RECURSIVE GRAPHICS
212 Project

Professor Jonathan Schrader

Nathaniel Brown
Rebecca Iselin
Aeyva Rebelo
Alexandria Sampalis

July 23, 2023

Table of Contents

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 reiterate we **assume** the **prior-possession** of the following **REQUIRED** resources:

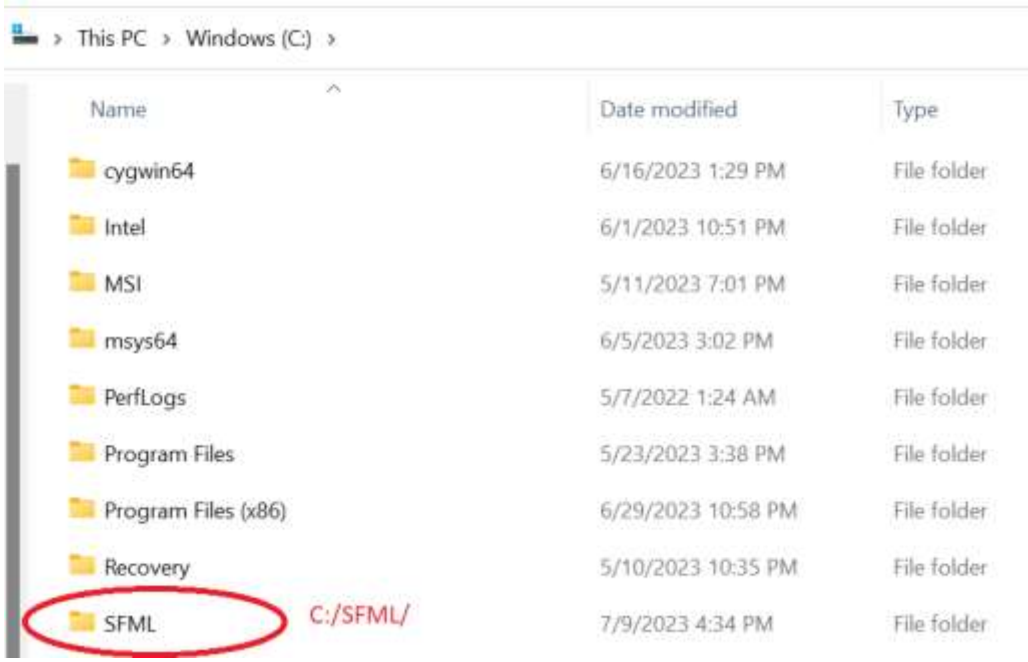
1. **Windows 11**
2. **Clion by Jetbrains**
3. **C++14**
4. **Simple and Fast Multimedia Library (Instructions found below).**

Instructions for Setup of SFML Library on CLion

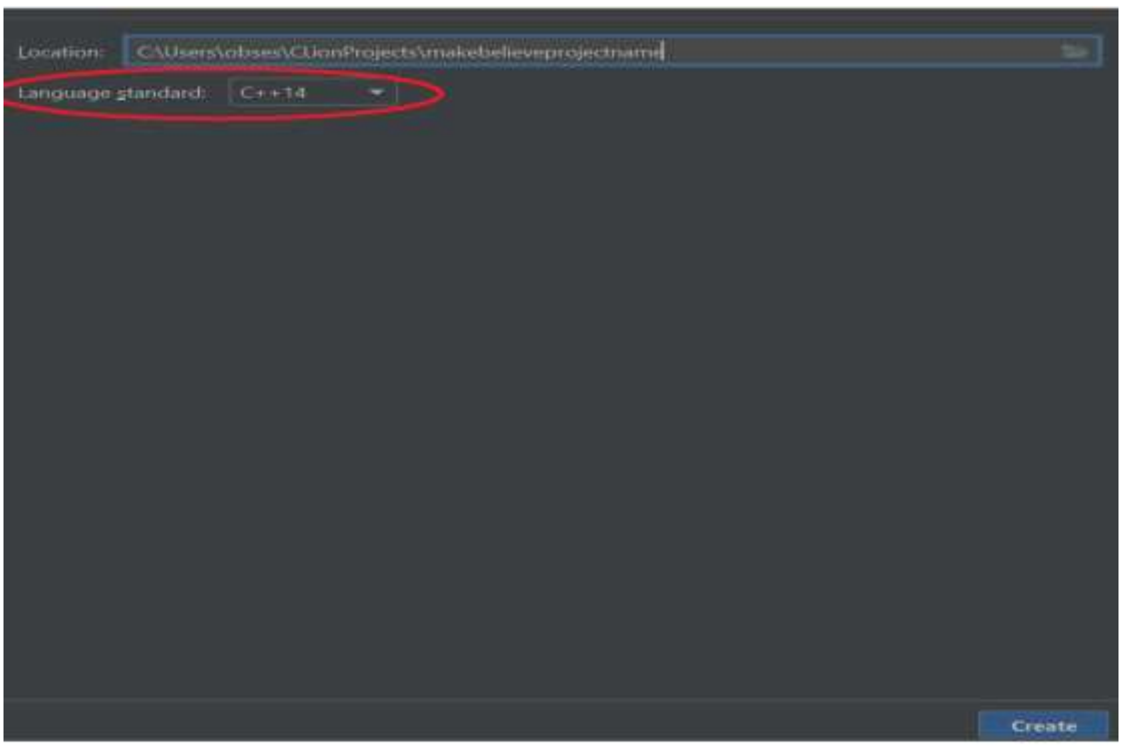
SFML Setup for Windows

1. Download here… *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*

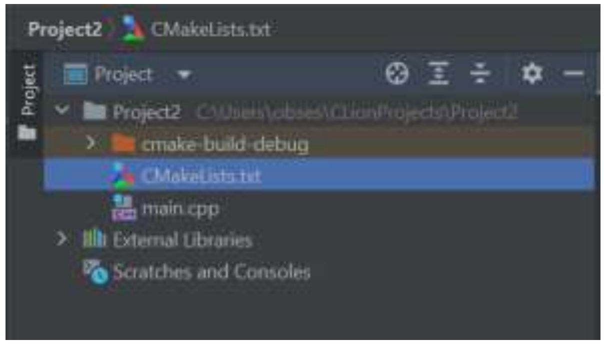
2. *Extract and move ENTIRE SFML folder to desired directory… for example I store mine directly within C:/*



3. *Create a new project in CLion… set standard language to C++ 14.*



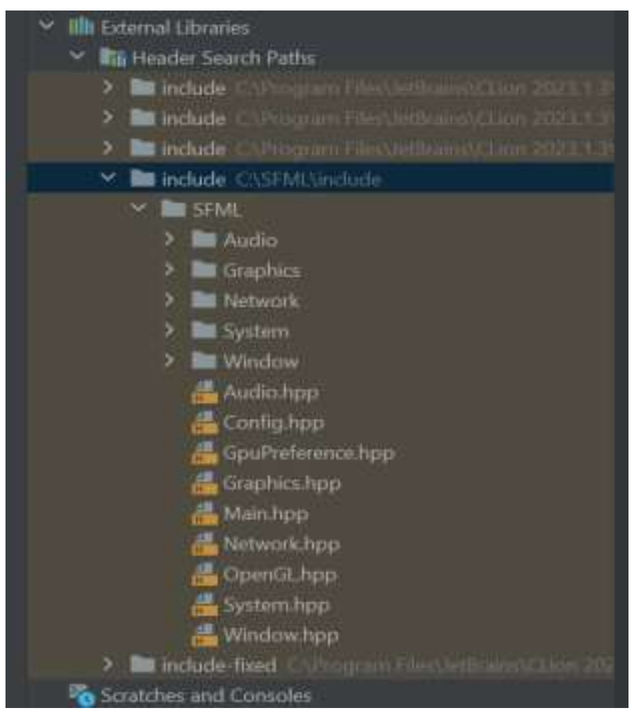
4. *Locate CMakeLists.txt within your new project (name depending on your choice… Mine is ‘Project2’ **NOTE:** Whenever you see project2 referenced; replace with your project folders name.*



5. *Edit CMakeLists.txt file to look like the following; **NOTE** adjust paths to reflect where you store SFML and directory name to reflect yours.*

```
cmake_minimum_required(VERSION 3.25)
project(Project2)

set(CMAKE_CXX_STANDARD 14)

add_executable(Project2 main.cpp)

set(SFML_STATIC_LIBRARIES TRUE)
set(SFML_DIR C:/SFML/lib/cmake/SFML)
find_package(SFML COMPONENTS system window graphics audio network REQUIRED)

include_directories(c:/SFML/include)
target_link_libraries(Project2 sfml-system sfml-window sfml-graphics sfml-audio sfml-network)
```

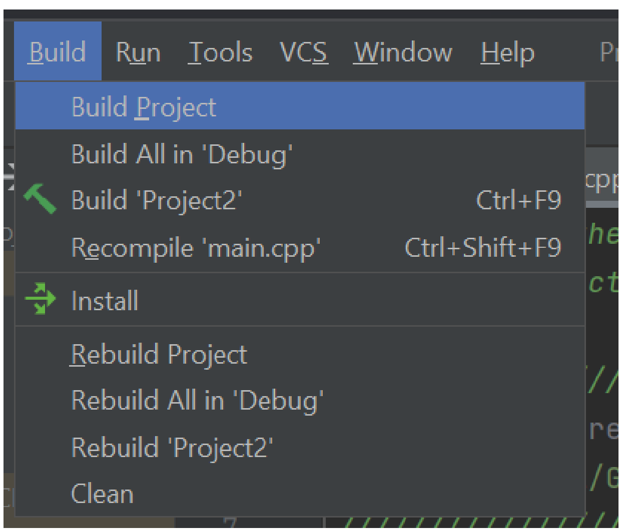
6. *This should create a new file in your external libraries that looks like this:*

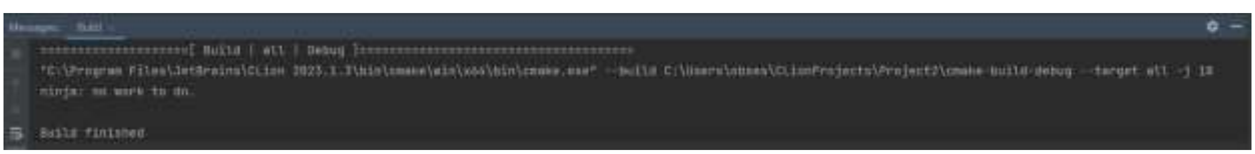


7. *From here you are ready to test if SFML library is properly installed: Copy this code into your main.cpp to test…*

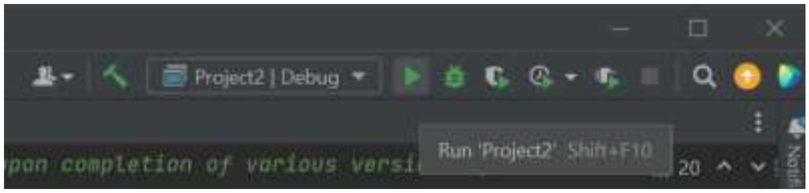
```
////////////////// Libraries //////////////////
#include <SFML/Graphics.hpp>
//////////////////////////////////////////////

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();
}


}
```

8. *You **CANNOT** compile using the traditional terminal methods. You must select **'build project'**.*

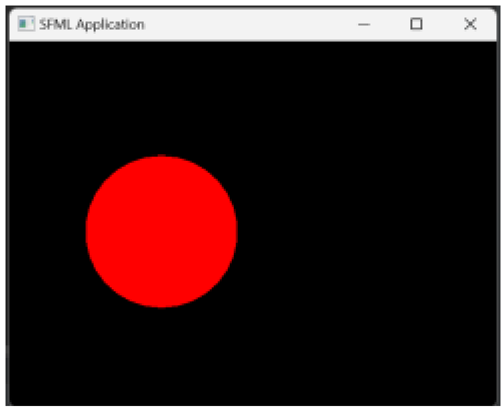




9. *If you do not get a build finished message; you have the wrong version of SMFL or pathing is inconsistent. Or you're not using C++14. Now you can select 'Run'.*



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



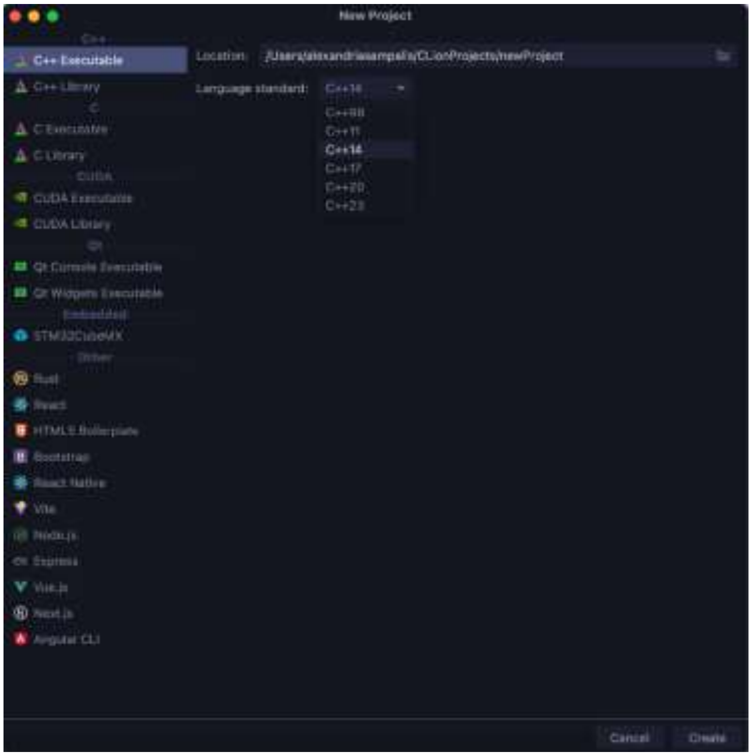
SFML Setup for MAC (Not Assured – Supplementary Instructions)

1. Download here. 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+)".

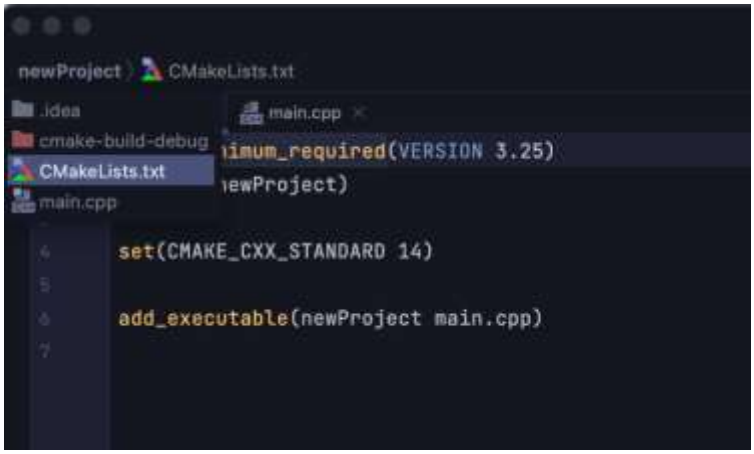
macOS	Clang - 64-bit (OS X 10.7+, compatible with C++11 and libc++)	Download | 5.07 MB
	Clang - ARM64 (OS X 11.0+)	Download | 5.02 MB
	macOS libraries are only compatible with 64-bit and ARM64 (M1 / M2) systems.	

2. Download the extension Homebrew here. It is recommended to do so to help streamline the process by avoiding any issues that come up with copying and locating the libraries.

3. Once homebrew is installed open your terminal and type: *‘ brew install sfml ‘*. This will begin the process of installing SFML to your machine.
4. We can now go ahead and create a new CLion project. Be sure to set standard language to C++ 14.



5. 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.



6. *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*

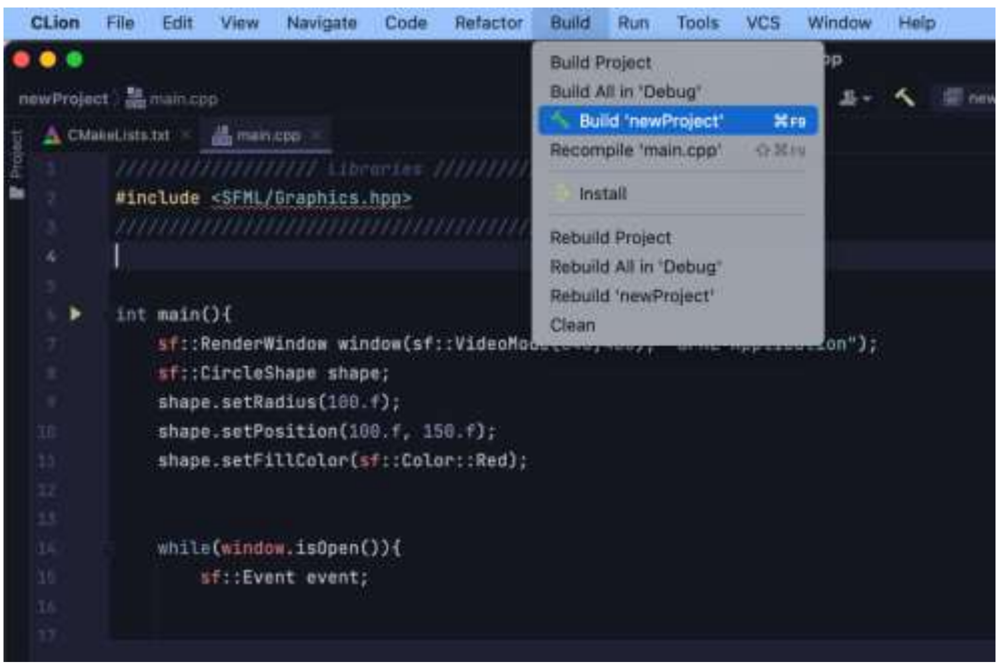
```
cmake_minimum_required(VERSION 3.14)
project(newProject)

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)
```

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

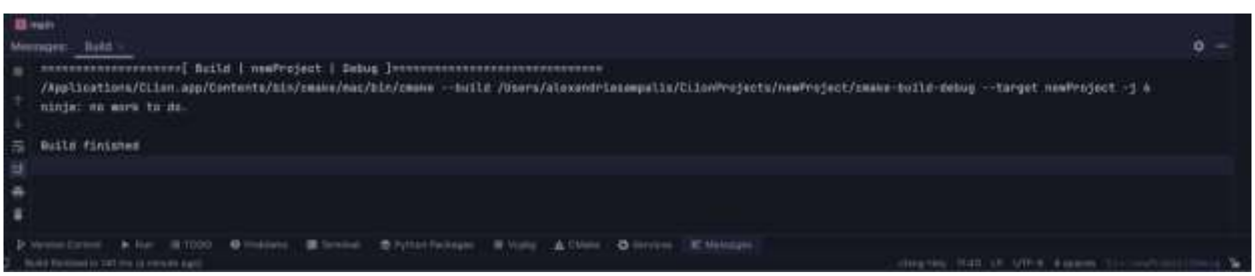
```
//////////////////// Libraries ////////////////////
#include <SFML/Graphics.hpp>
//////////////////////////////////////////////////


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();
    }


}
```

8. *You cannot compile this code, you must select '**build project**'*

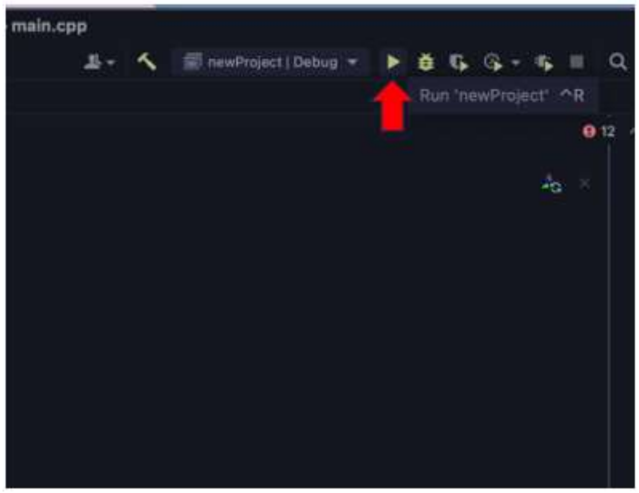


If the build is successful, you will see this message appear:

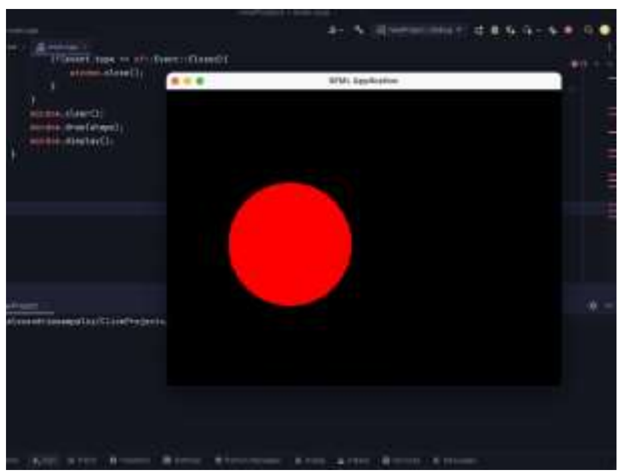


If not, you might need to troubleshoot, you may not be using C++14, or you could have run into an issue downloading SFML.

9. *Once your build is successful you can select ‘Run”.*



10. *If this menu pops up, you are ready to run and Compile the Project.*



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

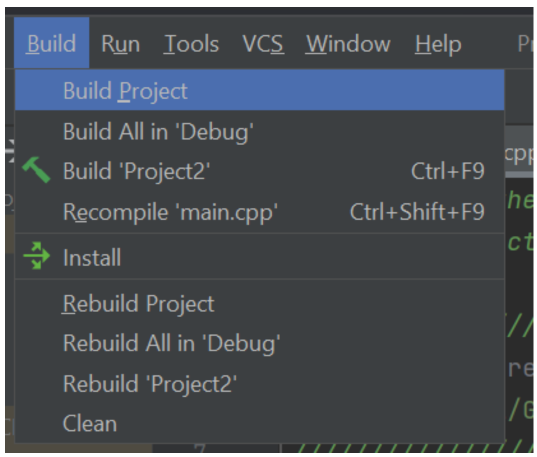
see the results of various SFML commands. I.E. This creates a window application in which you can view the final iteration of a recursive graphic call.

- 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

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).

```
Welcome to our project 212 user interface. Please read through following menu.
Upon selection of your algorithm of choice you will be requested for input of the
appropriate parameters required for the construction of the graphic.

1.) Hilbert's Curve
2.) Sierpinski's Triangle
3.) Koch's Snowflake
4.) End Program

Please input the number of your choice:
```

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.

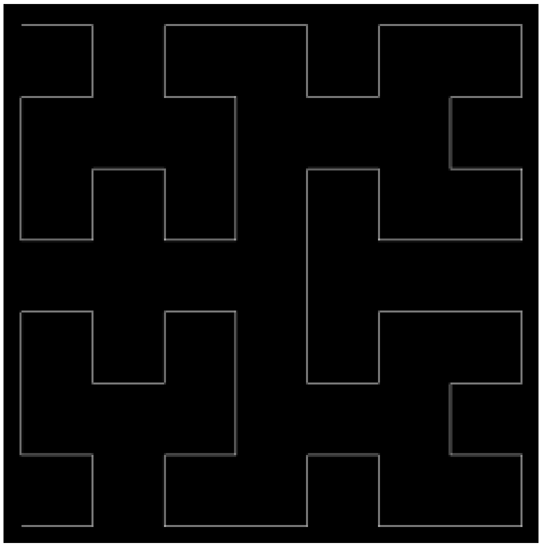
```
Welcome to the Hilbert's Curve Algorithm
Please follow the prompts and enter a file name for your image creation. *Must* end with .png.
```

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

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

```
Please enter the file name you wish to save the Hilbert's Curve to: curve.png
```

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:



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.

```
Welcome to the Sierpinski's Triangle Algorithm
Please follow the prompts and enter a file name for your image creation. *Must* end with .png.
Then select a color from the following menu (Input the corresponding number):
1.) Red
2.) Magenta
3.) White
4.) Yellow
```

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).

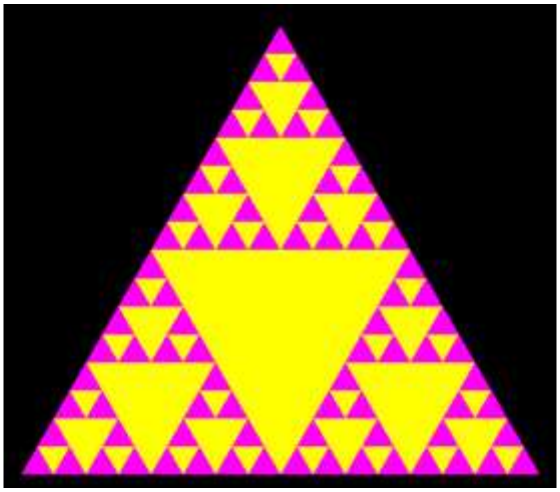
Note: The file name must contain a .png extension.

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).

```
Please enter the file name you wish to save the Koch's Snowflake to:  triangle.png
Please enter your main color selection. (Type the corresponding number): 2
Please enter your contrast color selection. (Type the corresponding number): 4
Please enter the stage of Sierpinski's Triangle you would like. (Enter a number from 0 - 5;
 anything higher will affect runtime.): 5
```

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:



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.

```
Welcome to the Koch's Snowflake Algorithm
Please follow the prompts and enter a file name for your image creation. *Must* end with .png.

Then select a color from the following menu (Input the corresponding number):
1.) Red
2.) Magenta
3.) White
4.) Yellow
```

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).

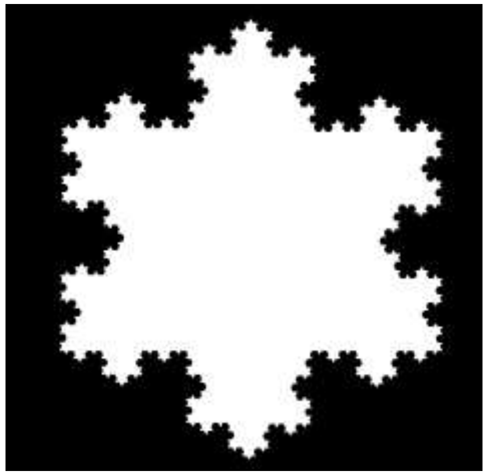
Note: The file name must contain a .png extension.

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).

```
Please enter the file name you wish to save the Koch's Snowflake to:  snowflake.png

Please enter your color selection. (Type the corresponding number): 3
Please enter the stage of Koch's Snowflake you would like. (Enter a number from 0 - 5;
anything higher will affect runtime.): 4
```

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:



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.

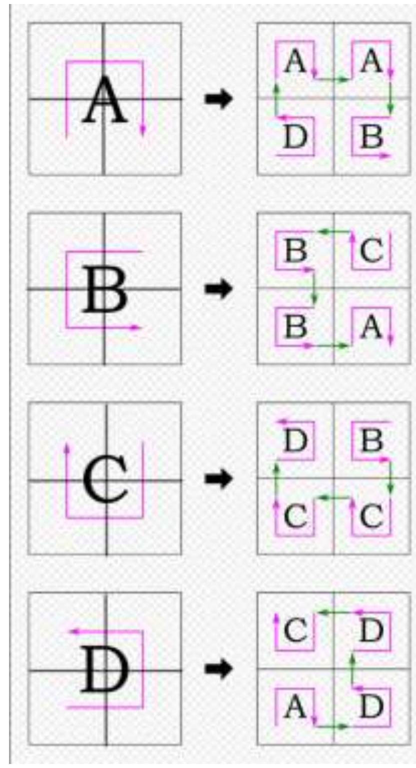
Introduction to the Project

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

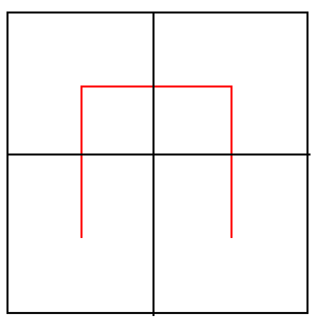
Hilbert's Space-Filling Curve is an algorithm that was first described and designed by German mathematician David Hilbert in 1891, and it is a continuous fractal space-filling curve. It can be implemented iteratively or recursively, but to most ends, it requires some form of iteration from the base case 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:



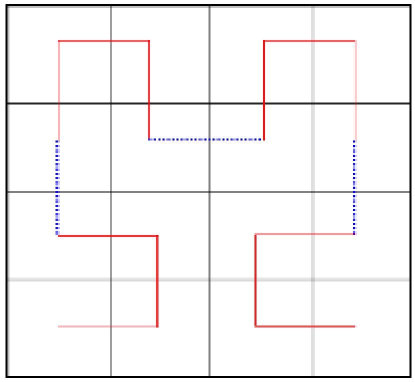
1. Construct a NxN grid, where N >= 2, for a total of 4 grid tiles.
2. Pick a point to start from, traverse each adjacent tile without revisiting, and draw a line from the starting point to the ending point. This is your 1st order curve.
3. For each order, N *= 2, resulting in subdividing the grid-space, 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

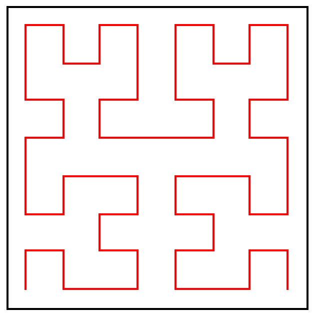
As described above, an example first order Hilbert Curve can be any permutation of the four starting points when drawing the first cup. For example, we'll start in the bottom-left quadrant, creating a downward cup shape:



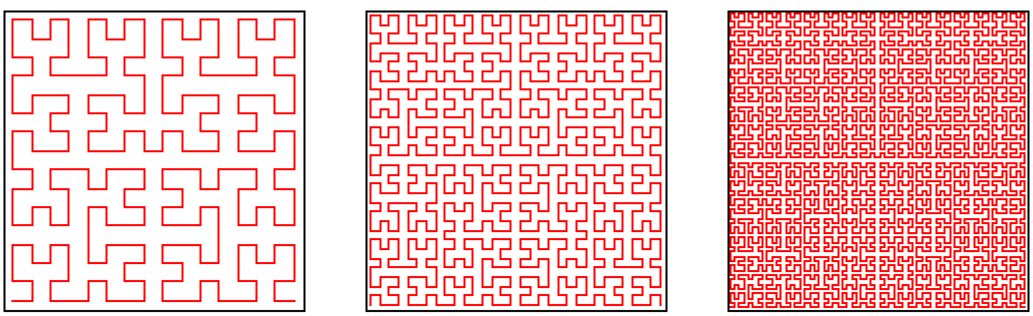
Using the base case, we can generate the second order curve by tracing over the base case, copying and transforming it twice, and then connecting the curves through one continuous line. As we had started in the bottom-left corner in our base case, we can see that the firsts point also starts in the bottom-left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



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.



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.



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.

```cpp
int initiateCurve(std::string fileName) {
    const int windowWidth = 1000;
    const int windowHeight = 1000;
    // 7 is the max you can achieve before the graphics become too small to see
    const int order = 3;
    const float sideLength = 600.0f;
    const sf::Vector2f startPosition(windowWidth / 2 - sideLength / 2, windowHeight / 2 - sideLength / 2);

    sf::RenderWindow window(sf::VideoMode(windowWidth, windowHeight), "Hilbert Curve");
    window.setFramerateLimit(60);

    HilbertCurve hilbert(order, startPosition, sideLength);
```

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.

```cpp
void HilbertCurve::generateCurve(int order, sf::Vector2f start, float sideLength, int direction) {
    // Base case: If the current order is zero or less, add the starting point to the points vector and return.
    if (order <= 0) {
        points.push_back(start);
        return;
    }

    // Calculate half of the side length for the smaller sub-curve.
    float half = sideLength / 2.0f;

    // Recursively generate the four smaller sub-curves based on the current direction.
    switch (direction) {
        case 0: /// OPENS LEFT ⅃
            // Generate the sub-curve upwards.
            generateCurve(order - 1, start, half, (direction + 1)%4);
            // Generate the sub-curve to the right.
            generateCurve(order - 1, sf::Vector2f(start.x + half, start.y), half, direction);
            // Generate the sub-curve downwards.
            generateCurve(order - 1, sf::Vector2f(start.x + half, start.y + half), half, direction);
            // Generate the sub-curve to the left.
            generateCurve(order - 1, sf::Vector2f(start.x , start.y + half), half, (direction + 2) % 4);
            break;
```

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.

```
        break;
    case 1: /// OPENS UP ^
        // Generate the sub-curve in the downward direction.
        generateCurve(order - 1, start, half, (direction + 3) % 4);
        // Generate the sub-curve to the right.
        generateCurve(order - 1, sf::Vector2f(start.x, start.y + half), half, direction );
        // Generate the sub-curve upwards.
        generateCurve(order - 1, sf::Vector2f(start.x + half, start.y + half), half, direction);
        // Generate the sub-curve to the left.
        generateCurve(order - 1, sf::Vector2f(start.x + half, start.y), half, (direction + 2) % 4);
        break;
    case 2: /// OPENS DOWN
        // Generate the sub-curve upwards.
        generateCurve(order - 1, sf::Vector2f(start.x + half, start.y + half), half, (direction + 1) % 4);
        // Generate the sub-curve to the left.
        generateCurve(order - 1, sf::Vector2f(start.x + half, start.y), half,  direction);
        // Generate the sub-curve downwards.
        generateCurve(order - 1, start, half, direction);
        // Generate the sub-curve to the right.
        generateCurve(order - 1, sf::Vector2f(start.x, start.y + half), half, (direction + 3) % 4);
        break;
    case 3:
        /// OPENS RIGHT :
        // Generate the sub-curve to the right.
        generateCurve(order - 1, sf::Vector2f(start.x + half, start.y + half), half, (direction + 1) % 4);
        // Generate the sub-curve in the downward direction.
        generateCurve(order - 1, sf::Vector2f(start.x, start.y + half), half, direction);
        // Generate the sub-curve to the left.
        generateCurve(order - 1, start, half, (direction + 2) % 4);
        // Generate the sub-curve upwards.
        generateCurve(order - 1, sf::Vector2f(start.x + half, start.y ), half, direction);
        break;
    default:
        break;
```

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 $V(x) = 4^n$, where n is the order of the graph being computed, meaning that the time complexity of generating the curve is $O(4^n)$.

Sierpinski's Triangle Algorithm Analysis and Breakdown

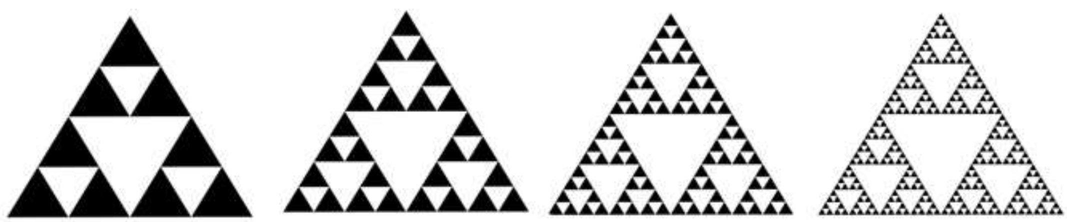
The Sierpinski triangle, sometimes referred to as the Sierpinski gasket, is a fractal named after its creator Waclaw Franciszek Sierpinski; the Polish mathematician. The discovery was made in 1915. The construction of this shape involves taking an equilateral triangle and dividing it into 4 smaller triangles 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.

2. 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.
3. Our next step is to remove the triangle in the center, which leaves behind 3 disconnected equilateral triangles.

4. 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).



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.

```
/// Helper - Find Top Point Vector
// Takes two 2d vectors containing coordinates for the left and right of base of the triangle.
sf::Vector2f find_top_vector(sf::Vector2f& baseLeftPoint, sf::Vector2f& baseRightPoint)
{
    /// Get height of triangle
    float side_length = std::sqrt( std::pow( baseLeftPoint.x-baseRightPoint.x, 2)*1.0f + std::pow( baseLeftPoint.y - baseRightPoint.y, 2) * 1.0f);
    float height_of_triangle = (side_length*std::sqrt( 3))/2.0f; // the side length * the sqrt of 3 divided by 2 is the height

    sf::Vector2f center = find_center_vector( baseLeftPoint, baseRightPoint); // define a center point using helper passed left and right; from active fun
    sf::Vector2f combinedPointVector = sf::Vector2f( (baseLeftPoint.x-baseRightPoint.x), (baseLeftPoint.y - baseRightPoint.y));

    combinedPointVector = normalized_vector( combinedPointVector); // normalizes the passed vector before rotating
    combinedPointVector = sf::Vector2f( -combinedPointVector.y, combinedPointVector.x); // redefine vector to the negated y value of vector AS THE X value,
    return sf::Vector2f( center.x + combinedPointVector.x*height_of_triangle, center.y + combinedPointVector.y*height_of_triangle); // return a 2d vector
}
```

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:

```
if(FinalStage == 0)
{
    //end recursive calls
    constructTriangle(top, left: leftPoint, right: rightPoint, setColor: mainColor, &: graphic_window);
}
```

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:

```
constructTriangle(top, left: leftPoint, right: rightPoint, setColor: mainColor, &: graphic_window); // cr

recursive_sierpinski_HELPER( pointA: leftPoint, pointB: rightPoint, currentIteration: 1, &: graphic_window);
recursive_sierpinski_HELPER( pointA: top, pointB: leftPoint, currentIteration: 1, &: graphic_window); // dra
recursive_sierpinski_HELPER( pointA: rightPoint, pointB: top, currentIteration: 1, &: graphic_window); // dr
```

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.

```
sf::Vector2f nextTopPoint = sf::Vector2f( 0.5 * (endPoint.x - startPoint.x) + startPoint.x, 0.5 * (endPoint.y - startPoint.y) + startPoint.y);
sf::Vector2f nextLeftPoint = find_top_vector( startPoint, nextTopPoint);
sf::Vector2f nextRightPoint = find_top_vector( nextTopPoint, endPoint);

//draw a triangle passed the above variables in the format of top > left > right > color > graphic_window
constructTriangle( nextTopPoint, nextLeftPoint, nextRightPoint, contrastColor, graphic_window);

// and recursively call the same function for all the newly created sides
/// Then... recursively call this function for EACH of the newly created sides, AND INCREMENT CURRENT ITERATION to ensure a stop point.
recursive_sierpinski_HELPER(startPoint, nextTopPoint, currentStage+1, graphic_window);
recursive_sierpinski_HELPER( nextTopPoint, endPoint, currentStage+1, graphic_window);
recursive_sierpinski_HELPER( endPoint, nextRightPoint, currentStage+1, graphic_window);
recursive_sierpinski_HELPER( nextTopPoint, nextLeftPoint, currentStage+1, graphic_window);
```

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

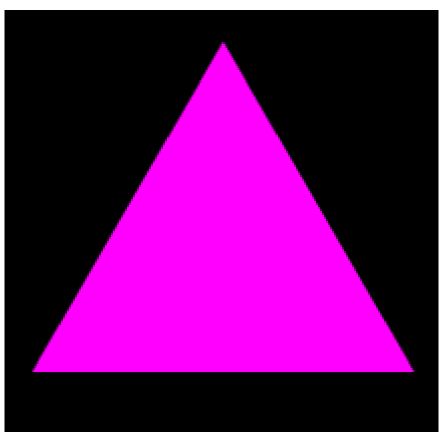
Koch's Snowflake is an algorithm designed by Helge von Koch, a famous mathematician. It recursively constructs a snowflake-like shape. The construction of which involves iteratively and recursively adding smaller and smaller triangles into the base of the previous triangle. To break this algorithm down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

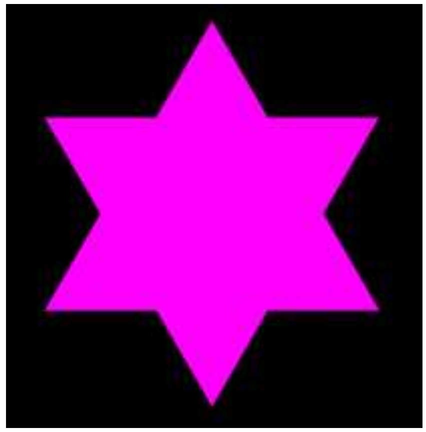
To help provide perspective as to the stages of the Koch's Snowflake construction please view the demonstration below:

Stages of Koch's Snowflake

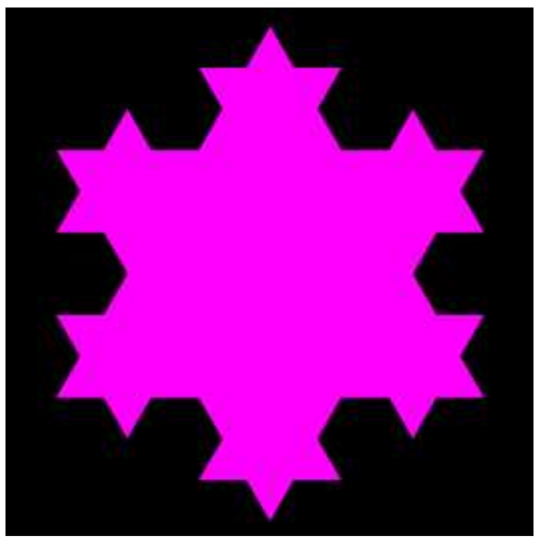
Typically, the first iteration of this algorithm will produce a traditional triangle. As seen below:



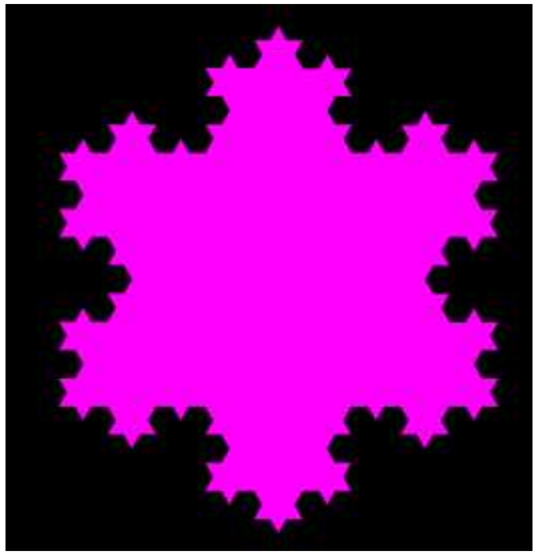
The second iteration of this algorithm will produce a hexagram, or two triangles on top of each other. As seen below:



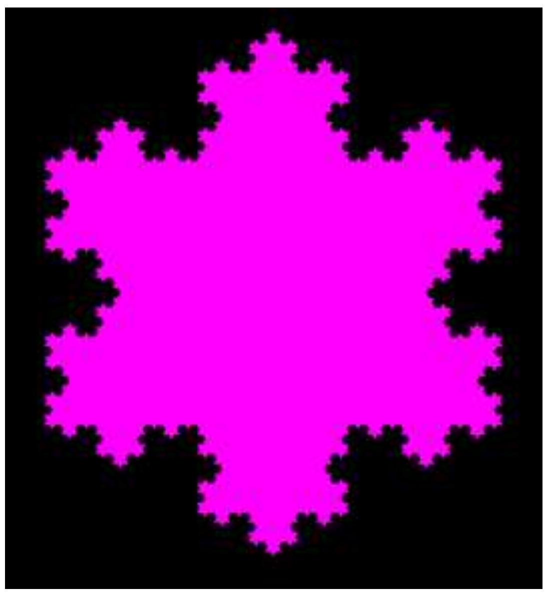
The third iteration of this algorithm will start to develop hexagrams on each of the cardinal points established in iteration 2. As seen below:



The fourth iteration is where the fractal turns into the traditional Koch's Snowflake. As seen below:



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:



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:

```
/// Helper - Find Top Point Vector
// Takes two 2d vectors containing coordinates for the left and right of base of the triangle.
sf::Vector2f find_top_vector(sf::Vector2f& baseLeftPoint, sf::Vector2f& baseRightPoint)
{
    /// Get height of triangle
    float side_length = std::sqrt( std::pow( baseLeftPoint.x-baseRightPoint.x, 2)*1.0f + std::pow( baseLeftPoint.y - baseRightPoint.y, 2) * 1.0f);
    float height_of_triangle = (side_length*std::sqrt( 3))/2.0f; // the side length * the sqrt of 3 divided by 2 is the height

    sf::Vector2f center = find_center_vector( baseLeftPoint, baseRightPoint); // define a center point using helper passed left and right; from active fu
    sf::Vector2f combinedPointVector = sf::Vector2f( (baseLeftPoint.x-baseRightPoint.x), (baseLeftPoint.y - baseRightPoint.y));

    combinedPointVector = normalized_vector( combinedPointVector); // normalizes the passed vector before rotating.
    combinedPointVector = sf::Vector2f( -combinedPointVector.y, combinedPointVector.x); // redefine vector to the negated y value of vector AS THE X value,
    return sf::Vector2f( center.x + combinedPointVector.x*height_of_triangle, center.y + combinedPointVector.y*height_of_triangle); // return a 2d vector
}
```

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:

```
/// If Stage 0; NO RECURSION.
if(FinalStage == 0)
{
    // Call the triangle creation; passed the three points of the triangle, the color, and the window object.
    constructTriangle(top, left: leftPoint, right: rightPoint, setColor: colorGraphic, &: graphic_window);
}
```

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:

```
else
{
    /// Draw the Triangle - Recursively calls snowflake draw; current iteration will increment within.
    constructTriangle(top, leftPoint, rightPoint, colorGraphic, graphic_window); // creates our initial triangle
    recursive_koch_HELPER( leftPoint, rightPoint, 1, graphic_window); // draws left -> right line
    recursive_koch_HELPER( top, leftPoint, 1, graphic_window); // draws top -> left line
    recursive_koch_HELPER( rightPoint, top, 1, graphic_window); // draws right -> top line
```

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:

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)

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:

Version 1.0	Tasks:	Notes on Task:	Start Date:	Initiated By:	End Date:	Finished By:	Notes:
	Meeting 1: Planning 7/5	First meeting to go over project specifications, our goals and preliminary ideas to start the project, as well as construction of major milestones and a breakdown of the first milestone. As well as setting up our member contribution sheet and our repository	7/5	Nathaniel (Present), Rebecca (Present), Aeyva (Present), Alexandria (Absent - Sick)	7/5	Nathaniel (Present), Rebecca (Present), Aeyva (Present), Alexandria (Absent - Sick)	Thoroughly discussed the project as a whole. Split the initial algorithms into three sections. Aeyva - Curve, Rebecca and Alexandria - Triangle, Nathaniel - Snowflake. With the intentions of combining our programs into a simple conditional based user interface that will ask the user which algorithm they'd like to run, followed by a request for necessary input, and then calling the respective algorithm to output an image of the recursive graphic. **Eventually,** upon completion of the 3 assigned graphics we will work together as a group to create a fourth graphic of some kind. Discussed the basic gameplan for our report/member contribution sheet / and how to properly use our repository. Set next meeting date - with the goals of getting barebones of our respective assignments together. As well as generic construction of a user interface that we will insert our individual code into after submission finalization.
	Meeting 2: Work 7/11	Report back on work with our barebones code. Figure out where to move forward, and set new tasks required for finalization of version 2. *Version 1 will be completed by this day.	7/11	Nathaniel (Present), Rebecca (Present), Aeyva (Present), Alexandria (Present)	7/11	Nathaniel (Present), Rebecca (Present), Aeyva (Present), Alexandria (Present)	General barebone and pseudocode has been created for each of the algorithms. Decided the best way to move forward from here is to continue implementing our programs. Koch's Snowflake needs the recursion element added. Traingle needs base case draw function and recursive elements. Curve needs base case draw function and recursive elements. Similarly we need to work as a group to figure out a save image functionality using our SFML library. As well as introducing a parameter menu to customize our graphics. Thinking even further, into V.3 development, we have decided to meet every day next week after finalization of Version 2.0 in order to write the Report and finalize documents needed for submission. As well as practicing our project presentation with our last day together.
	Objective 1: Curve Program	Start the barebones of the curve program; research algorithm and start construction of the .cpp functional classes related to the topic.	7/5	Aeyva	7/11	Aeyva	Completed the installation of the SFML library, as well as the pseudocode for the Hilbert's Curve algorithm. Detailed the plans for a recursive call feature to properly iterate through the curves algorithm. Started implementation of initial class files and construction of the draw base case function. Needs to be finished; then working on implementing the recursive call.
	Objective 2: Snowflake Program	Start the barebones of the snowflake program; research algorithm and start construction of the .cpp functional classes related to the topic.	7/5	Nathaniel Brown	7/10	Nathaniel Brown	Finished: Slightly more than the skeleton. Deep dived on the template classes within the SFML library. Started following the algorithm for the Koch's Snowflake and managed to get a functioning class together that can output triangles. Not entirely finished, needs methods and recursive element to properly implement the snowflake recursion call; but the base is there... Also needs reoptimization to allow for the collection of user input to determine construction aesthetics, i.e. ask for color, stage of the snowflake, filename. As well as implementation of save as png function. **TL;DR:V1.0 is completed. Program can construct a triangle. Developed helpers that should be usable for recursive calls as well!** New goals will be set after meeting 2 on 7/11
	Objective 3: Triangle Program	Create project - add pseudocode for planning of program - begin development	7/5	Rebecca & Alexandria	7/8	Rebecca	Completed set up of SFML library. Got template project working. Completed pseudocode to implement coding plan.
	Objective 4: Install & Configure SFML	As a group: figure out how to properly configure SFML so we can actually start programming. (This is **strictly** an objective as adding a library has not been covered in any course material at URI thusfar)	7/5	All Members	7/9	Nathaniel Brown	This was a pain, but proper installation methods have been heavily detailed and shared in this document. This document will need slight editing and formatting for input within our final project report.
	Task 1: Basic User Interface	Throw together a generic conditional based user interface, which will be used to call each of the individual algorithms the group works on.	7/5	Nathaniel Brown	7/5	Nathaniel Brown	Finished. Basic template has been constructed. Consisting of a main function that will be used to call various helper void display function. The program prompts the user for a main menu user input, then directs the user to the appropriate algorithms display message and prompts for whatever parameters are required for that algorithms output. This will be adjusted as Objectives 1 through 3 are finished. Invalid data outputs and redirects for inputs out of range within the main menu. Additional validity checks can be addressed in the future for parameter constrictions.

Version 2.0

Version 2.0	Tasks:	Notes on Task:	Start Date:	Initiated By:	End Date:	Finished By:	Notes:
	Meeting 3: Update and Work 7/13	Meet with updated code for the three algorithms; see where we are at and establish more accurate tasks as needed.	7/13	All Members	7/13	All Members	Met with mostly finalized code for the three algorithms; evaluated what needed to be done to finish implementation for the project and set tasks that needed to still be handled.
	Meeting 4: Update and Work 7/18	Meet with finished implementations of algorithms and combine everything into a master file for submission.	7/18	All Members	7/18	All Members	Met with mostly finalized code for the three algorithms; evaluated what needed to be done to finish implementation for the project and set tasks that needed to still be handled.
	Objective 1: Implement Koch's Snowflake Recursion Aspect	Work on a recursive helper to continuously call the triangle code made in Version 1.0	7/13	Nathaniel Brown	7/15	Nathaniel Brown	Recursive function is completed. Program now works for various stages of Koch's Snowflake algorithm; however it does crash on stages higher than 10. This is likely due to hardware; not code. I will begin additional enhancements to the program to include some level of customizability via parameters in the user interface.
	Objective 2: Implement Kierpinski's Triangle	Implement Triangle algorithm utilizing the SFML and handle proper creation of related objects and construction of the base case drawing.	7/13	Rebecca & Alexandria	7/18	Rebecca & Alexandria	Created the initial Triangle creation algorithm and worked partway through the recursive elements. More implementation and work needs to be done to complete the recursive call as intended.
	Objective 3: Implement Hilbert's Curve	Implement curve algorithm utilizing the SFML and handle proper creation of related objects and construction of the base case drawing.	7/13	Aeyva	7/15	Aeyva	Handled the construction of the initial base case graphic. Need to continue the recursive elements and iterations of the program as a whole.
	Task 1: Save as Image Function	Implement a function that can be called to save the graphic of our algorithms to an image. As per project requirements.	7/13	All Members	7/15	Nathaniel Brown	A function was not possible due to the nature of the template class Render Window; could not pass a current state of the rendered graphic into another function to save. Therefore the save image functionality needs to be manually written for each version of our windows. The method has been created and distributed amongst the group.
	Task 2: Configure Submenu Parameters	Implement some level of customizability of your graphic via setting parameters within the user interface	7/13	All Members	7/19	All Members	Needed to wait until finalized code was in for all members; submenu has been altered to recieve the appropriate parameters needed for output. I.e. File names, color, and stage where necessary.

Version 3.0 (Final Version)

Version 3.0	Tasks:	Notes on Task:	Start Date:	Initiated By:	End Date:	Finished By:	Notes:
Final Version of code. Anything other than touch-up work was outside of expected finish time.	Meeting 5: 7/19	Meet to finalize program and piece seperate programs together, troubleshoot any remaining issues. Get submission in by the end of the 19th.	7/19	Nathaniel (Present), Rebecca (Present), Aeyva (Present), Alexandria (Absent - Family Emergency)	7/19	Nathaniel (Present), Rebecca (Present), Aeyva (Present), Alexandria (Absent - Family Emergency)	Programs are not completed for the CURVE or the TRIANGLE; these projects need to be hyper focused over night and brought to a conclusion. So we can start presentation practice for tomorrow. Report is 80% done missing instructions for the programs that are not fully implemented.
	Meeting 6: 7/20	Practice presentation; finalize the remaining bits of documents for submission	7/20	All Members	7/20	All Members	Finalized documents, submitted and practiced presentation for Tuesday at 3:30pm.
	Curve Objective 1: Outline Data Structure, Methods, and Variables	Develop getters and setters necessary for the algorithms' needs, as well as outline base methods required such as 'drawCup' and 'drawLine'	7/18	Aeyva & Alexandria	7/19	Aeyva & Alexandria	Curve's implementation is outlined with the basic functions, member variables within the curve object, and the overall call stack to simulate Hilbert's Curve.
	Curve Objective 2: Calculation Solutions	Need to solve for the equations to implement a scalable implementation of the curve	7/19	Aeyva & Alexandria	7/20	Aeyva & Alexandria	Design a scaled size variable that will help subdivide the small square chunks properly and offsets appropriately, as well as other necessary computations for the curve methods, numerical formulas for number of cups and curves based on the order of the curve, etc.
	Curve Objective 3: Demonstrate Algorithm Design	Algorithm from start to finish needs to be demonstrated for group understanding and to push out the final stages of the curve collaboratively	7/19	Aeyva & Alexandria	7/20	Aeyva & Alexandria	Curve algorithm is the last implementation needed, collaborate with team members for a full understanding on a working solution to it, starting from the input, to the member variables needed to prime the recursive algorithm, consistently implement the higher order cases based on the first order case chosen.
	Curve Objective 4: Recursive Functions	Develop recursive call to implement various stages of algorithms graphic	7/19	Aeyva & Alexandria	7/22	Aeyva & Alexandria	Recursive function was solved and implemented by Alexandria, needed further calculation adjustments to display appropriate graphic.
	Curve Object 5: Fix Point Calculations/Calls	The way points are passed needs to be reformatted, to display the appropriate graphic.	7/22	Aeyva	7/22	Aeyva	Solved, graphic is now appropriately displaying.
	Curve **Rescue Objective:**	Code was not implemented by the 7/18 due date. During last meeting **Alexandria** was starting to assist **Aeyva** in development of the final algorithms code. They broke off and both worked on seperate versions to try and get the program started.	7/20	Alexandria	7/22	Alexandria	Since Alexandria couldn't contribute significantly to the Triangle, and Aeyva did not have their Curve working or implemented past theory on **7/18**, Alexandria started to breakdown the Curve algorithm and implement her own code. She got the code working over the weekend. On 7/21 she had a fully functioning graphic; albeit with a few off lines. From here Aeyva was able to complete the curve.
	Triangle Objective 1: Develop Helper Function	Create function that outputs a midpoint between two given coordinates	7/18	Rebecca Iselin	7/19	Rebecca Iselin	Developed necessary helper functions that can be used within the initiate sierpinski function. They help calculate midpoints and establish coordinates needed for a future recursive call function.
	Triangle Objective 2: Develop Recursive Function	Create function that recursively draw a triangle given 3 midpoints, then iterates and calls itself on newly defined midpoints.	7/19	Rebecca Iselin	7/19	Rebecca Iselin	Developed the recursive function which uses coordinates established via helpers, to decide where to draw recursive triangles on the graphic.
	Triangle Objective 3: Finalize	Implement into main, tweak parameters	7/18	Rebecca Iselin	7/19	Rebecca Iselin	Finalized parameters, tweaked calculations slightly to make the triangle more clear. Revamped some of the main menu displays, cleaned up the main file. And implemented the curve.
	Snowflake Objectives: Implement STAGE parameter	No snowflake objectives are needed at this point; algorithm was completed on schedule. Additional parameter is being added to allow the program to output a graphic at a particular iteration given by the user.	7/19	Nathaniel Brown	7/19	Nathaniel Brown	Parameter implemented; now the user can set the stage, rather than the stage of iterations being predetermined by the hard code within the snowflake.cpp
	Bonus Objective: File Saving Location	Edit pathing for file save	7/18	Rebecca Iselin	6/18	Rebecca Iselin	Updated the file name directory saving; now it will output our png images into the working directory, NOT within the testing folder within C-make
	Bonus Objective: Modularized Color Parameter	Create a helper function for the ui, to reduce rewritten code	7/19	Rebecca Iselin	7/19	Rebecca Iselin	getColor function has been created for the the parameter requests within the basic UI for our program.

Report and Presentation Contribution Breakdown

REPORT	Tasks:	Notes on Task:	Start Date:	Initiated By:	End Date:	Finished By:	Notes:
	REPORT - Task 1: Introduction Project	Write an introduction to the project as a whole; how we approached it, algorithms used.	7/19	Nathaniel Brown	7/19	Nathaniel Brown	Introduction completed
	REPORT - Task 2: Introduction Topic	Write an introduction to the topic; what are recursive graphics, how are they used, why are they used.	7/19	Nathaniel Brown	7/19	Nathaniel Brown	Introduction completed
	REPORT - Task 3: Instructions Compile	Write detailed instructions to compile and run our program.	7/19	Rebecca Iselin	7/22	Rebecca Iselin	Had to wait for late work to be turned in, finished the compilation instructions for the final section of our overall code.
	REPORT - Task 4: Instructions SFML	Write detailed instructions on SFML installation and setup	7/19	Nathaniel Brown	7/19	Nathaniel Brown & Alexandria	Detailed instructions have been copied over and reformatted from the PDF I created early on in the project explaining how to set up SFML. Nathaniel handled PC, Alexandria handled MAC
	REPORT - Task 4: SFML Info	Write section of body explaining SFML template classes	7/19	Nathaniel Brown	7/19	Nathaniel Brown	Section completed; details various class templates and some of their helper functions and methods.
	REPORT - Task 5: Triangle Breakdown	Write section of body explaining Algorithm in depth	7/19	Alexandria	7/20	Alexandria	Algorithm for the triangle described and visualizations were created
	REPORT - Task 6: Curve Breakdown	Write section of body explaining Curve algorithm in depth	7/19	Aeyva Rebelo	7/20	Aeyva Rebelo	Algorithm is sufficiently explained, detailed the construction of the points based on the quadrants, the general shape of the curve algorithm, and how to go about constructing higher orders using images
	REPORT - Task 7: Snowflake Breakdown	Write section of body explain snowflake algorithm in depth.	7/19	Nathaniel Brown	7/19	Nathaniel Brown	Algorithm is explained, included a section detailing various stages of koch's snowflake as well to emphasize the importance of the iterative recursive calls.
	REPORT - Task 8: Conclusions	Together, write a conclusion section relating the algorithms and importance of recursion. Include possible scenarios in which recursive graphics are used in the real world	7/19	Aeyva	7/22	Aeyva	Conclusion completed. Real world possible applications included
	REPORT - Task 9: Formatting and ToC	Take the google document and reformat within word, for submission.	7/19	Rebecca Iselin	7/20	Rebecca Iselin	Had to wait for late work to be turned in, finished the formatting of the entire document using the shared work in the google doc.
PRESENTATION	**Tasks:**	**Notes on Task:**	**Start Date:**	**Initiated By:**	**End Date:**	**Finished By:**	**Notes:**
	PRESENTATION - Slideshow and Practice	As a group, create presentation slideshow and practice presenting several times before Tuesday's presentation	7/20	All Members	7/20	All Members	Algorithm Breakdowns: Nathaniel - Snowflake, Rebecca - Triangle, Aeyva - Curve. Introduction to topic, and introduction to project (Alexandria). Conclusion was written by Aeyva. SFML Introduction (Nathaniel)

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