Project Report

Integration of SVG with Java for Scalable 2D Vector Graphics Drawings

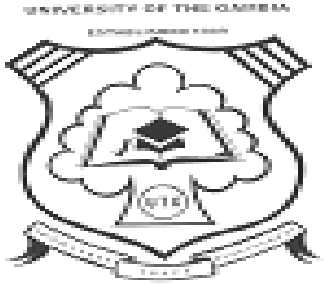
Submitted by:

Sheriff Ahmed Fadil Sonko

21718092

Under Supervision of:

Jacques Boillat



Department of Information Communication Technology

May 2020

Acknowledgement

I am pleased to acknowledge Prof. Jacques Boillat for his invaluable guidance during the course of this project work.

I will also like to extend my sincere thanks to Graduate Assistant Ousainou Jaiteh who continuously helped throughout the project and without his guidance, this project would have been an uphill task.

I am also grateful to other members of the class who co-operated with me regarding some issues.

Contents

1. Introduction

1.1 Overview

This report is centered on the result of the work done in development of a simple “vector graphics drawing tool” on Java Platform. In a graphical system, a windowing toolkit is usually responsible for providing a framework for drawing images. However, writing graphics applications in Java using Swing can be quite a daunting experience which requires understanding of some large libraries, and fairly advanced aspects of Java.

In this project, a simple drawing tool is created which enables users to draw any shape (simple of complex) in any color. Several elements are used ranging from geometric transformations on the objects to saving the images in different formats. The final result of the project is to provide an easy way to draw circles, lines, text and more complex shapes as well as saving them in various formats such as SVG which can then be viewed in any compatible viewing tool.

1.2 Aims & Objective of the Project

This project aims at the development of a platform facilitating the use of a methodological approach for the drawing of vector graphics images using the tools developed during the execution of the project. Upon completion of the project, users will be able to:

1. Create a wide range of geometric primitives in 2D (Two Dimensions) , such as circles, lines, and ellipses, as well as a mechanism for virtually rendering any geometric shape
2. Save those shapes in both a platform and browser independent way which will promote portability of both the tool and its products.
3. Enhanced color support that facilitates color management

1.3 Design

To implement the above goals, the project will be designed in a way that it will allow for the following:

1. Creation of a figure serving as a canvas where all objects to be drawn are placed. It also will determine the thickness, fill and color of each of the objects
2. Construction of different objects representing various graphical shapes by simply passing their coordinate points
3. Application of numerous geometric transformation of the objects in a figure by calling a single method
4. Modification of each of the shapes stored in a figure such as color, fill, thickness, resolution etc.

1.4 Scope of Work

The project is conducted primarily for the purpose of providing a simpler and easier model to draw 2D vector images. Drawing of these images with the windowing toolkit is problematic because it saves images in GIF or JPEG format. This project will extend to enabling the saving and exporting of those images in numerous formats one of which is SVG. It will be limited to drawing of simpler shapes as it does not promote more complex shapes such as arcs, curves, and the like.

The project begin to work 5 months from February to July in the school year 2019 - 2020

2. Detailed Description

2.1 Problem Statement

2.1.1 Problem of saving drawings as raster images

There are many ways to store graphics information in a file in Java APIs but two of the most popular formats are GIF and JPEG.

GIF

GIF stands for Graphics Interchange Format. GIF images can have 2 to 256 colors and are compressed before being stored. The compression algorithm is lossless, which means that the original picture will be restored verbatim when the image is decompressed and displayed. There are actually two common flavors of this format, GIF87a and GIF89.

JPEG

JPEG stands for Joint Photographic Experts Group. Unlike some other file formats, it was designed specifically for photographic images. JPEG images support more colors than GIF images, up to 24 bits per pixel. JPEG images are compressed before being stored using a “lossy” compression algorithm. This means that when the image is loaded and displayed, it will not be exactly the same as the original image.

Main Problems of GIF and JPEG

The problem of saving file formats in GIF or JPEG format is that is saves the drawings as raster images. Even if they need to be converted to vector images, there is the daunting task of conversion by using utilities that convert images between different file formats.

Raster Images

Raster images are bitmaps. They are compiled using pixels, or tiny dots, containing unique color and tonal information that come together to create the image. Since raster images are pixel based, they are resolution dependent. The number of pixels that make up an image as well as how many of those pixels are displayed per inch, both determine the quality of an image.

As you may have guessed, the more pixels in the image and the higher the resolution is, the higher quality the image will be. For example, if we scale a raster image to enlarge it, without changing resolution, it will lose quality and look blurry or pixilated. This is because we are stretching the pixels over a larger area, thus making them look less sharp.

2.1.2 Graphics Class Implementation

Part of Java's initial appeal was its support for graphics that enabled programmers to visually enhance their applications. It makes use of a graphics context which enables drawing on the screen. A Graphics object manages a graphics context and draws pixels on the screen that represent text and other graphical objects (e.g., lines, ellipses, circles and other polylines). Graphics objects contain methods for drawing, font manipulation, color manipulation and the like. When Java is

implemented on each platform, a subclass of Graphics is created that implements the drawing capabilities which supplies the interface.

The aforementioned feature of java and its APIs does not provide full portability because drawing is performed differently on every platform that supports Java: there isn't one implementation of the drawing capabilities across all systems. For example, the graphics capabilities that enable a PC running Microsoft Windows to draw a rectangle are different from those that enable a Linux workstation to draw a rectangle and they're both different from the graphics capabilities that enable a Macintosh to draw a rectangle. This meant that there is a bit of diversity among various platforms. There are also different looks & feels between each of the mentioned systems.

2.1.3 Difficulty in understanding the windowing toolkit

A java windowing toolkit is usually responsible for providing a framework to make it relatively painless for a graphical user interface (GUI) to render the right bits to the screen at the right time. Both the AWT (abstract windowing toolkit) and Swing provide such a framework but the APIs that implement it are not well understood by most. Consequently, there is a problem leading to programs not performing as well as they could because of the inability of developers to understand some large libraries, and fairly advanced aspects of Java. This is where the integration of Java with SVG comes in.

2.1.4 SVG for vector images

SVG, is fundamentally a vector graphics format (although SVG images can include raster images). SVG is actually an XML-based language for describing two dimensional vector graphics images. "SVG" stands for "Scalable Vector Graphics," and the term "scalable" indicates one of the

advantages of vector graphics: There is no loss of quality when the size of the image is increased. A line between two points can be represented at any scale, and it is still the same perfect geometric line. If you try to greatly increase the size of a raster image, on the other hand, you will find that you don't have enough color values for all the pixels in the new image; each pixel from the original image will be expanded to cover a rectangle of pixels in the scaled image, and you will get multi-pixel blocks of uniform color. The scalable nature of SVG images make them a good choice for web browsers and for graphical elements on your computer's desktop.

Unlike pixel-based raster images, vector graphics are based on mathematical formulas that define geometric primitives such as polygons, lines, curves, circles and rectangles. Because vector graphics are composed of true geometric primitives, they are best used to represent more structured images, like line art graphics with flat, uniform colors.

2.1.5 Limitation in manipulation of raster images

GIFs were never designed for animations. They produce exceptionally large file sizes. The more frames, the worse it gets. Raster graphics are less economical, slower to display and print, less versatile.

Unlike JPEG, PNG, and GIF format images, SVG images are just plain text documents that describe the lines, shapes, and colors that make up the image. As SVG files are plain text documents, they can be easily manipulated on the web using JavaScript, CSS, and HTML. SVGs also are resolution independent. They look sharp on any screen, while at the same time keeping file sizes low.

3. Theoretical Background

3.1 Figure

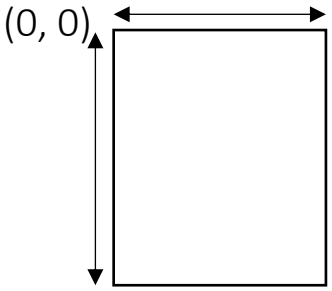
The figure in this application comprises of everything that should be drawn. To put it bluntly, it is basically the canvas. Each figure is created with its own set of instance variables that dictate the dimension of the canvas, resolution, thickness, fill and color of the contained shapes. There are also getters and setters for each of the aforementioned attributes but on instantiation of the figure, the values stored in the FigDefaultValues interface are used. All its shapes are stored in a data structure which can be used to access the objects independently. They are added by calling the methods with add prefix. The shapes that can be added are lines, polylines, circles, ellipses, text and binary trees.

The figure also contains methods from a shape visitor subclass for performing geometric transformations on its shapes and saving the figure in different formats. These visitors include scale, rotate, translate, clip, ExportSVG and save. These methods start off by creating and object of the visitor, traversing each shape stored in the data structure of the figure and then calling the accept method of that shape and passing the visitor object as its parameter.

The scale method expands or shrinks each shape horizontally (based on x) and/or vertically (based on y) while the translate method barely moves the shapes but maintains their sizes. The rotate transforms the figure's with respect to rotation angle and a center of rotation. The clip method performs line clipping on the figure by clipping shapes that fall partially in and eliminating those that are not within the clipping window determined by a top right and bottom left coordinate. The last two visitor methods i.e. the save and ExportSVG merely save the figure in different formats with the former saving in a self-defined JFIG format and the latter saving it in SVG format.

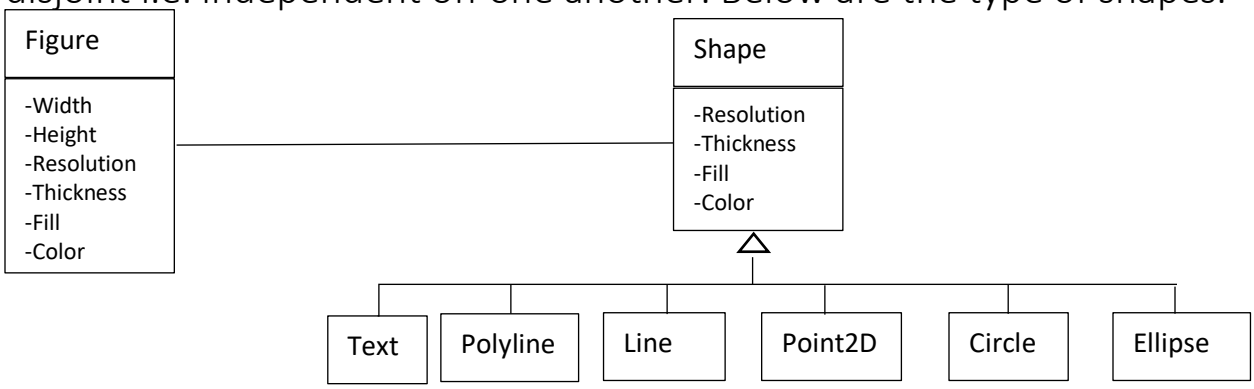
Consequently, there are methods that are not from visitor objects such as the open method which builds a figure, a tostring method which overrides the object tostring method and provides a string representation of the object.

The origin of the figure is the top left corner of the canvas not its center



3.2 Shapes

Within each figure we can have multiples shapes which can be of type line, polyline, circle, ellipse, text and tree. They are all subclasses of the shape class. Since this denotes inheritance, it seems logical to put the attributes that each shape should have here. These include the fill, resolution, color and thickness and are set with either the values set from the figure or passed as arguments on instantiation of the object. There is an additional accept method which is an abstract method and must be overridden by each shape subclass to supply a concrete implementation. The tostring method is overridden as well as the clone method for each subclass of shape. The clone method is overridden for every shape subclass as it makes a true copy of the shape which are disjoint i.e. independent off one another. Below are the type of shapes.



3.2.1 Line

A line is basically a distance between two points. To draw a line we need to set these two points. These will be its instance variables. In the instantiation of each line, we set all of its attributes which it inherited from shape to be the same as that of the figure. There are two constructors for this class with one using the fill, color and thickness of the figure while the other uses the values supplied into the object's constructor when it is instantiated.

3.2.2 Polyline

A polyline is a complex kind of line with poly denoting two or more points. It can be used to create any type of shape. To store these points we use a data structure e.g. a vector to store each of these points. As with the line it also has two constructors performing a similar function to those of the line class.

3.2.3 Circle

Circles are drawn as polylines. To make it as smooth as possible, we make the points closest to each other. For this to happen, the circle also has a vector data structure to store the points. It also has a central point which can be used to derive each of the points to be stored. It does this first by determining how many points there should be by dividing the radius by the resolution and it goes further to create the points by using a special mathematical formula.

The equation of a circle is $X^2 + Y^2 = r^2$

To get any point along the circumference of the circle:

[X ➔ cos (ø)*radius + x of center] [Y ➔ sin (ø)*radius + y of center]

A for-loop traverse each point and calculates it by first setting the angle to be the counter divided by the number of points multiplied by 360 and then convert it to radians since java works in radians. It then passes the

angle into the above formula to get the set of values of each point and stores it in the points' data structure.

3.2.4 Ellipse

An ellipse is a special type of circle which has a radius in the x axis independent off the radius in the y axis. This makes its equation different from the equation of a circle.

The equation of an ellipse is $X^2/a^2 + Y^2/b^2 = r^2$ where a is the x radius and b is the y radius

To get any point along the circumference of the ellipse:

[X ➔ cos (ø)*x radius + x of center] [Y ➔ sin (ø)*y radius + y of center]

Apart from this, it is completely similar to the circle.

3.2.5 Text

Setting up text is pretty much straightforward. We just supply a basepoint and the string message and we are done. However in order to be able to scale, clip and rotate the text, we set up

- horizontal and vertical scale for each text object set to default as 1
- Rotation angle set to 0 by default.
- Top left corner of the clip path rectangle
- The width and height of the clip path rectangle.

These instance variables come in handy when we want to export the text as SVG.

3.2.6 Tree

The binary tree is the most complex shape as of yet. It is complex in the sense that it contain an objects as instance variables of type node. Nodes are set as the elements of a binary tree. Each node contains data, a left child, and a right child. These are set as null in the constructor but can be modified by calling the addleft() and addright() methods. Each

Node also contains a point object to set its position on the canvas. In addition, we need to supply an x-radius and a y-radius for the ellipse of each node which we can enlarge and shrink when scaling the tree.

Back to the tree, since it also extends shape, it sets the color and thickness attributes to the values set by the figure.

The Tree has an additional instance variable for the root node. This node is used in the traverseTree() method which recursively adds the polylines connecting the nodes to the figure, followed by the circles which are filled with white to hide its overlapping lines and finally the text for the data with its basepoint at the center of the.

3.3 Shape Visitors

Visitors are used to perform transformation and exportation of the figure. The shape visitor is an abstract class that cannot be instantiated but can be extended to be inherited from. For this reason, we supply methods that should be common in each of its subclasses which visit each shape stored in a figure. They are all abstract methods so any subclass needs to provide a solid implementation of the method. The subclasses of the shape visitor abstract class are listed below.

3.3.1 Translate

The translate class is probably the simplest shape visitor subclass to implement. It has two instance variables determining how much to translate each point by in the x axis and y axis. These values are instantiated to the values supplied by the figure's translate method. For a line, it adds the translate x value to the x coordinate of the beginning and ending points and the translate y value to the y coordinate to get the updated coordinates. The same approach is used for the polyline but since it consist of multiple points, we loop the data structure adding the translate values to each point. The circle and ellipse since they are

polylines too work in the same way, however, their center is also translated. Translating the test is simplified to only adding the x translate value and y value to the x and y coordinate of the base point.

$$\begin{bmatrix} xprime \\ yprime \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + tx \\ y + ty \\ 1 \end{bmatrix}$$

3.3.2 Scale

This visitor is also very easy to implement. It expands or shrinks a shape based on x and y scale factor. There is uniform scaling if the scale factor of the object is the same in all directions. A non-uniform or anisotropic scaling is obtained when the scale factors are different. To scale the object by a vector $v = (v_x, v_y, 1)$, each point $p = (p_x, p_y, 1)$ will be multiplied by the scaling matrix

$$\begin{bmatrix} xprime \\ yprime \\ 1 \end{bmatrix} = \begin{bmatrix} c1 & 0 & 0 \\ 0 & c2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} c1 * x \\ c2 * y \\ 1 \end{bmatrix}$$

The scale class saves as instance variables the x and y scale factors obtained from the figure's scale method. The standard scaling matrix will only anchor at the origin. Scaling around the origin is trivial; it's simply scalar multiplication. We are scaling around an arbitrary point so we temporarily translate the point to coincide with the origin. Scaling around another point is three operations

- Translate the designated point to the origin
- Scale at the temporary origin
- Translate back so the origin goes back to the designated point

$$\begin{bmatrix} xprime \\ yprime \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} c1 & 0 & 0 \\ 0 & c2 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix}$$

This is pretty straightforward for the line and polyline, as well as the circle and ellipse since we traverse each point and scale. The complications begin when we want to scale the text.

For the text, we just have one point which is the basepoint and scaling that will not increase or decrease the size of the text. For that reason, we create two additional instance variables which will be set to the x and y scale factor. Finally, when exporting the text, we can space the text in the x axis by multiplying the length with the x scale factor and scale in the y axis by multiplying the font size with the y scale factor.

3.3.3 Rotate

Rotation is a bit more complex than the previous visitors. It rotates about a fixed point so these are also saved as instance variables. An addition angle variable which is obtained from the figure's rotate method determines the angle of rotation and we convert it to radians.

To rotate about the origin, we use the matrix

$$\begin{bmatrix} xprime \\ yprime \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & -sin0 & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} xcos\theta - ysin\theta \\ xsin\theta + ycos\theta \\ 1 \end{bmatrix}$$

If you wanted to rotate a point around something other than the origin, the steps taken are to

- Translate the whole system so that the point of rotation is at the origin
- Perform the rotation
- Undo the translation

This step can all be represented by the following matrix

$$\begin{bmatrix} 1 & 0 & px \\ 0 & 1 & py \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -px \\ 0 & 1 & -py \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

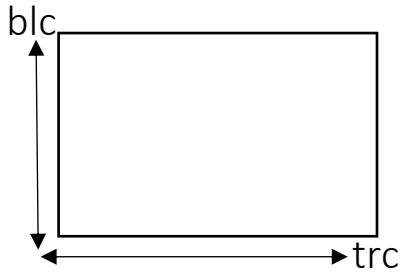
The rotation of the line and polyline as well as the circle and ellipse is pretty straight forward since the above algorithm is used for each point.

To rotate the text, we want to rotate the basepoint of the text but also change its angle and direction. For this reason, we make the text object store an angle instance variable initialized to zero. Then we can set this to whatever rotation angle in the rotate visitor but convert it to degrees. The text can then be rotated about its basepoint based on the angle (e.g. using SVG rotate method).

3.3.4 Resize

This visitor is responsible for performing line clipping function on the shapes. The algorithm used is the Liang-Banksy line clipping algorithm.

However, since the origin in this canvas is the top-left corner, the bottom left and top right corner are defined by



This is because the top left corner has the minimum x and y values and the bottom right corner has the maximum x and y values. The algorithm:

1. Read 2 endpoints of line as p1 (x1, y1) & p2 (x2, y2).

2. Read 2 corners (left-top & right-bottom) of the clipping window as (x-min, y-min, x-max, y-max).

3. Calculate values of parameters pi and qi for i = 1, 2, 3, 4 such that

p1 = -dx, q1 = x1 – x-min

p2 = dx, q2 = x-max – x1

p3 = -dy, q3 = y1 – y-min

p4 = dy, q4 = y-max – y1

4. if pi = 0 then line is parallel to its boundary

If qi < 0 then line is completely outside boundary so discard line

Else, check whether line is horizontal or vertical and then check the line endpoints with the corresponding boundaries.

5. Initialize t1 & t2 as

t1 = 0 & t2 = 1

6. Calculate values for qi/pi for i = 1, 2, 3, 4.

7. Select values of qi/pi where pi < 0 and assign maximum out of them as t1.

8. Select values of qi/pi where pi > 0 and assign minimum out of them as t2.

9. If (t1 < t2) {
xx1 = x1 + t1dx

xx2 = x1 + t2dx

yy1 = y1 + t1dy

yy2 = y1 + t2dy

Line (xx1, yy1, xx2, yy2)}

10. Stop.

This algorithm is used for just the line, polyline, circle and ellipse. For the text, we use the SVG defined clip path when exporting. This requires defining the top left corner which is set as the top left corner of the clipping window as well as the width and height. These are stored as

instance variables for the text which are used when exporting it. Each text when being exported in SVG has an id linking to the clip path. The syntax for setting the clip path id is

Syntax in SVG ➔<clip path id =”###”> <rect x = “#” y = “#” width = “##” height=”##”></clip>

3.3.5 ExportSVG

The primary purpose of this visitor is to export all the shapes in a figure in SVG format. To understand how it works, one has to understand the mechanism behind the figure’s ExportSVG method. This method first creates a file to which it writes the objects. It creates the SVG header to signify the file is of type SVG and then writes the format for each shape in SVG. It then terminates by closing the SVG tag.

For each of the shapes, the ExportSVG visitor returns a string for the representation of the shape in SVG. This is then written to the file

3.3.6 Save

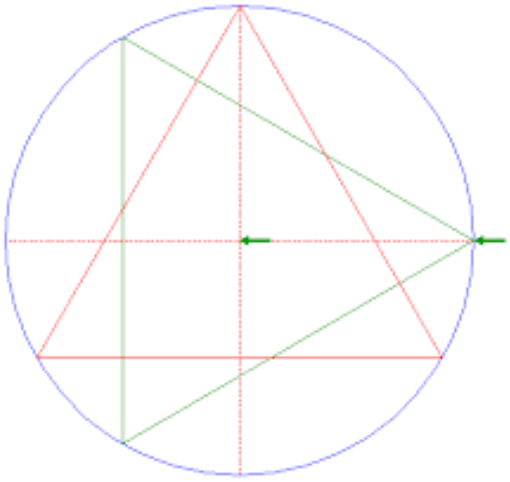
This visitor is also used for the exportation of the figure in a self-defined JFIG format. It starts off by creating a file to be written to in the figures’ save method. The very first line on the file is “JFIG 0.1” denoting the type of document. The width, height and resolution of the figure are written on the second line. It then loops the figures data structure containing all the shapes and writes the self-defined JFIG format of that shape onto the file.

3.4 Fractals

3.4.1 Triangles

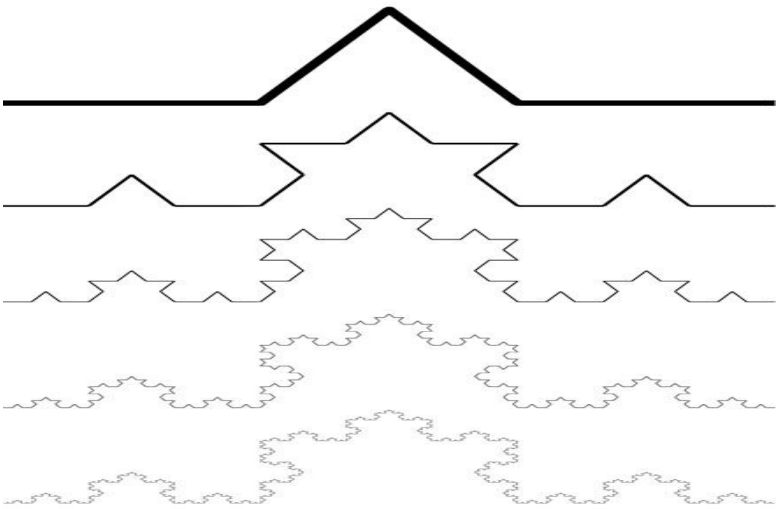
To draw the equilateral triangle centered at a point, we get the first point A which is at the top left corner of the center. The x coordinate of this point is found by subtracting half the radius of the circle from the x coordinate of the center and its y value is found by making a right angle triangle to find the opposite and then adding the opposite to the y value

of the center. The opposite is tan (60) * half the radius. The B point has the same y value as the center but its x value is found by adding the radius to the center's x value. The third point C same x as A but its y value is found by subtracting the opposite from the y of the center.

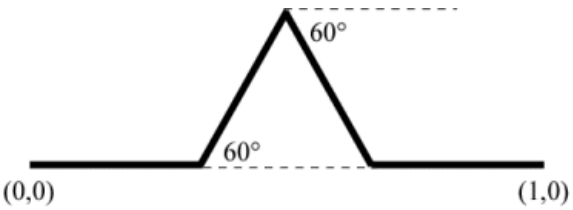


3.4.2 Koch Lines

The Koch Curve starts with a straight line that is divided up into three equal parts. Using the middle segment as a base, an equilateral triangle is created. Finally, the base of the triangle is removed, leaving us with the first iteration of the Koch Curve.



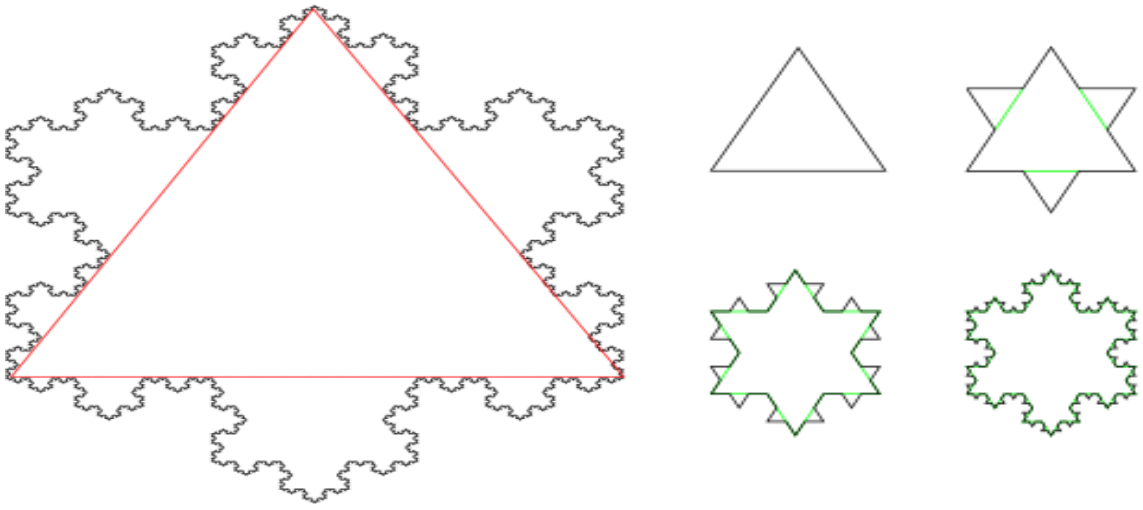
Getting two of the points of the equilateral triangle to be created from the base is pretty straightforward. It is simply the beginning and end point of the middle segment. However, for the third part i.e. the tip, to get the x value, we divide the difference in x of the middle segment into two and add that to the x value of beginning of the middle segment and also add the product of sin60 and the length of the middle segment. For the y value, we also divide the difference in y of the middle segment into two and add that to the y value of beginning of the middle segment and also add the product of sin60 and the length of the middle segment.



We recursively add these four Koch curves decrementing the iterations by one until we get to the base case where the level becomes zero.

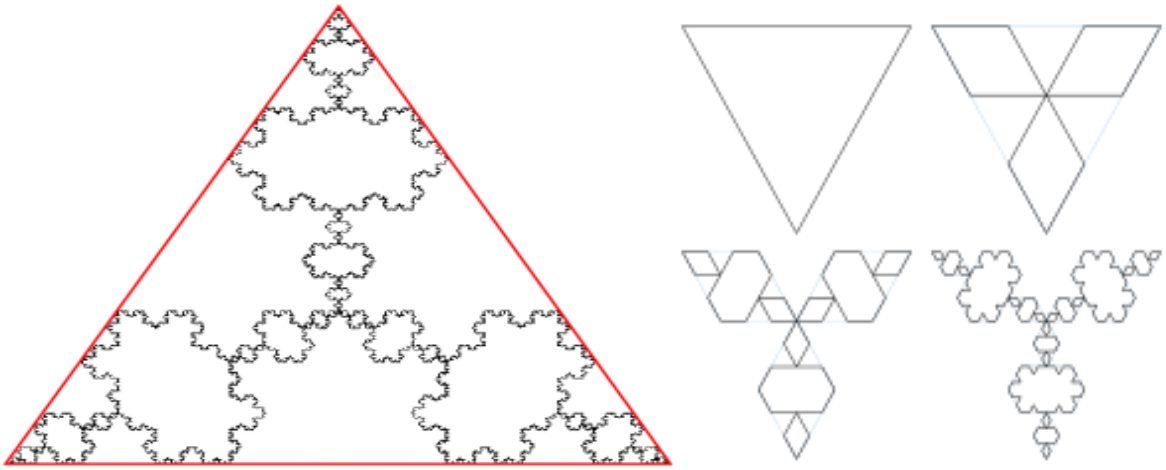
3.4.3 Koch Snowflake

From the Koch Curve comes the Koch Snowflake. Instead of one line, the snowflake begins with an equilateral triangle. The steps in creating the Koch Curve are then repeatedly applied to each side of the equilateral triangle, creating a "snowflake" shape.



3.4.4 Koch Anti-Snowflake

The Koch anti-snowflake curve, also known as anti-star curve, is constructed using the same principles as the regular Koch snowflake. The exception here is that every side of the starting equilateral triangle bends inwards at every iteration at 60 degrees rather than outwards.

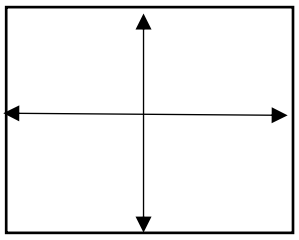


3.4.5 Sierpinski Triangle

The construction of this element starts with an equilateral triangle which is then chopped into four little equilateral triangles, getting rid of the middle upside-down triangle. This is done by finding the midpoint of all the lines and creating triangles from each of the corners of the triangle to the midpoint of the lines it is connected to. We follow up by repeating this process for every triangles based on the level of iterations.

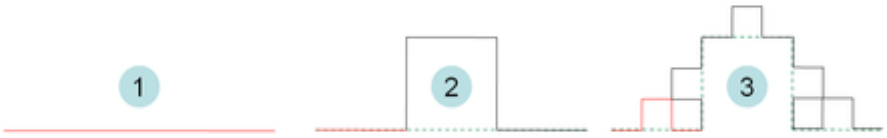
3.4.6 Squares

In order to construct a square centered at a particular point with a side length r, we find each of the corners by adding or subtracting half the length of the side from/to the center to get each of the four points.



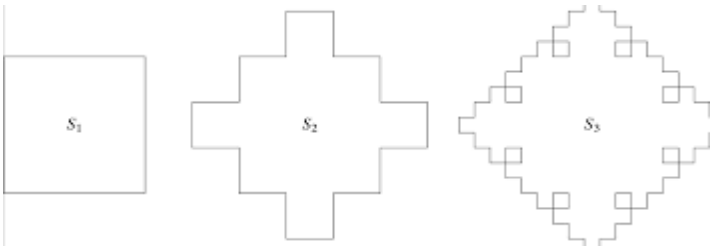
3.4.7 Square Koch Line

The type 1 curve is a direct analogy to the classic Koch curve, replacing each line segment with five smaller segments placed at right angles. We have five lines and six points. Getting two of the points of the square to be created from the base is pretty straightforward. It is simply the beginning and end point of the middle segment but for the two points that are outwards we use trigonometry. We start at the first point and calculate the opposite by multiplying sin60 by the adjacent and add that the x of the first point to get its x and use the same opposite calculated and add it to the y of the first point to get its y point. For the second outward point, we do the same thing but place the angle at the first point of the middle segment to do the calculations.



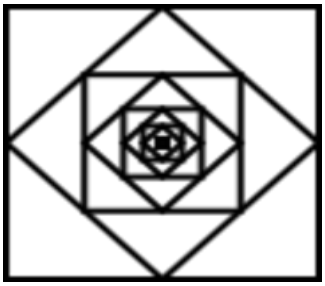
3.4.8 Partial Square Koch

This fractal makes use of the square Koch line by first drawing a square with each side as a square Koch line and on each iteration. Variations are created by using a square shape instead of a triangle in the replacement rule.



3.4.9 Internal Diamond Square Koch

This fractal is similar to the Sierpinski Triangle but instead works on squares. It draws a square and on each iteration it calculates the midpoint of all the square lines and then draws an internal square. This gives the illusion that it is drawing a diamond inside the square and then a square inside the diamond and so on.



3.4.10 Fractal Tree

The fractal tree starts off by drawing the trunk. The points of the trunk can be the center of the tree and another which has the same x as the center but with y being the y of the center plus the length. From the center, we can draw two branches. The first goes at 35 degrees to the left of the angle and the other goes at 35 degrees right of the angle. We can then repeat this at the end of each branch until a sufficient level of branching is reached. The thickness of a branch decreases by 0.5 in each recursion but it can only decrease to a basepoint of 0.5 so that the branch can still be visible.

3.4.11 Asymmetric Fractal Tree

The asymmetric fractal tree is synonymous to the fractal tree with the only difference being in the angles supplied of each branch. For the one used in the project, one of the branches is at 50 degrees left while the other is at 5 degrees right.

4. Software Documentation

4.1 Figure

The figure has its instance variables i.e. the color, fill, thickness and resolution set on instantiation with the default values stored in the FigDefaultValues interface. Its constructor sets up by defining:

```
public Figure (double width, double height) {
this. Width = width;
this. Height = height;
resolution = FigDefaultValues. RESOLUTION;
color = FigDefaultValues. COLOR;
fill = FigDefaultValues. FILL;
thickness = FigDefaultValues. THICKNESS;
}
```

These values can also be modified by the setters. As mentioned earlier, the figures stores all its shapes in a vector data structure using methods with the add prefix. These methods are the addLine(), addPolyline(), addCircle(), and addEllipse(), that stores these objects in the data structure containing the shapes. However, for the addBinary tree() method, it traverses the tree recursively from the root node all the way to the leaves.

For most of the visitors such as translate, rotate, scale and clip, the figure first creates an instance of the class. It then loops through all shapes stored in its data structure and pass the object created as the visitor to the shapes overwritten accept method.

To demonstrate, we look at the translate method. A translate object is created which is passed to all stored shapes using their accept method.

```
public void translate(double x, double y) {
Translate translator = new Translate(this,x,y);
for(int i=0;i<shapes.size();i++) {
Shape shape = shapes.get(i);
```

```
shape.accept(translator,shape);
}
 }
```

The ExportSVG creates a file using the printwriter which it writes the returned SVG format of all the contained shapes into.

```
PrintWriter writer = new PrintWriter(name);
ExportSVG sv = new ExportSVG(this);
```

The first line on this file is always the declaration showing it is an xml document. This is followed by a tag showing it is an SVG document with the SVG tag containing the width height and namespace.

```
String header = "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n";
header += String.format(
  "<svg width=\"%.2f\" height=\"%.2f\" "
    + "xmlns=\"http://www.w3.org/2000/svg\">\n",
  width, height);
writer.write(header);
```

It then loops through all the shapes and calls the accept method of the shape and passing in as a parameter the ExportSVG object created.

```
for (Shape shape : shapes) {
  writer.write((String) shape.accept(sv, null));
}
```

This returns a string representation of that object in SVG which is then written to the file. After all shapes have been looped, the SVG tag is closed marking the end of the document and the printwriter object is closed. These files may generate exceptions so it is caught in a try catch block.

```
writer.write("</svg>\n");
writer.flush();
writer.close();
```

The save method does a similar job to the ExportSVG with the difference being that it saves the file in a self-defined JFIG format.

```
PrintWriter writer = new PrintWriter(name);
Save sv = new Save(this);
```

The first line is always a declaration to show the document as JFIG and its version. The second line contains the width, height and resolution of the figure.

```
String header = "JFIG 0.1\n";
header += String.format(width + " " + height + " " + resolution + "\n");
writer.write(header);
```

The shapes are then looped similar to the ExportSVG and the self-defined JFIG format of each object is returned as a string which is written to the document.

```
for (Shape shape : shapes) {
writer.write((String) shape.accept(sv, null));
}
```

However, as opposed to the ExportSVG, the save method does not require closing the document, so only the printwriter is closed and everything is encapsulated in a try catch block.

```
writer.flush();
writer.close();
```

To build a figure from a file, we use the open method. This takes as a parameter the name of the file which is in the self-defined JFIG format. . The first thing this method does is to create a figure.

```
Figure fig = null;
```

It then creates a scanner object which takes the file as input.

```
File file = new File(name);
Scanner in = new Scanner(file);
```

Because the first two words of this file contain the declaration, they are consumed.

```
in.next();
in.next();
```

The next three inputs contain the width, height and resolution so these values are used to set the corresponding values for the figure.

```
double width = in.nextDouble();
double height = in.nextDouble();
double resolution = in.nextDouble();
```

Next comes the shapes object with the first value denoting what type of shape it is. By using this value, it can determine whether the shape is of type line, ellipse or text. Here, 1 is for ellipse, 2 is for a polyline and 4 for text.

```
int shape = in.nextInt();
```

The next three values for any shape read in contain the color, fill and thickness so this sets those corresponding values on the figure. For a line, the next input value following this one determines the number of points for the line.

```
int number_of_points = in.nextInt();
Point2D[] points = new Point2D[number_of_points];
for(int i = 0; i < number_of_points; i++) {
double xcoord = in.nextDouble();
double ycoord = in.nextDouble();
points[i] = new Point2D(xcoord,ycoord);
}
```

A for loop is created to traverse from 0 to the number of points, adding each point to an array which is passed to the constructor of the polyline.

```
Polyline line = new Polyline(fig,points);
```

fig.addPolyline(line);
On reading 1: for the ellipse, we consume the next input i.e. 3 which shows that it is going to be passed in by radius.

in.next();
We then read in the thickness, color, x of basepoint, y of basepoint, x radius and y radius

thickness = in.nextDouble();
color = in.next();
fill = in.next();
double xcoord = in.nextDouble();
double ycoord = in.nextDouble();
double xradius = in.nextDouble();
double yradius = in.nextDouble();
Using these values, we can create an ellipse and add it to the figure

fig.addEllipse(new Ellipse(fig,color,fill, thickness,new Point2D(xcoord,ycoord),xradius,yradius));
For the text however, the next two inputs contain the x and y values of the basepoint

double xcoord = in.nextDouble();
double ycoord = in.nextDouble();
Following these inputs comes the actual string message. We loop through all stings until the end which is denoted by the special characters “$$$$”

if (data.compareTo("$$$$")!=0)
build.append(data + " ");

After reading all these we can then build the text object and add it to the figure

fig.addText(new Text(fig, basepoint, build.toString().trim()));

4.2 Shapes

The shape abstract class cannot be instantiated but is extended by all shapes. It has three instance variable which all shapes inherit which are the color, fill and thickness.

```
public double thickness;
public String color;
public String fill;
```

There is also an abstract method called accept, which all shapes should overwrite and provide a solid implementation for. This enables for a shape visitor object to be passed for the shape.

```
public Object accept(ShapeVisitor shapeVisitor, Object o)
```

4.2.1 Line

The two points of the line are stored as instance variables

```
public Point2D from;
public Point2D to;
```

The line has two constructors. The first is set to uses the values of the figure by default

```
public Line(Figure fig, Point2D from, Point2D to)
```

The second one takes as parameters the color and thickness of the line and these are used on instantiation of the line

```
public Line(Figure fig,String color,int thickness, Point2D from, Point2D
to){
super(fig);
this.color = color;
this.thickness = thickness;
this.from = from;
this.to = to;
```

}

The accept method is overwritten as well as the tostring method with the latter showing the line in the following format on the console

The line class implements its clone method by returning an exact copy of the line

Syntax ➔ line c "color" f "fill" t "thickness" points "(from) (to)"'

4.2.2 Polyline

As with the line, the polyline has two constructors. Like all shapes, the first takes the figure and points as its parameters

public Polyline(Figure fig,Point2D... x)

The second one does not use the values of the color, fill and thickness from the figure but uses those defined in the constructor's parameters.

public Polyline(Figure fig,String color,String fill,int thickness,Point2D... x)
However, because a polyline can contain any number of points, we modify its constructor to accept this. These points can then be stored to a vector data structure

```
public Vector<Point2D> points = new Vector<Point2D>();
```

These points supplied are looped through and added to the data structure for each of the constructors of the polyline

```
for(int i= 0; i<x.length;i++) {
Point2D val = x[i];
points.add(val);
}
```

It loops through these points adding them to the vector data structure containing the points. Similarly, the tostring and accept method are overwritten. The polyline class implements its clone method by returning an exact copy of the polyline

4.2.3 Circle

The circle stores three instance variables. The first is its center which is stored as Point2D. The second is the radius of the circle and the third is the data structure containing the points since it is to be drawn as a polyline.

```
public Point2D center;
public double radius;
public Vector<Point2D> points = new Vector<Point2D>();
```

The circle also has two constructors with the first taking only the figure, center and radius as parameters thereby setting the fill, color and thickness with the values of the figure.

```
public Circle(Figure fig, Point2D center, double x)
```

The second constructor takes additional parameters for the color, fill and thickness which are used on instantiation.

```
public Circle(Figure fig,String color,String fill,int thickness,Point2D center, double x)
```

In each of the constructors, we determine the number of points the data structure should have by dividing the radius by the resolution. Using the special formula explained on the theoretical description, the points on the circumference of a circle given its center are derive each of which is added to the data structure

```
this.center = center;
radius = x;
double nop = radius / fig.RESOLUTION;
for (int i = 0; i <= nop ; ++i) {
        double angle = Math.toRadians((i / nop)*360);
        points.add(new Point2D(Math.cos(angle)*radius +
        this.center.x,Math.sin(angle)*radius + this.center.y));}
```

The circle class also overwrites the tostring and accept method. The circle class implements its clone method by returning an exact copy of the circle

4.2.4 Ellipse

The ellipse is almost identical to a circle with the only difference being that its radius in the x axis is different from that of the y axis. Because of this, it stores two instance variable for the radius with the first being the x-radius and the second being the y-radius

```
public Point2D center;
public double xradius;
public double yradius;
public Vector<Point2D> points = new Vector<Point2D>();
```

As with the circle, its first constructor just takes the figure, the center, the x-radius and the y-radius. This sets the color, fill and thickness to the values set by the figure

```
public Ellipse(Figure fig,Point2D center, double x, double y)
```

The second one takes those values as parameters instead of using those set by the figure

```
public Ellipse(Figure fig, String color,String fill,int thickness,Point2D
center, double x, double y)
```

To find the number of points the ellipse should have, we take the average of the x and y radius and divide that by the resolution. Then to get each of the points on the circumference of the ellipse, we use the formula described in the theoretical description and store these values in the data structure containing the points

```
double nop = ((xradius + yradius)/2) / fig.RESOLUTION;
for (int i = 0; i <= nop ; ++i) {
double angle = Math.toRadians((i / nop)*360);
points.add(new Point2D(Math.cos(angle)* xradius +
this.center.x,Math.sin(angle)* yradius + this.center.y));
```

The tostring and accept method are also overwritten for the ellipse class. The ellipse class implements its clone method by returning an exact copy of the ellipse

4.2.5 Text

A text requires attributes which describe its message and location on the canvas. These are set as the instance variables

public Point2D coord;
public String message;

In order for the text to be scaled: since we only know about its basepoint and the string, we rely on the functionality of SVG. Because of this, we add two instance variables for the text being the x-scale and y-scale. The x-scale determines the length to adjust in the x axis of the text: how the characters are spaced, while the y-scale determines the font size of the text which scales it in the y-axis. They are both set to 1 by default.

public double xscale;
public double yscale;

To rotate the text about its basepoint, there is another instance variable for the rotation angle which is set to be 0. Whenever the rotate shape visitor is called, it is modified to the angle of rotation supplied.

public double rotateangle;

The last set of instance variables are defined for text clipping. To clip text in SVG, we rely on the clip path which needs the top left corner of the window as well as its width and height. By default, this uses the top left corner of the canvas and its whole width and height. Whenever the clip visitor for the text is called, these values are modified to use the corner, the width of the clipping window, and its height.

public double clipheight;
public double rotateangle;
public Point2D ctlc;

Similarly, the text has two constructors with the first using the figures' values for the color, fill and thickness

public Text(Figure fig,Point2D p,String mess)

The second takes them in as parameters and sets the corresponding values to the values supplied in the parameter on instantiation.

public Text(Figure fig,String color,int thickness,Point2D p,String mess)
The tostring and accept method are also overwritten for the text class. The text class implements its clone method by returning an exact copy of the text.

4.2.6 Tree

The tree is basically made up of nodes. Each of these nodes need the data, left child, right child, its coordinate on the canvas as well as the default radius of the ellipse containing the data. This is set to 25 by default.

```
public String data;
public Node leftChild;
public Node rightChild;
public Point2D point;
public double xradius = 25;
public double yradius = 25;
```

There are two methods for a node i.e. to add the left child and to add the right child of the node

```
public void addLeft(Node left) {
        leftChild = left;
}
public void addRight(Node right) {
        rightChild = right;
}
```

The tostring method is also overwritten to return the value stored for the data of the node.

For the Tree class, it extends the shape abstract class and instantiates its color, fill and thickness to the values of the figure. Its most important method is the traverseTree(). This is the method that is called in the addBinaryTree() method of the figure.

```
public void addBinaryTree(Tree tree) {
tree.traverseTree(tree.rootNode);}
```

It starts at the root node of the tree and goes recursively in post order by drawing all lines linking a tree to its children.

```
if(root.leftChild!=null) {
fig.addLine(new Line(fig,color,thickness,new
Point2D(root.point.x,root.point.y),new
Point2D(root.leftChild.point.x,root.leftChild.point.y)));
traverseTree(root.leftChild);
}
if(root.rightChild!=null) {
fig.addLine(new Line(fig,color,thickness,new
Point2D(root.point.x,root.point.y),new
Point2D(root.rightChild.point.x,root.rightChild.point.y)));
traverseTree(root.rightChild);
}
```

After doing this, it creates a circle of radius 25 and fill white to cover the lines overlapping with the circle. Then the text is drawn on top of the circle.

```
If (root!=null) {
fig.addEllipse(new
Ellipse(fig,color,"#FFFFFF",thickness,root.point,25,25));
fig.addText(new Text(fig,color,thickness,new
Point2D(root.point.x,root.point.y),root.data));
```

}

The tostring and accept method are also overwritten for the tree class. The tree class implements its clone method by returning an exact copy of the tree with the rootnode.

4.3 Shape Visitors

The shape visitor abstract class sets up multiple abstract which must be overwritten for each concrete visitor class. The only exception is the tree class

```
public abstract Object visitLine(Line line, Object o);
public abstract Object visitPolyline(Polyline polyline, Object o);
public abstract Object visitText(Text text, Object o);
public abstract Object visitCircle(Circle circle, Object o);
public abstract Object visitEllipse(Ellipse ellipse, Object o);
```

4.3.1 Translate

The translate class has two instance variables for the value to translate a point by in the x- axis and the y-axis.

```
public double x;
public double y;
```

These values are instantiated with the parameters supplied by the figure when its translate method is called.

To translate a line, we add the x and y translate values to both points of the line to get its new coordinate.

```
line.from.x += x;
line.from.y += y;
line.to.x += x;
line.to.y += y;
```

To translate a polyline, we loop through all the points stored in its points' data structure and add the x and y translate value to all the points.

```
for(int i=0; i< polyline.points.size();i++) {
Point2D point = polyline.points.get(i);
point.x += x;
point.y += y;
}
```

To translate the circle, we translate the center and loop through all the points in the data structure and add the translate x and y value to each point to get its new coordinate.

```
circle.center.x += x;
circle.center.y += y;
for(int i=0; i< circle.points.size();i++) {
Point2D point = circle.points.get(i);
point.x += x;
point.y += y;
}
```

The same approach taken for the circle is used for the ellipse

```
ellipse.center.x += x;
ellipse.center.y += y;
for(int i=0; i< ellipse.points.size();i++) {
Point2D point = ellipse.points.get(i);
point.x += x;
point.y += y;
}
```

For translation of the text, we simply add the translation value along the x axis and along the y axis to the basepoint to gets its new translated coordinate.

```
text.coord.x += x;
text.coord.y += y;
```

4.3.2 Scale

The scale class has two instance variables for the scale factor along the x- axis and the y-axis.

```
public double x;
public double y;
```

For scaling of the line, we multiply the x value of each point by the x scale factor and multiply the y value of each point by the y scale factor to get its new coordinate.

```
line.from.x *= x;
line.from.y *= y;
line.to.x *= x;
line.to.y *= y;
```

The polyline uses a similar approach to the line but it loops through multiplies the x scale factor by the x value of the point and the y scale factor by the y vale of the point to get its new position.

```
for(int i=0; i< polyline.points.size();i++) {
Point2D point = polyline.points.get(i);
point.x *= x;
point.y *= y;
}
```

The circle, like the polyline, updates all its points by multiplying the x value of the point to the x scale factor and the y value of the point to the y scale factor. In addition, it also scales the center.

```
circle.center.x *= x;
circle.center.y *= y;
for(int i=0; i< circle.points.size();i++) {
```

```
Point2D point = circle.points.get(i);
point.x *= x;
point.y *= y;
}
```

The ellipse scales its points identical to the way that the circle does

```
ellipse.center.x *= x;
ellipse.center.y *= y;
for(int i=0; i< ellipse..points.size();i++) {
Point2D point = ellipse.points.get(i);
point.x *= x;
point.y *= y;
}
```

To scale a text, we multiply the x value of its basepoint by the x scale factor and the y value by the y scale factor

```
text.coord.x *= x;
text.coord.y *= y;
```

This is not enough since it does not actually scale the string. Therefore, we rely on SVG's lengthAdjust and font size attributes to scale the string. Because of this, we set the x scale and y scale instance variables of the text to be the x and y scale factor

```
text.xscale =  x;
text.yscale = y;
```

4.3.3 Rotate

The rotate class has three instance variables. These are the x and y values of the center of rotation and the angle of rotation. These are instantiated by the values passed as parameters when the figure calls its rotate method

```
double x;
```

double y;
double alpha;

However, the rotation angle passed by the figure is in degrees so it is converted to radians in the constructor

this.alpha = alpha * (Math.*PI*/180);

To rotate a line we use the special formula explained in the theoretical description.

New x ➔ (((old x – x)*Math.*cos*(angle)) - ((old y - y) * Math.*sin*(angle))) + x;
New y ➔ (((oldx – x)*Math.*sin*(angle)) + ((old y - y) * Math.*cos*(angle))) + y;

Here x is the x value of the center of rotation and y is the y value of the center of rotation

This modifies the x and y values of a point but because we will need the unmodified value of the point, we store it temporarily

double fxval = line.from.x;
double fyval = line.from.y;
double txval = line.to.x;
double tyval = line.to.y;

Then we can use the method on each of the points on the line to get its new coordinates

line.from.x = (((fxval - x)*Math.*cos*(alpha)) - ((fyval - y) * Math.*sin*(alpha))) + x;
line.from.y = (((fxval - x)*Math.*sin*(alpha)) + ((fyval - y) * Math.*cos*(alpha))) + y;
line.to.x = (((txval - x)*Math.*cos*(alpha)) - ((tyval - y) * Math.*sin*(alpha))) + x;
line.to.y = (((txval - x)*Math.*sin*(alpha)) + ((tyval - y) * Math.*cos*(alpha))) + y;

For the polyline, a similar approach is used but by looping through all the points stored in its data structure and updating the value using the formula

for(int i=0; i< polyline.points.size();i++) {

```
Point2D point = polyline.points.get(i);
double xval = point.x;
double yval = point.y;
point.x = (((xval - x)*Math.cos(alpha)) - ((yval - y) * Math.sin(alpha))) + x;
point.y = (((xval - x)*Math.sin(alpha)) + ((yval - y) * Math.cos(alpha))) + y;
}
```

This same method is applied to the circle with the exception being that the center of the circle is rotated as well.

```
double cxval = circle.center.x;
double cyval = circle.center.y;
circle.center.x = (((cxval - x)*Math.cos(alpha)) - ((cyval - y) * Math.sin(alpha))) + x;
circle.center.y = (((cxval - x)*Math.sin(alpha)) + ((cyval - y) * Math.cos(alpha))) + y;
```

This data structure containing the points is looped through and each point is rotated

```
for(int i=0; i< circle.points.size();i++) {
Point2D point = circle.points.get(i);
double xval = point.x;
double yval = point.y;
point.x = (((xval - x)*Math.cos(alpha)) - ((yval - y) * Math.sin(alpha))) + x;
point.y = (((xval - x)*Math.sin(alpha)) + ((yval - y) * Math.cos(alpha))) + y;
}
```

The ellipse is rotated the same way as the circle

```
double cxval = ellipse.center.x;
double cyval = ellipse.center.y;
ellipse.center.x = (((cxval - x)*Math.cos(alpha)) - ((cyval - y) * Math.sin(alpha))) + x;
ellipse.center.y = (((cxval - x)*Math.sin(alpha)) + ((cyval - y) * Math.cos(alpha))) + y;
for(int i=0; i< ellipse.points.size();i++) {
Point2D point = ellipse.points.get(i);
double xval = point.x;
double yval = point.y;
point.x = (((xval - x)*Math.cos(alpha)) - ((yval - y) * Math.sin(alpha))) + x;
point.y = (((xval - x)*Math.sin(alpha)) + ((yval - y) * Math.cos(alpha))) + y;
```

}

There is a bit of complexity when we want to rotate a text, the first step is to rotate the basepoint relative to the center of rotation

```
double xval = text.coord.x;
double yval = text.coord.y;
text.coord.x = (((xval - x)*Math.cos(alpha)) - ((yval - y) * Math.sin(alpha))) + x;
text.coord.y = (((xval - x)*Math.sin(alpha)) + ((yval - y) * Math.cos(alpha))) + y;
```

Then we use the SVG rotate transformation attribute to rotate the text based on the rotation angle supplied.

```
text.rotateangle = Math.toDegrees(alpha);
```

This angle is passed to the text and is consequently passed to the ExportSVG visitor for the text

4.3.4 Resize

This visitor is primarily responsible for performing line clipping on the shapes. To do this effectively, it requires setting a clipping window. Consequently, this requires the bottom left and top right corner so these are set to be instance variables of the Clip class

```
public Point2D bl;
public Point2D tr;
```

They are instantiated with the values passed as arguments when the figure calls its clip method.

For clipping of the line, we use the Liang Barsky algorithm to determine whether the line is visible, invisible, or a clipping case. If it is visible, we do not alter it and if it is a clipping case, we find the new coordinates. However, if the line needs to be discarded, we cannot simply delete it because it is the caller of the method. Therefore, we make the points on the line 0 so that they cannot show on canvas

```
line.from.x = 0;
line.from.y = 0;
line.to.x = 0;
line.to.y = 0;
```

The same approach is taken for the polyline. We take two points on a polyline and check to see if the line joining those two points is visible, invisible or a clipping case.

```
for(int j=0; j < polyline.points.size()-1;j++) {
int first = j;
double  x0 = polyline.points.get(first).x, y0 = polyline.points.get(first).y,
x1 = polyline.points.get(first+1).x, y1 = polyline.points.get(first+1).y;
```

If the line linking them is visible, we make a line object and store it on the figure.

```
fig.addLine(new Line(fig,new Point2D(nx0,ny0),new Point2D(nx1,ny1)));
```

We cannot make it a polyline because then a stack overflow will happen: the polyline will create a polyline which will create a polyline, and so on. If it is a clipping case we also make a line with the updated coordinates. After we have looped through all the points, we can then remove all points of the polyline since we cannot delete the caller of the method.

```
polyline.points.removeAllElements();
```

The circle and ellipse are implemented the same way as polylines since they also contain a data structure that has their points
The text however just requires setting the top left corner of the clip path rectangle as well as its width and height

```
text.ctlc.x = bl.x;
text.ctlc.y = bl.y;
text.clipwidth = tr.x-bl.x;
text.clipheight = tr.y-bl.y;
```

4.3.5 ExportSVG

The ExportSVG visitor class is primarily responsible for providing a string representation of the different shapes in SVG format. Each of these objects have their own various formats in SVG

Back to the visitor, the line and polyline are saved using the polyline tag. Their only difference is the line has only two points and for the polyline you have to loop the data structure containing the points. This approach is also used for the circle and ellipse since they too are polylines. The other attributes that are set for these tags is the stroke representing the outline color, the stroke width representing its thickness, and the fill.

Syntax in SVG➔<polyline points=”x1,y1 x2,y2 … xn,yn” stroke=”xxx” stroke-width=”xxx” fill=”xxx” />

The text uses the text tag in SVG. To scale the text in the x axis we have the text length attribute set to default as 40 plus the length of the string text multiplied by the x scale factor. In order to adjust it correctly, we have the textAdjust attribute set to spacingandglyphs. To scale it in the y axis we just multiply the font size by the y scale factor. The default font size is 16. In the rotation transformation of the text, we rotate the text about the center of rotation. It is then rotated about its basepoint too by using the rotate method of the transform attribute. Its parameter is set to the rotation angle instance variable of the text. Other attributes of the text include the stroke, stroke-width and fill and textAnchor to align the text to the start. Additionally, there is a clip path attribute linking to the lip path tag. The string of the text is between the opening and closing tag

Syntax

<text x=”x” y = “y” textlength=”xxx” lengthAdjust=”spacingAndGlyphs” font-size=”xx” text-anchor=”start” transform=”rotate (xx x,y)”

stroke=”xxx” stroke-width=”xx” fill=”xx” clip-path=”#url”>Message</text>

The clip path tag defines the clip path window. It contains a rectangle which has attributes for the x and y of the top left corner, the width and the height of the window. These are all stored as instance variables for the text which are modified whenever clip visitor for the text is called

Syntax ➔ <clipPath id=”url” ><rect x = “#” y = “#” width = “#” height = “#”/></clip>

4.3.6 Save

The format used in this project is similar to the format used of by XFIG. It first stores the object type i.e. 2 for lines, 3 for circles and ellipses, and 4 for text. For the lines, the next entry is the thickness, color and fill respectively. This is followed by the number of points, so for a simple line, it stores two (2) signifying two points and then stores the x axis and y axis of those points on the next line. Similarly, the polyline store the size of the data structure containing the points as the number of points and then loops over it storing the x and y axis of each point on the next line.

Syntax➔ 2 “thickness” “color” “fill” “number of points”

“x1 y1 x2 y2 … xn yn”

For the circles and ellipses, it stores 1 to signify an ellipse object followed by 3 to show that it is going to be passed in by radius. The next entries are the thickness, color and fill respectively. Then, the x and y points of the center are passed followed by the radius along the x axis and y – axis.

Syntax➔ 1 3 “thickness” “color” “fill” “center x” “center y” “x radius” “y radius”

The implementation for the text is similar but it uses 4 as the object type followed by the thickness, color, and fill. This is followed by the x axis and y axis of the basepoint and the message respectively. To signify the end of the message, we insert a static “$$$$”

Syntax➔ 4 “thickness” “color” “fill” “base x” “base y” “message” $$$$

This convention is unique to this project as it is self-defined and cannot be viewed in any JFIG compatible viewing tool.

4.4 Fractals

4.4.1 Triangles

In order to draw the triangle given a central point and length, we use the method defined in the theoretical description to calculate the three points

```
pOne.x = c.x - (double) r/2;
pOne.y = c.y + Math.sqrt(3) * r/2;
pTwo.x = c.x + r;
pTwo.y = c.y;
pThree.x = c.x - (double) r/2;
pThree.y = c.y - Math.sqrt(3) * r/2;
```

4.4.2 Koch Lines

Koch lines generally take two points as parameters and return five points for the four line segments to be calculated. It uses the level to determine the number of iterations and calls the method recursively until it gets to the base case i.e. where the number of iterations is 0.

On each iteration, it uses a set of two points from the five points created for a line and calls on the method recursively with those points. This mean that there are four recursive calls for each iteration

```
if (lev > 0){
Point2D pA = new Point2D(pOne.x + (pTwo.x-pOne.x) /3,pOne.y + (pTwo.y-
pOne.y)/3);
```

```
double sin60 = 0.866025403784438646763723170752936183471402626905190;
Point2D pTip = new Point2D(pA.x + (int)((pTwo.x-pOne.x)/3* 0.5 + (pTwo.y-
pOne.y)/3*sin60),pA.y + (int)((pTwo.y-pOne.y)/3* 0.5 - (pTwo.x-pOne.x)/3*sin60));
Point2D pB = new Point2D(pOne.x + 2 * (pTwo.x - pOne.x)/3,pOne.y + 2 * (pTwo.y -
pOne.y)/3);
        addKochLine(fig, lev - 1, pOne, pA);
        addKochLine(fig, lev - 1, pA, pTip);
        addKochLine(fig, lev - 1, pTip, pB);
        addKochLine(fig, lev - 1, pB, pTwo);
}
```

When it reaches the base case, it stops the recursive calls and just adds the polyline of the two points passed as arguments to the figure.

```
else{
fig.addPolyline(new Polyline(fig,pOne,pTwo));
}
```

4.4.3 Koch Snowflake

The snowflake starts from three points which are used to create a triangle.

```
Point2D pOne = new Point2D(0,0);
Point2D pTwo = new Point2D(0,0);
Point2D pThree = new Point2D(0,0);
triangle(center,length,pOne,pTwo,pThree);
```

It then calls the Koch line method on each of the sides of the triangle also passing in the number of iterations

```
addKochLine(fig, iterations, pOne, pThree);
addKochLine(fig, iterations, pThree, pTwo);
addKochLine(fig, iterations, pTwo,pOne);
```

4.4.4 Koch Anti-Snowflake

The anti-snowflake is similar to the snowflake with the only exception being how the sides are passed in to generate Koch lines

```
Point2D pOne = new Point2D(0,0);
```

```
Point2D pTwo = new Point2D(0,0);
Point2D pThree = new Point2D(0,0);
triangle(center,length,pOne,pTwo,pThree);
addKochLine(fig, iterations, pOne, pTwo);
addKochLine(fig, iterations, pTwo, pThree);
addKochLine(fig, iterations, pThree,pOne);
```

4.4.5 Sierpinski Triangle

To draw a Sierpinski triangle, we also draw a trangle from the three points and call the addSierpinski() method which takes those points as parameters

```
Point2D pOne = new Point2D(0,0);
Point2D pTwo = new Point2D(0,0);
Point2D pThree = new Point2D(0,0);
triangle(center,length,pOne,pTwo,pThree);
addSierpinski(fig, iterations, pOne, pTwo, pThree);
```

This is a recursive method which when the number of iterations is greater than 0, it calculates the midpoint of the sides of the triangle and makes three recursive calls each using the two points on the side of the triangle.

```
if (lev > 0){
Point2D pA = new Point2D (pOne.x + (pTwo.x-pOne.x) /2,pOne.y +
(pTwo.y-pOne.y)/2);
Point2D pB = new Point2D (pOne.x + (pThree.x-pOne.x) /2,pOne.y +
(pThree.y-pOne.y)/2);
Point2D pC = new Point2D (pTwo.x + (pThree.x-pTwo.x) /2,pTwo.y +
(pThree.y-pTwo.y)/2);
addSierpinski(fig, lev - 1, pOne, pA, pB);
addSierpinski(fig, lev - 1, pB, pC, pThree);
addSierpinski(fig, lev - 1, pA, pTwo, pC);
}
```

When the base case is reached, we draw a polyline linking those three points to form a triangle

4.4.6 Squares

To draw a square given a central point and length, we use the approach defined in the theoretical description to calculate the four points

```
pOne.x = c.x - (double) r/2;
pOne.y = c.y + (double) r/2;
pTwo.x = c.x + (double) r/2;
pTwo.y = c.y + (double) r/2;
pThree.x = c.x + (double) r/2;
pThree.y = c.y - (double) r/2;
pFour.x = c.x - (double) r/2;
pFour.y = c.y - (double) r/2;
```

4.4.7 Square Koch Lines

The square Koch line unlike the Koch line generates 6 points on each iteration. The middle segment is replaced by a square. The formula to generate these points is specified in the theoretical description. As long as the base case isn't reached, there are five recursive calls on the 6 points by taking consecutive points in pairs

```
if (lev > 0){
Point2D pA = new Point2D(pOne.x + (pTwo.x-pOne.x) /3,pOne.y +
(pTwo.y-pOne.y)/3);
Point2D pD = new Point2D(pOne.x + 2 * (pTwo.x - pOne.x)/3,pOne.y + 2
* (pTwo.y - pOne.y)/3);
double sin60 = -
0.86602540378443864676372317075293618347140262690519 0;
Point2D pB = new Point2D(pOne.x + ((pTwo.x-pOne.x)/3 + (pTwo.y-
pOne.y)/3*sin60),pOne.y + ((pTwo.y-pOne.y)/3 - (pTwo.x-
pOne.x)/3*sin60));
```

```
Point2D pC = new Point2D(pA.x + ((pTwo.x-pOne.x)/3 + (pTwo.y-
pOne.y)/3*sin60),pA.y + ((pTwo.y-pOne.y)/3 - (pTwo.x-
pOne.x)/3*sin60));
        addSquareKochLine(fig, lev - 1, pOne, pA);
        addSquareKochLine(fig, lev - 1, pA, pB);
        addSquareKochLine(fig, lev - 1, pB, pC);
        addSquareKochLine(fig, lev - 1, pC, pD);
        addSquareKochLine(fig, lev - 1, pD, pTwo);
}
```

Similarly, when it reaches the base case, it draws a polyline with the two points passed in as arguments

```
else{
fig.addPolyline(new Polyline(fig,pOne,pTwo));
}
```

4.4.8 Partial Square Koch

This creates a partial square Koch but starts off by creating a square based on the center and the length

```
Point2D pOne = new Point2D(0,0);
Point2D pTwo = new Point2D(0,0);
Point2D pThree = new Point2D(0,0);
Point2D pFour = new Point2D(0,0);
square(center,length,pOne,pTwo,pThree,pFour);
```

After creating the square, it uses the two points on each side of the square and calls the square Koch line recursive method. This means that there are 4 recursive calls since there are 4 sides

```
addSquareKochLine(fig, iterations, pOne, pTwo);
addSquareKochLine(fig, iterations, pTwo, pThree);
addSquareKochLine(fig, iterations, pThree,pFour);
addSquareKochLine(fig, iterations, pFour,pOne);
```

4.4.9 Square Inverse Koch Lines

Unlike the previous Koch lines, this takes in four points and finds the midpoint of each of the lines connecting those 4 points if the number of iterations is greater than 0. It draws a polyline for the four points supplied as arguments and then passes those four midpoint values to the method recursively till it reaches the base case.

```
if (lev > 0){
Point2D pA = new Point2D(pOne.x + (pTwo.x-pOne.x) /2,pOne.y +
(pTwo.y-pOne.y)/2);
Point2D pB = new Point2D(pTwo.x + (pThree.x-pTwo.x) /2,pTwo.y +
(pThree.y-pTwo.y)/2);
Point2D pC = new Point2D(pThree.x + (pFour.x-pThree.x) /2,pThree.y +
(pFour.y-pThree.y)/2);
Point2D pD = new Point2D(pFour.x + (pOne.x-pFour.x) /2,pFour.y +
(pOne.y-pFour.y)/2);
fig.addPolyline(new Polyline(fig,pOne,pTwo,pThree,pFour,pOne));
InverseKochLine(fig, lev - 1, pA, pB,pC,pD);
}
```

At the base case, it simple draws a polyline connecting those four points

```
else{
fig.addPolyline(new Polyline(fig,pOne,pTwo,pThree,pFour,pOne));
}
```

4.4.10 Internal Diamond Square Koch

As this creates a diamond inside a square and a square inside the diamond and so on, it begins by first creating the square.

```
Point2D pOne = new Point2D(0,0);
Point2D pTwo = new Point2D(0,0);
Point2D pThree = new Point2D(0,0);
Point2D pFour = new Point2D(0,0);
square(center,length,pOne,pTwo,pThree,pFour);
```

It then makes a recursive call to the square inverse Koch Line method using the points on the square.

InverseKochLine(fig, iterations, pOne, pTwo, pThree,pFour);

4.4.11 Fractal Tree

This method starts of by calculating the center of the tree. It then calls the addFBranch Method

```
Point2D pTwo = new Point2D(0,0);
pTwo.x = c.x;
pTwo.y = c.y+length;
addFBranch(fig,iterations,-90,length,thickness,pTwo);
```

This is recursive method which when the base case is not reached, it calculates the point using the mathematical formula described in the theoretical description.

```
if(thickness<0.5) {
thickness=0.5;
}
fig .setThickness(thickness);
Point2D pThree = new Point2D(0,0);
pThree.x = pTwo.x + (int) (Math.cos(Math.toRadians(angle)) * lev * 7.0);
pThree.y = pTwo.y - length+ (int) (Math.sin(Math.toRadians(angle)) * lev * 7.0);
fig.addPolyline(new Polyline(fig,pTwo, pThree));
```

This is followed up by two recursive calls with one decreasing the angle by 35 and the other increasing it by 35. It also reduces the thickness of the line by 0.5 on each recursion with the minimum value being 0.5

```
addFBranch(fig, lev - 1, angle - 50,length/2,thickness-0.5, pThree);
addFBranch(fig, lev - 1, angle + 15,length/2,thickness-0.5, pThree);
```

When the base case is reached, it simply returns to the caller

4.4.12 Asymmetric Fractal Tree

The asymmetric fractal tree is almost identical to the former. Its only exception is that whilst the fractal tree calls recursively by adding the

angle by 35 on one of the recursive calls and subtracts 35 from it in the other, it uses a different approach. On one of its recursions, it adds 50 to the angle and on the other, it subtracts 15 from it

addFBranch(fig, lev - 1, angle - 50,length/2,thickness-0.5, pThree);

addFBranch(fig, lev - 1, angle + 15,length/2,thickness-0.5, pThree);

On reaching the base case it also returns to the caller

5. User Manual

This project is going to be run as an executable Jar file (.exe extension). This executable file runs the main methods of the main class which in our case are the testers. However, since it does not have a GUI, The shapes to be drawn are created on a class that has a main method e.g. the TestFigure. Consequently, an executable jar file has to be created for every tester which can be run on any platform. Creating the executable Jar file requires exporting the project as a runnable Jar and setting the selected tester as the launch configuration. Furthermore, the location where it needs to be saved has to be specified.

The executable Jar file created for the tester can be run on any machine by simply double clicking it. This runs the main method and upon completion, all the files that are to be created are produced in the same location as the executable JAR file

6. References

Holiney, V., 2020. *Logaster.* [Online]
Available at: https://www.logaster.com/blog/vector-and-raster-graphics/[Accessed 3 April 2020].

Anon., 2020. *PsPrint.* [Online]
Available at: https://www.psprint.com/resources/difference-between-raster-vector/[Accessed 4 May 2020].

Jain, A., 2019. *GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/vector-vs-raster-graphics/?ref=rp[Accessed 17 March 2020].

GoodNight, E. Z., 2017. *How-to-Geek.* [Online]
Available at: https://www.howtogeek.com/howto/30941/whats-the-difference-between-jpg-png-and-gif/[Accessed 27 March 2020].

Anon., 2020. *TechTerms.* [Online]
Available at: https://techterms.com/definition/rastergraphic[Accessed 5 May 2020].

Praher, M., 2019. *TutorialsPoint.* [Online]
Available at: https://www.tutorialspoint.com/difference-between-jpeg-and-gif[Accessed 20 April 2020].

Weisstein, E. W., 2020. *MathWorld.* [Online]
Available at: https://mathworld.wolfram.com/Ellipse.html[Accessed 7 May 2020].

Weisstein, E. W., 2020. *MathWorld.* [Online]
Available at: https://mathworld.wolfram.com/Incircle.html[Accessed 27 April 2020].

Westfall, P., 2018. *Page Cloud.* [Online]
Available at: https://www.pagecloud.com/blog/web-images-png-vs-jpg-vs-gif-vs-svg[Accessed 12 March 2020].

Wick, J., 2019. *StickerMule.* [Online]
Available at: https://www.stickermule.com/blog/raster-vs-vector-images[Accessed 15 March 2020].

Anon., 2020. *w3schools.* [Online]
Available at: https://www.w3schools.com/graphics/svg_intro.asp [Accessed 14 April 2020].

Wikipedia Contributors., 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Ellipse[Accessed 15 April 2020].

Anon., 2019. *LumenLearning.* [Online]
Available at: https://courses.lumenlearning.com/waymakercollegealgebra/chapter/equations-of-ellipses/[Accessed 23 April 2020].

Wikipedia Contributors., 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Circle[Accessed 27 April 2020].

Anon., 2020. *MathisFun.* [Online]
Available at: https://www.mathsisfun.com/algebra/circle-equations.html[Accessed 4 May 2020].

Anon., 2012. *GamefromScratch.* [Online]
Available at: https://www.gamefromscratch.com/post/2012/11/24/GameDev-math-recipes-Rotating-one-point-around-another-point.aspx [Accessed 4 June 2020].

Anon., 2020. *TutorialsPoint.* [Online]
Available at: https://www.tutorialspoint.com/computer_graphics/2d_transformation.htm [Accessed 18 May 2020].

Wikipedia Contributors, 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Tree_traversal [Accessed 17 May 2020].

Wikipedia Contributors, 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Translation_(geometry) [Accessed 18 May 2020].

Wikipedia Contributors, 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Scaling_(geometry) [Accessed 19 May 2020].

Wikipedia Contributors, 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Rotation_(mathematics) [Accessed 4 June 2020].

Boillat, J., 2020. *Google Drive.* [Online]
Available at: https://drive.google.com/file/d/1Z4r7ir5ZWjRjcX_p6-QqgkGPC67QpJhS/view?usp=sharing[Accessed 25 June 2020].

Wikipedia Contributors, 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Koch_snowflake [Accessed 20 June 2020].

Wikipedia Contributors, 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Koch_snowflake [Accessed 20 June 2020].

Percy, X., 2020. *GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/sierpinski-triangle-using-graphics/?ref=rp[Accessed 5 May 2020].

Wikipedia Contributors, 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle [Accessed 22 June 2020].

Badbe, V., 2018. *Ques10.* [Online]
Available at: https://www.ques10.com/p/22053/explain-liang-barsky-line-clipping-algorithm-with-/ [Accessed 15 June 2020].

Brit, R. S., 2020. *Rosettacode.* [Online]
Available at: https://www.rosettacode.org/wiki/Fractal_tree [Accessed 27 June 2020].

Britten, D., 2020. *GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/ [Accessed 27 May 2020].

Chauhan, A., 2020. *GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/2d-transformation-in-computer-graphics-set-1-scaling-of-objects/ [Accessed 16 May 2020].

Dickar, R., 2011. *Wolfram Demostrations.* [Online]
Available at: https://demonstrations.wolfram.com/SquareKochFractalCurves/
[Accessed 23 June 2020].

Hosch, W. L., 2020. *Britannica.* [Online]
Available at: https://www.britannica.com/topic/set-mathematics-and-logic
[Accessed 20 June 2020].

Jindal, A., 2020. *GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/rotation-of-a-point-about-another-point-in-cpp/ [Accessed 4 June 2020].

Mishra, N., 2020. *The Crazy Programmer.* [Online]
Available at: https://www.thecrazyprogrammer.com/2017/02/liang-barsky-line-clipping-algorithm.html [Accessed 15 June 2020].

Pal, S., 2019. *Medium.* [Online]
Available at: https://medium.com/swlh/understanding-3d-matrix-transforms-with-pixijs-c76da3f8bd8 [Accessed 18 May 2020].

Parsons, M., 2019. *COE.* [Online]
Available at: http://jwilson.coe.uga.edu/EMAT6680/Parsons/MVP6690/Essay1/sierpinski.html [Accessed 22 June 2020].

Pennington, L., 2017. *Study.com.* [Online]
Available at: https://study.com/academy/lesson/what-is-the-sierpinski-triangle-pattern-history.html [Accessed 23 June 2020].

Roberts, D., 2020. *MathBitsNotebook.* [Online]
Available at: https://mathbitsnotebook.com/Geometry/Transformations/TRTransformationRotations.html [Accessed 29 may 2020].

Shiffman, D., 2020. *Nature of Code.* [Online]
Available at: https://natureofcode.com/book/chapter-8-fractals/ [Accessed 25 June 2020].

Wiesstein, E. W., 2020. *Math World.* [Online]
Available at: https://mathworld.wolfram.com/SierpinskiSieve.html [Accessed 22 June 2020].

Yang, A., 2020. *Study.com.* [Online]
Available at: https://study.com/academy/lesson/uniform-non-uniform-scaling-definition-examples.html [Accessed 22 May 2020].

Weisstein, E. W., 2020 Math World [Online]

Available at: http://mathworld.wolfram.com/KochSnowflake.html[Accessed 22 May 2020].