University of Waterloo
Faculty of Engineering
Department of Electrical & Computer Engineering

Creation of an Efficient Algorithm for Fractal Traversal

Prepared By

Mark Seufert
3A, Computer Engineering

December 24th, 2019

Summary

The main purpose of this report is to document the design of an interactive fractal viewer. The report will involve a selection process, where several efficient algorithms for the rendering of the fractal are provided, and a comparison between the algorithms is made to determine the most optimal. This report will also provide the background knowledge that is required to understand the basics on fractals.

The major points documented in this report are in the following order. The report begins with section 1.1, where the equation for a fractal is introduced, and any prerequisite information required for the understanding of the report will be covered. Section 1.2 will introduce the Mandelbrot set, and bring up a rendering technique to render it on the display. Section 1.3 will cover the computational difficulties associated with the rendering of the Mandelbrot set. Section 1.4 will discuss basic optimizations that can be done to optimize the rendering. Section 2.0 will explain the requirements for an efficient rendering algorithm and will cover the requirements for a viable solution. In section 3.1, three solutions that meet all the project requirements will be analysed: basic 2d iteration, boundary tracing, and GPU parallelization. These three solutions will be compared against one another and, in section 3.2, boundary tracing will be chosen as the optimal solution for rendering the Mandelbrot set. In the final section, section 4.0, the design process of implementing the chosen solution will be examined. After these sections comes the Conclusions and Recommendations, which marks the end of the report.

The major conclusion of the report is that the most efficient algorithm for rendering the Mandelbrot set with a high amount of detail is through the user of boundary tracing. The remaining parts of the pipeline that interact with the software will be concluded as well.

The major recommendations in this report is that the optimal algorithm, boundary tracing, should be used as of immediately, and that a powerful computer with a high end CPU should be used. Recommendations on additional functionality of the rendering software will be mentioned. Several future changes for the software solution will also be looked at.

Table of Contents

1.0 Introduction

The natural world contains repeating patterns, no matter where you look. The branches of trees resemble the tree itself, ocean waves have constant ups and downs, and the self-similar structure of a snowflake. It is an interesting question to ask, "how do these patterns occur naturally without human interaction?" These intricate patterns form as it is the most optimal in that situation. The tree, for example, has a self-similar branch pattern, as shown in Figure 1, in order to optimize the amount of sunlight received with the minimum amount of leaf surface area [1]. Although these patterns appear in nature, their origin stems from number theory in mathematics. Self-similar patterns in mathematics are defined as fractals, and this report will be investigating a specific fractal known as the Mandelbrot set, and the evaluation of efficient ways to implement it algorithmically.

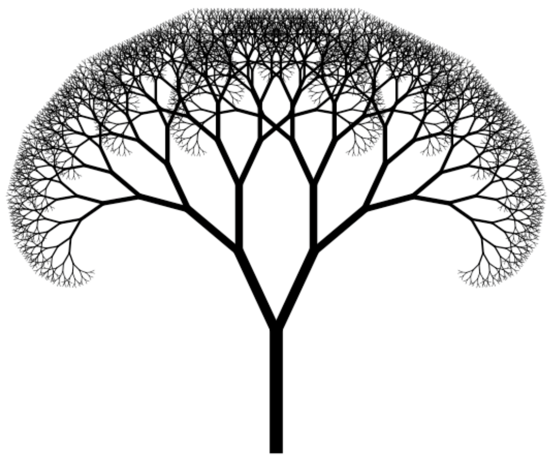


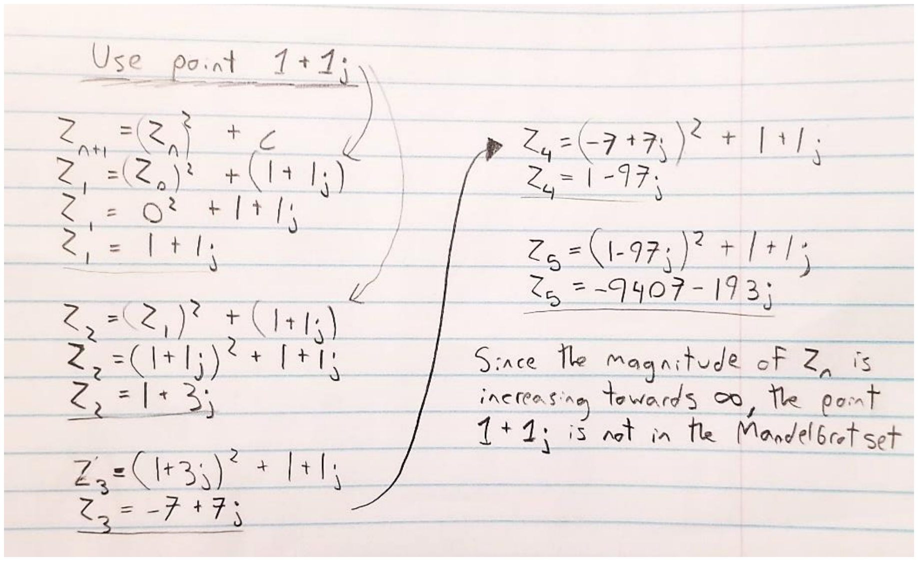
Figure 1: The Self-similar Pattern of a Tree [5]

1.1 Background Information

The Mandelbrot set is described in the complex plane. All points in the complex plane consist of two parts: a real and an imaginary component. Although similar to the cartesian plane with x and y coordinates, the complex plane relates the real axis to the imaginary axis through the equation $i * i = -1$. In other words, squaring an imaginary number produces a real number, which will prove important later on. With the complex plane established, the definition for the Mandelbrot set can be introduced.

The Mandelbrot set was first discovered in the 1980s by two French mathematicians while they were studying complex quadratic polynomials. They were looking at a specific case of the formula $y = x^2 + c$, where the output y is put back into the equation as the x value. Explicitly, $\mathbf{Z_{n+1} = Z_n^2 + c}$ where Z_{n+1}, Z_n, and c are complex numbers. If we were to let Z_n initially be 0 (Z_0 =

0), and let c be a complex constant, there are two possibilities as n is iterated towards infinity. First, the magnitude of the complex number Z_n would become larger with each iteration, diverging towards infinity as n goes to infinity. The second possibility would be that the magnitude of Z_n becomes smaller, converging to 0 as n goes to infinity. Figure 2 illustrates the complex point 1 + j being put into this equation as the complex constant c.



Use point $1 + 1j$

$Z_{n+1} = (Z_n)^2 + c$

$Z_1 = (Z_0)^2 + (1 + 1j)$

$Z_1 = 0^2 + 1 + 1j$

$Z_1 = 1 + 1j$

$Z_2 = (Z_1)^2 + (1 + 1j)$

$Z_2 = (1 + 1j)^2 + 1 + 1j$

$Z_2 = 1 + 3j$

$Z_3 = (1 + 3j)^2 + 1 + 1j$

$Z_3 = -7 + 7j$

$Z_4 = (-7 + 7j)^2 + 1 + 1j$

$Z_4 = 1 - 97j$

$Z_5 = (1 - 97j)^2 + 1 + 1j$

$Z_5 = -9407 - 193j$

Since the magnitude of Z_n is increasing towards ∞, the point $1 + 1j$ is not in the Mandelbrot set

Figure 2: Mandelbrot Equation Being Applied on a Point [5]

This equation, which is referred to as the Mandelbrot equation, can be applied to all complex numbers. The Mandelbrot set is defined as the set of complex numbers that converge to 0 when put into the equation.

1.2 Rendering the Mandelbrot Set

To visualize the Mandelbrot set, the complex plane can be colored depending on the result at each point. If the point's magnitude approaches 0, then it is inside the Mandelbrot set and is colored black. If the magnitude approaches infinity, then the point is colored blue. Although these two coloring criteria would successfully render the Mandelbrot Set, it would only contain two colors and would not render intricate details. A more advanced coloring algorithm involves modifying the color of blue by an amount proportional on the number of iterations it takes the point to reach a sufficiently large magnitude. The greater number of iterations it takes for a point to reach a threshold magnitude, the closer that point is to being inside the Mandelbrot Set. For example, a complex point that takes 5 iterations to reach a magnitude of 1000 is further from being inside the Mandelbrot set than a point that takes 50 iterations to reach a magnitude of 1000. These two points should not be colored the same. A third coloring criterion can now be applied, where the color of the point becomes shifted towards white as the number of iterations increases. Using these three basic coloring properties, Figure 3 is generated.

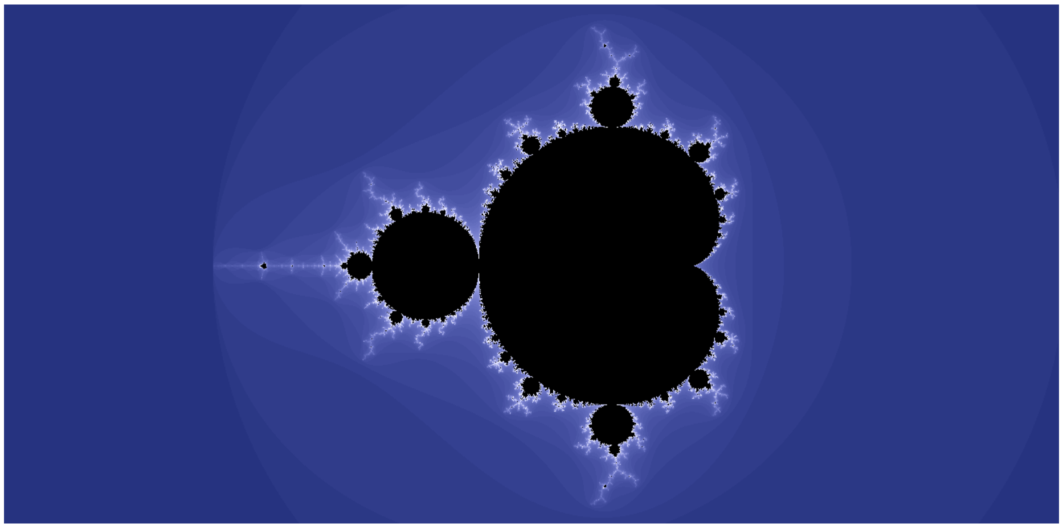


Figure 3: Mandelbrot Set with a Coloring Algorithm [5]

The range of complex values of this image is from (–4, -1.5i) to (2, 1.5i). Any subset of the Mandelbrot set can be rendered by changing the range of complex points that are calculated and stretching the results to fill the screen.

1.3 Computational Challenges

Rendering the Mandelbrot set has two main challenges that must be considered. First, depending on the screen resolution, there are a large number of points that need to be iterated over. If the screen has a resolution of 1920 by 1080, there are a total of 2 073 600 points, each of which needs to have the Mandelbrot equation applied to it in order to determine the color. This requires a significant amount of computational power to achieve and will cause the render to take a large amount of time to complete. Additionally, the number of iterations applied to each point changes depending on the zoom into the fractal. As the intricate parts of the fractal are zoomed into, it will require more iterations to differentiate the points inside the Mandelbrot set and those that are not. Experimentally, it is determined that there is a square root relationship between zoom and number of iterations required. For example, the Mandelbrot set at a x4 zoom will require double the number of iterations than at a x1 zoom.

The second challenge is maintaining an accurate rendering at high zooms. As mentioned before, complex numbers have two parts: real and imaginary. A computer program would typically use the *double* variable type to represent each part. Variables of double precision use 64 bits to represent a number [2], which is more than enough for most applications. At a high zoom of the Mandelbrot set however, the complex points will be extremely close together, and 64 bits will not accurately describe them.

In order to render the Mandelbrot set at runtime at any zoom, several basic optimizations can be made to the process to simplify the calculations.

1.4 Basic Optimizations

There are several characteristics of the Mandelbrot set that can be used to optimize the process.

When applying the Mandelbrot equation to a complex point, the goal is to determine whether or not the magnitude will diverge to infinity. One of the properties of the Mandelbrot set is that if the magnitude of the complex point Z_n exceeds 2 at any point, then that point is guaranteed to approach infinity. Therefore, the iterations for a point can stop once the magnitude exceeds 2. The calculation to compute the magnitude of a complex number is $mag = \sqrt{r^2 + i^2}$ where r and i are the real and imaginary components respectively. The $\sqrt{\ }$ function is computationally difficult, meaning it should be avoided if possible. To remove the square root, both sides of the original equation can be squared, resulting in $mag^2 = r^2 + i^2$. Therefore, instead of checking if the magnitude of the point exceeds 2, it is more efficient to check if the square magnitude of the point exceeds $4 = 2^2$.

The Mandelbrot set has several basic regions that are inside the set. Figure 4 highlights these regions, each of which has an equation associated with it. Before a point is put into the Mandelbrot equation, it should be checked to see if it is inside one of the regions. If a point is inside any of the major components, automatically put it into the set without applying the equation to it. This optimization will mainly affect the runtime at low zooms when a large percent of the screen is a set of these basic shapes. At higher zooms, less percent of the screen will be covered by the regions, and therefore this optimization's benefits decrease towards zero.

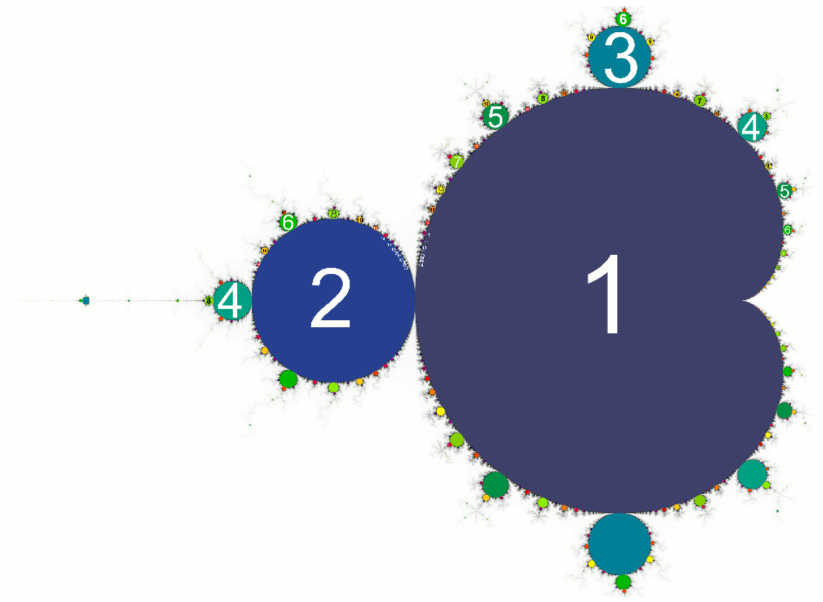


Figure 4: The Basic Components of the Mandelbrot Set [3]

One of the most intuitive properties of the Mandelbrot set is the symmetry about the X-axis. Using this property allows half of the points' calculations to be skipped by mirroring over the results from the other half. This approach initially doubles the runtime, as half the points are being calculated. However, this optimization loses its benefits if the view of the set is strictly in the positive or negative imaginary axis.

Each of these techniques results in an overall speedup of the algorithm. However, there are additional, more advanced modifications that can be made that would have greater effects towards the runtime. The functional requirements for the modified algorithms will be introduced in the next section.

2.0 Functional Requirements

There are several functional requirements for the algorithm, each with different weights of importance. The requirements will be listed below in no order of importance. It is assumed that the algorithm will be ran on a Windows computer with a designated GPU.

- The algorithm must be executed fast. The fractal rendering is going to be viewed in runtime, meaning that it is updated numerous times per second. Therefore, it is required to have the results generated in time for the frame to be displayed onto the screen. Failing to reach this requirement will cause the user to experience lagging and an overall decrease in the user responsiveness.

- The algorithm must be accurate. One of the interesting properties of fractals are the intricate levels of detail at any zoom. It is important to maintain a high level of accuracy at any zoom such that these patterns are not lost.

- The algorithm must use minimal memory. Failing to achieve this can cause the program to slow down or crash, as well as other unintended consequences. It is typical to see an increase in speed as a side effect of minimizing memory.

- The algorithm must be error-resistant. Any problems the program encounters should be logged to the user in a convenient format, such as a console output or log file. The program should not crash under any circumstances.

All together, there are four requirements: *Fast Execution, Accurate Calculations, Low Memory Consumption, and Error Resistance / Error Logging*

3.0 Solution Choices and Comparison

In section 3.1, three possible algorithm solutions will be introduced and briefly explained. Then, in section 3.2, each solution will be quantitatively compared against each other for each functional requirement. Table 1 contains a summary of the results from this comparison. Lastly, section 3.3 will choose the most optimal solution, and explain the rational behind the choice.

3.1 Solution Choices

1) Basic 2d Iteration Algorithm: This method does not require any modifications to the original algorithm, and simply iterates across the screen, applying the Mandelbrot equation at each point to render the fractal. As discussed in section 1.3, using double precision to represent the complex numbers would not maintain accuracy at large zooms. Therefore, a custom variable type must be made to allow for the number of bits per number to change. This technique will use this new datatype to store the real and imaginary components of the complex numbers instead of double to allow for high accuracy at large zooms.
2) Boundary Tracing Algorithm: One of the properties of the Mandelbrot set allows for a new type of algorithm to exist. The property defines that if a closed path is drawn on the complex plane, and each point on that path has the same outcome from the Mandelbrot equation, then all the points enclosed inside that path will also have the same outcome. This is useful because the algorithm would have a runtime dependant on the length of the paths instead of the area of the paths. This algorithm theoretically has a runtime of O(n), where n is the width of the screen. It is very rare to find a loop that has exactly the same

results along all of its points, so the algorithm would normally struggle to achieve any runtime benefit. To allow for greater speed increases, the algorithm can be modified to allow for slight differences in the results along the edges, as long as they fall within a certain delta value. The algorithm initially starts with a large rectangular path encompassing the entire screen. Each edge of the rectangle is iterated upon, checking to see if all the points of that edge return the same Mandelbrot value. If this holds true for all four sides, then the interior of the rectangular loop is filled with the potential of the four sides without needing to calculate them. If either of the four sides change potential along the way, then the rectangle is broken into four smaller rectangular paths and the algorithm recursively repeats itself. Figure 5 demonstrates the structure of the rectangular paths after the algorithm completes.

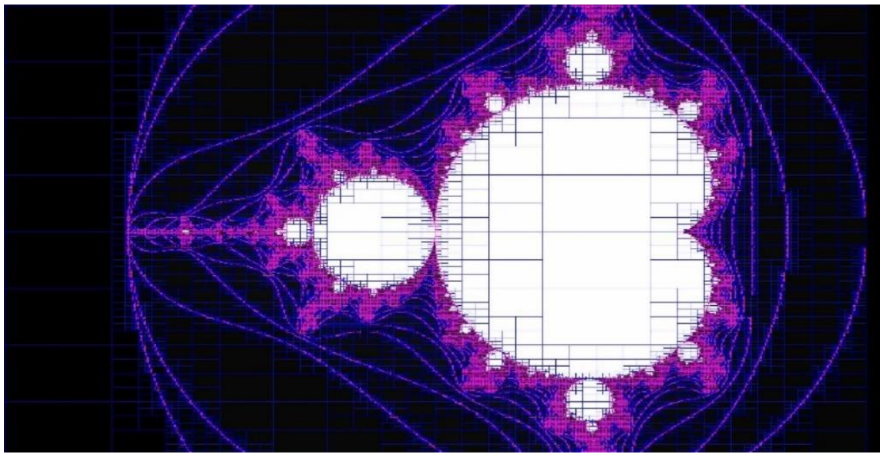


Figure 5: Mandelbrot Rendering with Border Tracing Rectangles Visible [5]

This algorithm would use the custom variable type mentioned previously.

3) GPU Parallelization Algorithm: Repeating the same Mandelbrot equation for thousands of points is very repetitive. Algorithms that run on the CPU, the central processing unit, are good at executing complex tasks, but are not optimized to repeat simple calculations a large number of times. To avoid using the CPU, the algorithm can be written in a low-level programming language and put onto the GPU, the graphics processing unit. The GPU is very well optimized for doing simple calculations because of specified hardware, and can do thousands of operations in parallel. It does not support custom variable type however and would suffer an accuracy loss at high zooms.

3.2 Solution Comparison

As explained in Section 2.0, the functional requirements for the solution are: Fast Execution, Accurate Calculations, Low Memory Consumption, and Error Resistance / Error Logging. The next three paragraphs will analyze each solution choice and compare it against the other choices.

Solution choice 1 (Basic 2d Iteration) has one main advantage. Since each point is independently being calculated, it is guaranteed that the desired level of accuracy is achieved. As well, if an error occurs at any point inside the algorithm, then it can easily be caught and handled. Unfortunately, since all of the points are being iterated over by the CPU, this approach is very slow. Experimentally, it took 4.34 seconds to generate the image in Figure 6. Additionally, this solution requires a large amount of memory to store the results from the calculations into a 2d array that would be put onto the screen.

Solution choice 2 (Boundary Tracing) is more well optimized than the first choice. Instead of iterating over all the complex points, this solution iterates over the edges of rectangles inside the complex plane. Because of the ability to skip large chunks of iterations, this approach is moderately faster than choice 1. Experimentally, Figure 6 was generated in 0.439 seconds. Although this provides a speedup of approximately a factor of 10, there is an accuracy disadvantage. Since this algorithm allows for slight differences in the edge values, as mentioned in section 3.1, there will be minor discrepancies in the final rendering. Therefore, the fractal will not be as accurate as choice 1. The memory usage would be decreased as well, since not as many complex values would have to be stored. Error logging will remain the same as the first choice, since the CPU can implement error handling.

Solution choice 3 (GPU Parallelization) is significantly different. Using the GPU to do the simple calculations in parallel allows for significant speed increases compared to using the CPU. Experimentally, the image in figure 6 was generated in 0.081 seconds which is approximately 50 times faster than choice 1. Although this solution seems optimal, there are accuracy drawbacks. GPU programming only allows for basic variable types, such as ints, floats, and doubles. It does not allow for custom variable types to be implemented, which are required for rendering highly zoomed parts of the Mandelbrot set. Therefore, the accuracy of the fractal degrades proportionally to the zoom level. The memory usage of using the GPU is the same as choice 1. In order for the GPU to execute the algorithm, the CPU must pass it a large 2d array containing all the points to apply the Mandelbrot equation to. Creating this 2d array would have similar memory requirements as the array containing the results from choice 1. The error logging abilities of this solution are not advanced. The GPU does not have substantial error logging and will likely crash if anything goes wrong in the algorithm.

Table 1: Solution Comparison

	Fractal Accuracy	Execution Speed (s)	Memory Usage	Error Logging
1) Basic 2d Iteration	High	4.34	Moderate	Yes
2) Boundary Tracing	Medium	0.535	Low	Yes
3) GPU Parallelization	Decreases with zoom	0.081	Moderate	No

The results from the table above show the comparison between the three algorithms choices. Out of the four functional requirements, the fractal accuracy is the most important. It does not matter if the fractal is rendered fast if the result isn't correct. Since the GPU becomes less accurate as the Mandelbrot set is zoomed into, it can be disregarded as a viable solution. Although the boundary tracing algorithm is not as accurate as the basic 2d iteration, its accuracy does not degrade with zoom. The second most important functional requirement is the execution speed. Boundary tracing outperforms basic 2d iteration by providing a speed increase by a factor of 10. The memory requirements for boundary tracing are also minimal, and the error logging capabilities are the same as that of the basic 2d iteration. Thus, the selected solution is the boundary tracing algorithm. The following section will describe the design that went into the implementation.

4.0 Solution Analysis

With the algorithm chosen, several additional steps must be done in order to provide the remaining functionality of the Mandelbrot renderer. First, a programming language must be chosen as a framework to implement the algorithm. Then, the ability to create a window and display pixels must be found. Lastly, the ability for user inputs to interact with the fractal is required. The next several paragraphs will document this process.

The first step in developing an application is choosing the language to write it in. Since runtime is an important factor, a language that has the ability to do its own memory management is required. Languages that use garbage collection for memory management suffer have lower complexity, with a trade off for speed decreases. Therefore, the language must not have garbage collection. C++ fits this criterion [4], and contains significant documentation and libraries that will be beneficial for later steps.

The next step in the process is to determine the structure of the application. Figure 5 shows a UML diagram with the high level class structure of the application.

Next is the requirement to create windows and display pixels. There are many libraries for C++ that allow for this functionality, but most of them are not efficient. What is required is a library that uses OpenGL as a backend to render graphics. OpenGL, which stands for Open Graphics Library, is a graphics standard that uses the GPU to render applications. Because it uses the GPU, it is efficient compared to CPU graphics libraries. Using OpenGL directly is difficult due to the high level of complexity, so the use of a helper library is desirable. GLFW, which stands for Graphics Library FrameWork, provides the basic functionality of OpenGL through the use of helper functions that make implementation simpler. Table 2 displays several functions that are provided.

Table 2: GLFW Functions

GLFW basic functions	
glfwCreateWindow(width, height, name)	Creates a window on the screen with a given width, height, and name for the window
glColor3f(r, g, b)	Sets the palette color to a specific RGB value
glVertex2f(x, y)	Colors a pixel at an x,y coordinate with the color selected by glColor3f
glfwGetCursorPos(&x, &y)	Sets the x and y variables to the current x and y of the cursor
glfwSetMouseButtonCallback(fctn)	Call the fctn() function whenever the mouse button is clicked
glfwInit()	Must be called at the beginning of the application
glfwTerminate()	Must be called at the end of the application

The ability for user input into the application is required. It would be ideal if the user could select a rectangular region of the screen by holding down the mouse button, and that region would be rendered on the release of the mouse. As seen in Table 2, there is a function called glfwGetCursorPos to return the current coordinates of the mouse, and an event handler called glfwSetMouseButtonCallback that handles mouse inputs. To achieve the zooming capabilities, the program would store the X and Y positions of the mouse when the mouse button is pressed down. A rectangle would be continuously drawn from the starting x,y position to the current x,y of the cursor to illustrate the region that would be zoomed into if the user released the mouse. When the mouse button is released, the fractal would be re-rendered with the new range of values.

With this functionality, the boundary tracing algorithm can be properly implemented to complete the Mandelbrot rendering application. Figure 6 shows the class structure of the application.

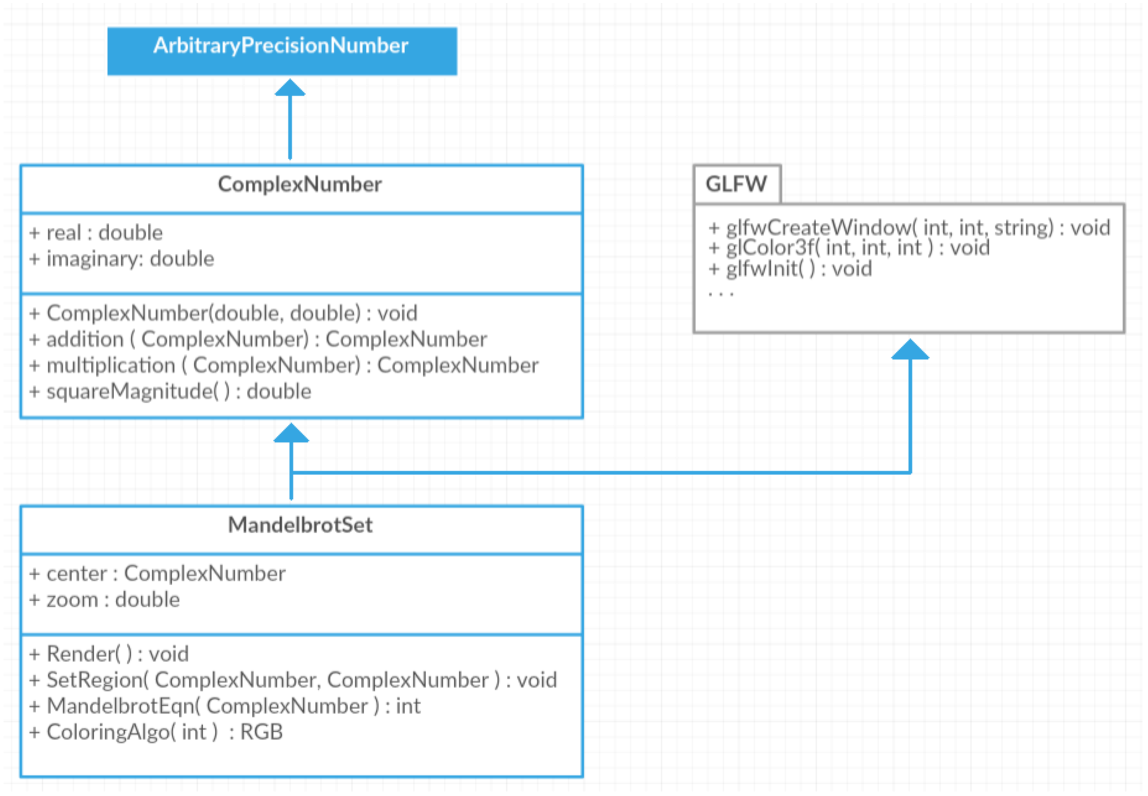

ArbitraryPrecisionNumber
ComplexNumber
+ real : double
+ imaginary: double
+ ComplexNumber(double, double) : void
+ addition (ComplexNumber) : ComplexNumber
+ multiplication (ComplexNumber) : ComplexNumber
+ squareMagnitude( ) : double
GLFW
+ glfwCreateWindow(int, int, string) : void
+ glColor3f(int, int, int) : void
+ glfwInit( ) : void
. . .
MandelbrotSet
+ center : ComplexNumber
+ zoom : double
+ Render( ) : void
+ SetRegion(ComplexNumber, ComplexNumber) : void
+ MandelbrotEqn(ComplexNumber) : int
+ ColoringAlgo(int) : RGB

Figure 6: UML Class Diagram of the Mandelbrot Render Application [5]

5.0 Conclusions

Through the report body, it was determined that the optimal algorithm for rendering the Mandelbrot set is boundary tracing (section 3.2). This algorithm finds loops on the complex plane and applies the Mandelbrot equation to the points that lie on the loop as a way to skip the points that the loop encompasses. It would be implemented in c++ with an OpenGL helper library called GLFW. Using these components, a real-time Mandelbrot set rendering application can be made with the ability for user inputs.

6.0 Recommendations

Based on the analysis and conclusion in this report, it is recommended that the boundary tracing algorithm be implemented. Although this algorithm is an optimized version of the original, it still requires the use of substantial compute power to achieve real time results, especially as the user attempts to render highly zoomed images. With that in mind, it is recommended that the application be used on a computer with a powerful CPU, such as the intel core i7. Although this report covers the complete development of an application, there are additional features that would be desirable for future development. Several of these are:

- Ability to print the screen and save as image
- Ability to change coloring algorithm
- Ability to render other fractals

References

[1] A. Bressan and Q. Sun, "On the Optimal Shape of Tree Roots and Branches", arXiv.org, 2018. [Online]. Available: https://arxiv.org/abs/1803.01042. [Accessed: 13- Jan- 2020].

[2] W. Kahan, "IEEE Standard 754 for Binary Floating-Point Arithmetic", People.eecs.berkeley.edu, 1997. [Online]. Available: https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF. [Accessed: 13- Jan- 2020].

[3] M. Tangora, Computers in geometry and topology. New York u.a.: Dekker, 1989, p. 217.

[4] D. Bulka and D. Mayhew, Efficient C++. Boston: Addison-Wesley, 2003, p. 15.

[5] M. Seufert, 2020.