Recursive Graphics  
212 Project

Professor Jonathan Schrader

Nathaniel Brown

Rebecca Iselin

Aeyva Rebelo

Alexandria Sampalis

July 23, 2023

Table of Contents

[Assumptions 2](#_Toc140996261)

[Instructions for Setup of SFML Library on CLion 2](#_Toc140996262)

[SFML Setup for Windows 2](#_Toc140996263)

[SFML Setup for MAC (Not Assured – Supplementary Instructions) 6](#_Toc140996264)

[Simple and Fast Multimedia Library (SFML) Key Information 10](#_Toc140996265)

[Instructions for Compiling 11](#_Toc140996266)

[Preliminary Compilation Instructions 11](#_Toc140996267)

[Running the Program 11](#_Toc140996268)

[Generating a Hilbert’s Curve Graphic 12](#_Toc140996269)

[Generating a Sierpinski’s Triangle Graphic 12](#_Toc140996270)

[Generating a Koch’s Snowflake Graphic 14](#_Toc140996271)

[Introduction to Recursive Graphics Topic 15](#_Toc140996272)

[Introduction to the Project 15](#_Toc140996273)

[Hilbert’s Curve Algorithm Analysis and Breakdown 16](#_Toc140996274)

[Stages of Hilbert’s Curve 16](#_Toc140996275)

[Sierpinski’s Triangle Algorithm Analysis and Breakdown 19](#_Toc140996276)

[Stages of Sierpinski’s Triangle 19](#_Toc140996277)

[Kock’s Snowflake Algorithm Analysis and Breakdown 22](#_Toc140996278)

[Stages of Koch’s Snowflake 22](#_Toc140996279)

[Member Contributions (Link to sheets HERE) 26](#_Toc140996280)

[Version 1.0: 27](#_Toc140996281)

[Version 2.0 27](#_Toc140996282)

[Version 3.0 (Final Version) 28](#_Toc140996283)

[Report and Presentation Contribution Breakdown 28](#_Toc140996284)

[Conclusion 29](#_Toc140996285)

[Links List 29](#_Toc140996286)

# Assumptions

This project was developed with the usage of **Windows 11** in mind, and the usage of **Clion** as a primary IDE for compiling and running the program. Similarly; it is assumed that the user will acquire and properly install the **Simple and Fast Multimedia Library** within C++. As well as the usage of **C++14**. Without all of these conditions met we can not guarantee the accurate compilation and processing of our program. We took steps to implement additional support of **MAC os**, however as none of our members own a Mac device we are unable to confirm the accuracy of the installation advice.

To reiteratewe **assume** the **prior-possession** of the following **REQUIRED** resources:

1. **Windows 11**
2. **Clion by Jetbrains**
3. **C++14**
4. **Simple and Fast Multimedia Library (Instructions found below).**

# Instructions for Setup of SFML Library on CLion

## SFML Setup for Windows

1. *A screenshot of a computer

   Description automatically generated*Download [here](https://www.sfml-dev.org/download/sfml/2.6.0/)… *Note: You MUST figure out which version of GCC you have. If you installed GCC in 211/201 you probably have MinGW64bt. Select GCC 13.1.0 MinGW(SEH)64-Bit*
2. *Extract and move ENTIRE SFML folder to desired directory… for example I store mine directly within C:/*

*A screenshot of a computer

Description automatically generated*

1. *A screenshot of a computer

   Description automatically generatedCreate a new project in CLion… set standard language to C++ 14.*
2. *Locate CMakeLists.txt within your new project (name depending on your choice… Mine is ‘Project2’* ***NOTE:*** *Whenever you see project2 referenced; replace with your project folders name.*

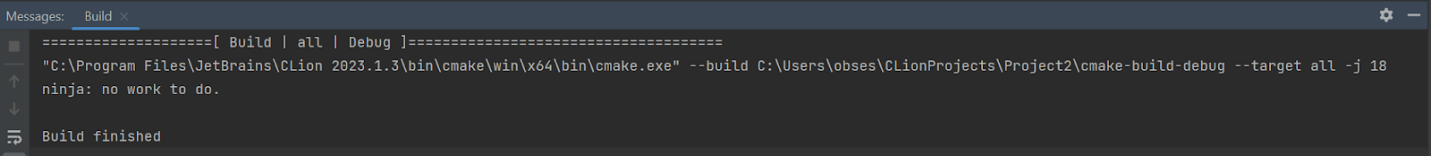
1. *A screenshot of a computer program

   Description automatically generatedA screenshot of a computer program

   Description automatically generatedEdit CMakeLists.txt file to look like the following;* ***NOTE*** *adjust paths to reflect where you store SFML and directory name to reflect yours.*
2. *A screenshot of a computer program

   Description automatically generatedThis should create a new file in your external libraries that looks like this:*
3. A screenshot of a computer program

   Description automatically generated*From here you are ready to test if SFML library is properly installed: Copy this code into your main.cpp to test…*
4. *You* ***CANNOT*** *compile using the traditional terminal methods. You must select* ***‘build project’****.*

*A screenshot of a program

Description automatically generated*

1. *If you do not get a build finished message; you have the wrong version of SMFL or pathing is inconsistent. Or you’re not using C++14. Now you can select ‘Run’.*

*A screenshot of a computer

Description automatically generated*

1. *If this menu pops up, you’re ready to run and Compile the Project*

*A screen shot of a computer

Description automatically generated*

## SFML Setup for MAC (Not Assured – Supplementary Instructions)

1. Download [here](https://www.sfml-dev.org/download/sfml/2.6.0/). Note: Make sure you scroll down to where the macOS downloads are. You will have to know the type of MAC you are using. To do this, check under ‘about this MAC’. If there is a section that says “Chip: Apple M1 (or M2)”, download the option that says “Clang-64-bit (OS X 10.7+, compatible with C++11 and libc++). If ‘about this mac’, says “Processor:” followed by some version of “Intel Core”, then be sure to download “Clang-ARM64 (OS X 11.0+)”.
2. *A screenshot of a computer

   Description automatically generated*Download the extension Homebrew [here](https://brew.sh/). It is recommended to do so to help streamline the process by avoiding any issues that come up with copying and locating the libraries.

*A screenshot of a computer

Description automatically generated*

1. Once homebrew is installed open your terminal and type:*‘ brew install sfml ‘*. This will begin the process of installing SFML to your machine.
2. *A screenshot of a computer program

   Description automatically generated*We can now go ahead and create a new CLion project. Be sure to set standard language to C++ 14.
3. Locate CMakeLists.txt in your new project (the project name will differ based on what you choose, for example: ‘newProject’ **NOTE**: Whenever you see newProject referenced or written; replace with your project folders name.

*A screenshot of a computer program

Description automatically generated*

1. *Replace the current contents of your CMakeLists.txt file with the following;* ***NOTE*** *be sure to change anywhere it says ‘newProject’ with your projects name*

*cmake\_minimum\_required(VERSION 3.14)*

*project(newProject)*

*set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -std=c++11")*

*set(SOURCE\_FILES main.cpp)*

*add\_executable(newProject ${SOURCE\_FILES})*

*include\_directories(/usr/local/include)*

*find\_package(SFML 2.5 COMPONENTS system window graphics network audio REQUIRED)*

*include\_directories(${SFML\_INCLUDE\_DIRS})*

*target\_link\_libraries(newProject sfml-system sfml-window sfml-graphics sfml-audio sfml-network)*

1. *From here you are ready to test if SFML library is properly installed: Copy this code into your main.cpp to test…*

*/////////////////// Libraries ////////////////////*

*#include <SFML/Graphics.hpp>*

*//////////////////////////////////////////////////*

*int main(){*

*sf::RenderWindow window(sf::VideoMode(640,480), "SFML Application");*

*sf::CircleShape shape;*

*shape.setRadius(100.f);*

*shape.setPosition(100.f, 150.f);*

*shape.setFillColor(sf::Color::Red);*

*while(window.isOpen()){*

*sf::Event event;*

*while(window.pollEvent(event)){*

*if(event.type == sf::Event::Closed){*

*window.close();*

*}*

*}*

*window.clear();*

*window.draw(shape);*

*window.display();*

*}*

*}*

1. ***A screenshot of a computer program

   Description automatically generated****You cannot compile this code, you must select ‘****build project’***

*If the build is successful, you will see this message appear:*

*A screenshot of a computer

Description automatically generated*

*If not, you might need to troubleshoot, you may not be using C++14, or you could have run into an issue downloading SFML.*

1. A screenshot of a computer

   Description automatically generated*Once your build is successful you can select ‘Run”.*
2. *If this menu pops up, you are ready to run and Compile the Project.*

# *A screen shot of a computer Description automatically generated*Simple and Fast Multimedia Library (SFML) Key Information

As previously mentioned, there are several template classes used within our program. This section will briefly discuss a few of the essential template classes to help formulate a better understanding of what their purpose and use is within project files.

* sf::Vector2f (x,y) is a specific type of class that is used primarily to store two floats which will represent a coordinate on a graph. It is very similar to the standard library’s ‘pair’ however; Vector2f includes a variety of utility functions and convenient methods used for 2D graphic operations. I.E. This allows the developer to focus primarily on the algorithm and construction at hand; rather than the mutation of a specific pair for the purposes of their program.
* sf::RenderWindow(Height, Width, Name) is another SFML class template; which is used to represent a window in which graphics can actually be drawn with a visible display to the user to see the results of various SFML commands. I.E. This creates a window application in which you can view the final iteration of a recursive graphic call.
* sf::ConvexShape is used to create a shape by connecting points with line segments. This can be done by using built in setters to establish the number of points within the shape. (For a triangle this would be 3, a rectangle would be 4.) Then setting each point passed an index for which point would be the start point, and a Vector2f object containing the coordinate locations of the line’s start. These helper setter functions are called setPointCount(numberPointers) and setPoint(index, Vector2f (x,y)).

# Instructions for Compiling

## Preliminary Compilation Instructions

*A screenshot of a program

Description automatically generated*The program should be built prior to execution. No commands are necessary to compile in order to run the program.

## Running the Program

Upon execution, the main menu will display, prompting the user to select an algorithm for which they wish to generate a graphic. The menu options include generating a Hilbert’s Curve graphic (1), a Sierpinski’s Triangle graphic (2), a Koch’s Snowflake (3), or End Program (4).

A screenshot of a computer error

Description automatically generated

Once a category to generate a graphic has been selected, a submenu will display prompting the user to enter the specific recursive graphic algorithm’s parameters.

## Generating a Hilbert’s Curve Graphic

The menu to generate a Hilbert’s Curve graphic will display when the user has entered option 1 from the men menu.



The Hilbert’s Curve submenu will prompt the user to enter a file name.

**Note: The file name must contain a .png extension.**

****

Once the file name has been entered, a window will open displaying the generated graphic and a .png file will be created holding a copy of the graphic image. The illustration above demonstrates a user generating a Hilbert’s Curve graphic which will save to a file named ‘curve.png’. The image below illustrates the graphic generated:

A black maze with white lines

Description automatically generated

Closing the graphic window will trigger the application to redisplay the main menu where the user may generate another graphic or end the program.

## Generating a Sierpinski’s Triangle Graphic

The menu to generate a Sierpinski’s Triangle graphic will display when the user has entered option 2 from the main menu.

A screenshot of a computer program

Description automatically generated

The Sierpinki’s Triangle submenu will prompt the user to enter a file name, select both a main and contrast color from the color menu, and a number to control the number of iterations used to generate the graphic (stage).

**Note: The file name must contain a .png extension.**

The color options available to generate the triangle graphic include Red (1), Magenta (2), White (3), and Yellow (4).

**Note: The stage number must be a number between 0 and 5 (anything higher will distort the resolution and may affect runtime).**

A screen shot of a computer

Description automatically generated

Once these parameters have been entered, a window will open displaying the generated graphic and a .png file will be created holding a copy of the graphic image. The illustration above demonstrates a user generating a Sierpinski’s Triangle graphic which will save to a file named ‘triangle.png’, have a main color of magenta, a contrast color of yellow and terminate its generation at stage 5. The image below illustrates the graphic generated using these parameters:

A yellow and pink triangle

Description automatically generated

Closing the graphic window will trigger the application to redisplay the main menu where the user may generate another graphic or end the program.

## Generating a Koch’s Snowflake Graphic

The menu to generate a Koch’s Snowflake graphic will display when the user has entered option 3 from the main menu.

A screenshot of a computer

Description automatically generated

The Koch’s Snowflake submenu will prompt the user to enter a file name, select a color from a menu of color options, and a number to control the number of iterations used to generate the graphic (stage).

**Note: The file name must contain a .png extension.**

The color options available to generate the triangle graphic include Red (1), Magenta (2), White (3), and Yellow (4).

**Note: The stage number must be a number between 0 and 5 (anything higher will distort the resolution and may affect runtime).**

A screenshot of a computer program

Description automatically generated

Once these parameters have been entered, a window will open displaying the generated graphic and a .png file will be created holding a copy of the graphic image. The illustration above demonstrates a user generating a Koch’s Snowflake graphic which will save to a file named ‘snowflake.png’, have a color of white and terminate its generation at stage 4. The image below illustrates the graphic generated using these parameters:

A white snowflake with black background

Description automatically generated

Closing the graphic window will trigger the application to redisplay the main menu where the user may generate another graphic or end the program.

# Introduction to Recursive Graphics Topic

Recursive graphics is a method of generating computer graphics. It can be used to create a wide variety of visual patterns and designs by establishing a set of rules and calling them to transform a shape through the process of recursion. This happens when a function is used and calls upon itself, using modified parameters each time, essentially causing a series of transformation stages to occur depending upon the number of times the recursive call happens.

There are a wide variety of different algorithms that implement recursive graphics. The key thing that all of these employ is the usage of a base case; to determine when the program will end. Followed by a conditional statement for when the base case is not active; this is used to organize and call the recursive function calls to properly increment stages of the program and output a ‘recursive’ graphic for the user to view.

# Introduction to the Project

We cover three specific algorithms within this project specifically: Hilbert’s Curve, Sierpinski’s Triangle, and Koch’s Snowflake. These three algorithms will be broken down into further detail within the sections below. To summarize them; Hilbert’s algorithm will generate a Curve graphic that recursively draws additional curves to portray a pattern of curves. Sierpinski’s algorithm will recursively draw triangles stacked to form larger triangles, containing more triangles. Finally, Koch’s Snowflake will recursively draw triangles; with additional triangles on each of the sides of the previous triangle(s), which will effectively draw a snowflake.

In order to more comfortably implement these algorithms, our project utilizes the Simple and Fast Multimedia Library (SFML); which provides a wide variety of template classes which we can use to more precisely and efficiently run various calculations and produce a graphic window in which our patterns and images can be visually represented. Some key information regarding some of the template classes can be found in the section below, then we will dive into our algorithm breakdowns.

## Hilbert’s Curve Algorithm Analysis and Breakdown

A screenshot of a computer

Description automatically generatedHilbert’s Space-Filling Curve is an algorithm that was first described and designed by German mathematician David Hilbert in 1891, and it is a continuous fractal space-filling curve. It can be implemented iteratively or recursively, but to most ends, it requires some form of iteration from the base case of a 1st order curve to Nth order curves. The base case can be implemented in four ways, in which it is a set of points that are connected to create a “cup” or a ‘U’ shape. The general form of the algorithm is as follows:

1. Construct a NxN grid, where N >= 2, for a total of 4 grid tiles.
2. Pick a point to start from, traverse each adjacent tile without revisiting, and draw a line from the starting point to the ending point. This is your 1st order curve.
3. For each order, N \*= 2, resulting in subdividing the grid-space, requiring N curves to be implemented, requiring N grid tiles for each curve.
4. Using the coordinates of the previous case, generate the required A, B, C, or D curves in the new unfilled quadrants, and connect them from a starting point to the ending point continuously.
5. Generate higher orders of the Hilbert Curve by recursively calling steps 3, 4, and 5, which is possible as curves that are higher than 2nd order are simple transformations of the N-1 order curves.

### Stages of Hilbert’s Curve

As described above, an example first order Hilbert Curve can be any permutation of the four starting points when drawing the first cup. For example, we’ll start in the bottom-left quadrant, creating a downward cup shape:

**A red line on a white background

Description automatically generated**

Using the base case, we can generate the second order curve by tracing over the base case, copying and transforming it twice, and then connecting the curves through one continuous line.  As we had started in the bottom-left corner in our base case, we can see that the firsts point also starts in the bottom-left most grid tile in the entire graph.

A grid with red and blue lines

Description automatically generated

Past the second order, we can see that it is simply four copies of the previous case: two identical copies, one rotated clockwise, and one rotated counterclockwise. As with the previous case, we connect the copies starting from the bottom-left corner to the right-most corner.

**A maze with red lines

Description automatically generated**

At any higher order curve, we are able to effectively generate a maze-like structure, where a line visits every new subdivided quadrant created from the previous order.

A red square pattern on a white background

Description automatically generated

Starting with the beginning of the algorithm, we first call our initiateCurve function, which takes in the passed name of the file that we want to save our image output to. To initialize our curve, we start by opening a window through the sf::RenderWindow function, by using the constant windowWidth and windowHeight of 1000x1000 pixels, we are able to calculate the start position by finding the center position of the top-left quadrant of the screen, which is where we start each and every curve from. By using the sf::Vector2f data type throughout our code, we can easily use a coordinate system that’s compatible with the drawing functions of the SFML library.

A computer screen with text

Description automatically generated

In the middle of the initialize curve function, we call the constructor for the HilbertCurve object, passing in the requested order, the starting position as a Vector2f, and the initial offset, which is thus re-calculated within the recursive function for higher order curves.

For the design of the recursive function, we needed to be able to pass in a new coordinate to be operated on, the order or ‘level’ of the curve’s generation, and the direction we want to draw a cup in. For example, we consistently call the left-ward opening cup as our initial case, as this decides the order in which our points are manipulated for all levels of the curve. Furthermore, if the order is greater than 1, our recursive call changes the direction of the curve we want to generate by manipulating the ‘direction’ variable, as well as decrementing the order to allow for a return to the higher calling function. By decrementing back to an order of 0, we push back the coordinate to the points member variable. Because each 1st case has a unique pattern in which its second case is consistently generated with, we are able to predictably generate the corresponding sets of cups with the function.

A screenshot of a computer program

Description automatically generated

For instance, if we call our recursive function with direction 0 and an order of 2, then direction 0 will call a rightward opening cup, itself twice, and then a downwards opening cup, which is synonymous with the standard Hilbert Curve pattern formation, as seen again below.

A computer screen shot of a program

Description automatically generated

Finally, once all of the coordinates are generated and pushed into the points vector, our program creates an image and texture object, calls our ‘draw’ function to create a sf::VertexArray, append the lines to it so they can be drawn, and calls itself until there are no more points. At the end of our initializeCurve function, we save the output graphic into a file that was passed into the beginning of the program, and we are done generating the Hilbert Curve.

As a general note for the algorithm, because the number of vertices can be found by raising the initial case’s amount to the power of two, we know that the equation for the amount of vertices can be interpreted as V(x) = 4n, where n is the order of the graph being computed, meaning that the time complexity of generating the curve is O(4n).

## Sierpinski’s Triangle Algorithm Analysis and Breakdown

The Sierpinski triangle, sometimes referred to as the Sierpinski gasket, is a fractal named after its creator Waclaw Franciszek Sierpinski; the Polish mathematician. The discovery was made in 1915. The construction of this shape involves taking an equilateral triangle and dividing it into 4 smaller triangles and removing the center one. You repeat this process over and over again creating more sub-triangles.

### Stages of Sierpinski’s Triangle

1. We begin with our first equilateral triangle.

A black triangle on a white background

Description automatically generated

1. We then divide this triangle into 4 congruent smaller triangles. We do this by finding the midpoint of each side and connecting those points together.
2. Our next step is to remove the triangle in the center, which leaves behind 3 disconnected equilateral triangles.

A black triangle with white triangle

Description automatically generated

1. This next step is where the recursion comes into play. You can repeat steps 2 and 3 to each of the smaller triangles left behind. You can call this recursion function as many times as you wish until you’ve reached your desired iteration of the triangle. (below are the next 4 iterations).

A black and white triangle shapes

Description automatically generatedA black and white triangle shapes

Description automatically generated

The number of subtriangles generated can be calculated using the following sequence relation: stage 1 = 1, stage 2 = 3, stage 3 = 9, stage 4 = 27, stage 5 = 81, and so on. The formula to calculate the number of triangles generated at any given stage is n = 3 ^ (i - 1) where n represents the number of subtriangles and i represents the number of iterations (or the current stage).

The triangle algorithm begins with an initiate function which takes the following parameters: height of graphic window, width of graphic window, file name, main color to create graphic, contrast color to create graphic, and stage to control the number of iterations when generating the graphic. It then creates a SierpinskiTriangle object and sets the object member fields to the values passed to the initiateSierpinski function. The RenderWindow graphic window object is then also created and the construct\_SierpinskiTriangle function is called to begin the actual creation of the Sierpinski triangle graphic.

The construct\_SierpinskiTriangle function sets the coordinates of the parent triangle left and right corner coordinates and calculates the top point using the find\_top\_vector function and passing it those coordinates.

A screenshot of a computer screen

Description automatically generated

The program then uses the coordinate information to generate the first triangle by setting the points of the triangle of a ConvexShape object, filling that object with the main color selected by the user and drawing the triangle in the graphic\_window. The base case would just draw this initial triangle as seen below:

A black background with white text

Description automatically generated

To construct a Sierpinski triangle, we use recursion to generate the smaller triangles within the parent triangle. The recursive\_sierpinski\_HELPER function is passed the current coordinates of the three triangle vertices representing the endpoints of all three edges of the triangle:

A screenshot of a computer program

Description automatically generated

Beginning with the parent triangle, the coordinates of the vertices of the inner triangle are calculated based on the coordinates of the parent triangle. This calculation finds the halfway mark between the original vertices at each edge of the parent triangle, where the new vertices will lie. Once those vertices have been calculated, they are passed to the constructTriangle function which draws the inner triangle using the contrast color selected by the user. The recursive\_sierpinski\_HELPER then takes the new coordinates and continues drawing subtriangles based on the new coordinate calculations until the terminating condition is met.

A screenshot of a computer program

Description automatically generated

The function will continue to iterate until it reaches the final stage which will print the final subtriangle and end the recursive graphic drawing.

## Kock’s Snowflake Algorithm Analysis and Breakdown

Koch’s Snowflake is an algorithm designed by Helge von Koch, a famous mathematician. It recursively constructs a snowflake-like shape. The construction of which involves iteratively and recursively adding smaller and smaller triangles into the base of the previous triangle. To break this algorithm down further:

1. Construct a triangle with three sides of the same length.
2. Divide these sides into three segments, representing each side of the triangle.
3. From here you can replace the middle segment with two separate segments: Removing the middle segment from each side and replacing it with two segments of the same length, forming another triangle.
4. Next recursively call steps 2 and 3, in order to create new smaller triangles within the appropriate segments passed to the recursive functions. Which will draw smaller triangles around the sides of the previous triangle. This step will be repeated X times determined by the number of iterations you wish to complete. I.e. which stage of the triangle you would like to construct.

To help provide perspective as to the stages of the Koch’s Snowflake construction please view the demonstration below:

### Stages of Koch’s Snowflake

Typically, the first iteration of this algorithm will produce a traditional triangle. As seen below:

A pink triangle on a black background

Description automatically generated

The second iteration of this algorithm will produce a hexagram, or two triangles on top of each other. As seen below:

A pink star on a black background

Description automatically generated

The third iteration of this algorithm will start to develop hexagrams on each of the cardinal points established in iteration 2. As seen below:

A pink star shaped object

Description automatically generated  
  
The fourth iteration is where the fractal turns into the traditional Koch’s Snowflake. As seen below:

A pink star shaped object

Description automatically generated

The fifth iteration and beyond; continue to modify the snowflake, to the limit of your computer's hardware. Eventually running the program will cause the application window to crash if you attempt to process too many iterations. Below is iteration 5:

A pink and black snowflake

Description automatically generated

You can calculate the number of sides the snowflake will have by following a sequence relation: stage 0 = 3. Stage 1 = 12, stage 2 = 48, stage 3 = 192, and stage 4 = 768, and so on. The formula for this is n = 3 \* 4^i. Where n represents the number of sides, and i represents the number of iterations (or the current stage).

The snowflake algorithm code functions off of a simple initiate function; which is passed a few parameters: a height, a width, a file name, a user chosen color, and what stage the user wants the graphic to create. From here, within our initiate function an object of our class type is created and the parameters are used to set basic member variables. The RenderWindow graphic window object is then also created and the first method is called… construct\_snowflake().

Construct snowflake sets two coordinates to the top left and bottom right of the screen. These will be used to determine the starting point of our snowflake. Then it creates a left and right point of a triangle using these modified values. It then needs to calculate a top point. So the program will call our first helper function find\_top\_vector. Which when passed a left and right point can calculate the appropriate coordinate above and in the middle of the two passed coordinates. Below you can see the breakdown of the find vector helper; which takes a left and right point and will return a vector containing the top coordinate based upon the parameters. This is done by defining a side length, and a height of the triangle. Then using another helper function to calculate the center of the left and right coordinates. Then Normalizing the coordinates before finally returning the top point coordinates set by the height of the triangle. As seen below:

A screenshot of a computer screen

Description automatically generated

After the top point has been established using the helper found above, the recursive aspect of the function will take place. As you can see below; our base case draws a triable on the center of the graphic window:

A screenshot of a computer

Description automatically generated

Otherwise, recursion must occur because we need to iterate through a multitude of stages to reach our end goal. If this is the case, the triangle will still be constructed, but the recursive helper function will be called given the three sides as parameters. You can see this in the image below:

A screenshot of a computer screen

Description automatically generated

The left -> right points create the base of the triangle. The Top -> left points create the left side, and the top -> right points create the right. From here within each of the recursive helper functions we have a simple if else. If the stage passed to it is equivalent to the end stage the user specified at the beginning, then the helper will simply calculate three new points; left, right, top and draw a final triangle before returning. If it still needs to iterate it will then calculate a point exactly one third and two thirds away from the current start/end points. Then using these two points it will find the top point using the helper described above. After these points are established; the recursive helper function is then called with each of the three sides of the smaller triangle. As seen below:

**A screenshot of a computer screen

Description automatically generated**

From here the function will continue to loop until the conditional If FinalStage == 0 is triggered as we discussed earlier. Which will print the final triangle and end the recursive graphic drawing. That concludes the runtime of a typical Koch’s Snowflake within SFML in C++14.

# Member Contributions (Link to sheets [HERE](https://docs.google.com/spreadsheets/d/19O4S9ikLLG0a7wsFFYNZiAYdWw9_vF0suITosEG15Ys/edit?usp=sharing))

*Below you will find screenshots of our member contributions sheet; for your convenience. Detailed within each version is a description of the intentions of our meetings as well as their outcomes. Included with date stamps and contribution by whom. Followed by any tasks created on certain days, and what the job entails, who started it and when, who finished it and when, and finally a description of the completed task or objective.* ***You can manually open this link within the header link, or via the links section at the bottom of the document.***

## Version 1.0:

A screenshot of a computer

Description automatically generated

## Version 2.0

**A screenshot of a computer

Description automatically generated**

## Version 3.0 (Final Version)

*A screenshot of a computer

Description automatically generated*

## Report and Presentation Contribution Breakdown

**A screenshot of a computer

Description automatically generated**

# Conclusion

Using recursion, we are able to create intricate patterns that utilize self-replicating structures. By designing a base case and a recursive case to work with, bringing mathematical algorithms to their completion through the usage of computational simulation, and their visualized products, to completion. Working with a graphics library in this context was integral to understanding how our algorithms worked, as studying the math algorithms and manually checking the arrays of coordinates can only go so far in debugging. Furthermore, being able to work with a library that we had never used before to reach our goals gave us more comfort and experience in understanding new libraries that can help break down large goals and objectives into manageable projects in the future.

As far as recursive graphics go, understanding and simulating the barebones implementation of mathematical algorithms is only the tip of the iceberg. In the past decades, vector graphics have been the proper foundation for disciplines such as artwork, where artists have made generations of artwork utilizing recursive graphics, such as with the Mandelbrot fractal. In game development, developers of popular retro games from the late 20th Century, such as Aztarec or Lunar Lander, largely utilized vector graphics to simulate the entire interface of their games. Without the use of vector graphics, we also would not have the earliest forms of CRT monitors, air traffic control monitors, or vital improvement in technology such as the advancements in graphical interfaces. For architecture, there even exists a very literal analogue to recursive graphics, in which architects utilize a recursive method for breaking down their structures, such as with fractal patterns, ribbed vaults, or arches in Gothic cathedrals and other historical locations.

Although not all of these implementations were designed behind a computer screen or through coding these structures, using recursion as a way to implement graphics that are based on these concepts is proof enough to show that the theorem still holds true across disciplines. As we have seen the analogues between computer science theory and fields such as art, architecture, or electronics, we now see how important recursion throughout historical advancements has been. Despite the recent advancements in computer science over the past century, we have learned that recursion has been present throughout the world’s creativity far beyond the development of the first electronic computers. With this knowledge in tow, it’s likely that if we were to travel far into the future, we would still be able to acknowledge and find elements of recursion there, albeit in new forms.

# Links List

* SFML Download Site: <https://www.sfml-dev.org/download/sfml/2.6.0/>
* Member Contribution Sheet: <https://docs.google.com/spreadsheets/d/19O4S9ikLLG0a7wsFFYNZiAYdWw9_vF0suITosEG15Ys/edit?usp=sharing>
* Github Repository: <https://github.com/NathanielJBrown97/CSC212-Project-2>