Recursive Graphics

Sierpinski Triangle, Hilbert's Curve, Koch Snowflake

Nathaniel Brown, Alexandria Sampalis, Rebecca Iselin, Aeyva Rebelo

Recursive Graphics & Fractals

- What are Recursive Graphic?
  - A recursive graphic is the creation and implementation of images using recursive algorithms
  - The recursive algorithms are called over and over to draw these shapes
- What is a Fractal
  - Fractals are generated through repeated application of a recursive algorithm, where a basic shape or pattern is changed, and that changed shape is applied to smaller copies of itself

Project Implementation

- After researching our three algorithms, and how they worked; we started to familiarize ourselves with SFML, an external library.
- Using this we were able to accurately and more efficiently render recursive graphics.
- Achieving this using the standard library, did not seem convenient or viable.

Honorable Mention: SFML

Simple and Fast Multimedia Library

- External library that needs to be manually installed and have proper settings installed within your IDE/c-make folder.
- Contains a **vast** library of template classes; as well as pre-built functions that assist with the development of graphics using C++.
- **ESSENTIAL** for producing any of our graphic outputs.

sf::RenderWindow - Is a template class that generates an object used to open a graphical window that we can draw our images to.

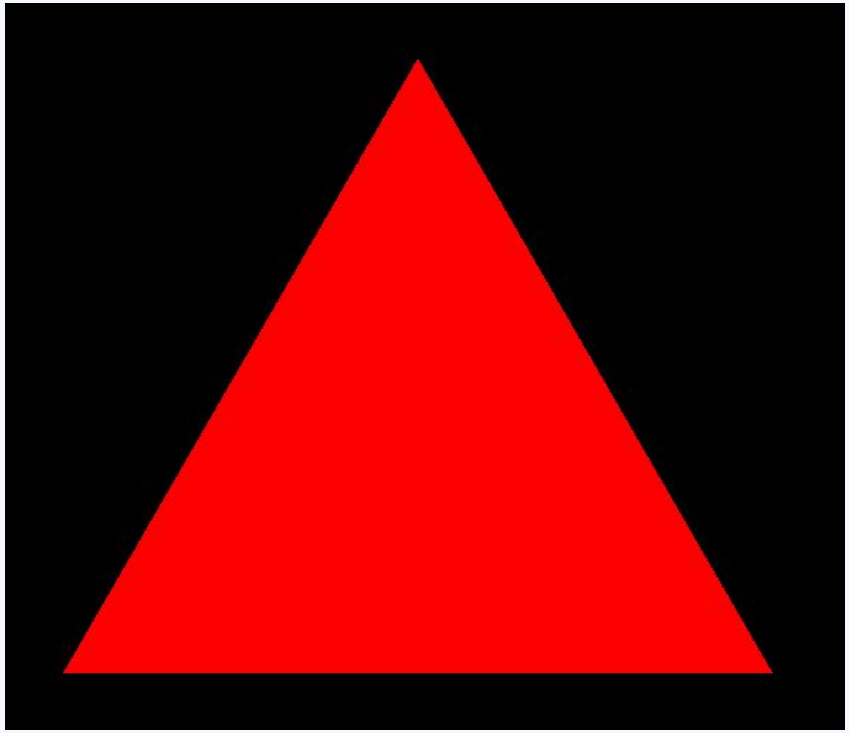
sf::Vector2f - Is a special type of vector that only accepts and x and y value of float type (Although you can specify an alternative type). They function similar to pairs; but have minor differences. They're more compatible with other functions and template classes.

sf::ConvexShape - allows you to draw a shape by assigning max points and coordinates for the points.

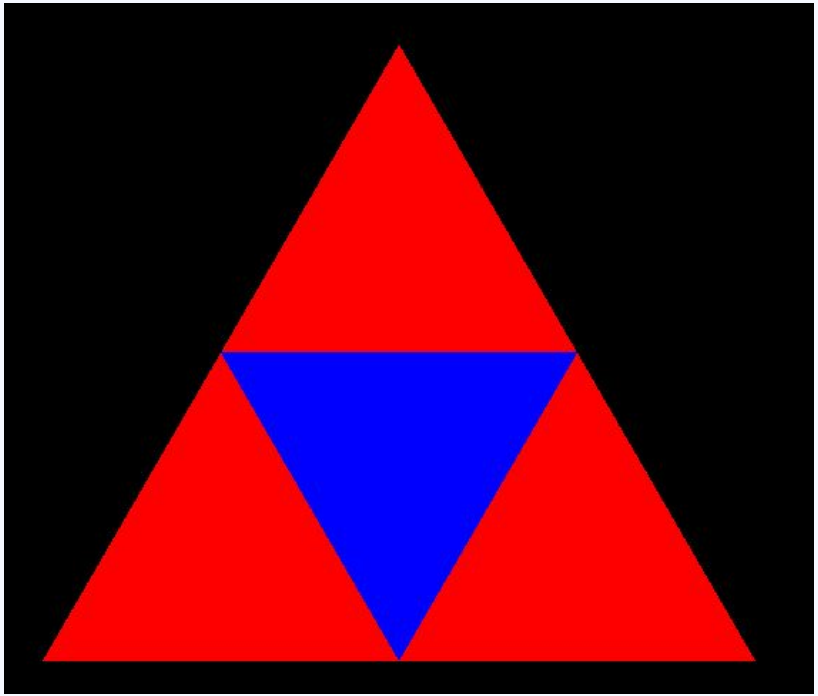
The Sierpinski Triangle

- The Sierpinski Triangle is a fractal named after it's creator Waclaw Sierpinski.
- The construction of this shape involves taking an equilateral triangle and dividing it into 4 smaller triangles and removing the center one. You repeat this process over and over again creating more sub-triangles.
- Steps in our algorithm:
    1. Begin with equilateral triangle
    2. Divide triangle into 4 congruent sub-triangles by locating midpoints of parent triangle - center triangle is removed
    3. Step 2 is repeated for sub-triangles until end case terminates recursive calls

Stage 1: Base Case - Triangle

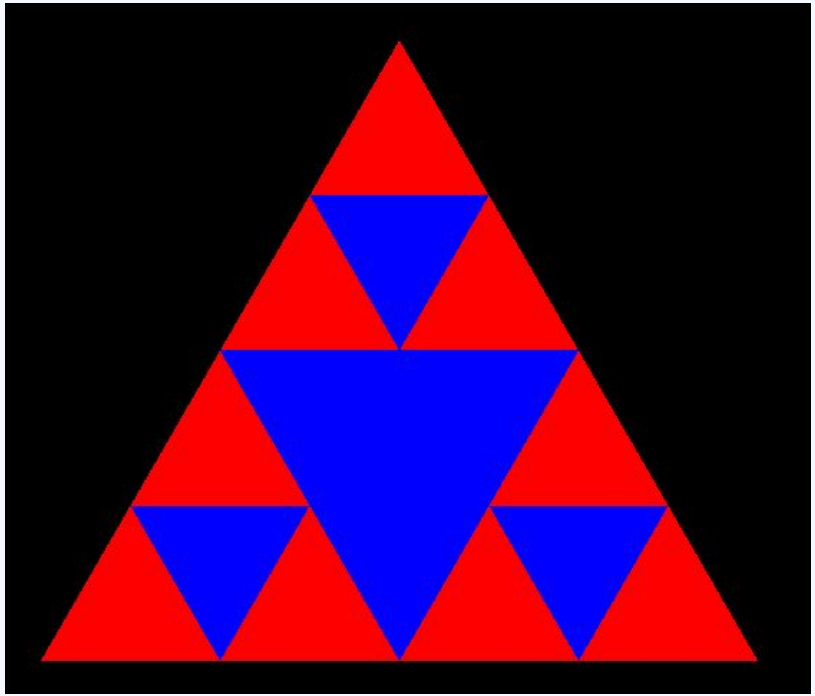


Stage 2: 1 Recursive Iteration



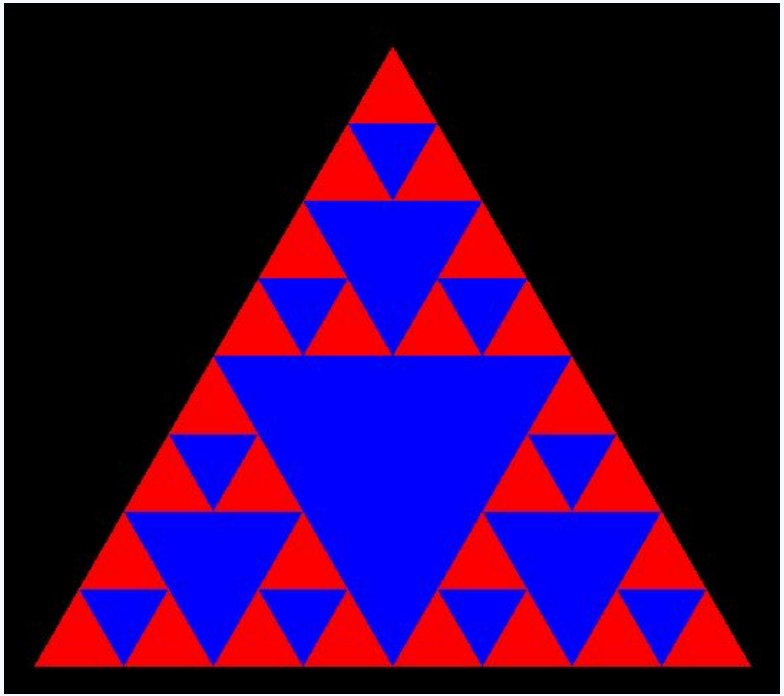
3 Triangles

Stage 3: 2 Recursive Iterations



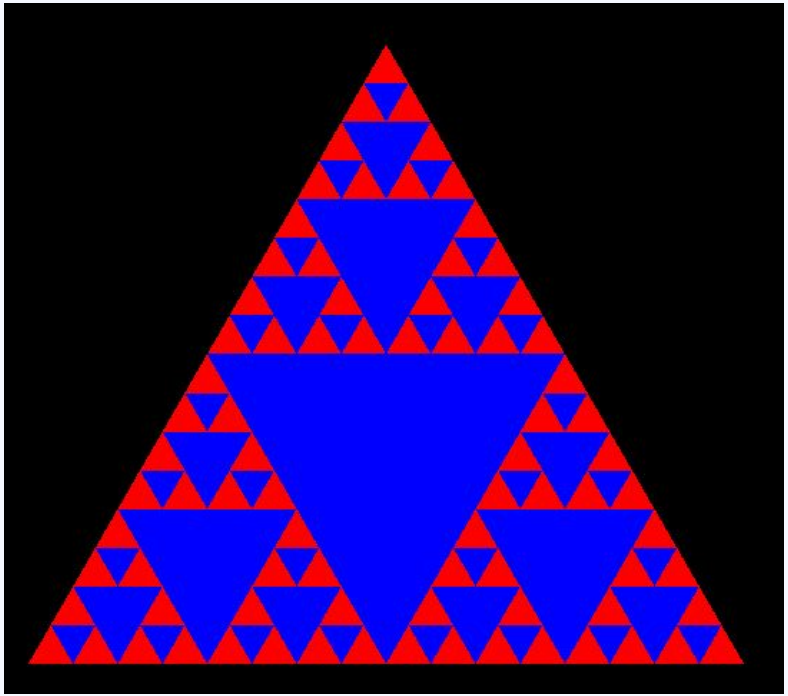
9 Triangles

Stage 4: 3 Recursive Iterations



27 Triangles

Stage 5: 4 Recursive Iterations



81 Triangles

Sequence? 1, 3, 9, 27, 81, …

As the stage of our graphic generation advances; the number of triangles increases by a rate of…

Num_triangles = 3 ^ (i-1)

'i' represents the current iteration

This formula can be used to determine the number of triangles for future iterations if you so desire.

Now let's take a look at it running…

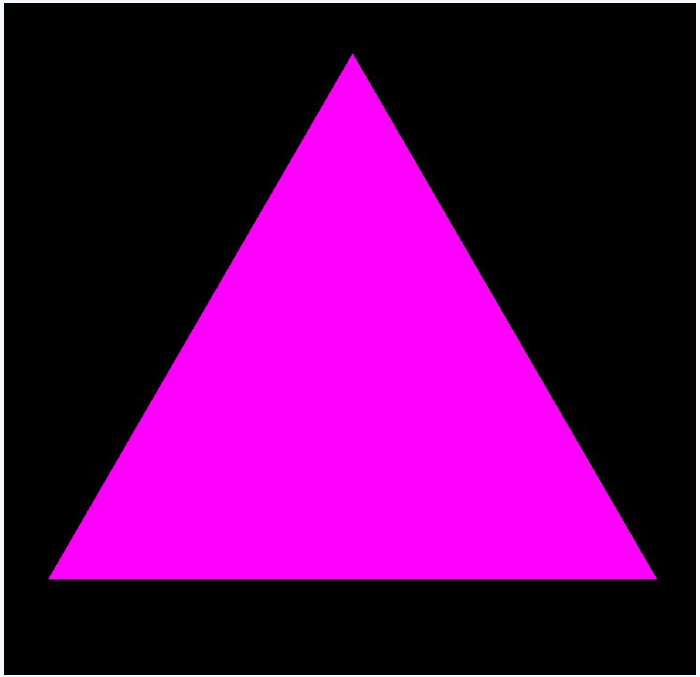
Koch’s Snowflake

Koch’s Snowflake is an algorithm designed by Helge von Koch (*’Coke’*), a swedish mathematician (1980 - 1924). It recursively constructs a snowflake-like shape. The construction of which involves iteratively and recursively adding smaller and smaller triangles into the base of the previous triangle.

The Steps are Simple:

1. Create an equilateral Triangle
2. On each side; create two points with equal distance in all three sections.
3. Using the two points, draw another equilateral triangle through the points with the top facing outward.
4. Repeat steps 2 and 3 until you reach your desired iteration count. (Try 4 or 5 iterations; for a standard Koch’s Snowflake.

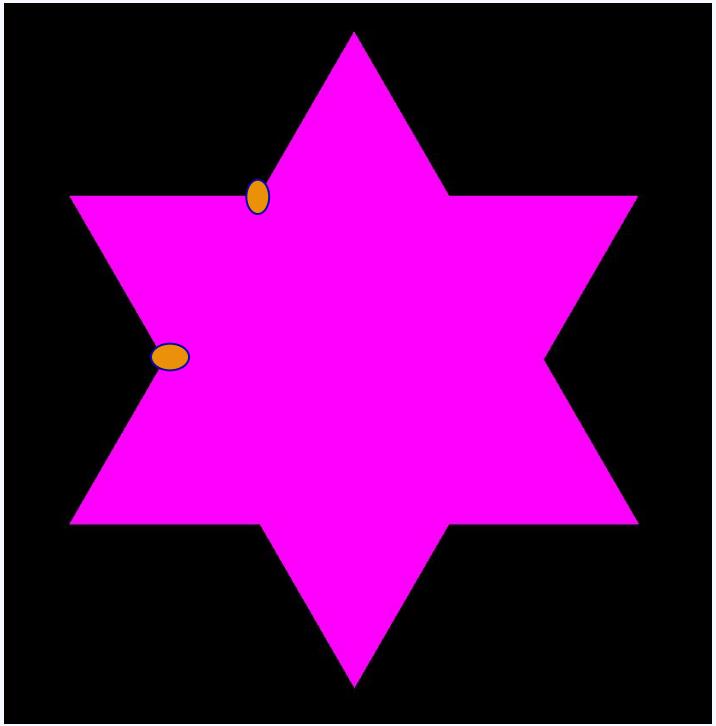
Stage 1: Base Case - Triangle



3 Sides

Stage 2 - 1 Recursive Iteration

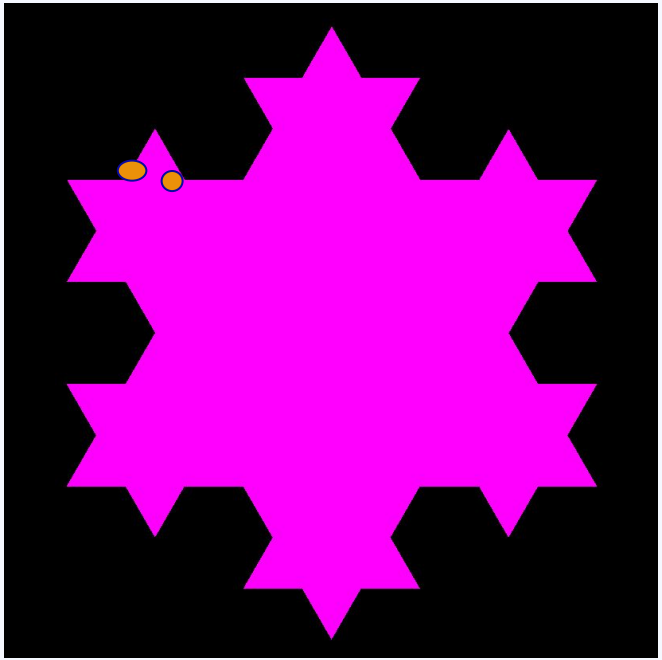
Hexagram:



12 Sides

Stage 3: 2 Recursive Iterations

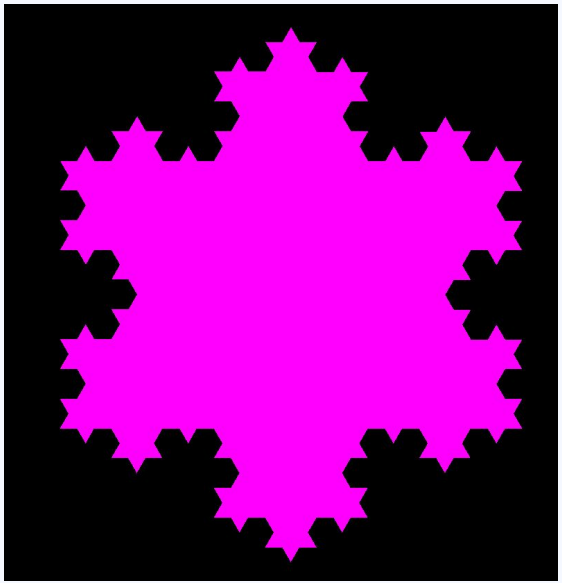
Whatever you'd call this shape:



48 Sides

Stage 4: 3 Recursive Iterations

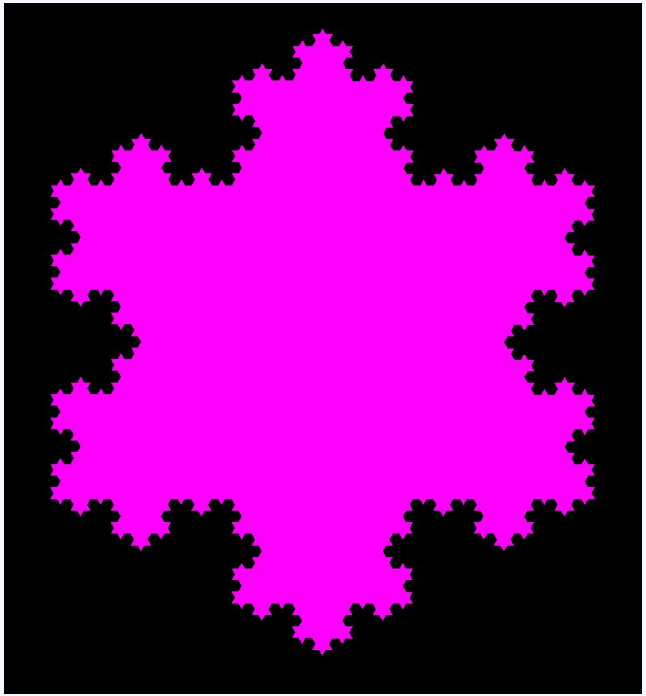
Kind of a snowflake:



192 Sides

Stage 5: 4 Recursive Iterations

Koch's Snowflake:



768 Sides

Sequence? 3, 12, 48, 192, 768, …

As the stage of our graphic generation advances; the sides increase by a rate of…

Num_sides = 3 * 4^i

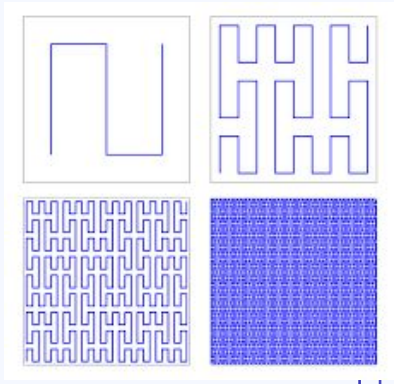
‘i’ represents the current iteration

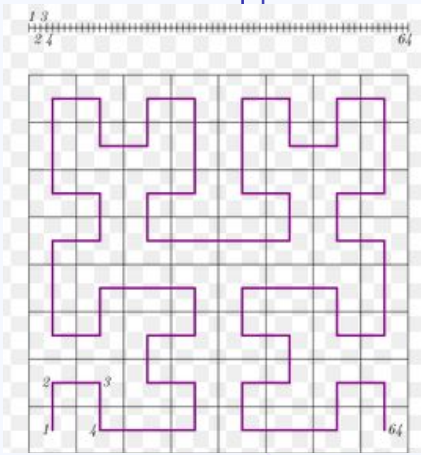
This formula can be used to determine the number of sides for future iterations if you so desire.

Now let’s take a look at it running…

Hilbert's Curve: Introduction

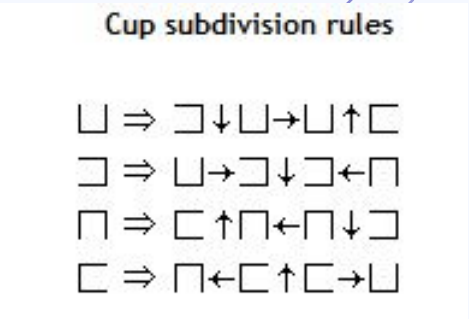
- Also known as Hilbert's Space-Filling Curve.
  - Defined by German mathematician David Hilbert in 1891.
  - Variant of the space-filling Peano curves defined by Giuseppe Peano in 1890.
  - Continuous fractal space-filling curve, meaning it:
    - Does not disconnect at any point.
    - Is a geometric shape with detailed structure at small scales.
    - Fills every space on a N-Dimensional grid-space.
    - Is a curve by line segment connection.



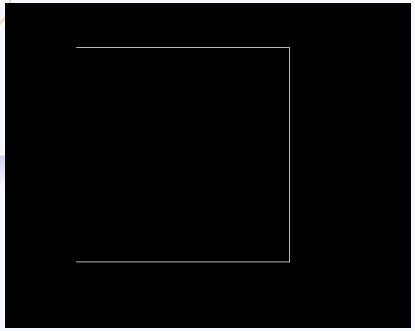


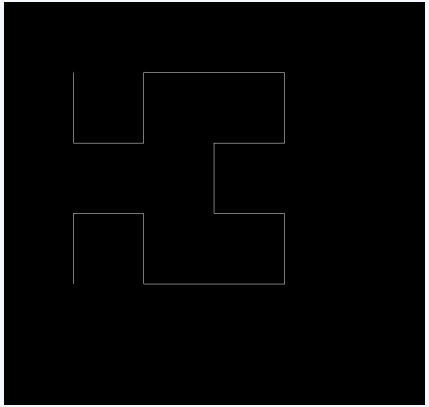
Hilbert's Curve: Generation

- 4^n grid tiles, where n is order of recursion, constructed in sub-quadrants.
- Can be constructed through either/or:
  - Iteration
    - Iterating through ruleset given amount of grids for each grid.
  - Iterative Recursion
    - For n ≥ 2, iterative generation of each previous order case in order to transform it for the current order case.
- From the base case, we determine the ruleset for which direction we will call next in our code.



Hilbert's Curve: 1st & 2nd Order Cases

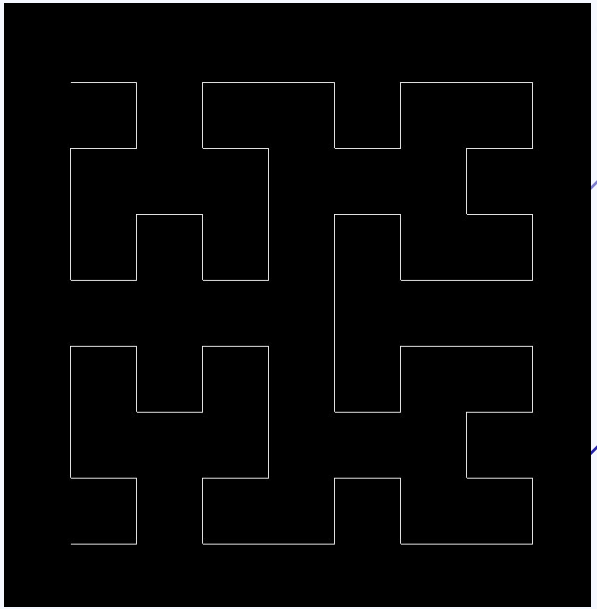




1. Pick a grid to construct a NxN grid, where N >= 2, for a total of 4 grid tiles.
2. Draw any of the four cup shapes, making sure to follow the correct start quadrant and end quadrant for whichever cup you choose.
    a. This determines the order for the rest of your cases.
3. Using the rules of the cup and line generation patterns:
    a. Repeat the generation, recursively following the pattern of which type of cup to generate next for each sub-pattern

Hilbert's Curve: 3rd Order Case

5. By following the precedent set by rule 2, repeat rule 3 with a higher order call to recursively generate a new curve, using the same rules.
    a. Essentially, the rules never change throughout any order, as the curve simply builds smaller curves by generating the previous order curves.



Conclusion

- Working with recursive graphics in C++ gave us invaluable experience learning, using, and comprehending libraries for the first time.
- Without recursion, vector graphic advancements may not have been possible, in applications such as
  - Retro video games - Aztarec or Lunar Lander
  - MRI/CT scanner graphical interface and image overlaying
- And even in outside disciplines such as architecture, we can see very similar approaches to recursion, for example, with Gothic Cathedrals, a very literal form of graphics.
- It is clear at this point how much of a part recursion plays in generative graphics or construction, particularly from our experience in this project.