FRANKFURT UNIVERSITY OF APPLIED SCIENCES
OOP JAVA PROJECT, WS22/23. 2022-2023

CONCURRENT VISUALIZATION OF SPACE FILLING CURVES FOR CREATING ART IMAGES

Phi Dinh Van Toan - 1403923
Nguyen Truong Duy - 1403266
Nguyen Khac Hoang - 1403402

Frankfurt University of Applied Sciences
5th February 2023

Contents

Preface

During our college years, geometric mathematics always be an indispensable part of our study program. Despite of the abstraction and the complexity of those things, we found geometry as an interesting aspect which will play a significant role in our programming career in the future. Having a chance of dealing with geometry problems through a completely new project made us feel very excited and inspired. Learning something new is an enormous challenge, especially for those who are studying the major computer science because it takes lots of time and sometimes, sitting in front of the computer all day may lead to some osteoarthritis problems. The documentation demonstrates the result of the whole project in order to prove the efforts and achievements after studying the Java course. This documentation is very important because after finishing and getting feedback from the lecturer, we will gain lots of experience from making mistakes and moreover, it helps us to develop ourselves on the way to become successful programmers in the future. Due to the first time writing documentation, we mainly hope this documentation be a communication to share knowledge about the space-filling curves with others students in the university.

Chapter 1: Team Work

The chapter explores the processing of the project through the timelines. It includes a basic idea about steps for illustrating the result of the space-filling curve project through a GUI. The tasks of all team members are also contained in this chapter. It brings an overview of the author's contribution to the documentation and the goals of this project.

1.1 Team member tasks

Phi Dinh Van Toan - 1403923:

- Find - Build algorithm
- Code Hilbert Curve - Lebesgue Curve - "Matrix" curve - Plier Curve - Triangle Curve - Sierpinski Triangle
- Chapter 4, Chapter 5.2,5.3 , chapter 6.4, 6.5, chapter 7

Nguyen Khac Hoang - 1403402:

- Code - Design and coding GUI, created animation interaction, responsible for JavaFX and Processing integration, code Gosper curve
- General planning operation and setting target of project
- Reformat code, compiled executable jar file
- Chapter 1, Chapter 5.1, Chapter 6.1, Chapter 6.2, Chapter 6.3, Chapter 6.6

Nguyen Truong Duy - 1403266:

- Develop Latex document
- Code Koch Snowflake

1.2 Work flow

+) 24th November - 2nd December: Find out about the general idea of the topic.
+) 5th December - 28th December: Coding 7 space-filling curves and creating GUI at the same time.
+) 3rd January - 15th January: Update and combine the code to create the first version of the application.
+) 16th January - 23rd January: Fixing and demonstrating the second version which is the final version of the project.
+) 24th January - 8th February: Focus on building the final documentation.

1.3 Basic idea:

Write algorithms by coding to demonstrate the space-filling curves on a GUI based on fractals. Firstly, design and generate some space-filling curves. At the same time, a GUI will be created on different platforms. Finally, algorithms of space-filling curves will be embedded in a GUI environment. Basically, with different algorithms + acknowledgement of Java coding, construct programs to build some space-filling curves and run them at the same time in order to create images in a given area.

Target:
+) Create some space-filling curves.
+) Create GUI for adding those space-filling curves.
+) Show the images run on the GUI.
+) Set colors, backgrounds, and other parameters.

Chapter 2: Introduction

In this chapter, we will consider a brief introduction to the space-filling curves. It also allows us to consider the milestone of the history of space-filling curves. Overall information is also included in order to gain some basic acknowledgements about the space-filling curves. Besides that, the application of this topic in real life also is discussed through examples. Overall, this is the base chapter for readers to get to know and imagine what space-filling curves look like.

Georg Cantor, a German mathematician, made a startling discovery in 1877. He found that a one-dimensional line has more points than a two-dimensional surface. Cantor linked the collection of points that make up a square's area to the collection of points along a certain segment of the square's perimeter and tried to demonstrate how the two sets are of equal size. But finally, this idea was rejected by intuition. More than a decade later, based on Cantor's exciting discovery, Giuseppe Peano found the first continuous space-filling curve, which was known as the Peano curve. The result of Cantor's discovery helped Peano to construct a continuous mapping from the unit interval onto the unit square. After the foundation of the continuous space-filling curve of Peano, some other space-filling curves were created such as the Hilbert curve, which was generated by David Hilbert in 1891.

Those curves have become more familiar to us. They have been specifically suited to serve as programmers' tools to illustrate specific algorithmic techniques (especially recursion). Even more surprisingly, the curves have proved to have real-world uses. They help with allocating resources in massive computational jobs, they are used to encode geographic information, and they are used in image processing. Furthermore, they also catch the attention of people who enjoy complicated geometric designs.

Space-filling curves are found in a variety of computer science domains, particularly those where it is crucial to linearize multidimensional data. Matrix, picture, and table, generated by the partitioning of partial differential equations are samples of multidimensional data. When we select an effective way of navigating the data, data operations like matrix multiplications, updating, and partitioning of data sets

can be streamlined. Space-filling curves offer precisely this ideal method of mapping high-dimensional data onto a one-dimensional sequence in many applications.

The discovery of the space-filling curve in 1890 opened the door and welcome mankind to a new vivid world. Space-filling curve plays an important role in creating images these days. Moreover, in the geography field, it helps scientists to analyse and do research about some natural phenomena such as waves or the construction of coastlines in sketching.

Figure 2.0.1: A piece of coast line before zooming

Figure 2.0.2: A piece of coast line after zooming

Chapter 3: Description

The target of the project is to help readers to have more understanding of the space-filling curves and how they are created. But first of all, we need to contemplate some fundamental aspects. Before finding out further acknowledgement about the space-filling curve, we must consider its definition and its related information. Moreover, images are the best way to illustrate this abstract geometry. In this chapter, we include a variety of pictures which describes comprehensible examples of space-filling curves, from those that are already widely known to some curves which are self-customized.

Actually, the research of Cantor in 1877 had point out that a space-filling curve is a continuous function that converts a line to a plane even when the line and the plane are not topologically equal. It is almost considered impossible but in the space-filling curve world particularly and the mathematics world in general, it is found as an important aspect of dealing with a multidimensional environment.

According to mathematical analysis, the term ”space-filling curve” is a curve which reaches every point in a higher dimensional region, normally by a unit square. It passes through every cell element in the multi-dimensional space so that every cell (unit square) is visited exactly once.

In this beginning explanation about considering more details about space-filling curves, introducing the levels and the pattern of a specific curve is very crucial, which is the base step for further understanding. Levels and patterns are the two main parts which contributed significantly to constructing a space-filling curve drawing.

Space-filling curves have lots of examples to consider, but let’s take a basic example, the Hilbert curve, to see in different levels of its change and how the pattern works in a grid plane.

The first level of the Hilbert Curve is very simple. In the above figure as a 2x2 grid, we easily saw the order that the Hilbert Curve will go on:

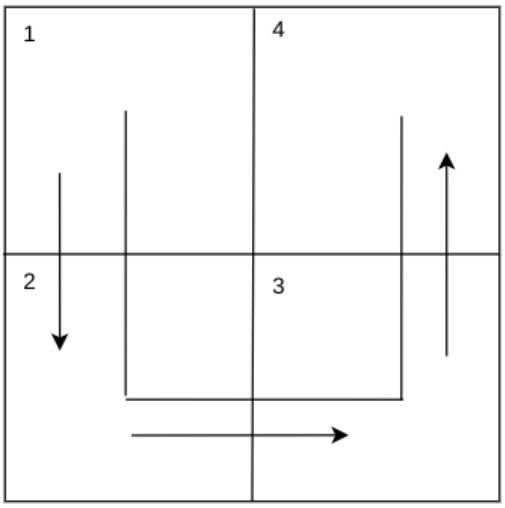


Figure 3.0.1: Hilbert Curve Level 1

$$1 \rightarrow 2$$
$$2 \rightarrow 3$$
$$3 \rightarrow 4$$

And the second level of the Hilbert Curve will be:

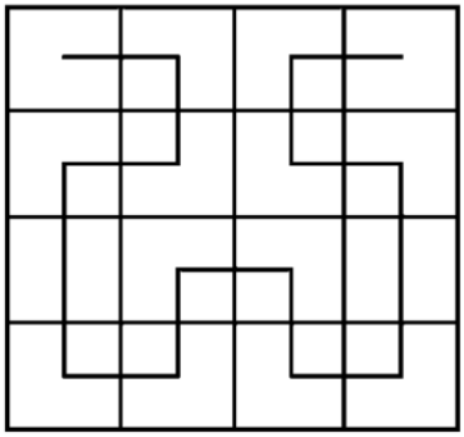


Figure 3.0.2: Hilbert Curve Level 2

With the level of 5, the Hilbert curve will give us:

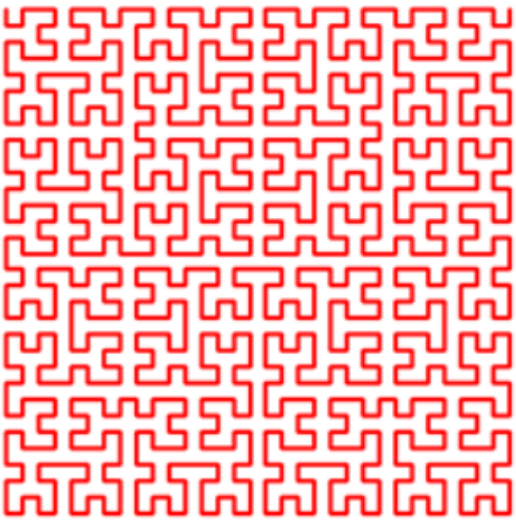


Figure 3.0.3: Hilbert Curve Level 5

Moreover, there are some popular space-filling curves that can be created such as:

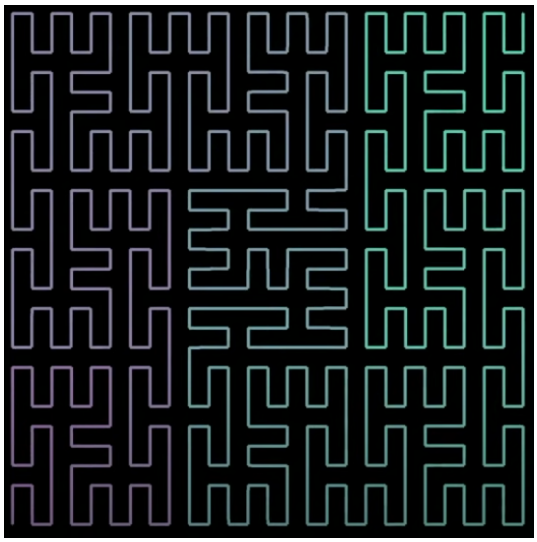


Figure 3.0.4: Peano Curve Level 3

Figure 3.0.5: Lebesgue Curve (Z-Curve) Level 3

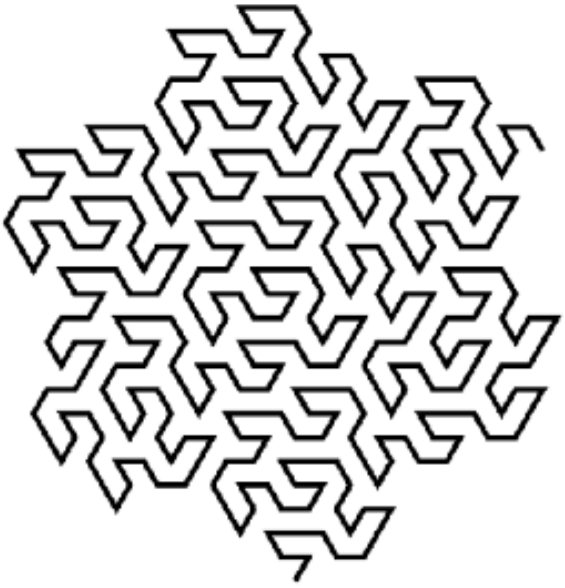


Figure 3.0.6: Gosper Curve Level 3

Or some curves that are self-customized:

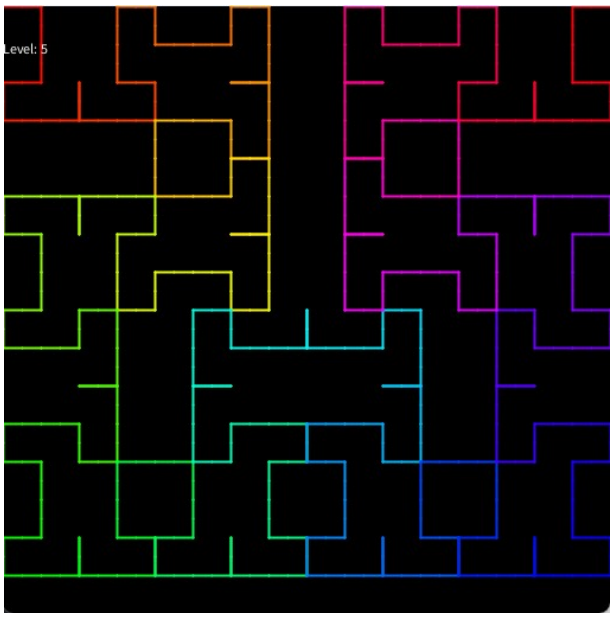


Figure 3.0.7: ”Matrix” Curve Level 5

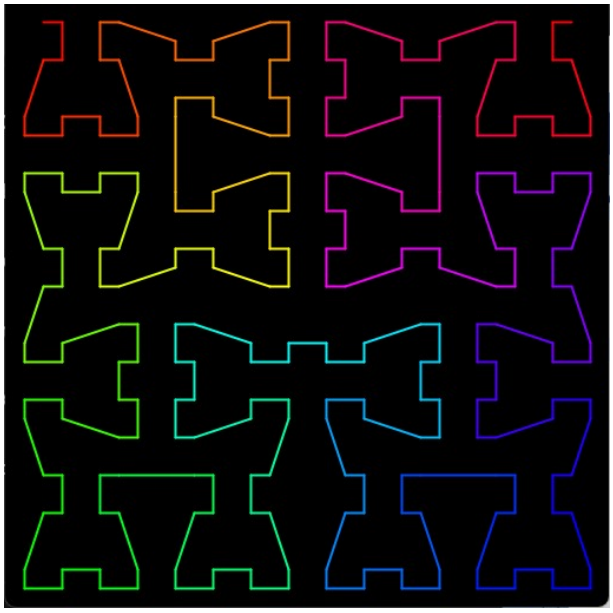


Figure 3.0.8: ”Plier” Curve Level 5

Chapter 4: Related Work

Before embarking on this project, it was crucial for us to understand the fundamentals of constructing fractals and transforming lines in a 2D space. This led us to delve into the existing literature and research on the topic of space-filling curves and fractal art. In this chapter, we present a comprehensive overview of the related work in this field, highlighting the various methods and algorithms used for generating fractal images. Our aim is to provide a clear understanding of the techniques and how they inspired our approach in developing our concurrent visualisation application.

4.1 Fractal

In some special cases, fractals could be considered space-filling curves. According to mathematics definition, fractal is a geometric shape containing detailed structures at arbitrarily small scales, usually having a fractal dimension, in other words, fractals can be thought of as never-ending patterns. For instance, you crush something into countless small pieces of shells, these pieces can be called fractals.

In our daily lives, fractals can be found as snowflakes, tree branching, and lightning,...

Moving on, the fractal dimension is simply a measure of how ”complicated” a self-similar figure is. For example, we have a line with 1-dimensional, a plane such as a square with 2-dimensional or a cube with 3-dimensional. Moreover, the number of dimensional is not just calculated in integer numbers, it also is calculated by decimal numbers, e.g. Sierpinski Triangle has 1.585-dimensional or the Koch Snow Flake has 1.262-dimensional. The real fact is, the fractal dimensional is a made-up concept but it is very useful to construct in modelling.

Let’s consider some interesting about the fractal dimension. Firstly, take an example of a square with 2-dimensional, the edge of the square will be considered as the length L, and the mass will be

considered as the measure M. If we have the scaling factor is $\frac{1}{2}$, the mass scaling factor of the square will be:

$$\frac{1}{4} = (\frac{1}{2})^2 \tag{4.1}$$

Similar to Sierpinski Triangle, suppose if the scaling factor also is $\frac{1}{2}$, and we divide a big Sierpinski triangle to 3 small Sierpinski triangles, we will have the mass scaling factor:

$$\frac{1}{3} = (\frac{1}{2})^D \tag{4.2}$$

Now, D represents the dimension by calculating the above formula, which gives the result of 1.585 by using a calculator. That's why the Sierpinski triangle has a 1.585-dimensional.

One of the most fascinating aspects of fractals is some self-similarity dimensions, which means we can build a shape with lots of copies of itself, for instance, this shape with the 1.668-dimension:

Figure 4.1.1: Flower shape

With 5 smaller copies of itself:

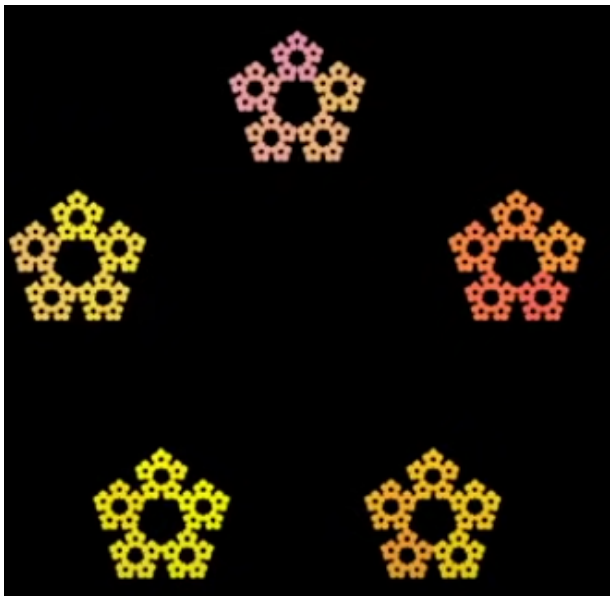


Figure 4.1.2: Separate flower shape

Some examples of fractal curves:

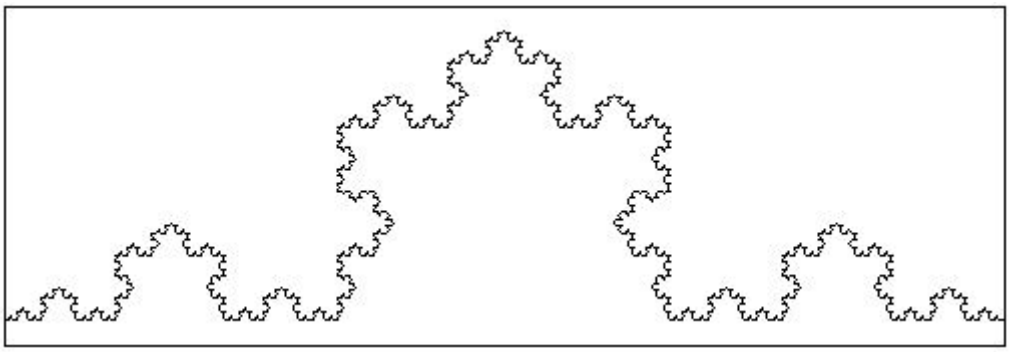


Figure 4.1.3: Snowflake Curve

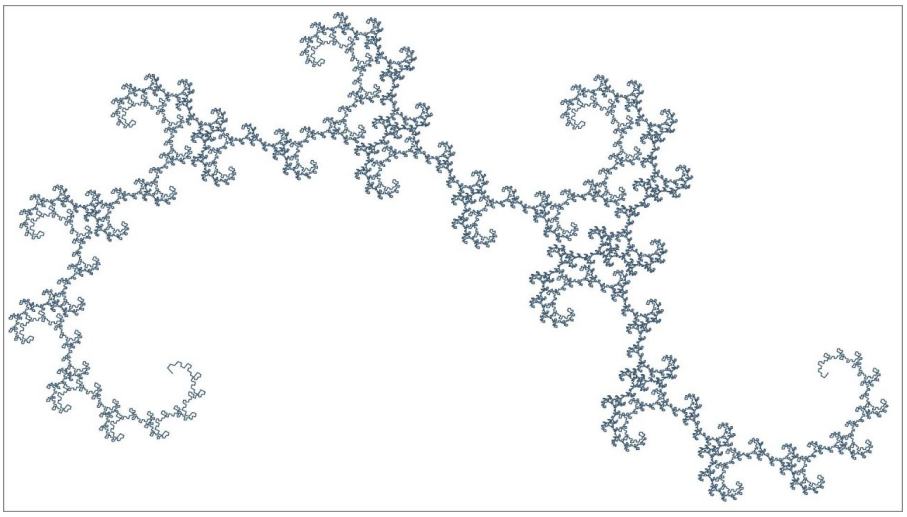


Figure 4.1.4: Dragon Curve

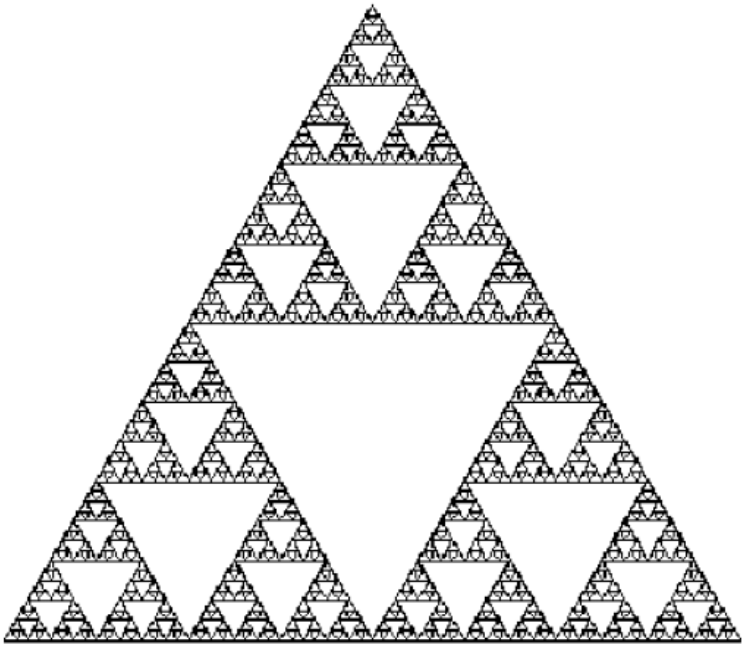


Figure 4.1.5: Sierpinski Curve

4.2 Rotate in 2D plane

When working with fractals, especially the Hilbert curve, understanding 2D transformations, primarily rotation, is essential. A straight line segment can be transformed, rotated, and translated several times to create this space-filling curve. Understanding rotational principles and how they are mathematically represented in matrices enables the exact and accurate manipulation of the curve, producing aesthetically pleasing and complex patterns. Mathematically, a 2D rotation can be represented by a rotation matrix of the form:

$$\begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

Where θ represents the angle of rotation in radians. Two important cases that we have to know is when θ is equal to 90 degrees or -90 degrees (Rotate of two top quadrants of Hilbert curve), the rotation matrix simplifies to:

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}, \text{ respectively}$$

Chapter 5: Algorithms In Pseudocode

Moving to this chapter, the algorithms will be exemplified through pseudocode. There are lots of curves with fascinating animations but we will get started with some fractal examples, which are the base of space-filling curves. After that, we will discuss a special case of the fractal, which is the continuous space-filling curve (Hilbert curve) to help readers to be more insightful about designing curves through coding. Writing algorithms by pseudocode is easier to understand because it demonstrates how the construction works step by step clearly.

5.1 First idea of Algorithm:

We initially used a recursive method to implement the algorithm, drawing the fundamental element first before repeatedly recursing to cover the entire plane. We soon discovered, however, this approach was only appropriate for the creation of fractals, not space-filling curves, and was not appropriate for making animations. As a result, we modified the Iteration method's approach, which offered greater flexibility and better outcomes.

Despite this modification, we were still able to use the recursive approach to produce several intriguing fractals, such as the Sierpinski triangle. The Sierpinski triangle is made by repeatedly splitting a triangle into smaller triangles and filling the resulting gaps with the same pattern. A stunning fractal design is produced by repeating this procedure until the appropriate level of detail is reached.

The Pseudo code of Sierpinski Triangle is represented below:

```
function divide(x,y,len, level,max){
  if level >= max
    draw an equilateral triangle :
```

```
        head up
        left vertex is at x,y
        length = length
else
    increment level
    recursive with x,y is the left vertex of the triangle
    recursive with x,y is the middle of point left side
    recursive with x,y is the middle of point bottom side
}
```

Moving on with another example of a fractal, Koch Snowflake is also an extremely typical type of fractal. Starting with a basic idea, which is also known as the Koch snowflake curve: the line segment is divided into 3 equal parts, then the middle part will be divided into 2 separate parts which are created the original one into an equilateral triangle. As a result, the initial middle part of the line segment will be disappeared and be replaced by the 2 new lines after separating.

Figure 5.1.1: The original line segment

Figure 5.1.2: The Koch Snowflake Curve Level 1

Continuing with the higher level of an original line segment, an initial equilateral triangle will be considered. In an equilateral triangle, each edge of the triangle will be exactly the same way that a line segment is separated.

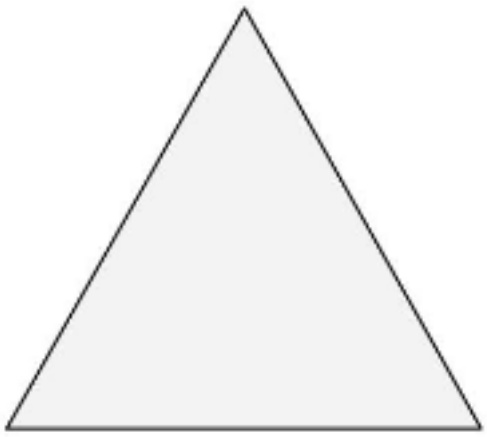


Figure 5.1.3: The initial equilateral triangle

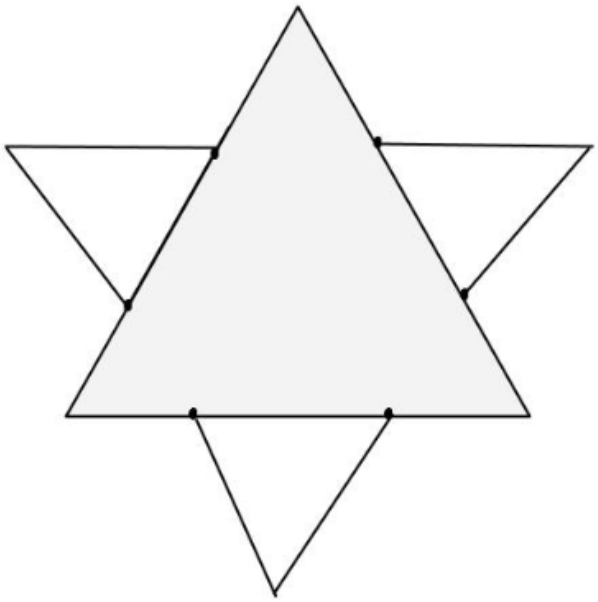


Figure 5.1.4: Koch Snowflake Level 1

And after increasing the level by keeping continuously separated, a full construction of Snowflake is set up:

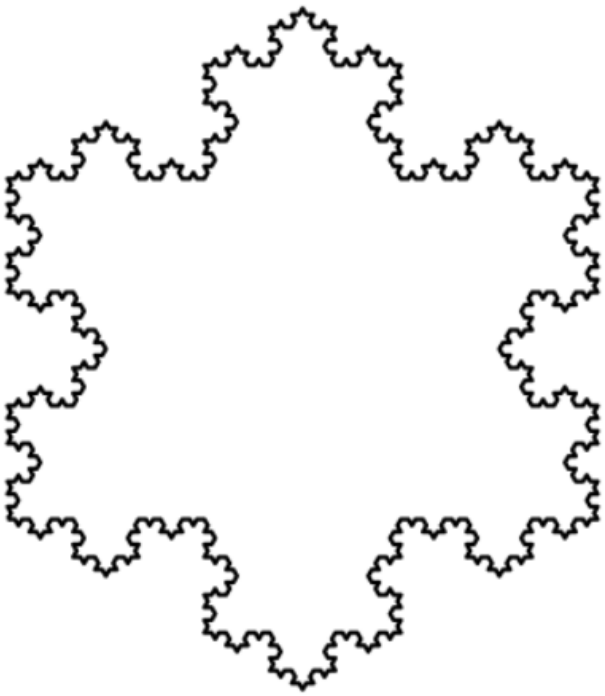


Figure 5.1.5: The image of a high-level Snowflake

Pseudocode:

```
Segment class:
    properties:
        declare 2 vector a, b
    constructor:
        input: PVector a_, PVector b_
    method create:
       create a segment which is divided into 4 parts
        another = PVector.sub(b, a)
        another.div(3)
        a1 = PVector.add(a, another) // part 1
        parts[0] = new Segment(a, a1)
        b1 = PVector.sub(b, another) // part 4
        parts[3] = new Segment(b1, b)
        another.rotate(-PI / 3)
        c1 = PVector.add( a1,another)
```

```
        parts[1] = new Segment(a1, c1) // part 2
        parts[2] = new Segment(c1, b1) // part 3
    method show:
         create a line

Snowflake class:
    property:
        ArrayList<Segment> segments //create an ArrayList of
        Segment to present Koch Snowflake
    method addAll:
        input: Segment[] array, ArrayList<Segment> listofsegment
        for each segment s in an array:
            listofsegment.add(s) // add parts to the ArrayList
    method settings:
        size()
    method setup:
        segments = new ArrayList<>()
        a = new PVector(100, 400)
        b = new PVector(600, 400) // create vector a,b for
        an equi. rectangle
        edge = PVector.dist(a,b) //store distance between a
        and b in edge
        height = edge * (sqrt(3)/2)  //formular in
        an equilateral triangle
        c = new PVector(350,400+height) // create vector c for
        the remaining edge of the triangle
        segment1 = new Segment(a, b)
        segment2 = new Segment(b, c)
        segment3 = new Segment(c, a) //create new Equi.Triangle
        segments.add(segment1)
        segments.add(segment2)
        segments.add(segment3) // add 3 new Segment to the
        segments in the ArrayList

    method draw:
        for each Segment s in segments:
            s.show()
        nextLevel = new ArrayList<Segment>() //create a new arraylist
        to iterate each segment in
        the old ArrayList
```

```
for each Segment s in segments:
    parts = s.create()
    addAll(parts, nextLevel)
segments = nextLevel // For each segment, use the "create"
method to generate four new line segments
and add them to the "nextLevel" ArrayList.
```

5.2 Final idea of Algorithm:

Explain:

The idea is to draw Hilbert Curve iteratively by using the *Hilbert* function to find the position of the node on the Hilbert curve and then use the position to calculate the cartesian coordinates of the node. The algorithm starts with a basic unit of the curve and repeatedly applies a set of rules to generate the next level of the curve. The input of the function is an integer *i* which represents the index of the node and the output return its cartesian coordinates *v*.
To index the curve, we assume that the curve starts at the top left corner and ends at the top right corner. The total index is 4^{level}, and starts from 0. Figure 5.2.1 at the bottom is an example of a *level 3* Hilbert Curve.

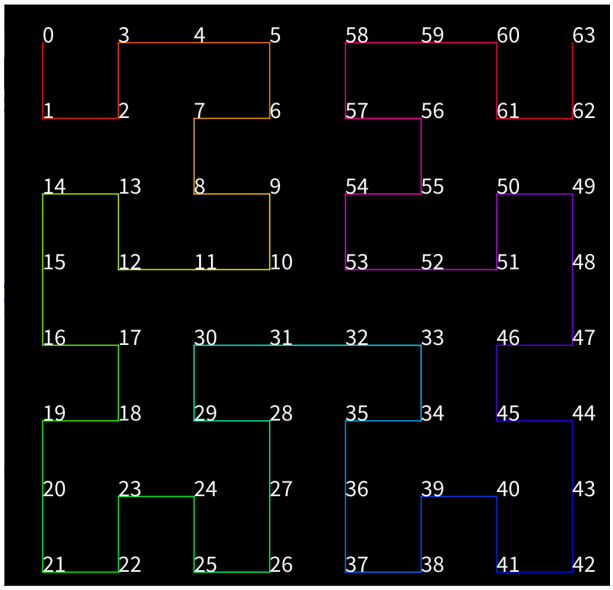


Figure 5.2.1: Hilbert curve 3rd iteration

The expected return of *Hilbert(18)* is *(x:5,y:1)*, of *Hilbert(60)* is *(x:6,y:0)*.

The Function *Hilbert()* uses a bottom-up approach to compute the coordinate of the curve. When we look at the binary representation of the index, we may find that the last two bits represent the position of the node inside the *level = 1* curve, the next two bits represent the position of the node inside the *level = 2* curve, and etc.

The *figure 5.2.2* shows how the algorithm work for *index = 11* in *level = 2* curve. We start by convert

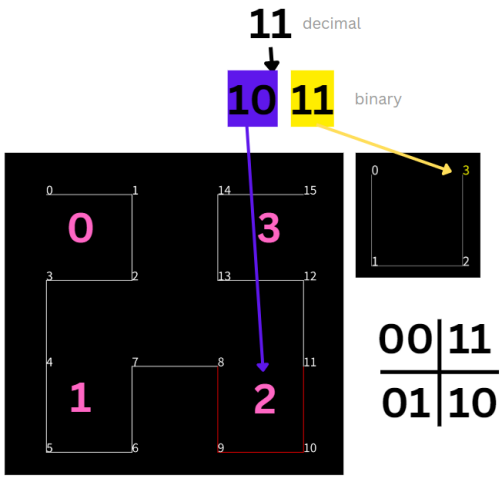


Figure 5.2.2: shifting bit algorithm

the index $11_{decimal}$ into 1011_{binary}. Here, last two bits 11_{binary} represent the position of that point in a *level = 1* curve, which is $3_{decimal}$ represents the cartesian coordinate *(x = 1,y = 0)*. To process the next two bits, we consider *level = 1* curve is the smallest element in *level = 2* curve. Now, we may notice that the next two bits 10_{binary} represent the smaller curve now in quadrant 3. To get the cartesian coordinate, we have to apply the translation $(x : 1; y : 1) * 2$ to point $(x : 1, y : 0)$:

$$(x : 3, y : 2) = (x : 1; y : 1) + (x : 1, y : 0)$$

In the case of doing with a *level = K* curve, the cartesian coordinate of a point :

$$(x, y) \quad \{0 \leq x, y < N\}$$
$$N = 2^{K-1} : maximum\ coordinate\ possible$$

When applying our *Hilbert* algorithm, we can split the whole curve into 4 big cases(figure 5.2.3). Case 2 and case 3 are simple because they are an exact copy of the *level K-1* curve. In case 2 we use the equation:

$$coordinate(K) = coordinates(K-1) + (x : 0, y : N_{K-1})$$

Figure 5.2.3: Four cases of the curve

In case 2 we use the equation:

$$coordinate(K) = coordinates(K-1) + (x : N_{K-1}, y : N_{K-1})$$

The real problem comes when we deal with cases 1 and 4, because they must be rotated to minimize the length of the connecting line.

In case 1, we have to rotate the *K-1* curve to the left by flip the *level K* curve around the 1st diagonal. In case 4, we have to rotate the *level K* curve around the 2nd diagonal(Figure 5.2.4)

In case 1 we use the equation:

$$coordinate(K).x = coordinates(K-1).y$$
$$coordinate(K).y = coordinates(K-1).x$$

In case 4 we use the same equation as in case 1 but also translate the coordinator to the right :

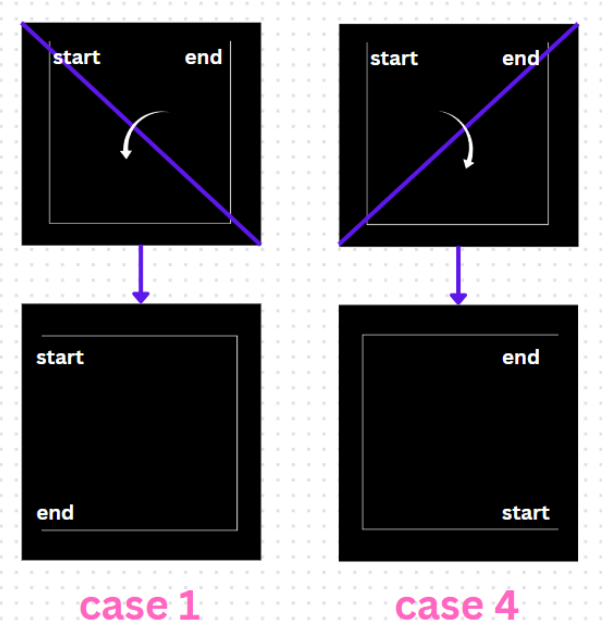


Figure 5.2.4: Hilbert curve 3rd iteration

$$tempVariable = N_{K-1} - 1 - coordinates(K)$$
$$coordinate(K).x = 2 * tempVariable - 1 - coordinates(K).y$$
$$coordinate(K).y = tempVariable$$

Finally, we have a pseudocode in Processing-Java:

Pseudocode:

```
int level = {Level of iteration user want};

int total = total lines of the curve;

Point path[total];

public void settings(){
    size(512,512);
}
void setup(){
```

```
    add color for background
    loop i from 0 to total{
        path[i] = Hilbert(i);
        resize the path in comparison to the real size of the window
    }
}

void draw(){
    rerun draw funcion every tic
    add colour to by line;
    draw all the lines, take data from path[];
}



public Point Hilbert (int i){
    Point[] point = {
        Set up all the points for the first iteration of the curve
    }

    int index = last two bits of i;

    Point v = point[index];

    v = point[index];

    for loop j from 1 to level{
        int length = N(level j);
        shift two bits of i to the left
        index = last two bits of i
        if (index = 1){
            rotate left
        }
        else if (index = 2){
            translate the coordinator
        }
        else if (index = 3){
            translate the coordinator
        }
        else if (index= 4){
```

```
            translate the coordinator
            rotate right
        }
    }
    return v;
}
```

To add animation, we create a variable *count* to track the position of the line and restart that value every time the curve finish drawing - the count reach to the maximum value.

```
int count = 0;
void draw(){
    rerun draw funcion every tic
    add colour to by line;
    draw all the lines, take data from path[];
    restart the animation after the drawing is done;
   increment the count
   if (count >= total line possible) count = 0;
}

}
```

To add the change speed of the animation function, we create a global variable speed and increment the count by speed, each time the function draw run. The program can change the speed to be faster-slower each time user press right-left arrow. We also add the change the level function, when user press up-down arrow, the program will change the level up-down corresponding. When changing the level between the animation, we also have reinitialize all the value of the curve by using class *reInitialize()*.

```
int count = 0;
int speed = 1;
void draw(){
    rerun draw funcion every tic
    add colour to by line;
    draw all the lines, take data from path[];
    restart the animation after the drawing is done;
   increment the count by speed
   if (count >= total line possible) count = 0;
```

```
}

void keyPressed(){
    change the level
        up when the user press upArrow
        down when the user press downArrow
        then reInitialize all the attributes of the curve;
    change the speed of the animation
        faster when the user press left arrow
        slower when the user press right arrow
}
void reinitialize(){
    count = 0;
    loop i from 0 to total{
        path[i] = Hilbert(i);
        resize the path in comparison to the real size of the window
    }
}
}
```

5.3 Other variance of Hilbert curve algorithm:

Every curve that is using the same concept as the Hilbert curve, which can split into 4 smaller cases, can be created by using this algorithm by changing the shape of the first iteration and the way to get the index, the increment length after each time goes up.

For example in the ”Triangle Curve”, there are only 3 points for the 1st iteration. Therefore, the first index and the quadrant are got by modulo i by 3 and the second index is get by dividing i by 3. The connecting line is also omitted by changing the value length inside the loop. The code now will be:

```
public Point tri (int i){
    Point[] point = {
        Set up all the points for the first iteration of the curve
    }

    int index = i%3;
    i = i/3
    Point v = point[index];
```

```
    v = point[index];

    for loop j from 1 to level{
        int length = N(level j-1);

        index = last two bits of i
        if (index = 1){
            rotate left
        }
        else if (index = 2){
            translate the coordinator
        }
        else if (index = 3){
            translate the coordinator
        }
        else if (index= 4){
            translate the coordinator
            rotate right
        }
        shift two bits of i to the left
    }
    return v;
}
```

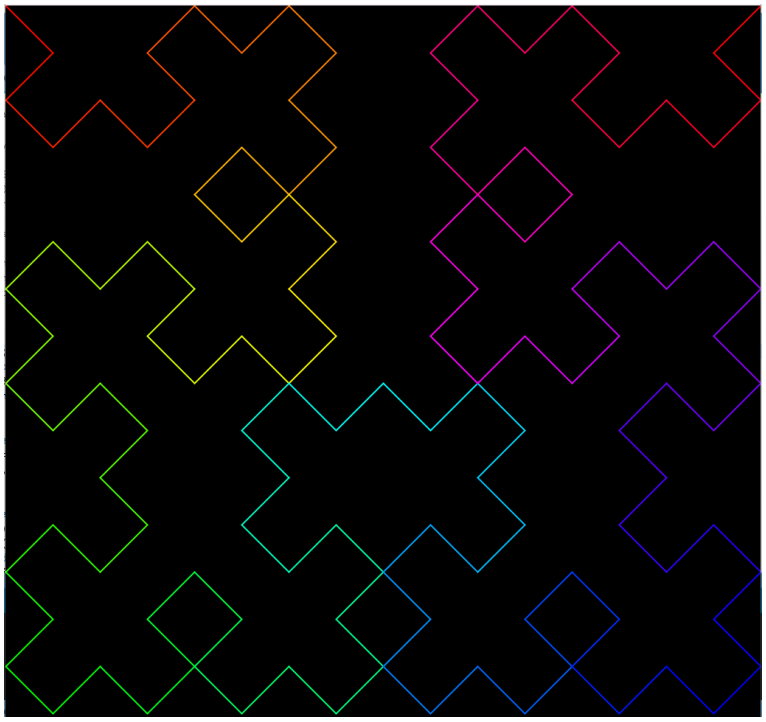


Figure 5.3.1: Triangle Curve Level 4

Chapter 6: Implement Details

The chapter explains the application which is known as the user interface. It allows users to interact with animations of space-filling curves. To run this application, lots of code snippets and libraries must be used. Each separate part has its own crucial function, contributing significantly to the process of designing smooth animations of curves. The structure of the application guides the users through different stages which are described clearly with pictures. In addition, 2 UML diagrams will also be drawn in order to express the actions of the users through the utilizing process and the construction of 7 specific space-filling curves that are generated.

6.1 GUI & Application Structure

GUI is an abbreviation for Graphical User Interface. The simple definition of GUI is an interface which allows users to interact with devices through graphical icons, menus, buttons, cursors or other graphics. It brings information, and actions, that can be taken by the users. Some examples of GUIs that can be considered are computer monitors, smartphones, tablets, gaming systems, etc.

In this space-filling curve project, the common tool for building up the GUI for the code to run is Processing. Using Processing to create a Java project is a good idea because it contains lots of online resources where users can find its commands or functions easily. It also has a great community and clear graphics, generating a friendly and interactable environment during the process.

Implement JavaFX and SceneBuilder kit, FXML files are created to design a beautiful and interactive interface for users to visualize space-filling curve concepts throughout animations from Processing.

+) SceneBuilder: provides a visual interface for a JavaFX application, and has a drag-and-drop interface for adding, positioning and configuring UI components. Additionally, it generates the necessary FXML

code that can integrate into the Java development environment.

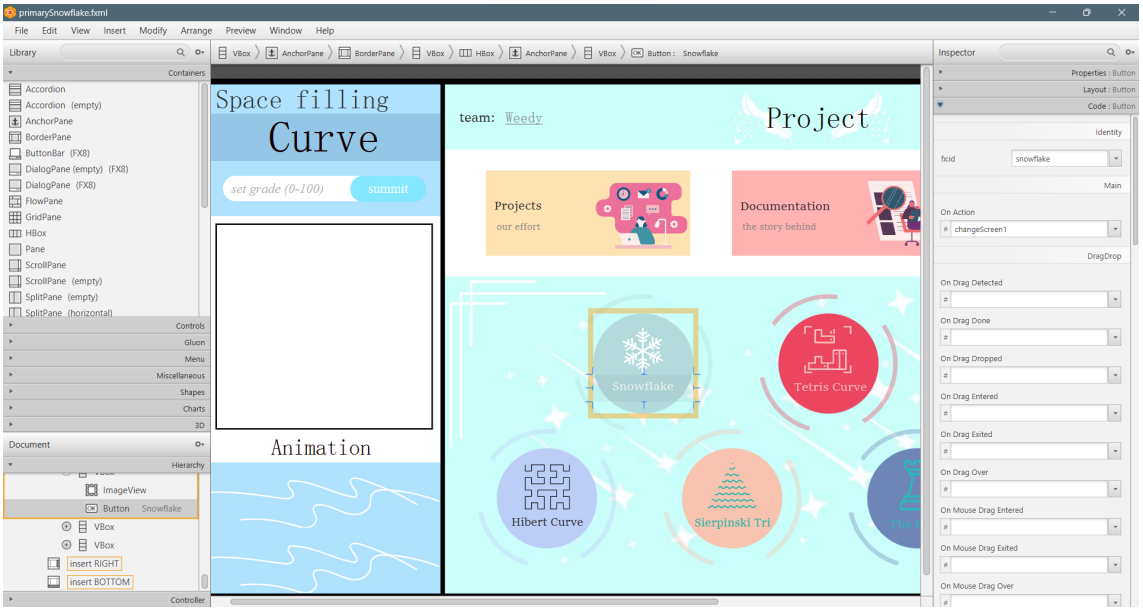


Figure 6.1.1: SceneBuilder

+) NetBeans IDE: provides a comprehensive development environment for creating JavaFX GUI, including features such as a visual layout editor, integrated debugging, and a control system. Additionally, it can integrate with SceneBuilder, making it easier to switch between the visual design interface and code editor, providing JavaFX CSS for styling GUI.

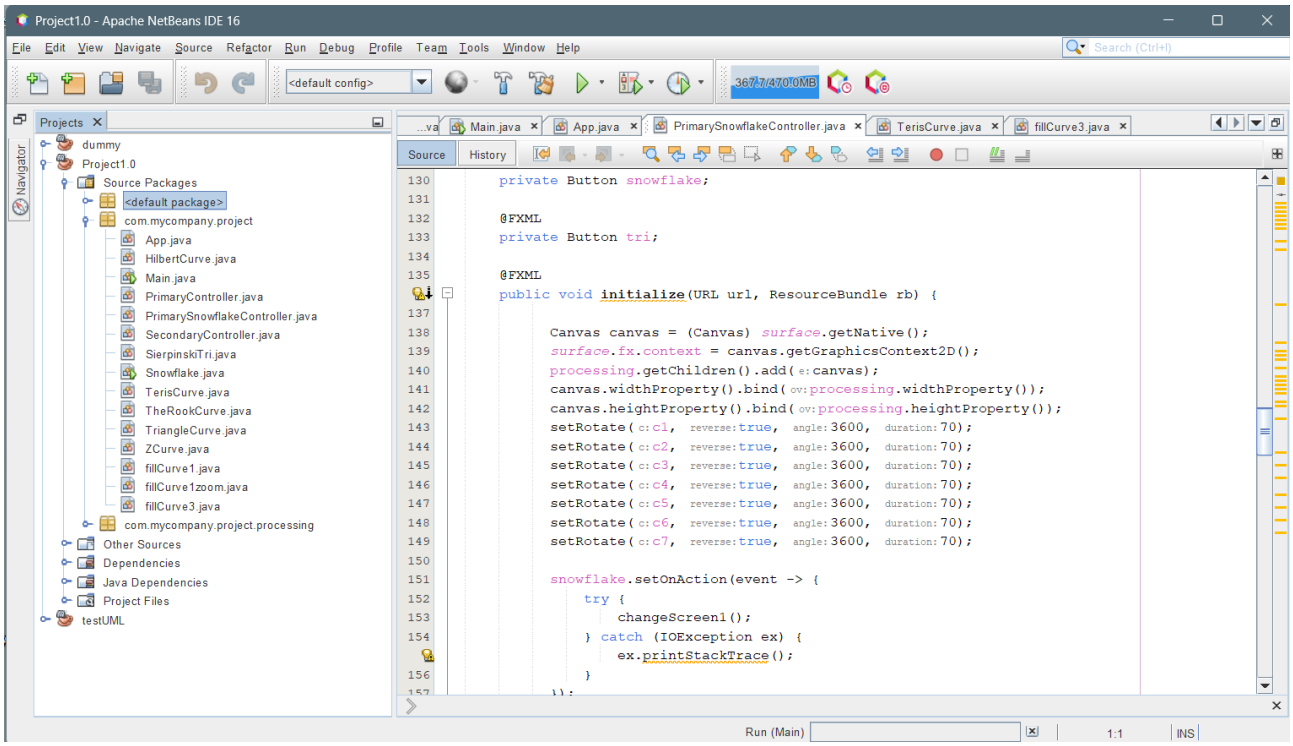


Figure 6.1.2: NetBeans

Project Main Screen:

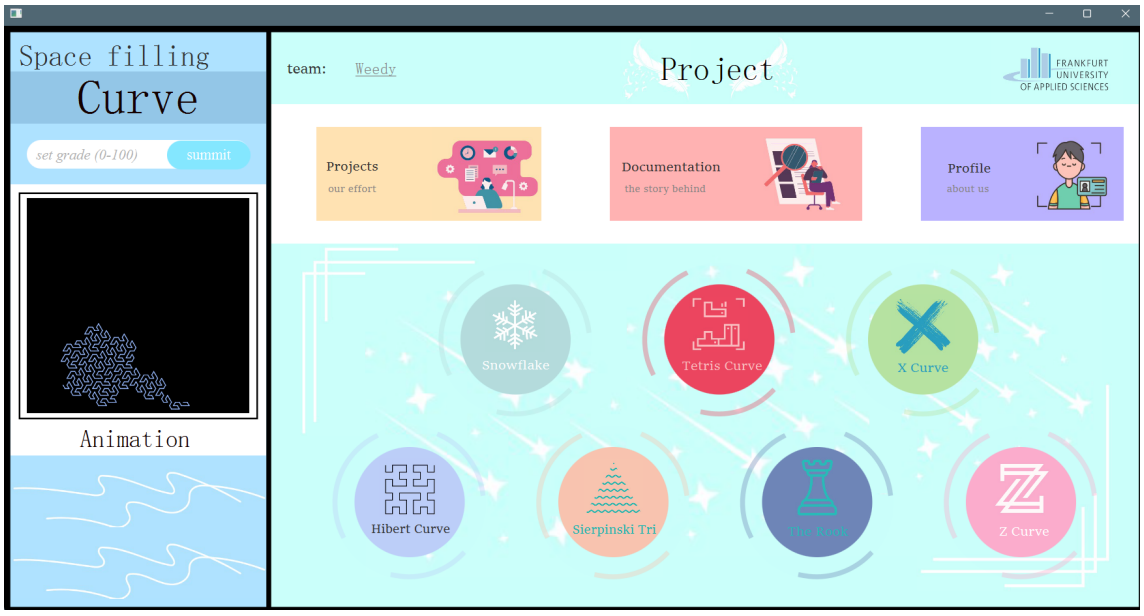


Figure 6.1.3: Project Main Screen

When the user runs the main file, the figure above is the first screen to pop up. In this screen, the users are able to open and interact with 7 different space-filling curves drawing animations. The interface also gives the users the option to navigate between other screens such as ”Documentation” and ”Profile”

+*) Left bar:*
In the left side of the screen, it is the preview animation created by Processing and allocated on the FXML interface along with the title and the set grade bar. Through the set grade bar, users are able to mark their satisfaction on the scale from 0-100 points, by clicking on the submit button and the grade will pop out at the bottom left corner. The previewed animation is set by default as Gosper curve and it will run continuously in the loop as long as the main screen displays.

+*) Right pane:*
General information including the team name, the current screen ”Project”, and the logo of the Frankfurt University institution can be found in the label on top of the right pane.

Right below the label is the navigation bar. Inside the bar, the user can move to their desired screen by clicking on 1 of 3 labels. The ”Project” label will move the user to the above figure screen (so if the user is already in the ”Project” screen, there won’t be any modifications) while the ”Documentation” label and ”Profile” label will move the user to other interfaces.

Finally, the remaining buttons will open up the PApplet interface and run the drawing animations of

the corresponding space-filling curves. This project offers 7 different space-filling curves drawing that user can interact with: Snowflake or Gosper curve, Tetris curve, X curve, Hilbert curve, Sierpinski triangle, Rook curve, Z curve.

Figure 6.1.4: Among right click on the X curve button

Inside the PApplet interface, the user can interact with the interface by clicking the keyboard, this is the operation for each key:

- ”ArrowUp”: increase the complexity of the curve.
- ”ArrowDown”: decrease the complexity of the curve.
- ”ArrowLeft”: decrease the speed of the animation.
- ”ArrowRight”: increase the speed of the animation.
- ”Enter”: close the current animation window.

Figure 6.1.5: Change level by pushing key

The next label of the navigation bar is the ”Document” scene.

Figure 6.1.6: Documentation interface

When the user clicks the documentation label, the root of the previous FXML file will change to

profile FXML file, change the scene to the above figure. In this scene the user have the options see this project or detail documentation by scanning the QR code: +) Project: go to the Github project +) Document: go to the Overleaf PDF document. Moreover, the user can have a preview of the content of the document at the pane inside the right bar.

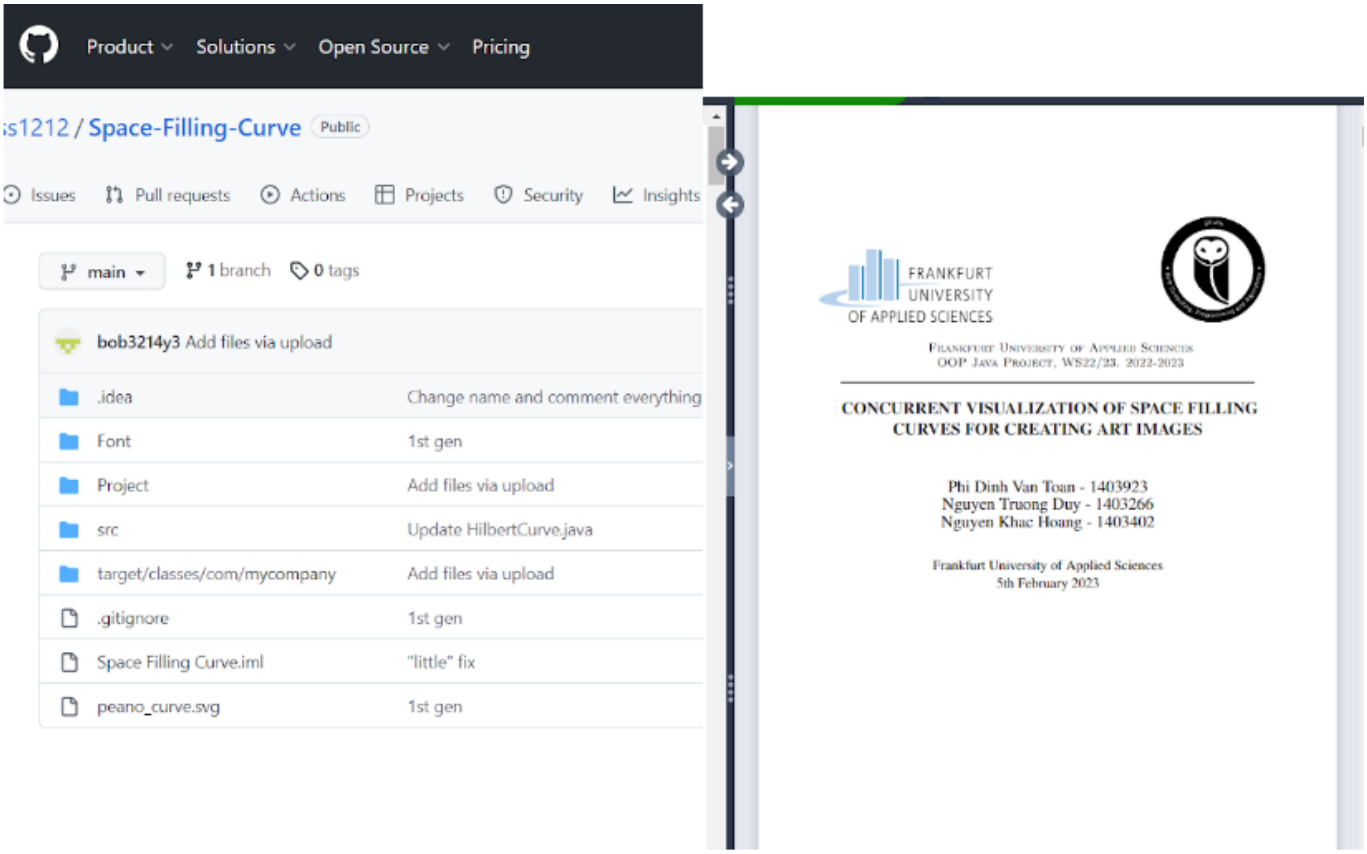


Figure 6.1.7: Preview content of the documentation

The navigation bar is also available in this scene and allows the user to navigate between scenes similarly to Project main Scene, which has been explained above.

Profile scene:

This scene appeared when the user clicks on the ”Profile” label inside the navigation bar. In this scene the user will be able to gather information about the main contribution of the creators along with their contacts information for further request regarding to this project:

+) GUI designer: main creator and designer of this JavaFX interface.
+) Animation: main creator of graphical drawing of multiple space filling curve PApplet.

+) Documentation: main creator for the documentation of this project.

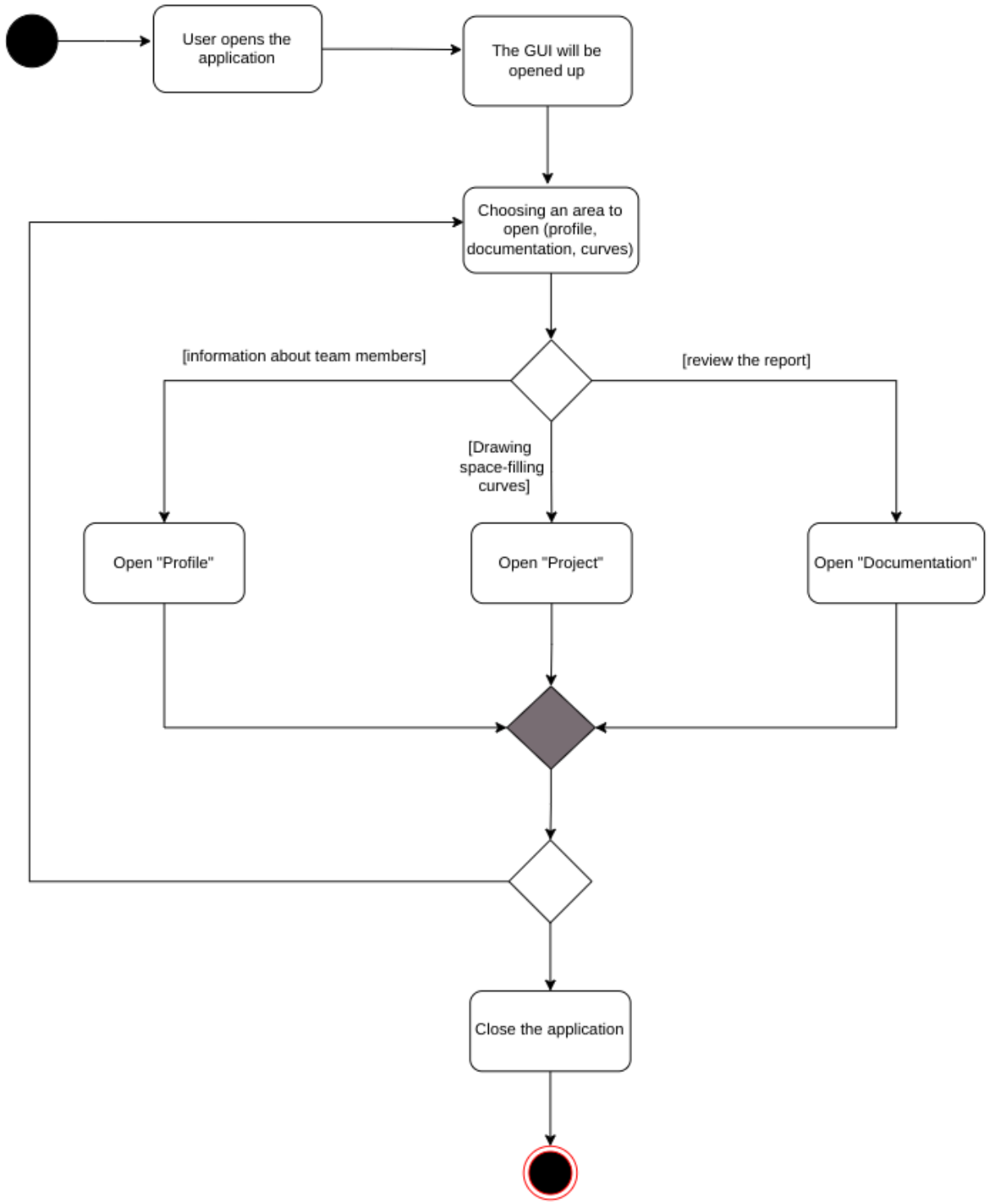
In the ”About Us” panel is the note of creator send to the user and general information about the team.

For the study purpose of this project, the ”Matriculation” is included for identification of creators inside the Frankfurt University campus. Feedbacks and recommendations can be sent directively via the email address of 1 of 3 creators or by the QR code below the ”About Us” panel.

6.2 UML Diagrams

By definition, UML diagrams can be used as a powerful tool to visualize the way that a project works. This type of diagram is very useful when having a project with working in teams. It provides new ideas and the workflow process in order to achieve a goal of a team’s needs.

Overall, a simple UML diagram called an activity diagram will demonstrate about the flow of the process when the user uses the application to run the images. The process will look like this:


User opens the application
The GUI will be opened up
Choosing an area to open (profile, documentation, curves)
[information about team members]
[review the report]
[Drawing space-filling curves]
Open "Profile"
Open "Project"
Open "Documentation"
Close the application

Besides the activity diagram for users to utilize the application, there is another class diagram which illustrates the whole construction of the application. The attributes, the methods of 7 space-filling curves with ”Document Scene”, ”Profile Scene” and ”App”.

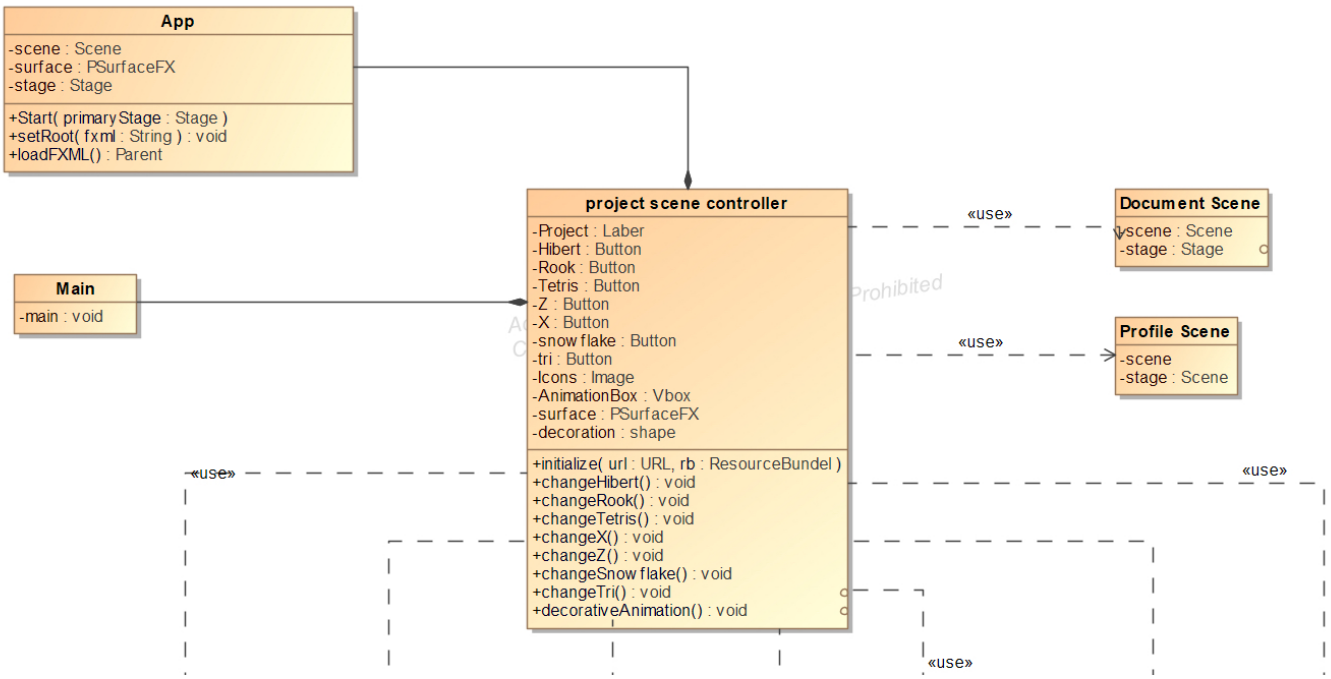


Figure 6.2.1: The Upper Class Diagram

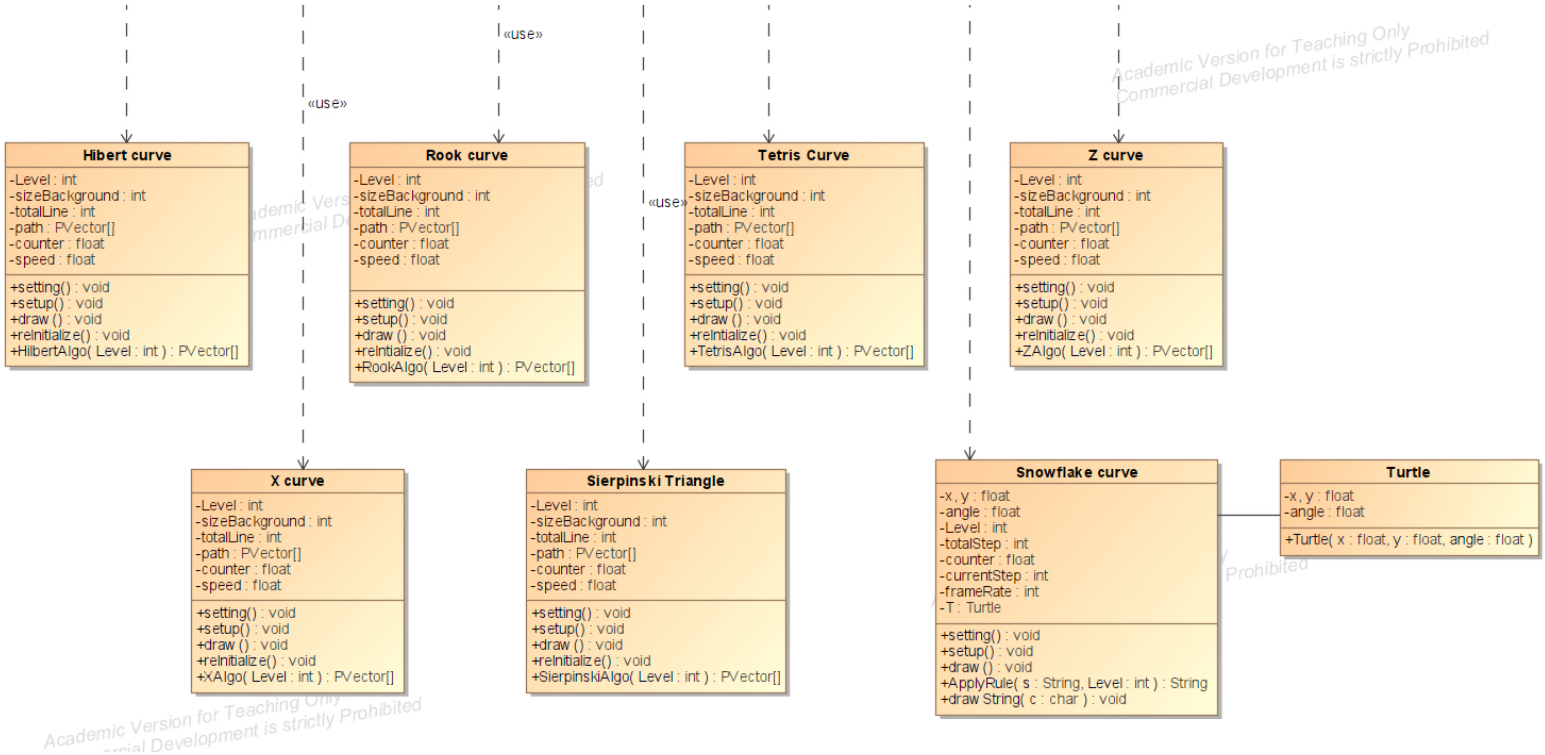


Figure 6.2.2: The Lower Class Diagram

6.3 Used Libaries

In coding 7 space-filling curves, we declare 2 libraries which are PApplet and PVector. The function of the PApplet is to use the *setup()* method to make the code executed when the applet is started. The PVector provides data components and techniques for vector math operations as well as a way to represent vector objects. These 2 libraries can be declared in Processing by:

```
import processing.core.PApplet;
import processing.core.PVector;
```

These libraries stand for Processing - a Java-based language that is widely used for visual arts, design, and data visualization. We choose Processing for this project due to its simplicity and visualization capabilities. Processing has a straightforward syntax, making it easier to learn, and its built-in functions and libraries make it easy to create and visualize our Curve and make animation.

In comparison, JavaFX is a more complex framework for developing user interfaces, requiring a deeper

understanding of programming concepts. While it can be used for data visualization, the simplicity of Processing and powerful visualization library make it a superior choice for generating space-filling curves.

Moving on with the used libraries of GUI, the integration between Javafx and Processing library must be made and implemented correctly for a smooth operation. There are 2 main libraries for this GUI:

+) javafx JDK 19: combination of the JavaFX library and the 19th version of the Java Development Kit, which provides a comprehensive set of tools for developing Java applications with graphical user interfaces.

+) modified processing core (core-10.jar): modified version of the Processing library in which has the modifications to the PSurfaceFX and PGraphicsFX2D classes

Detail library information:

+) java.io.IOException: A checked exception that indicates an IO operation has failed or been interrupted.

+) java.net.URL: Represents a uniform resource locator (URL) and provides access to the parts of a URL and the ability to manipulate URLs.

+) java.util.ResourceBundle: Provides a way to store locale-specific resources that can be loaded by a program at runtime.

+) javafx.animation: Package of the JavaFX library that provides classes and interfaces for animation.

+) javafx.fxml: Package of the JavaFX library that provides a way to declare JavaFX GUI elements in an XML file and load them into a JavaFX application.

+) javafx.scene: Package of the JavaFX library that provides the base for constructing and rendering the JavaFX scene graph.

+) javafx.stage: Package of the JavaFX library that provides the top-level container for all JavaFX applications and is responsible for managing the scene graph.

+) javafx.application: Package of the JavaFX library that provides the entry point for JavaFX applications and the base class for JavaFX applications.

+) processing.javafx.PSurfaceFX: PSurfaceFX class is used in Processing to represent the JavaFX Surface object.

+) processing.core.PSurface: The PSurface interface is implemented by a Surface object, used in

Processing to describe the way an output window is drawn to the scene.

6.4 Hilbert Curve Code Snippets

Below is the snippet of code to generate a Hilbert Curve continuously, a space-filling curve. It is one of the first programs we successfully used the algorithm mentioned above, and we also used this program as a template for subsequent programs.

When initializing attributes of the plane, we set the total vertex of the curve as 4^{10} instead of their actual value as 4^{level} because when changing the level, the total vertex of the curve changes also. In order to make the animation smooth, we decided to initialize an Array of all vertex with a size as big as the maximum level's curve in the beginning (which is 10).

The colour filled in the Curve is highest in brightness and saturation, and the *hue* value changes from 0 to 360 degrees corresponding to the index of the line. 0 corresponds to the beginning of the line and 360 corresponds to the end of the line.

The setup - settings function is called once when the program is started, we include a for loop inside to generate all the cartesian coordinates of the points to reduce the overhead for the program when drawing and making animation. Here, we also set the colour of the background, colour mode and attributes of the panel.

The draw function taken from Processing is the function that continuously executes the code inside it. It runs "as fast as possible" so the time that is needed to finish the Curve "practically" can't be calculated because it is based on the speed of the processor. Said it "practically" can't be calculated because "theoretically" there is a function in Processing named *frameRate()* to specify the number of times *draw()* runs each second, but it seems does not to work, except when we put in a very low value, like 5 frame/second or 10 frame/second. They will look very slow - broken if we do so, so we decided not to use that function and let the code run as fast as possible to make a smooth animation.

The *reInitialize()* function is needed here because as mentioned above, the setup function only runs once when the program starts. When you change the level of the Curve, all the attributes change. The *reInitialize()* here is to generate all the cartesian coordinates again according to the new level.

To apply the algorithm, or in specific, get the last two values of the coordinate of a point when in binary form. We use "&" operator to get the last two values of the coordinate in binary form and ">>>" operator to shift the binary form two values to the right.

It does not have GUI, the user can only interact with the program through the keyboard. The program

has 5 functions ”ArrowUp”, ”ArrowDown”, ”ArrowRight”, ”ArrowLeft” and ”Enter” with the functions had been mentioned on the above GUI section.

Below is the snippet of the code to generate the Hilbert Curve.

```
package com.mycompany.project;
import processing.core.PApplet;
import processing.core.PVector;
public class HilbertCurve extends PApplet{
   int level =2; // number of Iteration
   int N = (int) pow(2,level); // size of the background (for drawing)
   int total = N*N; // total line of drawing
   PVector[] path = new PVector[(int)(pow(4,10))];

   public void settings(){
      size(512,512); // set the size of the image
      smooth();
   }
   public void setup(){
      noStroke();
      colorMode(HSB,360,255,255);
      background(0);
      for (int i = 0; i < total; i++) {
         path[i] = Hilbert(i);
         float len = width/(float)N ;
         path[i].mult(len);
         path[i].add(len/2,len/2);
      }
      surface.setTitle("Hilbert Curve!"); // name of the window
//      surface.setLocation(100, 100);
      surface.setAlwaysOnTop(true);
//      surface.getNative();
   }

   float counter = 0; // this one is to change the position(stage
      :V)(iteration) of the animation
   float speed = 0.5F; // this one is to change the speed of the
      animation
   public void draw(){
      background(0);
      stroke(255);
      strokeWeight(2);
      for (int j = 1; j < counter; j++) {
```

```
      /*
       * Set the color of the image
       * the j change from 0 to 360 directly proportional to 0->
          path.lenth
       * or (j/(path.length))*361-1
       * */
      float hue = map (j, 0 , total,0,360);
      stroke(hue,255,255); // set the color for the line
      line(path[j].x,path[j].y,path[j-1].x,path[j-1].y); // draw
   }

   counter+= speed; // using this one to increase speed of the
      animation
   if (counter >= total){
      counter = 0;
   } // reset the counter (animation) when the picture is finish
   String shown_speed = String.format("%.2f",(speed));
   text("Speed:" + shown_speed,0,60);
   text("Level: "+ level,0,40);



}
/*
 * recalculate all the value of the curve
 */
public void reInitialize(){
   counter =0; // restart the counter


   N = (int)pow(2,level);
   total = N*N;
   for (int i = 0; i < total; i++) {
      path[i] = Hilbert(i);
      float len = (width /(float)N) ;
      path[i].mult(len);
      path[i].add(len/2,len/2);
   }
}
public void keyPressed(){
   if (key == CODED){

      if (keyCode == UP){
```

```
            if(level <= 9){
            level++;
            reInitialize();
            }
         }
         if (keyCode == DOWN){
            if (level >1){
            level --;
            reInitialize();
            }
         }
         if (keyCode == LEFT){
            speed = (float) (speed * 0.9);
         }
         if (keyCode == RIGHT){
            speed = (float) (speed * 1.1);
         }

      }
      if (key == ENTER){
         this.getSurface().pauseThread();
         this.getSurface().setVisible(false);
      }
   }
   // this one run when a keyboard is pressed
   public PVector Hilbert (int i){
      /* in here we consider all the plane is just a grid
       * for example
       *  in the 1st level is 2*2
       *  in the 2nd level is 4*4
       * `...
       * Input: index of the node
       * */
      PVector[] point = {
            new PVector(0,0),
            new PVector(0,1),
            new PVector(1,1),
            new PVector(1,0)
      };
      int index = i &3; // get the quadrant
      PVector v = point[index];

      for (int j = 1; j < level; j++) {
```

```
            int len = (int)pow(2,j);
            i = i >>>2;
            index = i & 3;
            if (index == 0 ){ // roltate left
               float temp = v.x;
               v.x = v.y;
               v.y = temp;
            }
            else if (index == 1){
               v.y += len;
            }
            else if (index == 2){
               v.x +=len;
               v.y +=len;
            }
            else{ // roltate right
               float temp = len - 1 - v.x; // len now =
               v.x = 2* len-1-v.y ;
               v.y = temp;
            }
         }
         return v;
      }

}
```

6.5 Code Snippets For Others Curve

We also generate other curves, by changing some constants and some methods of the Algorithm such as:

6.5.1 Plier Curve

For this curve, we only need to change the first iteration of the curve

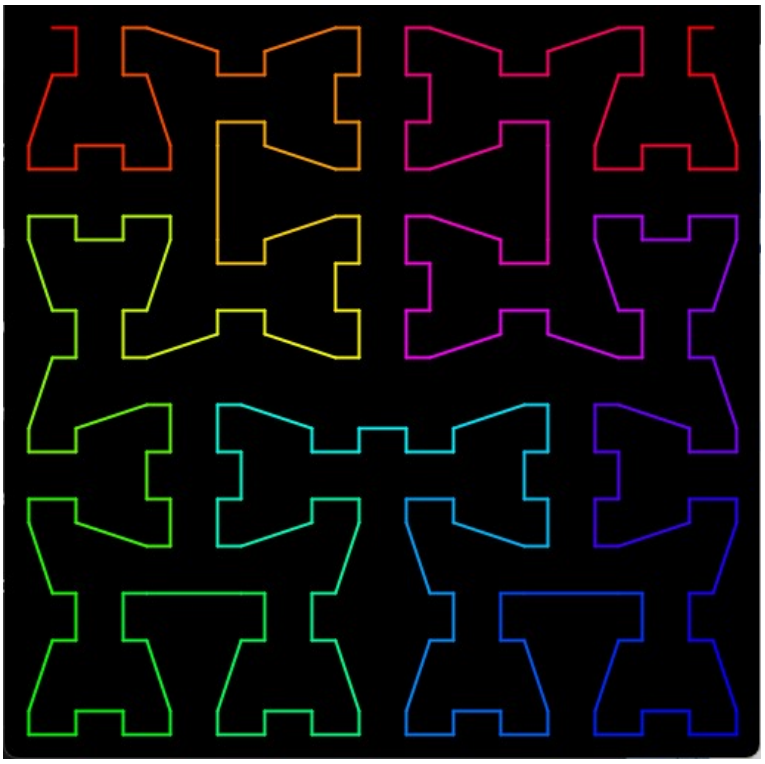


Figure 6.5.1: Plier Curve

```
public PVector Plier(int i){
   PVector[] point = {
         new PVector(0,0.5F),
         new PVector(0,1),
         new PVector(1,1),
         new PVector(1,0.5F)
   };
   int index = i &3;
   PVector v = point[index];

   for (int j = 1; j < level; j++) {
      int len = (int)pow(2,j);
      i = i >>>2;
      index = i & 3;
      if (index == 0 ){ // roltate left
         float temp = v.x;
         v.x = v.y;
         v.y = temp;
      }
```

```
        else if (index == 1){
            v.y += len;
        }
        else if (index == 2){
            v.x +=len;
            v.y +=len;
        }
        else{ // roltate right
            float temp = len - 1 - v.x; // len now =
            v.x = 2* len-1-v.y ;
            v.y = temp;
        }
    }
    return v;
}
```

6.5.2 ”Triangle Curve”

This one we created by accident while trying to change the number of vertex of the first iteration of the curve.

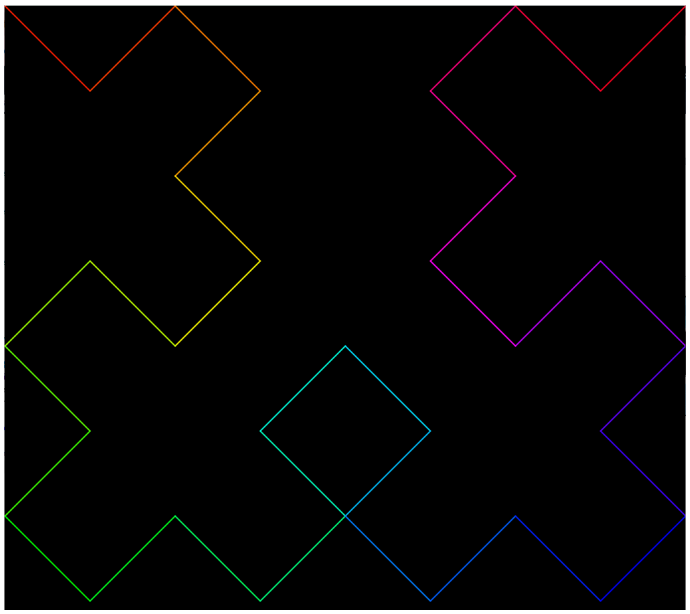


Figure 6.5.2: Triangle Curve

To create this ”Triangle Curve”, we have to change not only something about the process of the first recursion but also the attribute of a curve.

The initialisation of the curve has a little change in total and size of the *path* array.

```
int level =1 ;
int N = (int)pow(2,level);
int total = 3 * (int)pow(4,level-1);
PVector[] path = new PVector[3 * (int)pow(4,10)];
```

The algorithm also changes the way to index the first recursion.

```
    public PVector Tri(int i){

    PVector[] point = {
          new PVector(0,0),
          new PVector(0.5F,0.5F),
          new PVector(1,0)
    };
    int newIndexMod = i %3;
    i = i /3;
    PVector v = point[newIndexMod];

    for (int j = 1; j < level; j++) {
       int len = (int)pow(2,j-1); // now = 1,2

       int newIndexDiv = i & 3;
       if (newIndexDiv == 0) {
          float temp = v.x;
          v.x = v.y;
          v.y = temp;
       }
       else if (newIndexDiv ==1) {
          v.y += len;
       }

       else if (newIndexDiv ==2 ) {
          v.x +=len;
          v.y +=len;
       }
       else {
          float temp = len - v.x; // len now = 1 , 2
          v.x = 2* len-v.y ;
```

```
            v.y = temp;

         }
         i = i >>>2;
      }
      return v;
   }
```

6.6 GUI Code Snippets

App class: to run the JavaFX application:

```
  package com.mycompany.project;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

import java.io.IOException;
import processing.javafx.PSurfaceFX;

/**
 *
 * App class to run the JavaFX application
 */
public class App extends Application {

// Variables to hold the surface, scene and stage
   public static PSurfaceFX surface;
   public static Scene scene;
   public static Stage stage;

   /**
    *
    * Start method to set up the scene and display the primary stage
    *
    * @param primaryStage The main window of the application
    *
```

```
     * @throws Exception
     */
    @Override
    public void start(Stage primaryStage) throws Exception {

// Load the FXML file and set it as the root node of the scene
        FXMLLoader loader = new
            FXMLLoader(getClass().getResource("Project.fxml"));
        Parent root = loader.load();

// Set the stage in the ProjectController class
        ProjectController.stage = primaryStage;

// Create the scene and set its size
        scene = new Scene(root, 1400, 720);

// Set the scene in the primary stage and display it
        primaryStage.setScene(scene);
        primaryStage.show();

// Set the stage in the PSurfaceFX class and the ProjectController class
        surface.stage = primaryStage;
        ProjectController.stage = primaryStage;
    }

    /**
     *
     * Method to change the root node of the scene
     *
     * @param fxml The name of the FXML file to be loaded
     * @throws IOException
     */
    static void setRoot(String fxml) throws IOException {
        scene.setRoot(loadFXML(fxml));
    }

    /**
     *
     * Method to load the FXML file and return its root node
     *
     * @param fxml The name of the FXML file to be loaded
     * @return The root node of the FXML file
     * @throws IOException
```

```
     */
    private static Parent loadFXML(String fxml) throws IOException {
        FXMLLoader fxmlLoader = new FXMLLoader(App.class.getResource(fxml
            + ".fxml"));
        return fxmlLoader.load();
    }
}
```

fillCurve1 class: default animation in project Main Scene

```
    package com.mycompany.project;

import java.util.Stack;

import com.mycompany.project.fillCurve3.Turtle;

import javafx.application.Application;
import javafx.scene.canvas.Canvas;
import processing.core.PApplet;
import static processing.core.PApplet.cos;
import static processing.core.PApplet.pow;
import static processing.core.PApplet.radians;
import static processing.core.PApplet.sin;
import processing.core.PSurface;
import processing.javafx.PSurfaceFX;

/**
 *
 * default animation inside the Project main Scene
 */

public class fillCurve1 extends PApplet {
    /*
     * default setting
     */
    float x, y;
    float angle;
    float depth = 4;
    int totalSteps;
    int currentStep;

    int frameRate = 300; // set the initial frame rate
```

```
/*
 * Initializes the graphics surface for the animation.
 * creates an instance of the PSurfaceFX class, which
 * represents a Processing surface within a JavaFX environment,
 * and sets it as the surface for the animation.
 */
@Override
protected PSurface initSurface() {
   g = createPrimaryGraphics();
   PSurface genericSurface = g.createSurface();
   PSurfaceFX fxSurface = (PSurfaceFX) genericSurface;

   fxSurface.sketch = this;

   App.surface = fxSurface;
   ProjectController.surface = fxSurface;

   new Thread(new Runnable() {
      public void run() {
         Application.launch(App.class);
      }
   }).start();

   while (fxSurface.stage == null) {
      try {
         Thread.sleep(5);
      } catch (InterruptedException e) {
      }
   }

   this.surface = fxSurface;
   Canvas canvas = (Canvas) surface.getNative();

   return surface;

}

// Configures the size of the graphics window and enables
   anti-aliasing.
public void settings() {
   size(800, 800, FX2D);
   smooth();
```

```
}

/*
 * Sets up the initial conditions for the animation,
 * such as the background color, angle, starting position,
 * and depth of the fractal. It also starts the JavaFX application.
 */
public void setup() {
   // noStroke();
   // colorMode(HSB,360,255,255);
   ProjectController.p = this;
   background(0);
   angle = radians(60);
   x = 180;
   y = height + 97;
   translate(x - 300, y - 300);
   String curveString = applyRules("A", depth);
   totalSteps = curveString.length();
   currentStep = 0;
   frameRate(frameRate); // set the frame rate
}

/*
 * draw() method: Draws the animation on the graphics window. It uses
    the
 * applyRules() method to
 * generate the curve string, then iteratively calls the drawString()
    method to
 * draw the characters of the string.
 * The background color is reset and the position is reset to the
    center of the
 * window when the animation is complete.
 */
public void draw() {

   if (currentStep < totalSteps) {
      // stroke(255);
      // strokeWeight(2);
      String curveString = applyRules("A", depth);
      drawString(curveString.charAt(currentStep));
      currentStep++;
   } else {
```

```
      currentStep = 0;
      background(0);
      x = 180;
      y = height + 97;
      translate(x - 300, y - 300); // reset the origin to the center
         of the frame

   }
}

/*
 * Responds to keyboard inputs,
 * allowing the user to increase or decrease the frame rate of the
    animation by
 * pressing the '+' or '-' keys, respectively
 */

public void keyPressed() {
   if (key == '+') {
      frameRate(frameRate += 10); // increase the frame rate when the
         "+" key is pressed
   }
   if (key == '-') {
      frameRate(frameRate -= 10); // decrease the frame rate when the
         "-" key is pressed
   }
}

/*
 * Generates the curve string for the fractal animation by applying a
    set of
 * rules to a starting string.
 * The depth of the fractal is specified as an argument drawString()
    method:
 * Draws the characters of the curve string on the graphics window
 */
public String applyRules(String s, float depth) {
   if (depth == 0) {
      return s;
   }
   String result = "";
   for (int i = 0; i < s.length(); i++) {
      char c = s.charAt(i);
```

```
      if (c == 'A') {
         result += "A-B--B+A++AA+B-";
      } else if (c == 'B') {
         result += "+A-BB--B-A++A+B";
      } else {
         result += c;
      }
   }
   return applyRules(result, depth - 1);
}

/*
 * Draws the characters of the curve string on the graphics window.
 * The method uses the angle and length of the line to determine the
    endpoints
 * of the line segments,
 * and the '+' and '-' characters to adjust the angle of the line
 */
public void drawString(char c) {
   Stack<Turtle> stack = new Stack<Turtle>();
   stroke(153, 187, 255);
   strokeWeight((float) 1);

   float len = pow((float) 2.4, (float) (depth - 2.2));
   if (c == 'A' || c == 'B') {
      line(x - width / 2, y - height / 2, x - width / 2 + len *
         cos(angle), y - height / 2 + len * sin(angle));
      x += len * cos(angle);
      y += len * sin(angle);
   } else if (c == '+') {
      angle += radians(60);
   } else if (c == '-') {
      angle -= radians(60);
   } else if (c == '|') {
      angle += radians(180);
   } else if (c == '[') {
      stack.push(new Turtle(x, y, angle));
   } else if (c == ']') {
      Turtle previous = stack.pop();
      x = previous.x;
      y = previous.y;
      angle = previous.angle;
   }
```

```
    }

    class Turtle {

        float x, y;
        float angle;

        Turtle(float x, float y, float angle) {
            this.x = x;
            this.y = y;
            this.angle = angle;
        }
    }
}
```

ProjectController class: controller for Project scene

```
    package com.mycompany.project;

import java.io.IOException;
import java.net.URL;
import java.util.ResourceBundle;
import javafx.animation.RotateTransition;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.canvas.Canvas;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;
import processing.core.PApplet;
import processing.javafx.PSurfaceFX;

/**
 * FXML Controller class
 *
 * @author ACER
 */

public class ProjectController implements Initializable {

    public static PSurfaceFX surface; // a static variable for the
```

```
   Processing surface
public static fillCurve1 p; // a static variable for the Processing
   sketch
protected static Stage stage; // a static variable for the stage of
   the application

@FXML
private Button Hibert; //button to run the HilbertCurve sketch

@FXML
private Button Rook;//button to run the RookCurve sketch

@FXML
private Button Tetris;//button to run the Tetris sketch

@FXML
private Button X;//button to run the XCurve sketch

@FXML
private Button Z;//button to run the ZCurve sketch

//circles to be rotated
@FXML
private Circle c1;

@FXML
private Circle c2;

@FXML
private Circle c3;

@FXML
private Circle c4;

@FXML
private Circle c5;

@FXML
private Circle c6;

@FXML
private Circle c7;
```

```
@FXML
private Button documentationButton; //button to change to the
   documentation screen

@FXML
private Button profileButton; //button to change to the profile screen

@FXML
private StackPane processing; // a stack pane for displaying the
   Processing sketch

@FXML
private Button snowflake; //button to run the SnowflakeCurve sketch

@FXML
private Button tri; //button to run the TriangleCurve sketch

/*
initializes the class and sets the properties of the Processing
surface and the circles to be rotated. It also sets up event handlers
   for each button
to change the screen to a different Processing sketch or FXML file.
 */
@FXML
@Override
public void initialize(URL url, ResourceBundle rb) {

   Canvas canvas = (Canvas) surface.getNative();
   surface.fx.context = canvas.getGraphicsContext2D();
   processing.getChildren().add(canvas);
   canvas.widthProperty().bind(processing.widthProperty());
   canvas.heightProperty().bind(processing.heightProperty());
   setRotate(c1, true, 3600, 70);
   setRotate(c2, true, 3600, 70);
   setRotate(c3, true, 3600, 70);
   setRotate(c4, true, 3600, 70);
   setRotate(c5, true, 3600, 70);
   setRotate(c6, true, 3600, 70);
   setRotate(c7, true, 3600, 70);

   snowflake.setOnAction(event -> {
      try {
         changeScreen1();
```

```
    } catch (IOException ex) {
    }
});

Hibert.setOnAction(event -> {
    try {
        changeScreen2();
    } catch (IOException ex) {
    }
});

Rook.setOnAction(event -> {
    try {
        changeScreen3();
    } catch (IOException ex) {
    }
});

X.setOnAction(event -> {
    try {
        changeScreen4();
    } catch (IOException ex) {
    }
});

Z.setOnAction(event -> {
    try {
        changeScreen5();
    } catch (IOException ex) {
    }
});

tri.setOnAction(event -> {
    try {
        changeScreen6();
    } catch (IOException ex) {
    }
});
Tetris.setOnAction(event -> {
    try {
        changeScreen7();
    } catch (IOException ex) {
    }
```

```
        });

        profileButton.setOnAction(event -> {
            try {
                changeProfile();
            } catch (IOException ex) {
            }
        });

        documentationButton.setOnAction(event -> {
            try {
                changeDocumentation();
            } catch (IOException ex) {
            }
        });

    }

    /*
    methods to change the screen to the corresponding Processing sketch.
     */
    @FXML
    private void changeScreen1() throws IOException {

//      surface.pauseThread();
        PApplet sketch = new fillCurve3();
        String[] processingArgs = {""};
        PApplet.runSketch(processingArgs, sketch);

    }

    @FXML
    private void changeScreen2() throws IOException {
//      surface.pauseThread();
        PApplet sketch = new HilbertCurve();
        String[] processingArgs = {""};
        PApplet.runSketch(processingArgs, sketch);

    }

    @FXML
    private void changeScreen3() throws IOException {
        PApplet sketch = new TheRookCurve();
```

```
    String[] processingArgs = {""};
    PApplet.runSketch(processingArgs, sketch);
}

@FXML
private void changeScreen4() throws IOException {
    PApplet sketch = new TriangleCurve();
    String[] processingArgs = {""};
    PApplet.runSketch(processingArgs, sketch);

}

@FXML
private void changeScreen5() throws IOException {
    PApplet sketch = new ZCurve();
    String[] processingArgs = {""};
    PApplet.runSketch(processingArgs, sketch);

}

@FXML
private void changeScreen6() throws IOException {
    PApplet sketch = new SierpinskiTri();
    String[] processingArgs = {""};
    PApplet.runSketch(processingArgs, sketch);

}

@FXML
private void changeScreen7() throws IOException {
    PApplet sketch = new TerisCurve();
    String[] processingArgs = {""};
    PApplet.runSketch(processingArgs, sketch);
}

/*
 methods to change the screen to the profile or documentation FXML
     files
 */
@FXML
private void changeProfile() throws IOException {
    App.setRoot("Profile");
}
```

```
    @FXML
    private void changeDocumentation() throws IOException {
        App.setRoot("documentation");
    }

    /*
     a method to rotate a circle by a given angle over a given number of
         cycles.
     */
    private void setRotate(Circle c, boolean reverse, int angle, int
        duration) {
        RotateTransition rt = new
            RotateTransition(Duration.seconds(duration), c);
        rt.setAutoReverse(reverse);
        rt.setByAngle(angle);
        rt.setDelay(Duration.seconds(0));
        rt.setRate(3);
        rt.setCycleCount(18);
        rt.play();

    }
}
```

DocummentController class: controller for Documment scene

```
package com.mycompany.project;

import java.io.IOException;
import java.net.URL;
import java.util.ResourceBundle;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;

/**
 *
 * @author ACERsss
 *
 */
public class DocummentController implements Initializable {
```

```
    @FXML
    private Button ProjectButton;

    @FXML
    private TextArea info;

    @FXML
    private Button profileButton;

    @FXML
    @Override
    public void initialize(URL url, ResourceBundle rb) {

        //PSurface surface = new PSurfaceJFX(sketch1);
        info.setEditable(false);
        info.setWrapText(true);
        info.isWrapText();

        ProjectButton.setOnAction(event -> {
            try {
                changeProjects();
            } catch (IOException ex) {
            }
        });

        profileButton.setOnAction(event -> {
            try {
                changeProfile();
            } catch (IOException ex) {
            }
        });

    }

    @FXML
    private void changeProjects() throws IOException {
        App.setRoot("Project");
    }
     @FXML
    private void changeProfile() throws IOException {
    App.setRoot("Profile");
}
}
```

ProfileController class: controller for Profile scene

```
   package com.mycompany.project;

import java.io.IOException;
import java.net.URL;
import java.util.ResourceBundle;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;

public class ProfileControler implements Initializable {

   @FXML
   private Button ProjectButton;

   @FXML
   private TextArea info;

   @FXML
   private Button documentationButton;

   @FXML
   @Override
   public void initialize(URL url, ResourceBundle rb) {

      //PSurface surface = new PSurfaceJFX(sketch1);
      info.setEditable(false);
      info.setWrapText(true);
      info.isWrapText();
      ProjectButton.setOnAction(event -> {
         try {
            changeProjects();
         } catch (IOException ex) {
         }
      });

      documentationButton.setOnAction(event -> {
         try {
            changeDocumentation();
         } catch (IOException ex) {
         }
```

```
        });

    }

    @FXML
    private void changeProjects() throws IOException {
        App.setRoot("Project");
    }

    @FXML
    private void changeDocumentation() throws IOException {
        App.setRoot("documentation");
    }

}
```

Main class: execute program

```
package com.mycompany.project;
import processing.core.PApplet;
public class Main {
public static void main(String[] args) {
      PApplet sketch = new fillCurve1();
      String[] processingArgs = {""};
      PApplet.runSketch(processingArgs,sketch);
}
}
```

Chapter 7: Conclusions

Team-working plays an important skill in the way being a programmer. A good project is mostly based on contributors through each step of the project. The project gives us knowledge of how fractals and space-filling curves work. This is an interesting topic, we learned how to create images from those space-filling curves, building a GUI as an application for illustrating those images, and moreover, it also develops logical thinking through some complex algorithms. Besides that, self-study is also an indispensable skill, however, we have lots of problems throughout the project but supporting and working together make those problems become much easier to solve. We learned things best by taking examples, so more examples will bring more understanding about the algorithms.

7.1 Working of team members

First, the team working on this project faced a number of challenges, including difficulties in reaching agreements on certain aspects of the project. This led to some rushing towards the end of the project, as we tried to make up for lost time. Despite these challenges, we were able to successfully complete the project and deliver a final product that we believe meets the requirements of the project and our own expectations. While our teamwork was not always smooth, we learned a lot from this experience and are already looking forward to future opportunities to collaborate and build upon the skills we developed through this project.

7.2 What we have learned through this project

The development of this concurrent visualisation of space-filling curves for creating art images has been a valuable learning experience for the team. Not only did we gain a deeper understanding of algorithms for generating space-filling curves, but we also learned about the challenges and benefits of concurrent programming. We developed skills in designing and implementing an efficient and effective program to handle multiple processes and threads.

In addition, we had the opportunity to explore and apply various graphic libraries and programming languages, furthering our expertise in these areas. We also gained experience working with complex data structures and algorithms, as well as testing and debugging our code. Through this project, we evolved a better understanding of the process of software development, from design and implementation to testing and maintenance.

Finally, we discovered the importance of collaboration and effective communication within a team. We learned the hard way that working together plays an important role in making good programs. Only by working together, sharing knowledge and skills, and providing constructive feedback, we were able to develop a high-quality application that met our goals and exceeded our expectations.

7.3 Ideas for the future development of our application

The development of our application has only scratched the surface of the potential for concurrent visualisation of space-filling curves in creating art images. There are many possibilities for future development, including new algorithms, improved performance, and enhanced user interface which are not so good in the current version due to the lack of time.

One of the areas of future development would be to incorporate additional algorithms for generating space-filling curves. For example, we could implement more complex fractals, such as the Mandelbrot set, or custom curve shapes designed by the user like the websites ”onlinefractaltools.com” or ”sciencevsmagic.net/fractal”. This would allow for even more unique and interesting art images to be generated.

Another exciting idea for the future development of our application is the ability to export the generated curve to vector file formats such as EPS or SVG or even make this program an extension for some graphic design applications. This would make the curves compatible with popular graphics editings software such as Adobe Illustrator or Photoshop, allowing for even greater customization and manipulation of the generated art images. The export feature would also make it easier for users to incorporate unique and beautiful curves into their own projects, whether for personal or professional

use. This added functionality would open up new possibilities for creativity and collaboration in the field of concurrent visualisation of space-filling curves.

Enhancing the application's effectiveness and performance would be another area for our improvement. This can entail enhancing the application's concurrent programming capabilities to handle more curves concurrently and optimizing the methods for creating space-filling curves.

The application's user interface is another crucial area for future development. Although the current user interface is useful, not all users will find it to be ideal. We want to make it easier and more natural for users to interact with the curves while also giving them more customization choices. This might entail giving users a more aesthetically pleasing and user-friendly interface, as well as the ability to modify the curves' parameters in real-time. The application would be simpler to use thanks to the improved user interface, and users would be able to take full advantage of the special and lovely curves that the algorithm can produce.

Lastly, we would like to take this chance to express our sincere gratitude to the course lecturer, professor, and teaching assistant who gave us invaluable direction and assistance throughout the project. We were able to overcome the obstacles we encountered and successfully complete the project thanks to their knowledge, persistence, and encouragement. We appreciate their dedication to our technical advancement and believe that without their help, this project would not have been feasible. We appreciate the opportunity to work on this project and look forward to using the skills and knowledge we learned in future endeavours.

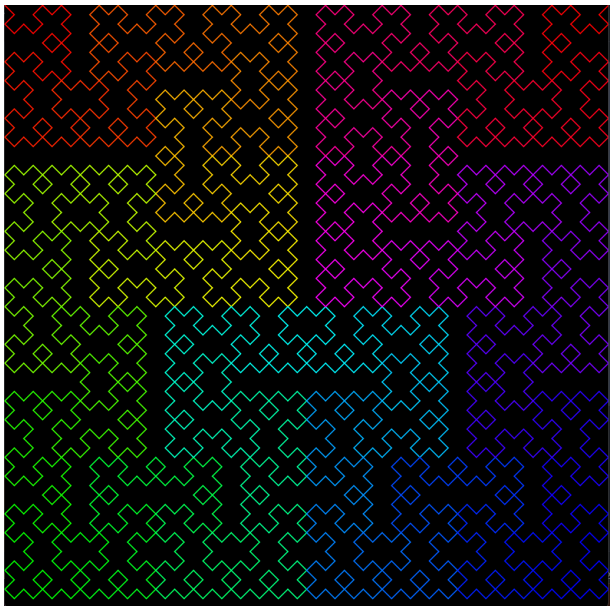


Figure 7.3.1: Our custom curve based on the Hilbert Curve's algorithm

Chapter 8: Citation

Bader, M. (2012). Algorithms of Scientific Computing Space-Filling Curves and their Applications in Scientific Computing. https://www5.in.tum.de/lehre/vorlesungen/asc/ss12/sfc.pdf

Chwedczuk, M. (2016, August 6). Iterative algorithm for drawing Hilbert curve. Blog Marcin Chwedczuk. http://blog.marcinchwedczuk.pl/iterative-algorithm-for-drawing-hilbert-curve

Crinkly Curves. (2017, February 6). American Scientist. https://www.americanscientist.org/article/crinkly-curves: :text=The%20first%20space%2Dfilling%20curve

Guan, X., van Oosterom, P., Cheng, B. (2018). A Parallel N-Dimensional Space-Filling Curve Library and Its Application in Massive Point Cloud Management. ISPRS International Journal of Geo-Information, 7(8), 327. https://doi.org/10.3390/ijgi7080327

Lamb, E. (2016, March 31). A Few of My Favorite Spaces: Space-Filling Curves. Scientific American Blog Network. https://blogs.scientificamerican.com/roots-of-unity/a-few-of-my-favorite-spaces-space-filling-curves/

Reference. (n.d.). Processing. https://processing.org/reference

Sanderson, G. (2017, January 27). 3Blue1Brown - Fractals are typically not self-similar. Www.3blue1brown.com. https://www.3blue1brown.com/lessons/fractal-dimension

Wikipedia Contributors. (2019, August 8). Graphical user interface. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Graphical_user_interface

Workman, K. (2019). Processing in Java. Happy Coding. https://happycoding.io/tutorials/java/processing-in-java