CS51 Final Project

Mandelbrot Set Viewer

Shane Kissinger

May 2, 2023

1 Introduction

Fractals have often captured the imaginations of many due to their infinite, complicated structure and their self-similarity. The first popularized fractal was the Mandelbrot set, which was first computed by Benoit Mandelbrot in 1980 using the high-powered computers at IBM.

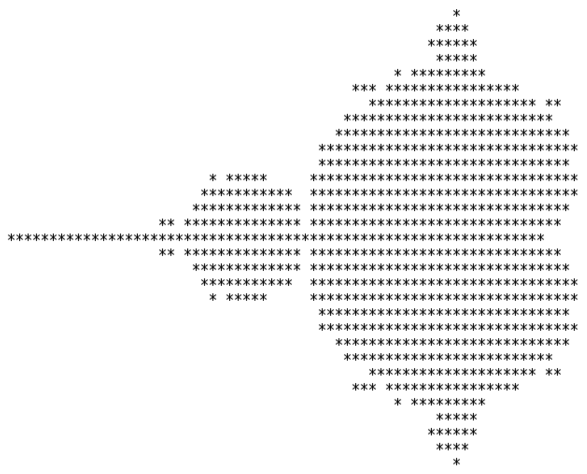


Figure 1: The Mandelbrot set computed by Mandelbrot on the IBM computers.

My objective for this project is to create software to illustrate fractals for arbitrary complex polynomials, as opposed to the singular function $f(z) = z^2 + c$ used for the Mandelbrot set. OCaml is a great candidate to write this program in because of implementations of functors making code organization much easier for variable complex polynomials.

1.1 Mathematical background

Let $f(z) = z^2 + c$, where $z, c \in \mathbb{C}$. Note that a critical point of this function is located at $z = 0$. We look at the dynamics of the system by letting c change continuously and look at the actions of iterations of f, evaluated at $z = 0$.

A point $c_0 \in C$ is said to be contained within the Mandelbrot set if $|f \circ \cdots \circ f(0)| = |f^n(0)| < \infty$ for $f(z) = z^2 + c_0$. In plain english, we take a complex number $c_0 \in C$, plug it into f, and then see if the point goes off to infinity or not as we repeatedly iterate.

2 Implementation

To implement this, I had to write several different helper files to handle the computation, graphing, and user interface. The structure of the code is below.

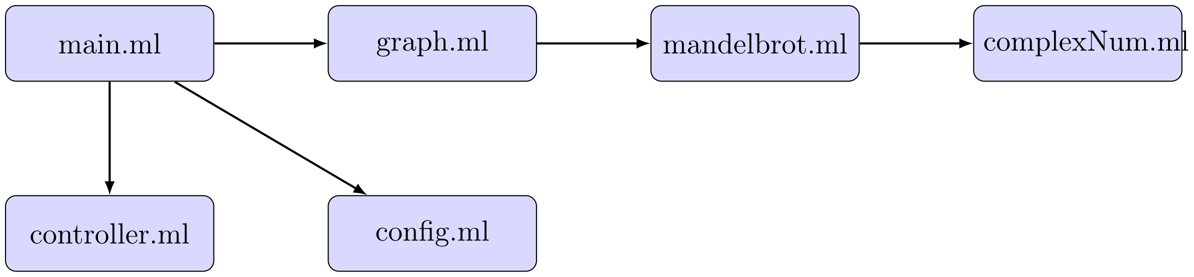


Figure 2: Code organization

The code structure for this program is designed to create a cohesive and modular system for generating and interacting with fractal images. The program is centered around the file `main.ml`, which serves as the main entry point and calls all necessary functions to display the graphical user interface and handle user interactions. This layout ensures that the different components of the program are well-organized and maintainable.

The `config.ml` file contains all the settings for the program, such as the type of fractal to generate, color options, and resolution. This configuration file allows users to easily customize the output and behavior of the program. The file `graph.ml` is responsible for rendering the fractals on the screen using the Graphics module. It utilizes functions from `mandelbrot.ml` to create the necessary graphics.

The file `mandelbrot.ml` contains a functor that instantiates a Mandelbrot module using settings defined in `config.ml`. This module encapsulates all the required functions to determine if a number is in the Mandelbrot set for a specific function. With the functor, it is possible to pass any arbitrary complex polynomial using the `CNum` module found in complexNum.ml. In the implementation, the function `in_mandelbrot` takes a complex number c and repeatedly applies the function fed to the functor. The function that outputs a tuple of type `int*bool` with the number of iterations it took to escape the threshold, i.e. the number $n \in \mathbb{N}$ it took such that $|f^n(0)| > t$ where t is the threshold, along with a boolean value saying if it is contained within the Mandelbrot set. Note that since a point may diverge after the specified number of iterations. If a point diverges, we know that it is *not* in the Mandelbrot set and not that it is. This is why increasing the number of iterations in `config.ml` causes the definition of the fractal to become clearer.

The `complexNum.ml` file is focused on creating a complex number data type as a record and defining various functions for manipulating complex numbers, such as absolute value, conjugate, addition, and multiplication. This module provides the necessary tools for working with complex numbers throughout the program.

User interactions with the GUI are managed by the `controller.ml` file, which is called by `main.ml` after the fractal is written to the screen. This module handles all user inputs, ensuring a smooth and responsive experience when using the program. The file displays the position of the cursor, in complex coordinates, if the screen is currently loading, and allows the user to press `e` or `q` to enhance the image or quit, respectively.

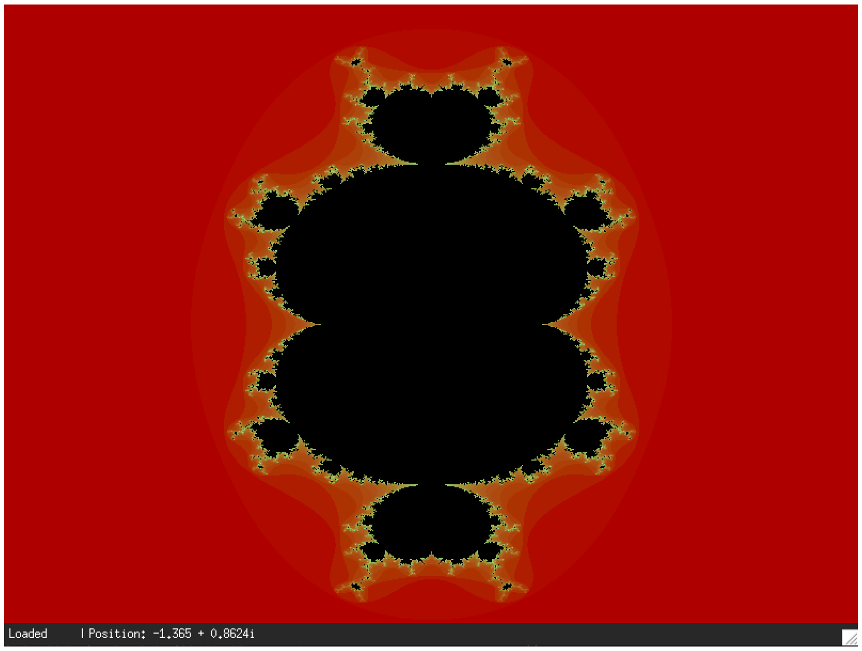


Figure 3: Fractal of $f(z) = z^3 + c$

Overall, the modular design of the code ensures that each component is responsible for a specific task, making it easier to understand, maintain, and expand the program's functionality.

3 Challenges & obstacles encountered

There were several unexpected challenges that I encountered. However, these served as a great learning experience.

3.1 Color mapping

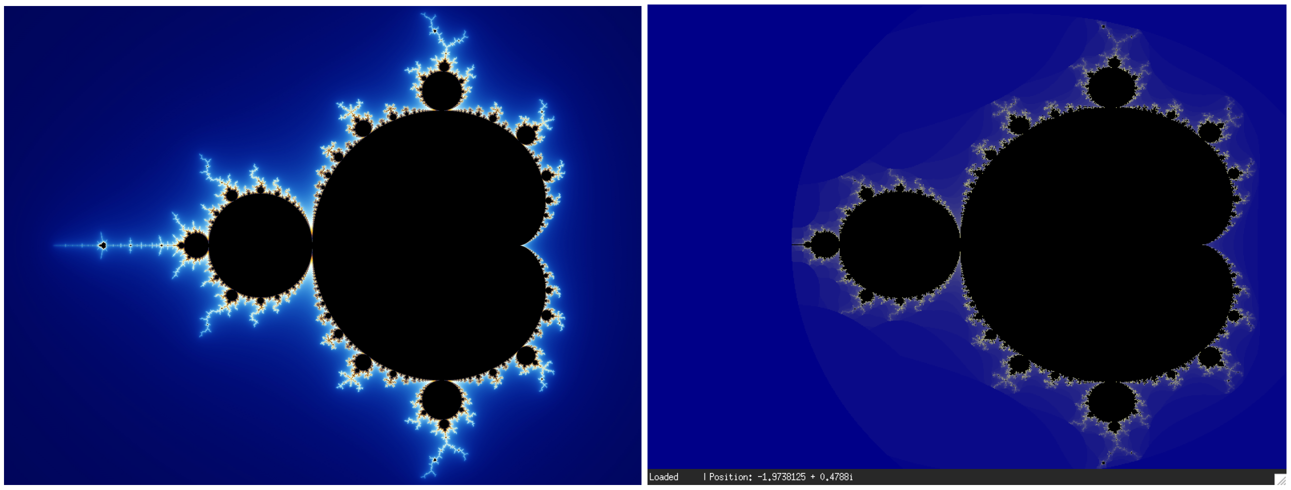
I struggled initially with implementing a way to map integers to a range of colors that was sensible and aesthetically pleasing. Though this is a small feature, this was difficult to find a function that worked well. I ended up using the function

$$\psi(g) = 255(1 - e^{g\frac{i_{\text{count}}}{i_{\max}}})$$

where g is a growth rate I adjusted as needed and i_{count} is the number of iterations it took to escape the threshold and $i_{\max}$ is the maximum number of iterations allowed. Using this approach, we can get similar pictures to ones we see on the internet.

3.2 Implementing the draggable rectangle to zoom

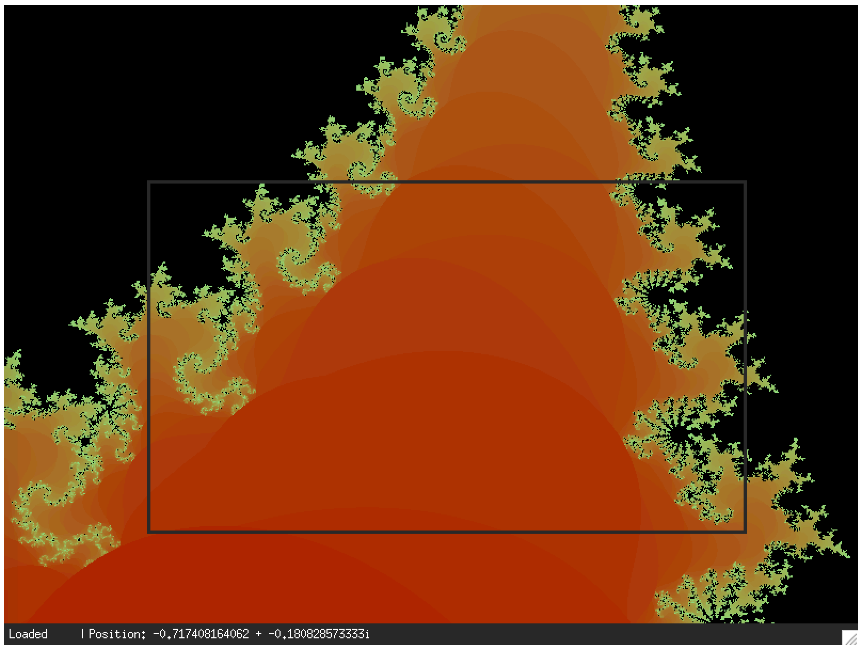
I felt that it was necessary, both for the accuracy of the zoom and the experience of the user, for the user to be able to drag a rectangle in real-time to denote an area to zoom into. The Graphics



```
Loaded      | Position: -1.9738125 + 0.4788i
```

Figure 4: Left: Image from the Mandelbrot set Wikipedia, Right: Image from my program.

module was very basic for this need and I needed to overcome a few different obstacles because of this. First, there was no way to animate the box on top of the fractal as a background. Instead, to animate the rectangle changing size with the user's cursor, I needed to first save the background as an image and then clear the screen, display the image, draw the new rectangle, and repeat whenever the user moved their cursor.



```
Loaded      | Position: -0.717408164062 + -0.180828573333i
```

Figure 5: An image showing the user dragging a rectangle to denote an area to zoom into.

Second, the box that the user denotes on the screen returns the pixels, but not the complex coordinates. This was a difficult challenge because the complex coordinates of each pixel changes on each zoom, as the viewing pane is now different and corresponds to a different area on the complex plane. So, the rectangle coordinates had to be converted into complex coordinates using the old complex coordinate values, and then these are used to set the boundaries for converting follow-on pixels to complex coordinates. This was a fun exercise in arithmetic.

4 Conclusion

Overall, I believe I was successful in implementing a bare-bones program to view and interact with fractals. It was exciting to use OCaml's tools of abstraction to make the program a lot more modular and flexible.

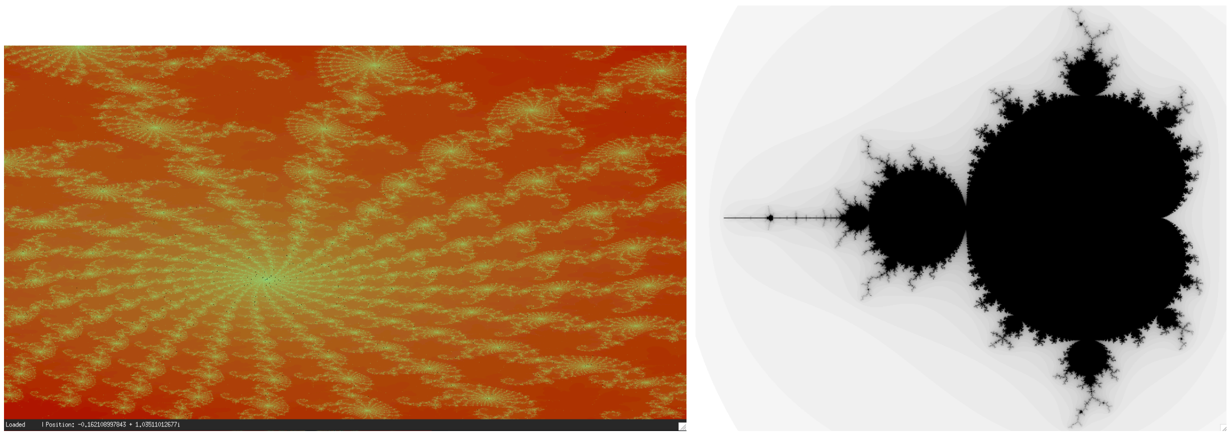


Figure 6: Some more images I generated because I thought they were cool...

4.1 Future work

There are still many functionalities that the program does not have. Further improvements could be made in the speed of the fractal generation. The computation speed was much better interacting with color matrices directly and then plotting, instead of plotting each point individually and showing them all at the end. However, for some reason, the implementation using color matrices introduced a significant amount of error in the calculations. Additionally, it is a natural thing from here to implement a way to generate Julia sets of functions, which are a similar fractal generated from the dynamics of complex functions.

One could also drastically improve the computational speed by introducing a sort of *memoization*. If a point reaches a point that has been reached before, and we know it diverges, then we don't need to check the current point anymore. This would remove a lot of repeat calculations.