ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 458E
Functional Programming

Homework 1

Ali Emre Kaya
150210097
kayaemr21@itu.edu.tr

SPRING 2025

Contents

1 Hilbert Curve

Hilbert Curve is a continuous fractal space-filling curve described by David Hilbert.

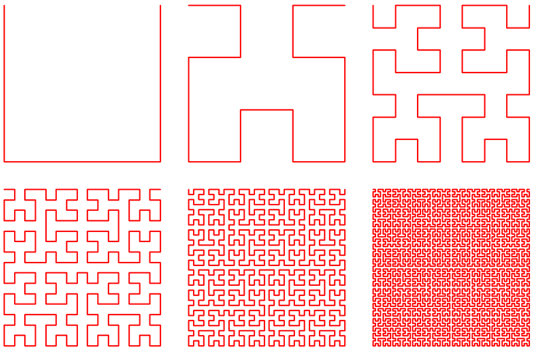


Figure 1: Hilbert curver order 1 to 6 [1]

Hilbert curve is a curve that grows by repeating itself recursively. It is used in different fields such as computer science, image and video rendering. The reason for using it in image and video rendering fields is its ability to preserve the locality of the data.

Let's say you have a one-dimensional array, numbers between 0 and 1 with only 1 digit after the decimal point.

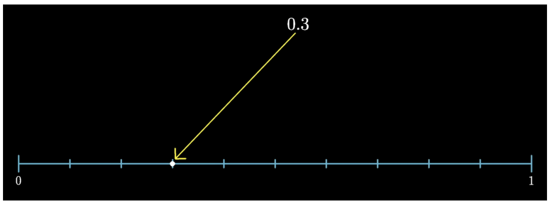


Figure 2: 1d array [2]

You can represent this array using a Hilbert curve as follows.

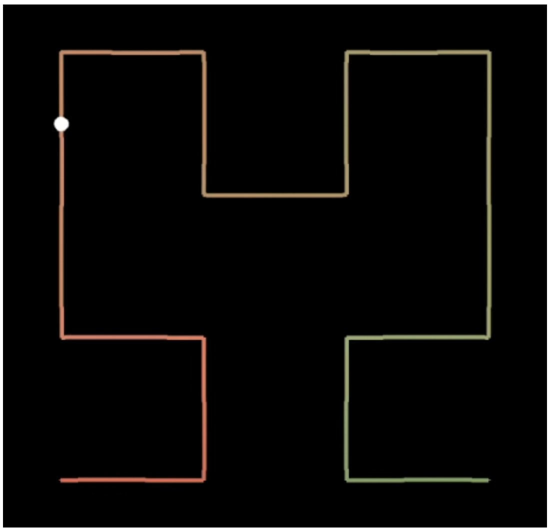


Figure 3: 2nd order Hilbert Curve [2]

If you add numbers with 2 digits after the decimal point to this array in a sequential manner (it can also be thought of as expanding an audio), you will see that there is a minimal change in the places of the old values in the 2D grid.

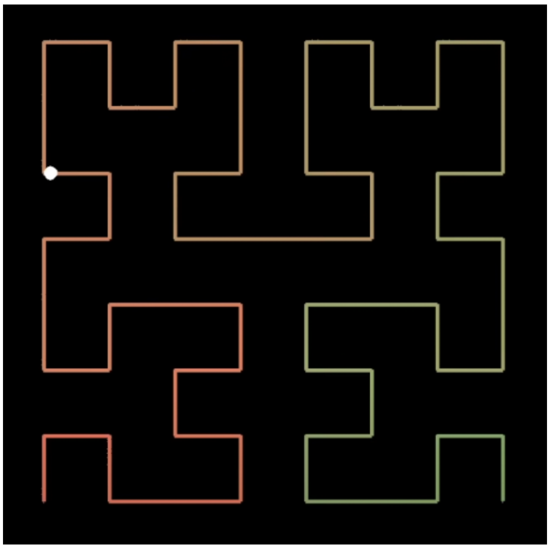
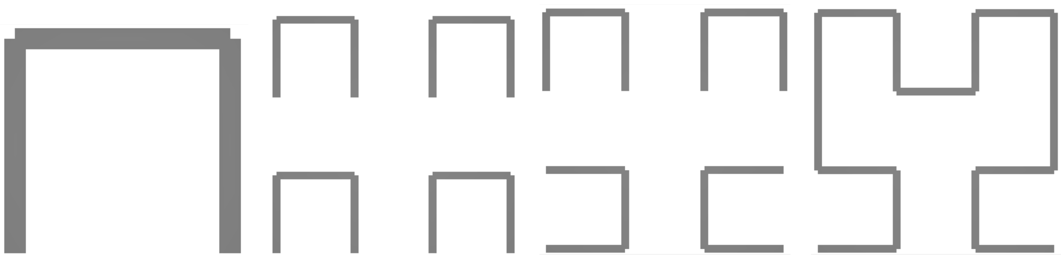


Figure 4: 3th order Hilbert Curve [2]

2 Code

My Hilber curve generation code follows the steps below recursively.



(a) Order 1 (b) Order 2 prep (c) Order 2 prep (d) Order 2

Figure 5: Passing order 1 to order 2

For example, to create a hilber curve in step 2, the program first comes to the Order 1 level and takes the default Lines 5a.

```
hilbert_n :: Integer -> [Line]
hilbert_n 1 = map (scaleLine 1) line_list
hilbert_n n =
    let
        prev = hilbert_n (n-1)
```

The function that appears here as `scaleLine` adjusts the thickness and length of the Line according to given number. The reason I used this here was that I could not see it clearly in high orders due to the thinness of the bars. If you were going to calculate an order higher than 5, increasing the value of 1 here would make it easier for you to see.

The content of the `scaleLine` function is:

```
scaleLine :: Float -> Line -> Line
scaleLine factor ((x1,y1),(x2,y2),w) = ((x1*factor, y1*factor), (x2*factor, y2*factor), w*factor)
```

After keeping the base lines it received as *prev*, it continues to work to calculate the 2nd order.

Here, first of all, the scale Line function appears in the figures again, the reason for using this function here is to reduce the size of the Lines in order to create four new Order 1 copies.

```
scale = 1 / (2^(fromIntegral n - 1))

scaled = map (scaleLine scale) prev
```

The reduced Order 1 is kept in the `scaled` array.

The reason for using `fromIntegral` is to convert the `Integer` n to `Float` since I want a decimal number. The reason I use `map` is that since `scaleLine` only takes one Line, I cannot give all the Lines to the function at once. I solve this by using map and adding the Lines in `prev` to the function one by one.

Then I create 4 different Order 1s using scaled Lines.

```
182 bottomLeft_scaled = scaled
183 bottomLeft_origin = findOrigin bottomLeft_scaled
184 bottomLeft = rotateLineArray90CW bottomLeft_origin bottomLeft_scaled
185 bottomRight_scaled = map (translateLine (lineHeight, 0)) scaled
186 bottomRight_origin = findOrigin bottomRight_scaled
187 bottomRight = rotateLineArray90CCW bottomRight_origin bottomRight_scaled
188 topLeft = map (translateLine (0, lineHeight)) scaled
189 topRight = map (translateLine (lineHeight, lineHeight)) scaled
```

If we examine lines 182, 185, 188 and 189 above, we will notice that 4 different small Order 1s are created using the `scaled` array. These 4 Order 1s are created at the same index, I use the `translateLine` function to move them to the correct positions.

```
translateLine :: (Float, Float) -> Line -> Line
translateLine (dx, dy) ((x1,y1),(x2,y2),w) = ((x1+dx, y1+dy), (x2+dx, y2+dy), w)
```

The `translateLine` function takes a Float tuple and a Line, shifts the Line according to the data in the tuple, and returns the new Line. In this way, I get the view in the figure 5b.

After this process, I need to rotate the left copy clockwise and the right copy counterclockwise. To do this, I use the `rotateLineArray90CW` and `rotateLineArray90CCW` functions.

```haskell
rotateLineArray90CW, rotateLineArray90CCW :: (Float, Float) -> [Line] -> [Line]
rotateLineArray90CW (ox, oy) = map rotateLine
  where
    rotateLine ((x1, y1), (x2, y2), w) =
      let newP1 = rotate (x1, y1)
          newP2 = rotate (x2, y2)
      in (newP1, newP2, w)
    rotate (x, y) = (ox + (y - oy), oy - (x - ox))
```

These functions take a tuple with two Floats as the center of rotation and a Line (I will explain how to find the center in the next code snippet). The `rotateLineArray90CW` function takes all the Lines one by one and rotates them. While doing this, the function creates a different function inside itself (`rotateLine`), the newly created function applies the rotation process by calling another function (`rotate`) defined in the same function for the start and end points of the bar, the details of the process are available on this page [3].

Finding the origin of the lines is not a very difficult application, I just take the farthest corners and their average gives the origin.

```haskell
findOrigin :: [Line] -> (Float, Float)
findOrigin lines = ((xmin + xmax) / 2, (ymin + ymax) / 2)
  where
    -- Concatenate all the points
    points = concatMap (\((x1, y1), (x2, y2), w) -> [(x1, y1), (x2, y2)]) lines
    xs = map fst points
    ys = map snd points

    xmin = minimum xs
    xmax = maximum xs
    ymin = minimum ys
    ymax = maximum ys
```

The function rotates all Lines one by one, selects the max and min of x and y coordinates from them, then takes the average and returns it as a tuple. `concatMap` is used to put the start and end points of all points into a single list, `fst` and `snd` are used to separate the pair, fst takes the first element while snd takes the second, so I can reach all x indexes with `fst` and all y indexes with `snd`.

After doing these rotation operations, I have the image in 5c. After this step, all I have to do is connect the separate pieces to each other in an orderly manner.

```
createConnector :: [Line] -> [Line] -> [Line] -> [Line] -> [Line]
createConnector lines1 lines2 lines3 lines4 =
    let
        -- Bottom Left
        points1 = concatMap (\((x1, y1), (x2, y2), _) -> [(x1, y1), (x2, y2)]) lines1
        ymax1 = maximum (map snd points1)
        xmin1 = minimum (map fst points1)

        -- Top Left
        points2 = concatMap (\((x1, y1), (x2, y2), _) -> [(x1, y1), (x2, y2)]) lines2
        xmin2 = minimum (map fst points2)
        ymin2 = minimum (map snd points2)
        xmax2 = maximum (map fst points2)

        -- Top Right
        points3 = concatMap (\((x1, y1), (x2, y2), _) -> [(x1, y1), (x2, y2)]) lines3
        ymin3 = minimum (map snd points3)
        xmin3 = minimum (map fst points3)
        xmax3 = maximum (map fst points3)

        -- Bottom Right
        points4 = concatMap (\((x1, y1), (x2, y2), _) -> [(x1, y1), (x2, y2)]) lines4
        xmax4 = maximum (map fst points4)
        ymax4 = maximum (map snd points4)

        (_, _, w) = head lines1
    in
        [((xmin2, ymin2), (xmin2, ymax1), w), ((xmax2, ymin2), (xmin3, ymin3), w), ((xmax3, ymin3), (xmax4, ymax4), w)]
```

I do the connecting process with the `createConnector` function. This function simply finds the vertices that need to be connected and connects these vertices with a string of the required length. With this connections, finally I achieve to take figure 5d.

Below are some Hilbert Curve Orders that I created with my own code and visualized using some web sites (viewstl, 3dviewer).

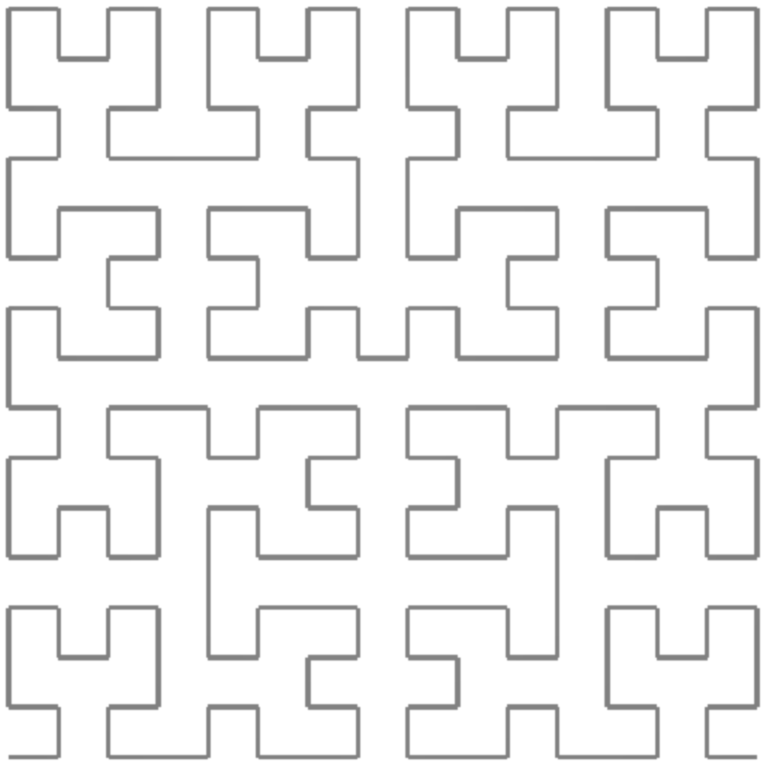


Figure 6: Hilbert Curve Order 4

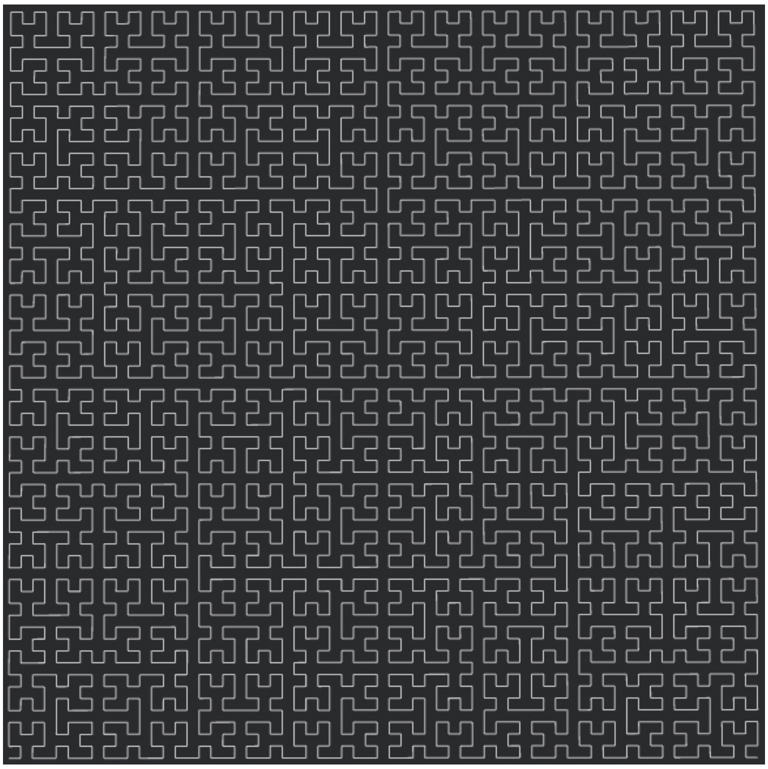


Figure 7: Hilbert Curve Order 6

REFERENCES

[1] Nick Berry. Hilbert curves, 2013. Accessed: 2025-04-11.

[2] Grant Sanderson. Hilbert's curve: Is infinite math useful? https://www.youtube.com/watch?v=3s7h2MHQtxc, 2017. 3Blue1Brown, YouTube. Accessed: April 11, 2025.

[3] Maison et Math. 90 degree rotation around a point that is not the origin. https://maisonetmath.com/transformations/video/571-90-degree-rotation-around-a-point-that-is-not-the-origin, 2017. Accessed: April 11, 2025.