BLG458E - Functional Programming
Homework 1 Report
Hilbert Curve Drawing with Haskell

Feyza Nur Keleş
820220332

April 17, 2025

Contents

1 Introduction

In this report, I explained how I approached the problem, implemented the recursive construction of the Hilbert curve in Haskell, and visualized the output using STL models. I also included sample outputs of Hilbert curves of orders 1 through 6 and mentioned the challenges I had during the implementation.

2 Problem Definition

In this homework assignment, we were expected to change the Hilbert Curve Function so that our "turtle" creates a hilbert pattern with given order.

3 The Path to My Solution

3.1 Curve Construction

First, I implemented this in python. It was fairly simple, I thought. First, I realized I could generate the first layer and then split one edge of the space into $2n+1$ with n being initially 0. And then copy the previous pattern into 4 corners with correct scaling and rotation given in this table:

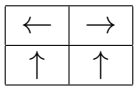


Figure 1: Rotation Table

And with just 3 new connection lines every order, two on the left and right edges of the space, and one in the bottom of the center grid, I could achieve this pattern.

However, upon thinking I realized this solution was unnecessarily hard to implement, and instead of copying the old pattern and connecting them and repeating this step until we get to the requested order of pattern, we could just change where the turtle was going to turn his head while creating this pattern, and my implementation was just defeating the whole point, which is making the turtle generate the pattern starting from a line. Therefore, I searched for other algorithms and found one that I could implement in Haskell.

3.2 Overview of the Skeleton Code

The skeleton code provides basic functions to help convert a list of 2D line segments into a 3D STL file format. Here's a quick summary of what each part does:

- **Point, Line, Triangle types:** Points are 2D coordinates `(Float, Float)`, lines are defined by two points and a width, and triangles are used to model surfaces in the STL file.
- `sortClockwise`: Sorts four corner points in a clockwise order, used for building rectangles correctly.
- `get_rectangle_corners`: Takes a line and calculates the four corners of the corresponding rectangle with a given thickness.
- `create_triangles_from_rectangle`: Splits a rectangle into four triangles using its center point. This helps in STL formatting.
- `linelist_to_rects`: Converts a list of thick lines into triangles by using the two functions above.
- `createTriangleDef` and `createObjectModelString`: These functions turn the triangle list into a proper STL-formatted string.
- `writeObjModel`: Saves the STL string into a file so it can be viewed in tools like CloudCompare.

Basically, this code sets up everything needed to draw and export 2D shapes in 3D, and all we had to do was generate the list of Hilbert lines in the correct formatting.

3.3 Changes to the `hilbertcurve` Function

Like I explained in the previous part, the code uses a turtle graphics approach to generate the Hilbert curve. The turtle moves and turns recursively to draw the path.

Initial Part of the Function Sets up the canvas, computes step size based on the order requested, and initializes the turtle at the top-left corner facing right.

```
hilbertcurve :: Integer -> [Line]
hilbertcurve order =
  let
    spanLen     = 10.0                          :: Float
    size        = (2 :: Int) ^ order
    steps       = size - 1
    stepLen     = spanLen / fromIntegral steps
    startPos    = (-spanLen/2, spanLen/2)       :: Point
    initState   = (startPos, 0)                 :: (Point, Float)
    (_, segments) = hilbertTurtle order (pi/2) stepLen initState
  in
    segments
```

This code defines a constant line thickness, and sets the type signature for the recursive turtle function:

```
  where
    lineWidth :: Float
    lineWidth = 0.05

    hilbertTurtle
      :: Integer                    -- recursion depth
      -> Float                      -- turn-angle in radians
      -> Float                      -- step length
      -> (Point, Float)             -- current (position, heading)
      -> ((Point, Float), [Line])
```

Base Case At depth 0, return current state and no lines.

```
hilbertTurtle 0 _ _ st = (st, [])
```

Recursive Construction In each call of

`hilbertTurtle lvl ang step st0`

we execute the following code, to kind of implement Hilbert's "a" and "b" functions with the sign of `ang` controlling orientation:

1. **Turn into sub-curve B:**
   - `st1 = turnRight st0` (rotate heading by $-ang$)
   - `(st2,s1) = hilbertTurtle (lvl-1) (-ang) step st1` ← "B" at order n-1
2. **Draw one segment:**

   `(st3,f1) = forward st2`

   where

   $$nx = x + step\,\cos d, \quad ny = y + step\,\sin d$$

   creates a single line: $\big((x, y), (nx, ny), \texttt{lineWidth}\big)$.
3. **Turn into sub-curve A:**
   - `st4 = turnLeft st3` (rotate by $+ang$)
   - `(st5,s2) = hilbertTurtle (lvl-1) ang step st4` ← "A" at order $n-1$
4. **Draw one segment:** `(st6,f2) = forward st5`.
5. **Another sub-curve A:**

   $(st7, s3) =$ `hilbertTurtle (lvl-1) ang step st6`
6. **Turn and draw:**

   $st8 =$ `turnLeft st7`, $\quad (st9, f3) =$ `forward st8`

7. **Final sub-curve B and cleanup:**
   - `(st10,s4) = hilbertTurtle (lvl-1) (-ang) step st9` ← "B" at order $n-1$
   - `st11 = turnRight st10` (restores original heading for parent call)
8. **Combine all segments:**

$$\texttt{combined} = s1 + f1 + s2 + f2 + s3 + f3 + s4$$

and return (`st11`, `combined`).

3.4 Line Generation and STL Output

I used the skeleton code provided in the homework, so I didn't have to implement the STL conversion part myself. Most of the heavy lifting — like turning a line into a rectangle, splitting it into triangles, and writing everything into a proper STL file — was already done for me.

However, I realized that the pdf was explaining the overall point point weight srtucture as **"Any line segment is represented using a tuple of type (Point, Point, Float) where the first parameters represent start and end points of the line segment and the last parameter represent the thickness."** and I was implementing my functions that way but the function in skeleton code expected a line like `get_rectangle_corners ((x1,x2), (y1,y2), width)`, which is not how lines are actually defined. So, after having so much trouble with my outputs, I changed the function parameters to be:

```
get_rectangle_corners ((x1,y1), (x2,y2), width) = ...
```

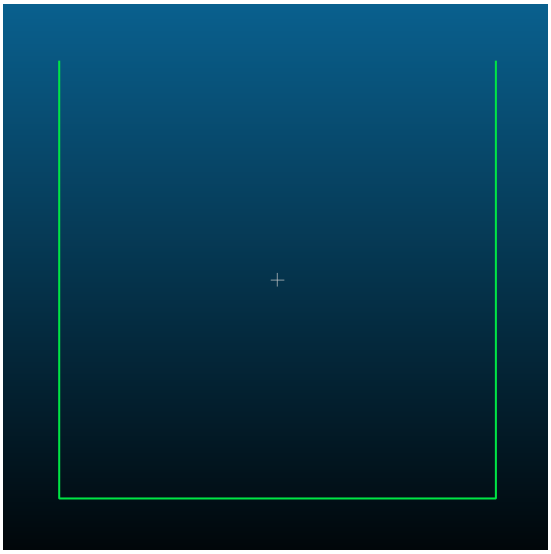
After that, the outputs looked normal. So overall, I didn't need to fully understand all the geometry code. I just had to make sure it was using the line format correctly, as described in the PDF, to fit my implementation.

4 Conclusion

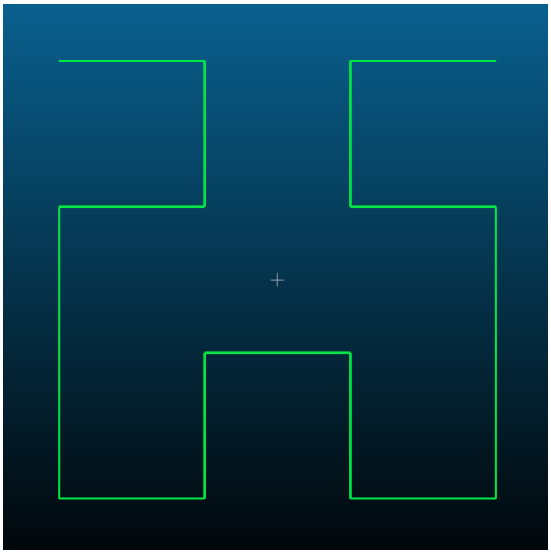
At first, the problem felt complex, especially when thinking about how to implement a visual, recursive structure into point by point creation. But once I found a turtle-graphics-based algorithm, everything started to make more sense, especially in the context of Haskell's recursive strengths. Even though I didn't write the STL export logic myself, I had to understand just enough of it to identify and fix a bug that caused invalid outputs. Overall, this assignment helped me practice recursion in a more visual and practical way and was fun to implement and think of the pattern.

5 Results

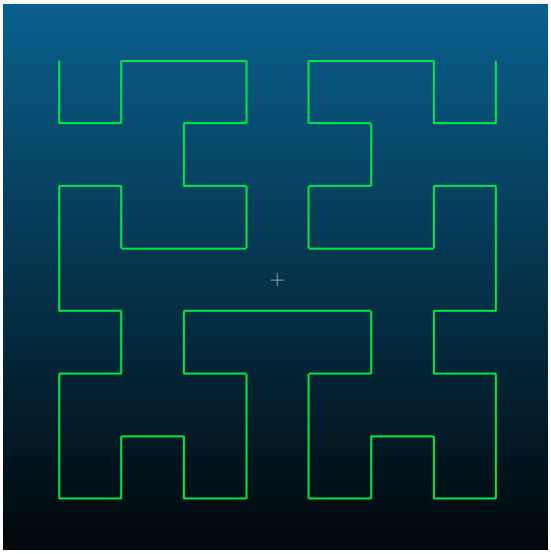
Hilbert Curve Outputs



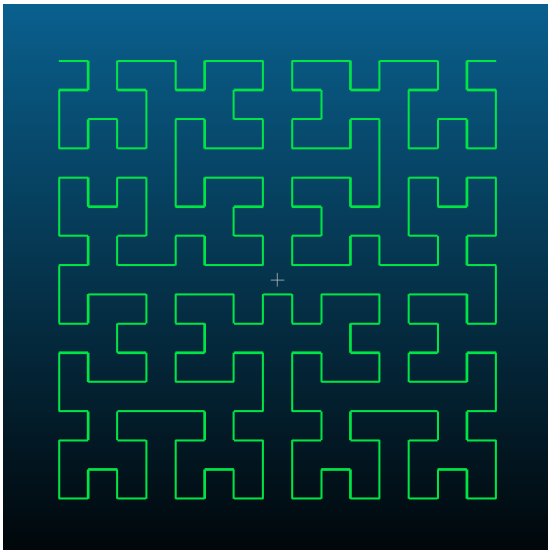
(a) Order 1



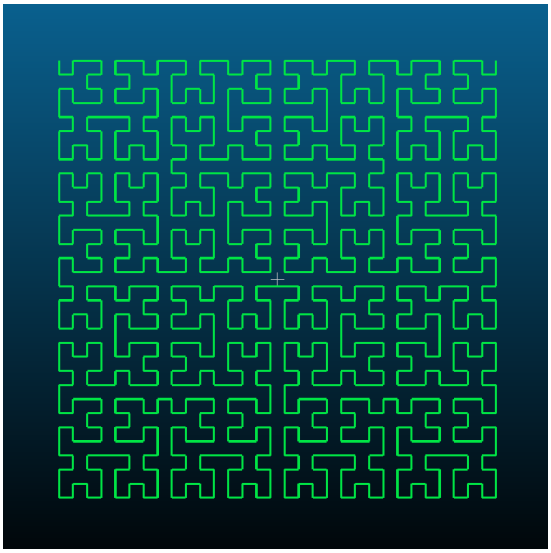
(b) Order 2



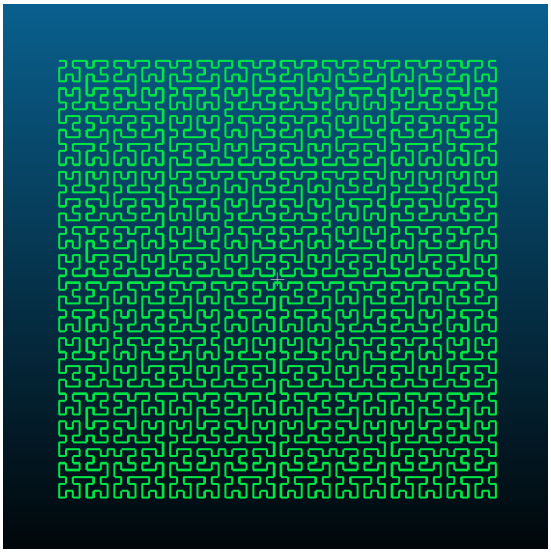
(c) Order 3



(d) Order 4



(e) Order 5



(f) Order 6