1 Introduction

In this project, we are to draw and export a Hilbert curve using Haskell and turtle graphics. A Hilbert curve is a fractal that fills a square by making a path that never crosses itself. It's useful because it keeps nearby points close together, which is useful in things like image processing or printing really detailed patterns.

2 Implementation

Here I describe how I implemented the Hilbert curve part step by step.

2.1 Turtle State and Drawing

I used a simple turtle graphics approach. Turtle is a common library for python and first I have implemented function in python using turtle. The turtle has a position, a direction (angle), and a list of lines it has drawn so far:

```
type TurtleState = (Point, Float, [Line])
```

- ‘forward d‘ moves the turtle forward by distance ‘d‘, adding a line segment to the list. - ‘left‘ and ‘right‘ rotate the turtle by 90 degrees without drawing.

2.2 Algorithm

The Hilbert curve is built using two functions, ‘a‘ and ‘b‘, that call each other recursively. Each function handles one "orientation" of the curve:

- ‘a(n)‘: Draws the positive-orientation Hilbert curve of order ‘n‘.
- ‘b(n)‘: Draws the negative-orientation version, basically the mirror image of ‘a‘.

At each level ‘n‘:

1. Turn and call the other function with ‘n-1‘.
2. Move forward one step.
3. Turn back and call self or the other function again.
4. Repeat this pattern four times to form the full curve of that order.

This pattern ensures that as ‘n‘ increases, the curve fills the square more finely.

2.3 Recursive Curve Construction

The core functions look like this in pseudocode:

```
function a(n, length):
  if n == 0: return
  right()
  b(n-1, length)
  forward(length)
  left()
  a(n-1, length)
  forward(length)
  a(n-1, length)
  left()
  forward(length)
  b(n-1, length)
  right()

function b(n, length):
  if n == 0: return
  left()
  a(n-1, length)
  forward(length)
  right()
  b(n-1, length)
  forward(length)
  b(n-1, length)
  right()
  forward(length)
  a(n-1, length)
  left()
```

As you can see, the Hilbert functions are directional opposites of each other. When passing the turtle's direction, some Hilbert functions required multiplying the direction by -1. Instead of doing that, I wrote reversed versions of the same functions and made them call each other accordingly. Since the recursion always decreases the level, this approach worked without any issues.

2.4 Curve Entry Point

The function 'hilbertcurve order' sets up the turtle in the bottom-left corner, calculates the step size as 'spanSize / $(2^order - 1)$', and starts with function 'a'. After the recursion, we get a list of line segments.

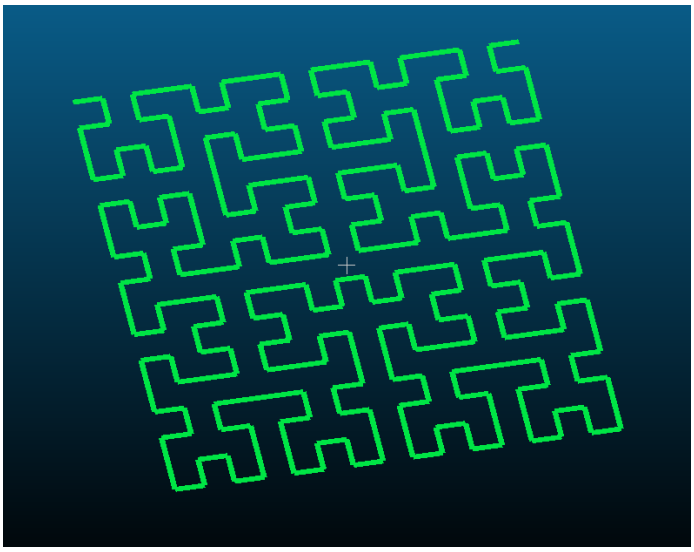

Figure 1: Level 4 Hilbert Curve visualization

3 Conclusion

I learned how a simple set of turtle commands and mutual recursion can generate a complex fractal like the Hilbert curve. Overall, it was a fun way to practice both recursion and graphics in Haskell.