper Curve

The Peano-Gosper curve is named after mathematician Bill Gosper and Giuseppe Peano. It is another fractal curve that resides in the same family grouping as the other self-similar curves described above.

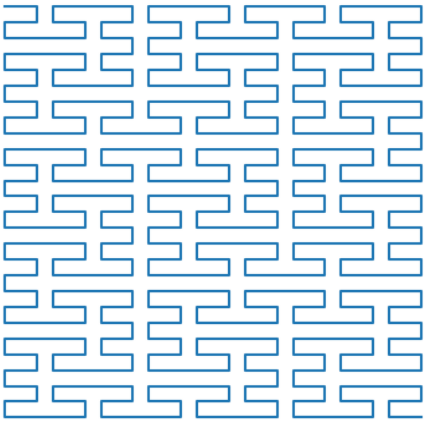
Fig 5. 4th order Peano-Gosper Curve



Hilbert Curve II

This fractal pattern is a variant of the original Hilbert Curve, however instead of starting with a U shape, it begins with an S shape. This emphasizes the butterfly effect found in fractal geometry, as the end result is an entirely different shape with a slight change to the initial conditions.

Fig 6. 3rd order Hilbert Curve II



Square Curve

The square curve, otherwise known as the sierpinski square is a triangular or rhombus shaped structure, consisting of many step shaped fractals. The base case consists of what looks like a three leveled pyramid, symmetrical on both sides. You can see this by looking at the bottom left corner of the square curve seen below.

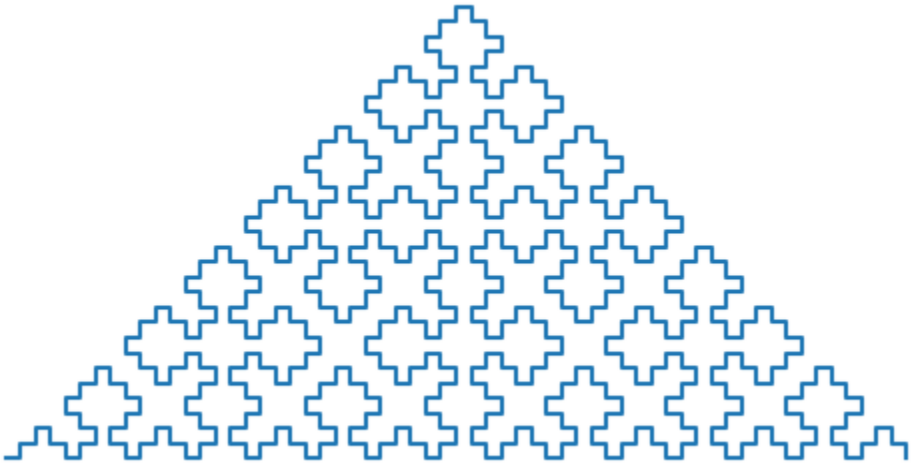
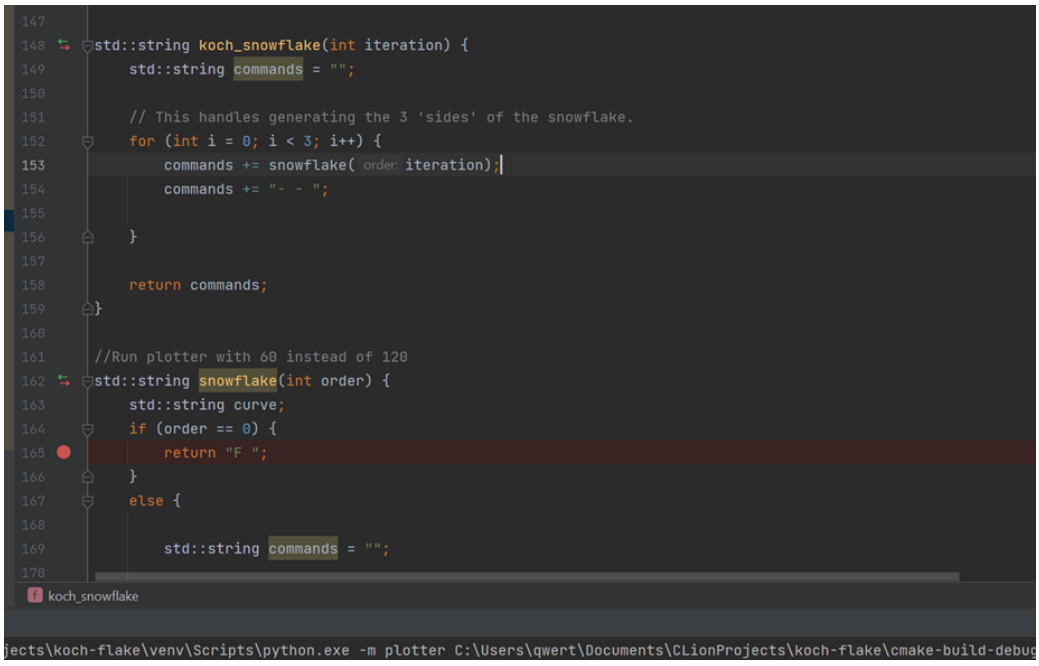


Figure 7. 4th order Square Curve

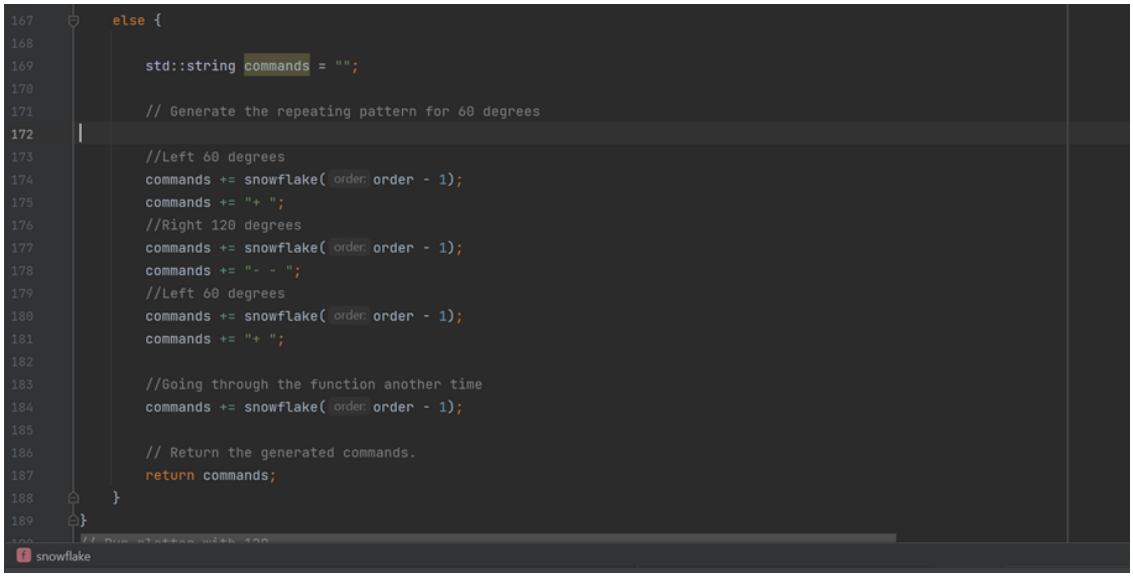
Implementations

Koch's Snowflake

Our implementation for Koch's Snowflake involved having an outside function call the recursive function three times using a for loop, which then adds two negatives to the string before iterating (which stands for a 120-right degree). The reason I implemented it like this is that each pattern generated by the program naturally has three sides that are almost completely identical, with the only differences being that they are rotated by different angles. This means that one call to the recursive function should generate one side and rotate before generating the next side using the same methodology, as a result of the sides being identical. As can be seen in the koch_snowflake function in the following code:

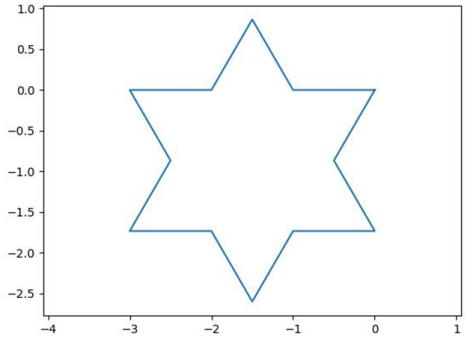


The base case is “F “since that should always be the end point of the recursive call for any iteration in order to ensure proper results. To show why this needs to be the base case, it is important to remember that the base “iteration” of Koch’s snowflake is a simple triangle. In this code, that would be “F - - F - - F - - “, which would generate upon running the program and passing in a variable with the value of 0 to the function koch_snowflake. The next few recursive calls, which can be seen here:



, are calls that generate the turns required to make a triangle(as a koch snowflake is essentially a series of nested triangles), and one additional recursive call that goes through the directions for the koch snowflake of a magnitude below it, something that is critical to the functioning of a koch snowflake as each iteration builds off the previous iterations. To prove that my code is correct, then we can consider the scenario in which the snowflake recursive function did not have just “F” as the base case, then it would run whatever else is in the base case, which more than likely, would not generate the proper triangle. In the case that we decide to have not just “F” as the base case, but also the “- -“

from the outside calling loop, since it also generates a basic triangle, we need only to consider the next iteration of the Koch's snowflake, a six pointed star, to see what the problem with that iteration would be. The proper string of that iteration would be "F + F - - F + F - - F + F - - F + F - - F + F - - F + F - -", but in this new scenario, the recursive function would be called, skip the base case since the value being passed in would be higher than the one needed to execute the base case, and run the next few lines of code, which would result in each turn calling the function again, and getting the base case "F - -" , before the function would then add either a "+" or a "- -" in order to make a turn. In this scenario, not only would a left degree turn of 60 degrees would no longer be possible, since the "- -" would negate it completely, but also a right degree turn of 120 degrees would become a right degree turn of 240 degrees. As such, my base case as well as my recursive function calls are the correct ones, since they not only generate the first iteration of Koch's Snowflake(the triangle), but also the second iteration of Koch's Snowflake(the six sided star) as well as those beyond it. To prove that, here is the output when that string is put into the python plotter:



Which proves that my recursive function and iterative function can properly generate the L-system output needed for the second iteration of a Koch Snowflake, and that it is capable of generating subsequent iterations as well.

Sierpinski Triangle

The implementation of this graphic involved the use of two functions, one to generate the pattern, and another to close the edges of the triangle afterwards. The L-system formula involves a base case of “F”, and the recursive cases that include two patterns. We used a boolean parameter to determine the pattern that would be returned. The overarching L-system structure is: “A-B+A+B-A”, where A is the first pattern and B is the second pattern. In the base case, this results in a single “F”, and then the closing formula to create a singular triangle. It next turns into “F-FF+F+FF-F”. The 3rd order of magnitude is where it gets more complex, as each F is replaced with either pattern A or pattern B. The closing function appends a “-” and 2 to the nth power number of “F”s, where n is the order of magnitude of the graphic being created. It does this twice to close the two open sides of the triangle. To visualize this fractal, we run the python plotter with 120 degrees in the CLAs.

Hilbert Curve

The Hilbert curve uses a single function to generate the L-system. This function utilizes this L-system formula: L -> +RF-LFL-FR+, R -> -LF+RFR+FL- where L denotes the first pattern and L denotes the second pattern, and F represents an F that is being placed into the text file, as the base case returns an empty string in this function. One noteworthy aspect of this formula is that the patterns are inverses of each other.

Other Graphics

We implemented various other graphics such as Dragon Curve, Peano-Gosper Curve, Square Curve, and the variant Hilbert Curve. These extra graphics were fairly easy to create after we came to understand the L-System formula by creating the first three fractal patterns. Using the well documented formulas and substituting them with a recursive function enabled us to show a deeper understanding of how recursive graphics work.

Contributions

Contributor	Code	Report	Group
Kyle Nadeau	Hilbert Curve, Square Curve, Python coloring script	Styling, Introduction, Project	Scheduling & Research
Zach Albanese	Sierpinski Triangle, Dragon Curve, Peano-Gosper, Hilbert Curve II	Background, Methodology	Research
Stephen Brown	Koch Snowflake, UI	Implementations, Methodology	Research
Alex Bergeron	UI, Comments, Spacing for code readability	Styling Presentation Slides information and organization	Research

Conclusion

Fractal geometry is an incredibly interesting field in mathematics. These patterns can often be found in nature, and as such are a useful tool in examining the world around us. Fractal patterns are also used in other areas of science, such as in the study of chaos theory with specific areas such as fluid dynamics and the notorious “Three body problem”. Image compression can use fractal image coding (FIC) as a way to be data efficient when working with large 3-D models. Fractal analysis has also been used in medical research to analyze the structure of tissues and organs, and to detect and diagnose diseases such as cancer and heart disease. Of course the most obvious practical use is in art, as we demonstrated with our visualization of these fractal patterns. The field of computer graphics is currently using fractals to generate virtual landscapes, due to the already fractal nature of mountain ranges, coastlines, and other patterns in nature. The infinite nature of fractals are hard to comprehend, but useful in various practical and theoretical applications.

Through our research, we found that L-system fractals are illustrated via simplistic formulas, as described earlier. These are much easier to understand than their mathematical counterparts, decomposing a once complex theorem into a set of rules; patterns, alterations in degrees (as + or - signs) and forward movements. After finding this specific rule set it made the process of implementing the recursive algorithms significantly more efficient, hence the addition of the Dragon Curve, Hilbert Curve II, Peano Gosper, Square Curve and ultimately, the Schrader Structure.

Bibliography

Berry, Nick. “Koch Snowflake.” *Koch Snowflake*, 2016,

http://datagenetics.com/blog/january12016/index.html.

Dickau, Robert. “2-D L-Systems.” *2-D L-Systems - Robert Dickau*,

https://www.robertdickau.com/lsys2d.html.

“Fractal Geometry.” *IBM100 - Fractal Geometry*,

https://www.ibm.com/ibm/history/ibm100/us/en/icons/fractal/transform/