How to Draw Infinity

Alexander Martin

1 Introduction

How do you measure the length of the British coastline? This might seem like a trivial question to answer; take some long rulers, let's say 100km long, and see how many you can place around the coastline. Doing this, you'll be able to put down 28 rulers, so your final answer is 2,800km. But what if we want to measure the length using 50km rulers? The shorter rulers allow us to include more inlets and small details of the coastline, so the length of the coastline suddenly increases to 3,400km. You can actually keep zooming into the coastline infinitely only to find increasingly smaller details[1], so using shorter rulers simply makes the measured length of the coastline increase exponentially towards infinity. This counter-intuitive observation is called the Coastline Paradox and happens because the coastline of Britain is a fractal.[2]

Image: The Coastline Paradox.[3]

[1] Arguably, at the atomic/Planck scale, measurement is not meaningful in the same way as at the macroscopic scale, so coastlines are only infinitely detailed from a purely mathematical perspective.
[2] Mandelbrot, B. (1967a)
[3] *Britain Fractal Coastline* (2006)

But what is a fractal? The most basic definition of a fractal is that a fractal is a never-ending pattern. A better definition is that a fractal is any kind of pattern or shape where zooming in reveals increasingly smaller, repeating details. When zooming into the British coastline, similar patterns of jagged edges and inlets appear at different scales, and a mathematical representation of a coastline would have these patterns repeat infinitely as you zoom in. Fractals are found all over the place in nature, from fern leaves to snowflakes.

1.1 Drawing Fractals with Code

It's important to understand that all depictions of fractals are fundamentally just approximations of fractals, because creating a truly infinitely detailed drawing would take an infinite amount of time. So, when we draw recursive fractals, we always pick a certain number of iterations to go before stopping. While it is technically possible to draw these fractals by hand, these problems are especially suited for computation, because computers are very good at running recursive algorithms. In simpler terms, drawing these fractals by hand would take way too long, and computers can do a lot of the hard work for us.

For this essay, I've written a C++ program that draws fractals such as the Mandelbrot Set using OpenGL fragment shaders. By leveraging parallel processing on the GPU for its calculations, it achieves performance optimal enough for drawing fractals in real time.[4]

[4] Applications are usually considered "real time" when at least 30 frames can be processed in a single second. This means that we'll have to compute and draw the Mandelbrot Set in less than 1/30th of a second.

2 The Mandelbrot Set

The Mandelbrot Set is a famous fractal with a surprisingly simple definition. It is the set of all complex numbers, c, where the sequence:

$$z_{n+1} = z_n^2 + c$$

$$z_0 = 0$$

does not diverge to infinity. Drawing the set of these numbers on the complex plane leads to beautiful fractal patterns.

We have computers to thank for the discovery of the Mandelbrot Set. Benoit Mandelbrot first formally discovered the fractal while working at IBM in 1980.[5] The fractal is way too complex for plotting by hand, so the first images of the Mandelbrot Set were created using computation.

2.1 Drawing the Mandelbrot Set with Code

A fragment shader is a program that calculates the colour of a single pixel on your screen. Fragment shader algorithms run in parallel (i.e. simultaneously) for every pixel on screen, utilising the multiprocessing capabilities of a GPU. Fragment shaders provide a very elegant way to compute and render the Mandelbrot set: each pixel represents a number on the complex plane, like so:

$$PixelCoordinate(x, y) \rightarrow x + iy$$

In GLSL (a language used to write shaders), the code for this looks like the following:

```
// Map fragment to complex plane

vec2 c = fragCoord / u_zoom + u_center;
```

“fragCoord” is the coordinate of the pixel we are calculating the colour of. It is measured in Normalised Device Coordinates, which range from -1 to 1. “u_zoom” and “u_center” are parameters related to the app’s pan and zoom functionality.

2.2 How do we know if the sequence is diverging?

This is the most important part of the code: checking if our pixel is within the Mandelbrot Set. The code for this is the following:

[5] Nakos, G. (2024), p322

```
vec2 z = vec2(0.0);

int iter = 0;



for (int i = 0; i < u_maxIter; i++) {

    if (dot(z, z) > 4.0) break;                    // |z|^2 > 4.0 is divergent (outside the set)

    z = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + c;  // z = z^2 + c

    iter++;

}
```

We can't compute the sequence to infinity as that would require an infinite amount of computation. Instead, I've set the limit for the maximum iterations, "u_maxIter", at 256 in most examples. This means some values are included in the set that shouldn't be because they take more than 256 iterations before the program can detect whether they diverge. This is a problem that cannot truly be fixed due to the infinite nature of fractals and the limited nature of processing power, however increasing u_maxIter makes it less noticeable.

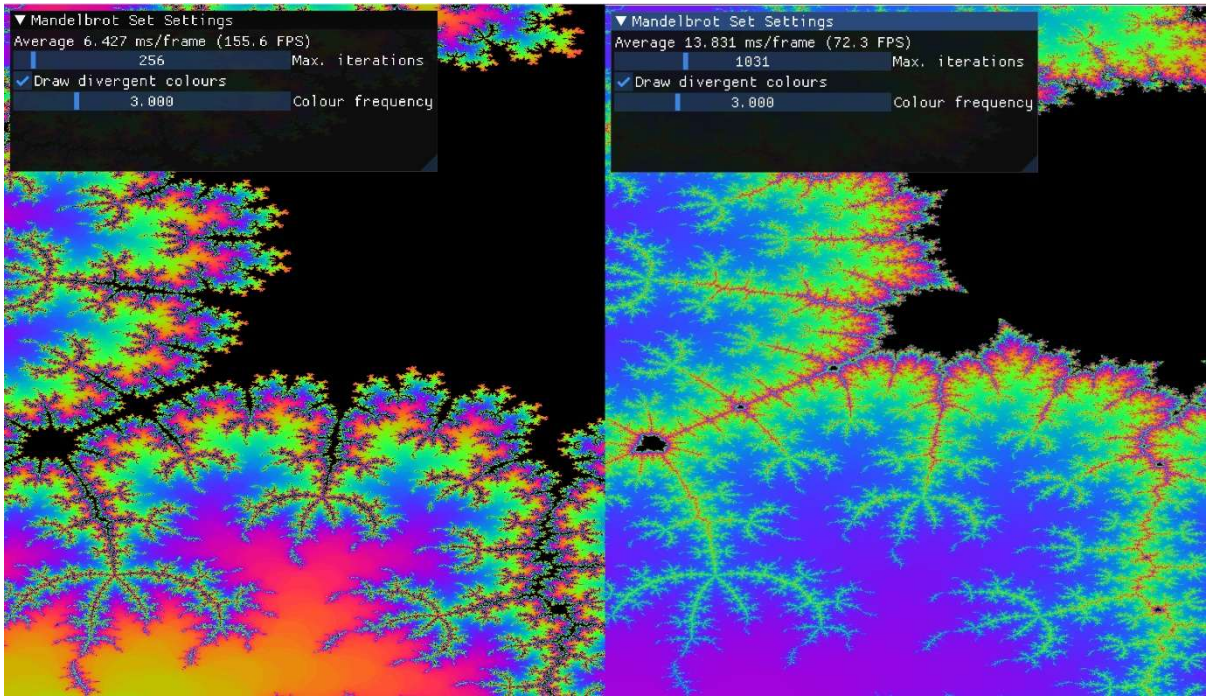

Image: The same render created with different values for maximum iterations. Notice how more iterations increases accuracy but decreases frame rate.

In each iteration, we calculate the next complex number in the sequence using the formula from before:

$$z_{n+1} = z_n^2 + c$$

We can show that if the modulus of any term in the sequence is greater than 2, the sequence is guaranteed to diverge to infinity. This is because, if $|z_n| > 2$ and $|c| \leq 2$, z^2 will always grow the sequence faster than c can pull it back.

When calculating $|z_n|$, we need to keep in mind that GLSL doesn't understand what complex numbers are. We can calculate $|z_n|^2$ using the dot product, since:

$$|a + bi|^2 \equiv a^2 + b^2$$

$$\begin{bmatrix} a \\ b \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} \equiv a^2 + b^2$$

We can then compare $|z_n|^2$ and 4 to see if $|z_n| \geq 2$. This is done instead of getting the square root of $|z_n|^2$ because computers are quite slow at calculating square roots. All of this is done in a single line of code:

```
if (dot(z, z) > 4.0) break;  // |z|^2 > 4.0 is divergent (outside the set)
```

If $|z_n| \geq 2$, we stop calculating the sequence because we already know that c is not in the Mandelbrot Set. Otherwise, we compute the next number in the sequence like so:

```
z = vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y) + c;  // z = z^2 + c
```

Again, we must handle the complex numbers carefully: GLSL won't calculate z^2 as it thinks z is a 2D vector, not a complex number. Instead, we need to calculate the real and imaginary components of z^2 individually.

2.3 Colours!

The difficult part is now out of the way. Now, we just need to assign a colour to each pixel depending on whether it is part of the Mandelbrot Set or not. For numbers that are included, we simply set the colour to black like this:

```
// |z|^2 <= 4.0 (inside the set)

if (dot(z, z) <= 4.0) {

    fragColor = vec4(0.0, 0.0, 0.0, 1.0);

    return;

}
```

I've decided to make the colour of pixels not within the Mandelbrot Set reflect how quickly that sequence diverges to infinity. The cosine function is used to smoothly cycle

through different colours depending on how many iterations we went through before the algorithm found a term bigger than 2.

```
// Diverges faster = different colour

// I came up with these colours using an advanced technique called picking random numbers and
seeing if they look good

float t = float(iter) / float(u_maxIter);

fragColor = vec4(

    0.5 + 0.5 * cos(6.3 * (t * u_colourFrequency + 0.0)),

    0.5 + 0.5 * cos(6.3 * (t * u_colourFrequency + 0.33)),

    0.5 + 0.5 * cos(6.3 * (t * u_colourFrequency + 0.67)),

    1.0

);
```

3 Chaos from Simple Rules

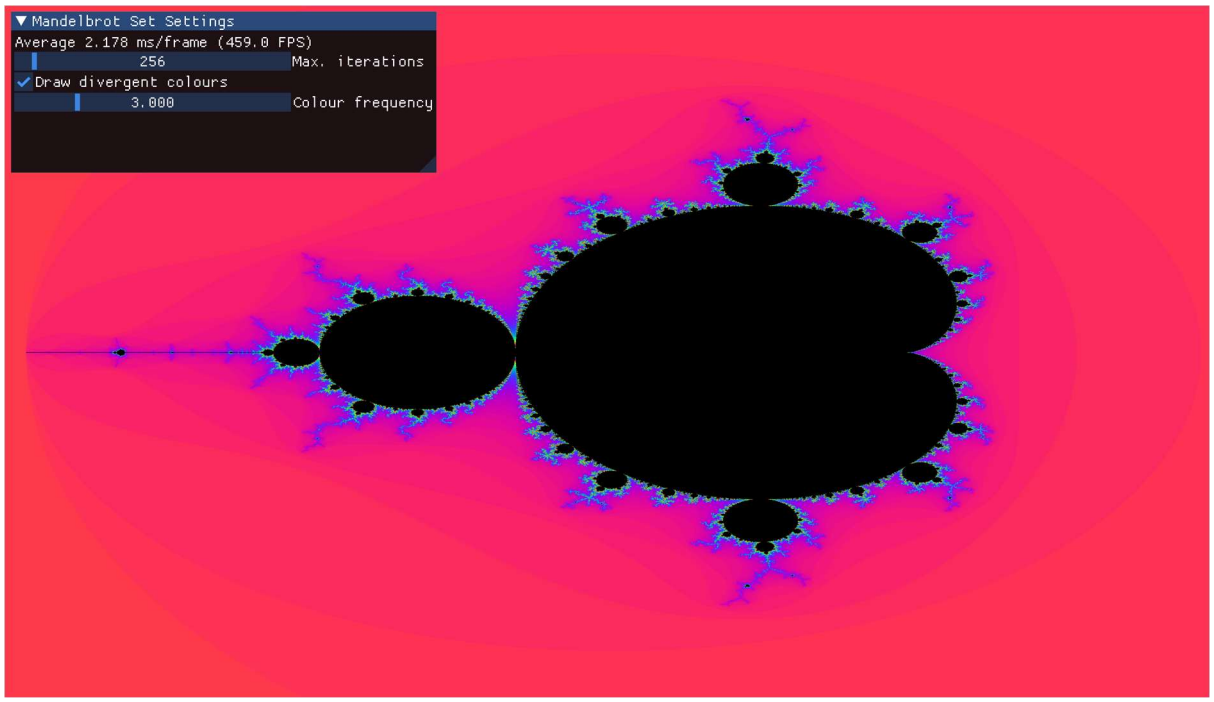

Image: The Mandelbrot Set. Note that the image has been stretched due to the 16:9 aspect ratio.

This is the Mandelbrot Set, drawn using the code discussed in the previous section. The black areas represent values of c where the sequence converges, and the coloured areas represent values of c where the sequence diverges. The colours themselves show how quickly the sequence diverges. Zooming in, you can find even more interesting shapes:

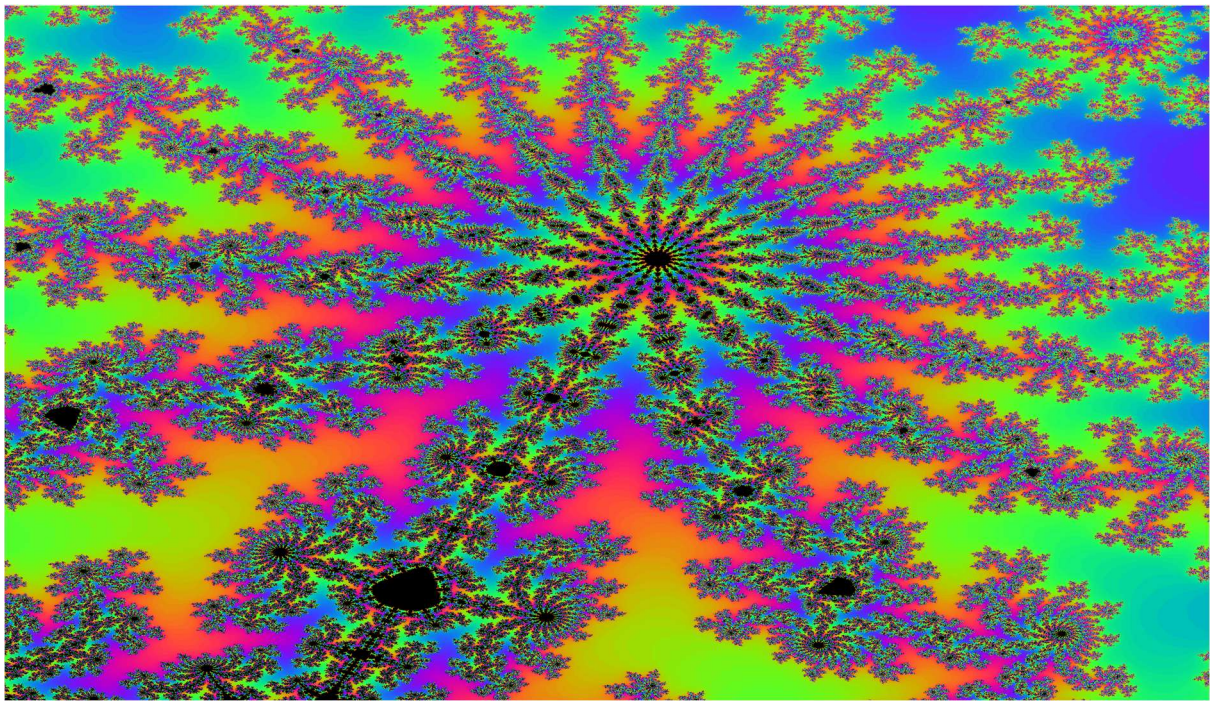

Image: Mandelbrot Set fractal patterns

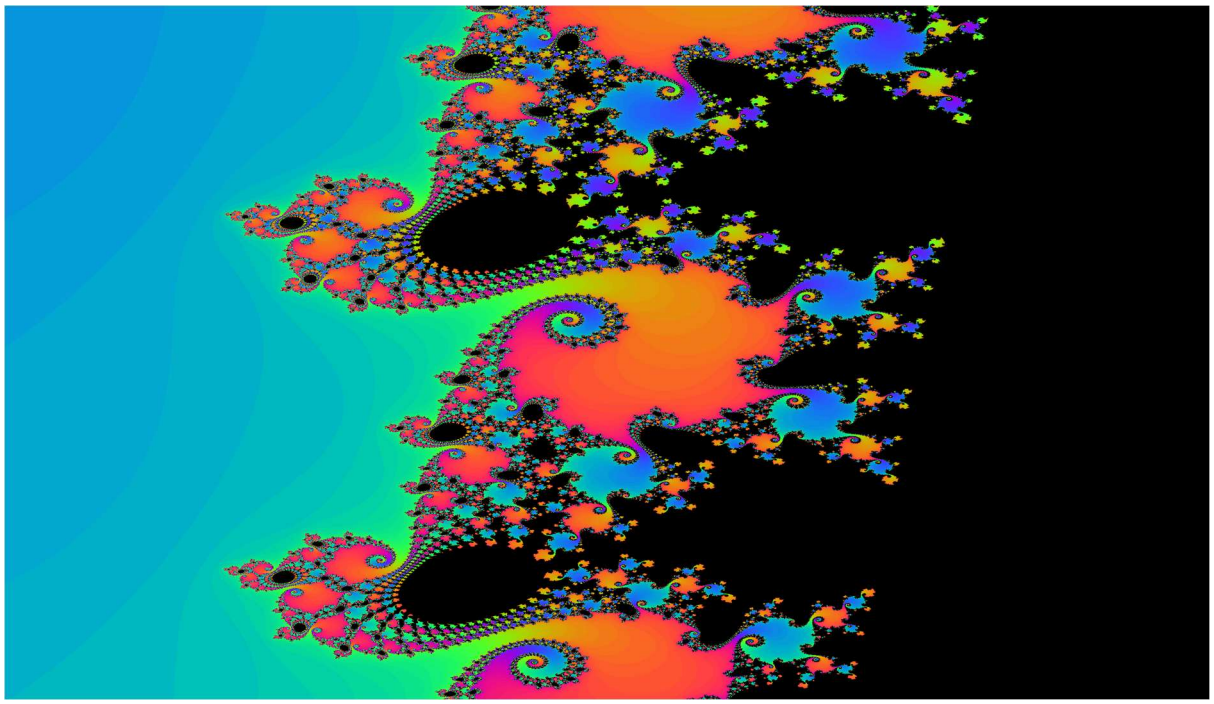

Image: Mandelbrot Set “seahorse valley”

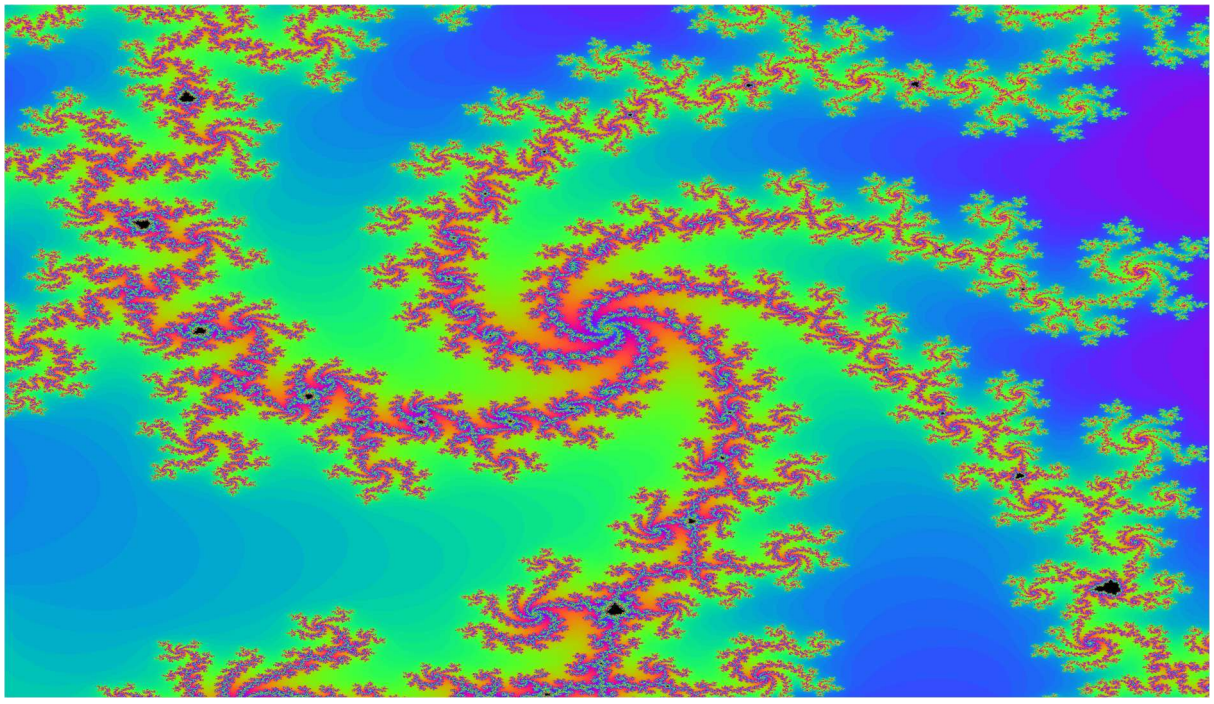

Image: Mandelbrot Set fractal spiral. See how each arm in the spiral is composed of smaller, identical spirals!

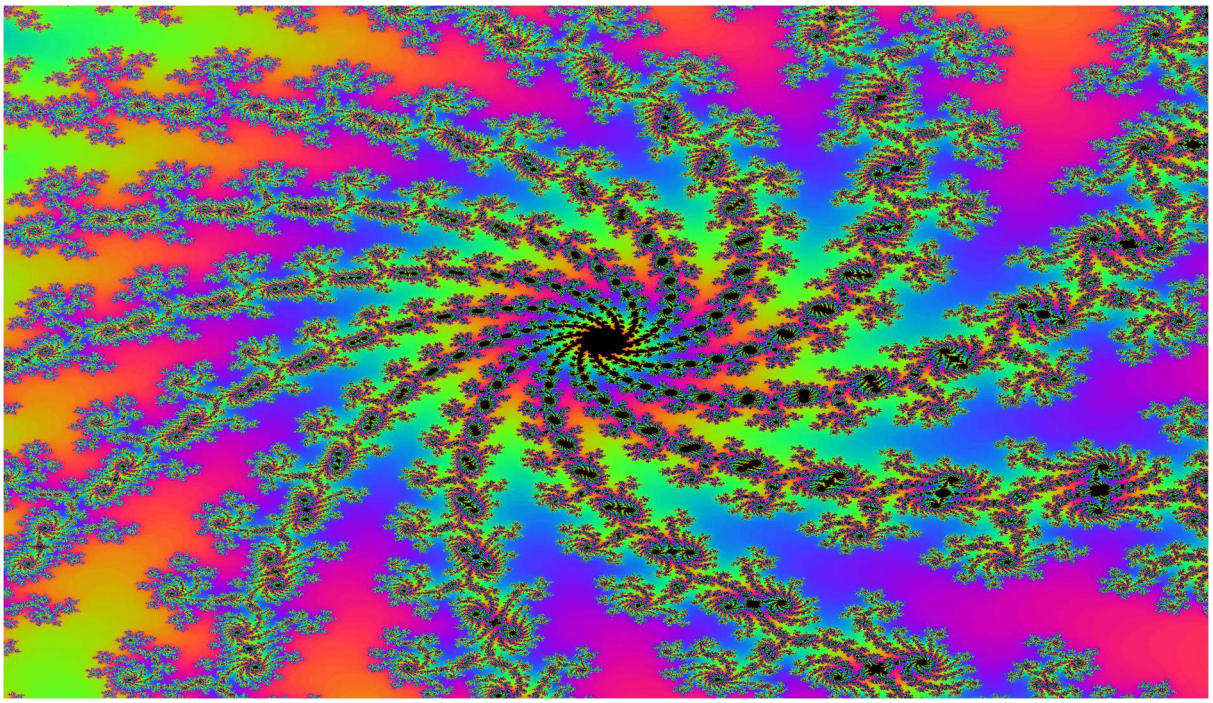

Image: Mandelbrot Set fractal spiral

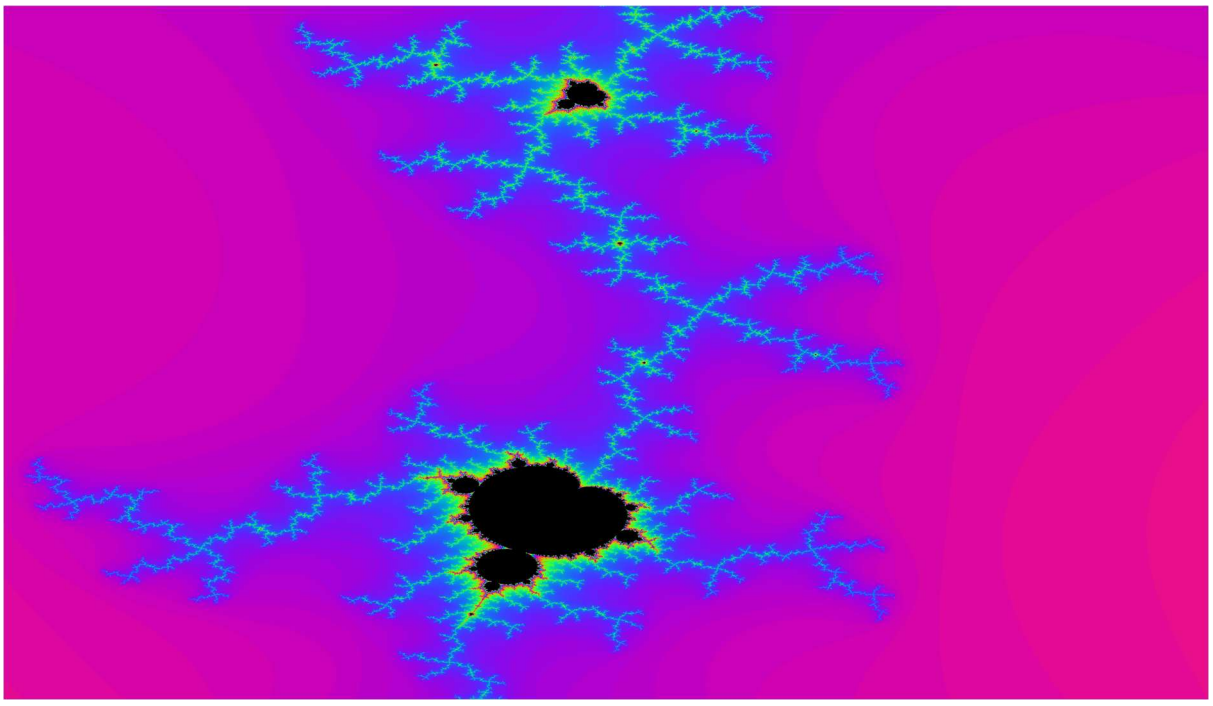

Image: “Mini-Mandelbrots” in the Mandelbrot Set.

Doesn’t it look beautiful? The quasi-similar “Mini-Mandelbrots” found inside the set, the “seahorse valleys” and the spirals all show that we undeniably are looking at a fractal, and one that is far more complex than other famous fractals like the Sierpinski Triangle or Koch Snowflake. So, how can such a simple rule lead to such complexity?

3.1 The meaning of the Mandelbrot set

In order to understand the fractal, we will make three observations.

Observation 1: We can start by looking at what happens in each region of the set. If we calculate the sequence for any value in the cardioid (the large, heart-like shape), the sequence always converges to a single value. In the large circle to the left of the cardioid, the sequence always ends up cycling between two values. In the smaller circle directly above the cardioid, the sequence cycles between three values. In fact, every single "bulb" has a different cycle length.

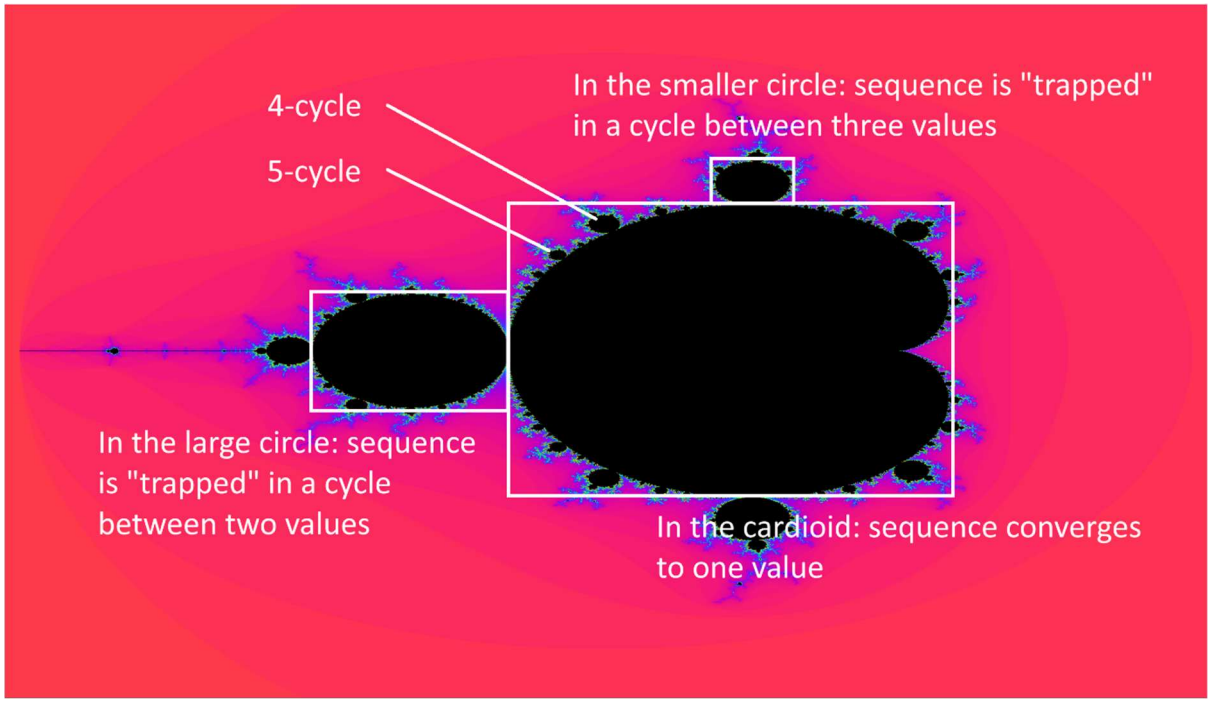

Image: Mandelbrot set bulb cycles

Observation 2: c values deeply inside the set are very stable; if you move them slightly, the sequence still converges. c values far outside of course quickly escape to infinity. The values right on the set's boundary don't quickly converge or diverge, so slightly changing the sequence can drastically push it in either direction. We can call these values "chaotic."

Observation 3: Remember that our rule is:

$$z_{n+1} = z_n^2 + c$$

This is called a recurrence relation, or in other words, we apply our rule to its own output. The self-similarity found when zooming in is not designed, instead it comes from

the recursive nature of the sequence.[6] This recursion is why the “mini-Mandelbrots” appear everywhere.

Bringing these three observations together, we can now understand the fundamental reason why the Mandelbrot Set looks the way it does. Different regions of the set can have an infinite number of different behaviours: the number of cycles, stable versus chaotic, and so on. Our boundary must encode all this information, yet a smooth curve can only encode a finite amount of information. Therefore, the boundary must be infinitely complex.

[6] In this context, I am using the loose mathematical definition of “recursion” which is just that each term is defined in terms of previous terms. In computer science the definition is stricter and the Mandelbrot sequence would instead be described as iterative.

4 Conclusion

While this certainly wasn't the deepest possible dive into fractals or complex dynamics, I've hopefully shown that computer-aided visualisation can be a powerful tool for doing maths and shed a little light on the meaning behind the Mandelbrot Set. Fractals are an interesting, beautiful area of mathematics and much of the research behind them is only made possible using computation.

5 Appendix: Mandelbrot Set Renders

All images of the Mandelbrot Set were created using the app made specifically for this essay. Unless stated otherwise, all images were created with the following settings:

- Max. iterations: 256
- Draw divergent colours: enabled
- Colour frequency: 3.0

The following hardware was used to produce the images:

- 2880x1800 display
- Intel Core Ultra 9 185H
- Intel Arc Integrated Graphics

The full source code (MIT licensed) can be found at:

- https://github.com/AbsoluteNarwhal/trmfractals

6 References

[2] Mandelbrot, B. (1967a) ‘How long is the coast of Britain? statistical self-similarity and fractional dimension’, *Science*, 156(3775), pp. 636–638. doi:10.1126/science.156.3775.636.

[3] *Britain Fractal Coastline* (2006) *Coastline Paradox*. Wikimedia Commons. Available at: https://commons.wikimedia.org/wiki/File:Britain-fractal-coastline-100km.png (Accessed: 24 March 2026).

[5] Nakos, G. (2024) *Elementary linear algebra with applications: MATLAB, Mathematica and maplesoft(tm)*. Berlin: Walter de Gruyter GmbH.