Universiteit Leiden
The Netherlands

Opleiding Informatica

Efficiently generating the Mandelbrot and Julia sets

Luc de Jonckheere

Supervisors:
Rudy van Vliet
Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
`www.liacs.leidenuniv.nl` 29/08/2019

Abstract

The Mandelbrot and Julia sets are very famous fractals. They became widespread because of the visually appealing images they generate. Generating images of these sets is computationally very expensive, especially the more appealing images. This prevents users from interactively exploring these fractals. In this thesis, we research methods to generate fractals efficiently and try to build a user friendly application to explore fractals. This application and its code should also serve as a tool for people interested in learning about developing and optimizing fractal viewers.

There are two main methods of optimizing the computation of fractals. One tries to prevent calculating some pixels using mathematical properties of the fractals. In this thesis we describe symmetry pasting, shape checking and border tracing. The other method utilizes the fact that pixels can be calculated in parallel. The speed-up of these optimizations depends on the part of the fractal which is generated. Experiments show speed-ups of almost 10 to 100 times when using all optimizations with 8 cores compared to a single core brute force implementation.

Contents

Chapter 1

Introduction

This thesis describes methods to generate the Mandelbrot and Julia sets efficiently. In this chapter, we first give an overview of the history behind the sets followed by the motivation for this work. Then, we take a look at current software for generating the Mandelbrot fractal. Finally, we give an overview of this thesis. Figure 1.1 shows some examples of the Mandelbrot fractal.

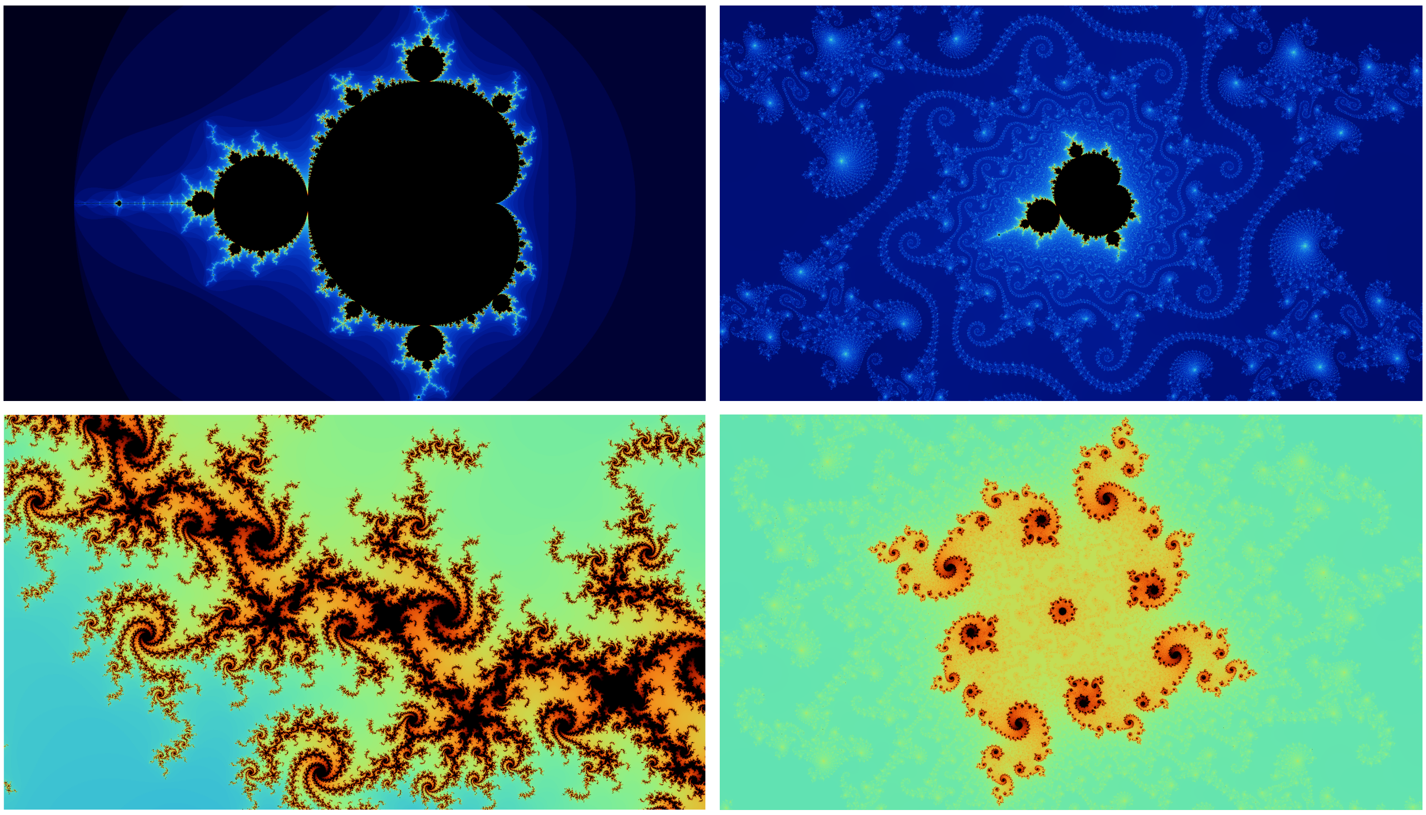


Figure 1.1: Some examples of the Mandelbrot fractal.

1.1 History of the Mandelbrot and Julia fractals

The study of the Mandelbrot and Julia sets is a part of complex dynamics. This is a field of mathematics which studies the behavior of iterative functions in the complex plane. Complex dynamics was pioneered by Pierre Fatou and Gaston Julia around 1920. Fatou was the first to study the function which defines the Mandelbrot set. Unfortunately for him, computers and computer graphics were not advanced enough to visualize his works for another 60 years [BM09]. Julia first described the Julia sets and his publication on this became very famous [Jul18]. Hubert Cremer made the first drawing of a Julia set based on this work [Cre25], see Figure 1.2. Even though Julia became famous, his works were mostly forgotten until much later.
In the 1980s, fractals became popular again through the work of Benoit Mandelbrot in the years 1975–1980. In 1978, the Mandelbrot set was first defined and drawn [Mat78], see Figure 1.3. Note that in this publication, the word fractal was not yet used. Later, the set was named after Benoit Mandelbrot in honor of his work in fractal geometry. Computers became more powerful and computer graphics more advanced. The Mandelbrot set was often used as a graphics demonstration and an increasing number of people learned about it. Even people who were not mathematicians got interested because fractals are visually appealing. Since the Mandelbrot set is one of the first fractals to become popular, it is the most researched fractal.

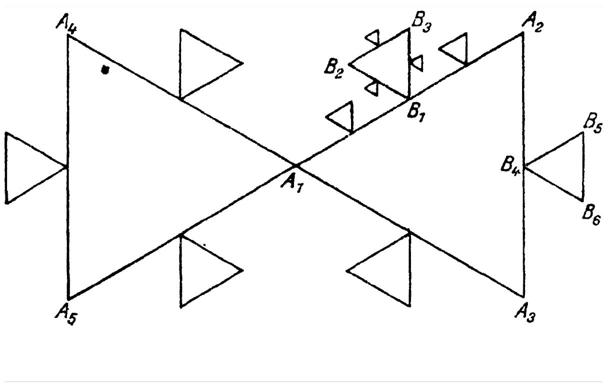


Figure 1.2: First drawing of a Julia set by Hubert Cremer, 1925.

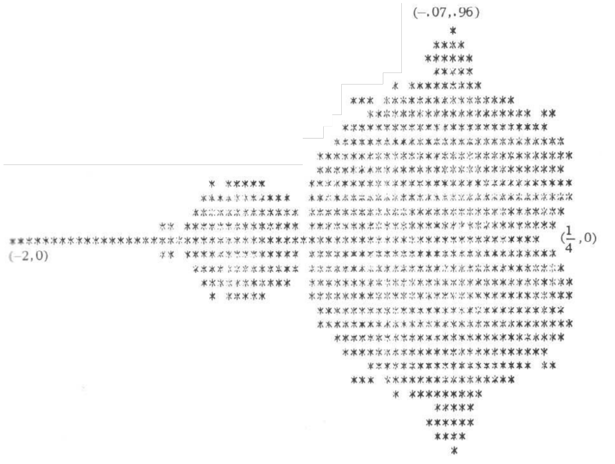


Figure 1.3: First visualization of the Mandelbrot set by Robert W. Brooks and Peter Matelski, 1978.

1.2 Motivation

Current software for generating fractals is still very slow. It can easily take seconds to generate a computationally cheap image. This prevents users from viewing fractals interactively. One possible way to achieve interactivity is by precalculating a specific area. However, it is difficult to guess where interesting or visually appealing parts are, so precalculating has to be done carefully. Also, the deeper the zoom level is, the more visually appealing the resulting images become and calculation times increase significantly.
The goal of this thesis is to research methods to efficiently generate fractals. We hope to perform the calculations fast enough to facilitate interactive viewing. The real-time constraint should be low enough so the user is not hindered by the latency between pressing a button and seeing the result. We set our real-time constraint to one second. Even though a latency of one second is very high, it still is a reasonable time to be asked to

wait, especially on high zoom levels.
In this thesis, we often refer to 'front-end' and 'back-end' when talking about fractal viewers. The task of a back-end in this context is handling the computation of the fractal. The task of the front-end is displaying the fractal and providing an interface for the user which allows interactivity.
We need a tool to test if our results are correct and are calculated within the real-time constraint. Since we could not find a tool which would easily support a different back-end, we decided to develop our own fractal viewer named Fraccert. Fraccert should have a clear division between the front-end and back-end, so it will allow others to change one of the two components. This way, others can easily research other methods of efficiently generating or displaying fractals. To facilitate this, the code should be easy to understand and easily expandable, so modularity and clear high level design is very important. So our second goal is to develop a tool which can be used to research fractal generation methods with. Other fractal viewers are not well suited to perform experiments with, so our tool will focus on a structure which allows timing different parts of the calculation process.
The performance of the front-end is as important for interactivity as the performance of the back-end. So our front-end should efficiently use our back-end to display the calculated fractals. Not only are there some optimizations which can be applied in the front-end to increase interactivity. A user friendly front-end can also save many calculations. For instance, if the rendered screen is not what the user wanted, the screen has to be adjusted slightly and rendered again. If the user had better control, this second render could have been prevented. So our third goal is making an efficient, user friendly front-end to display our results with.

1.3 Related work

There are many articles available on generating the Mandelbrot and Julia sets. Unfortunately, due to the popularity among non-mathematicians, there is much incorrect information available, especially in the easier to understand pieces. Many sources blindly copy from other sources (mostly Wikipedia) without referencing them. Sources with correct information are difficult to find, because there are many sources with incorrect information. The best available information comes from mathematical papers. These papers are very technical and require a thorough understanding of complex dynamics which is a difficult field of mathematics which describes these fractals. They also require knowledge of (algebraic) topology.

1.3.1 Current publications and articles

All important background knowledge of fractals is covered in [Cro95]. This work introduces the mathematical concepts without being too technical. This makes the mathematics describing the sets easy to understand.
There exist many methods to color a fractal. We need a coloring algorithm which takes one parameter and returns a color, which is a three dimensional parameter. So we need a function $f : \mathbb{R} \to \mathbb{R}^3$. We would like this function to be continuous so that two input values which are close together produce two output values which are close together[1]. Many methods rely on expensive operations like the sine function. [Sil13] presents a function which is relatively inexpensive to compute and produces a visually appealing color spectrum.

[1]This does not follow from the definition of continuous. However, if the function is continuous, we can easily scale this function to get the desirable behavior.

[Wil] shows an easy method to prove the Mandelbrot set is symmetric around the real axis and [Bea16] proves this for the Julia sets. These symmetries can be used to optimize the computation of these sets.
[Cro05] shows a method to algebraically determine functions which define parts of the Mandelbrot set. Using these functions, we can skip calculating some pixels. The article expresses the functions in a polar coordinate system. In Fraccert, we use a Cartesian coordinate system, so we have to transform these functions to our coordinate system.
There are general algorithms which perform border tracing, however, these are used in contexts outside of fractal rendering. The main disadvantage of most methods is that the number of considered pixels is not minimized. However, the ideas from these algorithm may be useful in a fractal rendering context. We only found one article on border tracing in a fractal rendering context [Rue12], however, this method is not well suited for SIMD optimizations, see Section 7.2. Border tracing is also mentioned by many other websites, however, these websites only mention the concept and do not give details on implementing a border tracing algorithm. The only details on an implementation of border tracing are segments of undocumented source code [qba].
Another optimization which is mentioned often is periodicity checking. Here we check if an orbit reaches a point it has reached before. If a certain point is reached a second time, we know it will not diverge because the orbit will always return back to this point. This check is very expensive and most orbits do not reach certain points twice, so its speed-up is very questionable. Also, because floats have limited precision (even the GMP ones), this check might introduce errors. If a point reaches a neighboring point so close a float cannot differentiate them, we would assume this point does not diverge. This may be incorrect, because the point is slightly different and may have a totally different orbit. XaoS, which is discussed in the next section, implements this optimization and also questions its use. To increase the effectiveness of this optimization, we could consider points to be equivalent within a certain distance, however, this would further increase the error.

1.3.2 Current software

There are many programs which can calculate and view fractals. Many of these use different techniques to optimize the required calculations. We will discuss four programs. These are chosen because they are well known within the fractal community. Fractint is one of the first popular fractal generators and it uses an interesting optimization which is not useful anymore. The others have some interesting optimizations and also have some problems we would like to avoid in Fraccert.

- **Fractint** [fra]: Fractint is one of the first widespread fractal generators. The name is a combination of fractal and integer. Fractint was created in 1988. Back then, many computers did not have special floating point math units. The ones available were very expensive. Instead of using floats, Fractint only uses integers for its calculations. How this is done is described in [Pet91]. Fractint gained much performance by using integers, since the floating point math units were much slower back then. Because it was one of the first programs which had high performance on many systems, it became very popular.

- **XaoS** [xao]: XaoS supports high frame-rate zooming. It achieves this by reusing pixels from the previous rendered frame. It is difficult to use previously calculated information when zooming. Using straightforward methods causes rendering artifacts. Many heuristics were tried to find a good pixel reusing scheme. In the end, this effort led to a fractal viewer with very efficient scaling. XaoS does not support easy translation of the view. Also, it has a lot of errors when it is not given enough time to correct them. The smooth zooming only remains smooth when the maximum number of iterations is fairly low, which limits the amount of detail.
- **Gnofract 4D** [gno]: Gnofract 4D is a versatile program. It supports custom iterative functions and has many features. Its front-end feels slow and awkward to control. This results in a less interactive experience. It also does not support smooth zooming. Rendering the Mandelbrot and Julia sets is not very fast compared to other programs. This might be because the program is a general fractal viewer. Many optimizations in rendering the Mandelbrot and Julia sets are specific to those sets. It also misses some optimizations like pixel reuse in translation, which makes translating much slower. The program tends to crash when performing too many actions to fast.
- **NVIDIA's Mandelbrot** [nvi]: NVIDIA has a Mandelbrot renderer as sample code for its CUDA SDK. This renderer uses GPU acceleration for calculating the Mandelbrot set. This program is faster than any CPU implementation will ever be when using floats. However, when using doubles or ever higher precision floats the performance quickly decreases, because consumer GPUs have little double precision floating point units. Also, it does not have any algorithmic optimization, because performing checks (branching) is much slower than brute-forcing on GPUs. Because of the performance drop on higher precision floats and large overhead on optimizations, CPU implementations are still relevant.

Fraccert should be interactive and easy/fast to control. It should also achieve these goals without noticeably impacting the performance. Fraccert should not only optimize the computation of the fractal, but also optimize actions in the front-end to achieve minimal latency. It should also support unlimited detail, although it may be hard to remain interactive beyond the limit of most other fractal viewers.

1.4 Our contributions

Distance estimation (Section 3.2) is normally used for smooth coloring, so there are many fractal viewers which use similar formulas as ours. However, we did not find literature or fractal viewers using distance estimation to make the thin filaments visible.
Symmetry pasting (Section 4.3) is used by many fractal viewers. Most implementations use the CPU for pixel copying. We did not find any work which uses GPU acceleration for this.
As discussed in Section 7.1, it is possible to use a part of the real axis as a shape. As far as we know, this shape is not considered in other publications or programs. Also, we did not see mentions of using hitbox detection to only check for shapes within screen bounds. Most fractal viewers hard-code these to shapes in the back-end, which prevents dynamically changing the shapes which are checked for, which in turn prevents an easy implementation of the hitbox detection. This is why we use a list of shapes which the back-end has to check for.

The general idea of border tracing (Section 4.5) came from other articles, but our implementation was our own creation. No work mentions multi-threading combined with border tracing (Section 4.5.2), so we had to devise our own threading scheme (dividing the screen in blocks). The only other understandable implementation we found did not foresee SIMD (Section 7.2) in the future, which is possible with our implementation. Other implementations coded border tracing into the function which computes the fractal, which prevent others from only copying the border tracing part. Our implementation is separated from calculating the fractal, which makes it easy to reuse in other projects.
When we developed the Julia window (Section 3.5) to visualize the Julia sets morphing, we did not know of any program which had this feature. However, long after implementing it, we found out XaoS has a similar feature. Not knowing this during development may have helped with designing our version which differs from XaoS' much, which resulted in a much less limited implementation.
In this thesis, all building blocks necessary to develop a fractal viewer are extensively described. We also systematically tested the optimizations we implemented to measure their effectiveness; something we did not find elsewhere when researching the optimizations methods. Current fractal viewers are not well suited to perform experiments with. With our tool, new optimizations can easily be implemented in a clear manner and experiments can be performed on them.

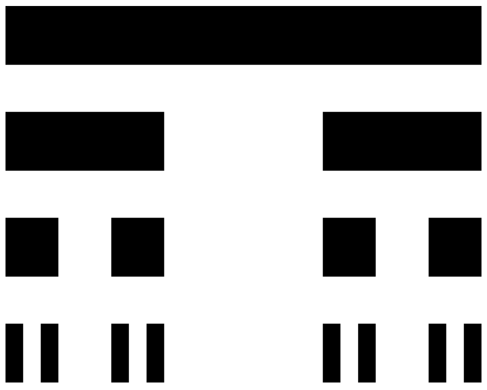
1.5 Thesis overview

This thesis is organized as follows. Chapter 2 contains background information about fractals, the math used to describe them and some interesting behavior of fractals. Chapter 3 describes the different features of Fraccert and the corresponding implementations. Chapter 4 describes more efficient implementations of some features and methods to speed render times up. The performance of Fraccert and its optimizations is measured in Chapter 5 and is concluded in Chapter 6. In Chapter 7 we discuss features and optimizations which could still be added or improved.

Chapter 2

Preliminaries

There is no real definition for fractal. Before the Julia sets were found, the only known fractals consisted of simple geometric rules or patterns. Examples of such fractals are the Cantor set and the Sierpinski triangle, see Figure 2.1. Back then, the word fractal was not used. However, the definition of these figures was along the lines of recursive self-similarity. Later, the definition became more refined and became along the lines of a figure whose complexity is invariant under scaling (i.e. when you zoom in, the amount of detail does not change). However, this definition is informal, as there is no good measure to define complexity of detail with. After Benoit Mandelbrot discovered fractals, he coined the term fractal in 1975 and defined it as "a set with a fractional Hausdorff dimension[1]" [Man77]. Later it was refined to "a set having Hausdorff dimension strictly greater than its topological dimension", since some fractals have a integer Hausdorff dimension. However, this definition was also found to be too restrictive. Now, there is no formal definition of fractal, only a list of characteristics which are often found in fractals such as self-similarity.


(a) Cantor set

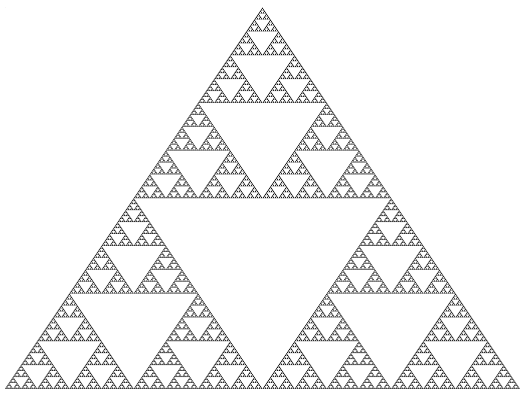

(b) Sierpinski triangle

Figure 2.1

[1]For integers, the Hausdorff dimension agrees with the topological dimension. A line is 1D, a square is 2D and a cube is 3D. When you halve the measure of each dimension for a shape with an integer dimension (cut the shape in halve through every dimension), the volume of the shape reduces by 2^D, where D is the dimensions of the shape. This leaves us with 2^D smaller copies of the shape. When you halve the measure of each dimension of the Sierpinski triangle (to get n smaller copies of the original shape), you'll get 3 smaller copies of the triangle so it divides the volume by 3. So the dimension of the Sierpinski triangle is $2^D = 3$, which is $log_2(3) \approx 1.585$. See [Man82] or [HOP92] for a more detailed explanation. Here, Hausdorff dimension is referred to as fractal dimension.

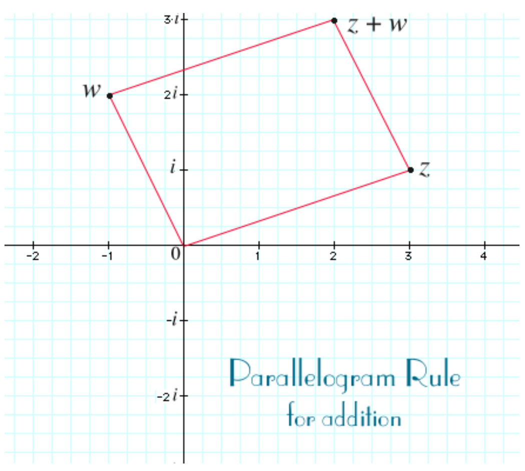
In this chapter, we quickly discuss basic complex math, because the Mandelbrot and Julia sets are sets of complex numbers. Subsequently, we define and explain the Mandelbrot and Julia sets. Then, there is an estimation of the amount of work needed to generate an image of the Mandelbrot set to show how important efficiency is. Finally, we show different kinds of self-similar behavior.

2.1 Complex mathematics

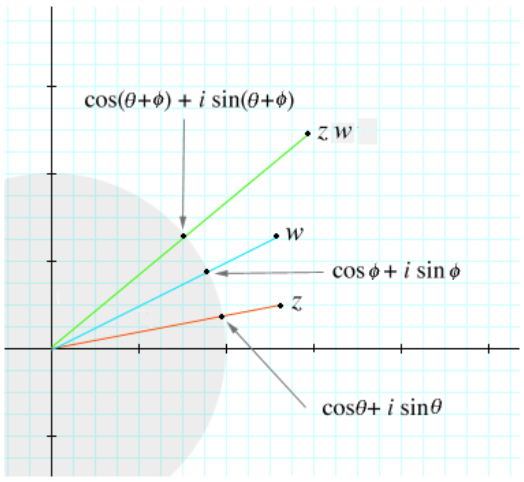
In this thesis, we will refer to a complex number as a pair $z = [x, y]$, which represents $z = x + yi$. The function $\Re(z) = x$ gives the real part of z and $\Im(z) = y$ the imaginary part as a real number.
A complex number can be represented as a point in a graph, see Figure 2.2. Here, a complex number represents coordinates in the complex plane. The addition of two complex numbers can be represented as the addition of the two vectors describing the points, see Figure 2.2a. You can also define a complex number as $z = r * (\cos(\theta) + i\sin(\theta))$, where $r = |z|$ (the modulus; distance to origin) and θ is the angle from the real axis. In this representation, a complex multiplication is defined as

$$\begin{aligned} z * w &= |z| * (\cos(\theta) + i\sin(\theta)) * |w| * (\cos(\phi) + i\sin(\phi)) \\ &= |zw| * (\cos(\theta + \phi) + i\sin(\theta + \phi)). \end{aligned}$$

This can be interpreted as adding the angles and multiplying the moduli of the complex numbers, see Figure 2.2b.



(a) Addition



(b) Multiplication

Figure 2.2: Visualization of complex operations.
Source: `https://www2.clarku.edu/faculty/djoyce/complex/`

2.2 Mandelbrot set

The Mandelbrot set is a set of complex numbers. A complex number c is in the Mandelbrot set if it does not diverge when iterated through the iterative function $m_c(z) : z_{n+1} = z_n^2 + c$ for $n = 0$ to ∞, where $z_0 = 0$. Figure 2.3 visualizes some iterations of the iterative function for two complex numbers. For any iterative function, the left side is always z_{n+1}, so we will omit this in the rest of the thesis. Also, on the right side the subscript for z is always n, so the subscript will also be omitted.

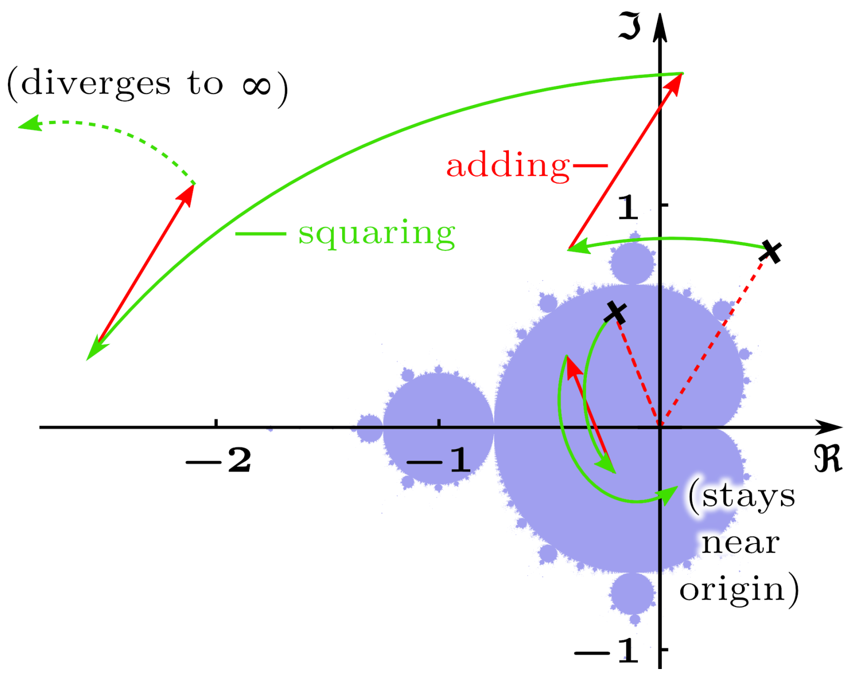


Figure 2.3: A visualization of iterating complex numbers in the complex plane.
Source: `https://commons.wikimedia.org/wiki/File:Complex_mandelbrot_illustration.png`
Edited by the author.

It can easily be proven that complex numbers z where $|z| > 2$ diverge [Cro95]. This implies that the Mandelbrot set is bounded by a closed disk around the origin of radius 2. Here is an example calculation for $c = [-1.5, 1]$.

$$\begin{array}{llll|ll} z_0 & = [0,0] & & & |[0,0]| & = 0 \\ z_1 & = [0,0]^2 + [-1.5,1] & = [-1.5,1] & & |[-1.5,1]| & \approx 1.803 \\ z_2 & = [-1.5,1]^2 + [-1.5,1] & = [-0.25,-2] & & |[-0.25,-2]| & \approx \underbrace{2.016 > 2}_{\text{diverges!}} \end{array}$$

So $[-1.5, 1]$ is not an element of the Mandelbrot set.
When visualizing the Mandelbrot set, complex numbers that have been found to be in the set are usually colored black. The complex numbers outside the set are usually colored based on how fast they diverge. There are many algorithms to calculate the color of a complex number outside the set, which are explained in Section 3.1.2.

2.3 Julia sets

A Julia set is defined as the boundary of a set of complex numbers which do not diverge when iterated through an iterative function. The filled in Julia set is the set of all complex numbers which do not diverge. In this thesis, we will always refer to the filled in Julia sets. In the case of Julia sets, the iterative function can take any rational function $f(z) = p(z)/q(z)$, where $p(z)$ and $q(z)$ are complex polynomials. For some functions, the Julia set is not a fractal, e.g. $f(z) = z^2$, where the Julia set is the unit disk. Note that other iterative functions which are not a rational complex polynomial may also result in a fractal. Even though these fractals may behave like Julia sets, they are not.
In this thesis, we will focus on the Julia sets $J(f_c)$ with iterative function $f_c(z) = z^2 + c$. When f_c is used in this thesis, it always refers to this iterative function. Also, when referring to Julia sets, we always refer to $J(f_c)$

unless stated otherwise. Here, c is a complex number which can be chosen arbitrarily and z_0 is the complex number of which its membership is checked.
There are two kinds of Julia sets, connected and fully disconnected [Cro95]. When a Julia set is connected, there is a path from every complex number in the set to every other without leaving the set. When it is fully disconnected, every connected subset has one element. In other words, there is always a complex number which is not in the set between two complex numbers in the set. The Mandelbrot set can also be defined as the set of complex numbers c for which the Julia set is connected [HOP92].
Calculating Julia sets is very similar to calculating the Mandelbrot set for an arbitrary iterative function. Where the Julia sets start with $z_0 = z$, the Mandelbrot starts with $z_0 = 0$. Note that for the Mandelbrot set $z_1 = z_0^2 + c = 0^2 + c = c$. Where the Julia sets pick a complex number z_0 which is iterated, the Mandelbrot set picks a complex number c which is iterated. This complex number corresponds to the location of a pixel in the complex plane.

2.4 Self-similarity

Even though the Mandelbrot set shows chaotic behavior, when looking at the set, its patterns feel predictable. This is likely due to its self-similar nature. Around some special points, called Misiurewicz points [Lei89], it shows infinite self-similar behavior. The pattern in a neighborhood of a Misiurewicz point will repeat infinitely when scaled with specific sequence of varying factors. After scaling, the rendered image will be unchanged except for a rotation. Images which are invariant under scaling (and rotation) show the Droste effect. When zooming in between two Misiurewicz points, the emerging pattern will look like a mix of the patterns around the two points. Julia sets also show the Droste effect, however, Julia set patterns repeat when scaled with a specific constant factor. In theory, the Julia sets could be fully rendered by rendering the parts which remain invariant under scaling and rotation. The Julia set can then be constructed by copying this part around. This would be very efficient, because the set only has to be calculated once and can then scale and rotate infinitely. Also, scaling and rotation are done very efficiently by GPUs, so this would be incredibly fast. However, it is very difficult to determine what parts are invariant under which scaling and rotating factors and no program has successfully implemented this technique as of yet.
The Mandelbrot set and Julia sets are also very similar to each other. The Julia set with complex parameter c will resemble the patterns of the Mandelbrot set when rendering around point c [HOP92], see Figure 2.4 for examples. Unfortunately, the amount of detail possible in the figure is very limited. See Section 3.5 for more information about exploring this phenomenon. The shapes of the Julia sets can also be found within the Mandelbrot set.

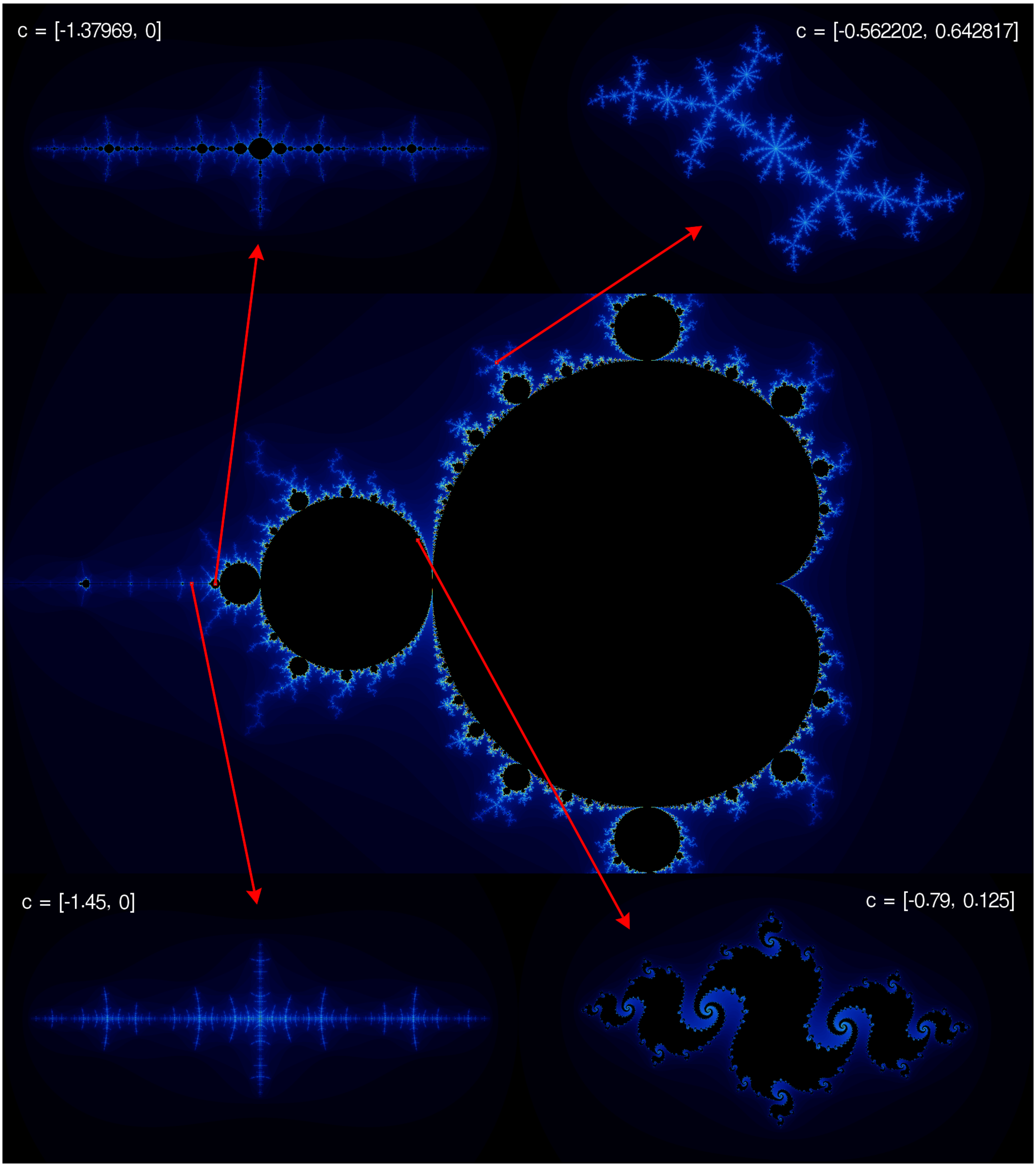


Figure 2.4: Different values of c in the Mandelbrot set and the corresponding Julia set.

Chapter 3

Implementation

The fractal viewer we implemented, Fraccert, is written in C++ and compiled with GCC's g++ (9.1.0), using SDL2 (2.0.9) for the graphics and user interaction. Fraccert is developed for Linux, however, all OS interaction is done through SDL, so it should be platform independent. Fraccert also uses GMP (6.1.2) (GNU Multiple Precision arithmetic library). There is no official Windows support for GMP, but it can be compiled on Windows. OpenMP is used for multi-threading, so your compiler has to support that. The source code for Fraccert can be found at `https://github.com/lucmans/fraccert`.
The controls for Fraccert can be found by running "./fraccert --help". When Fraccert is started, the terminal it is started from becomes a console which can be used to interact with Fraccert. The available commands are listed at startup. Note that all input actions are queued, which means that all given input will be processed, even if they are given when waiting for a fractal to render. The input queue can be emptied with the escape key ('Esc'), so the user does not have to wait for all input to process before regaining control over the application.
In this thesis, the following definitions will be used:

- Re_{min} and Re_{max}: These numbers represent the interval $[\text{Re}_{min}, \text{Re}_{max}]$ which is displayed horizontally.
- Im_{max} and Im_{min}: The interval $[\text{Im}_{max}, \text{Im}_{min}]$ which is displayed vertically.
- ΔRe and ΔIm: The size of our view in the real and imaginary axis ($\Delta\text{Re} = \text{Re}_{max} - \text{Re}_{min}$).
- `NMAX`: The number of iterations before concluding a complex number is part of a set.
- Pixel (x, y): The pixel at coordinates x and y, where x is in $[0, \texttt{screenWidth}]$ and y is in $[0, \texttt{screenHeight}]$.

Note that pixel (x, y) is x pixels from the left and y pixels from the top of the screen. When x increases, the corresponding real value increases. When y increases, the corresponding imaginary value decreases. So pixel $(0, 0)$ is at complex number $[\text{Re}_{min}, \text{Im}_{max}]$.
In this chapter, we will first show how the Mandelbrot and Julia sets are calculated and show different methods to color them. Then, details about translation and scaling are described. Subsequently, we explain the problem with using normal floats. Finally, we discuss some features of Fraccert and explain the architecture and design choices behind Fraccert.

3.1 Escape time algorithm

Rendering an image of a fractal consists of two steps. First, for each pixel, we calculate if the corresponding complex number is in the set. If not, this calculation tells us how fast it diverges. Then, using this information, we can color the pixel. This method of pixel coloring is called an escape time algorithm, because the time (iteration) when a point escapes (diverges) is used for coloring.

3.1.1 Calculating escape iteration

The function which calculates the Mandelbrot set requires Re_{min}, Re_{max} and Im_{max} (or Im_{center}). Im_{min} is calculated based on the aspect ratio to prevent stretching the fractal. Then, the size of a pixel in the complex plane is calculated. With this, we can start at (Re_{min}, Im_{max}) and iterate over all pixels by incrementing the current complex number by the pixel size. Each pixel then goes through the iterative function and afterwards its color is calculated. See Algorithm 1 for a pseudo-code implementation. Note that the actual implementation has some small optimizations. For instance, $|z|$ is calculated as $\sqrt{\Re(z)^2 + \Im(z)^2}$. When both sides of the inequality are squared, we get $\Re(z)^2 + \Im(z)^2 > 4$, which omits taking the square root of the left side.

```
input  : $\text{Re}_{min}$, $\text{Re}_{max}$ and $\text{Im}_{max}$
output : Pointer to pixelbuffer of rendered fractal
screenWidth ← getWidthPixels();
screenHeight ← getHeightPixels();
pixels[screenWidth * screenHeight] ← 0;
pixelSize ← ΔRe/screenWidth;
for $y$ ← 0 to screenHeight do
    for $x$ ← 0 to screenWidth do
        $c$ ← [$\text{Re}_{min} + (x *$ pixelSize), $\text{Im}_{max} - (y *$ pixelSize)];
        $z$ ← 0;
        $n$ ← 0;
        for $n$ to NMAX do
            if $|z| > 2$ then break;
            $z \leftarrow z^2 + c$
        end
        pixels[$y *$ screenWidth $+ x$] ← colorPixel($n$);
    end
end
return pixels
```

Algorithm 1: Escape algorithm for calculating the Mandelbrot set.

3.1.2 Coloring

The color a pixel gets is based on the speed with which iterations diverge from the origin. We use the iteration count of when the modulus of the complex number became greater than two, the escape iteration, as a measure of speed. If we divide the escape iteration by NMAX, we get a normalized value t in $[0, 1]$. With this normalized value, it is easy to color a pixel. The straightforward method would be mapping the normalized range to the range of the colorspace which is $[0, 16\,777\,216]$, because each color (red, green and blue) has a value in $[0, 255]$ and $256^3 = 16\,777\,216$. This method has a big disadvantage. The coloring is not continuous, which means two escape iteration counts that are very close together might have very different colors, see Figure 3.1 for an example. This is called banding. The color continuity is dependent on `NMAX`, so colors change wildly when `NMAX` is changed a little bit. For some values, it only uses one or two colors. This is because it will divide 16 777 216 in such a way, that some of the least significant bits are always zero. In the case of `NMAX` $= 256$, the sixteen least significant bits are zero, so only the most significant color channel will be used. One advantage of this coloring method is that its visualization shows the difference in high/low detail areas very well for most values of `NMAX`, because of the strong banding.

There is another potential problem with coloring pixels. If `NMAX` is bigger than the number of colors, some escape iteration levels will get the same color. This only happens when `NMAX` is very high, so it is not a big problem. We further discuss this in Appendix B.1.

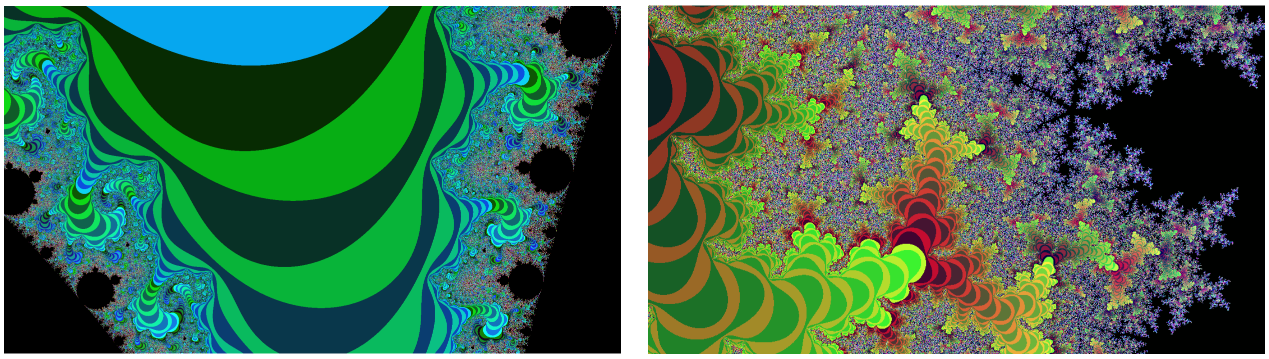


Figure 3.1: Example of banding which occurs on bad coloring algorithms.

To get a smooth transition between color bands, we need a continuous mapping from the normalized iteration count to a 3D color space. We have implemented the method described in [Sil13]. This method uses a slight modification of Bernstein polynomials to achieve a continuous coloring scheme. We use three polynomials to map each color channel to a value in $[0, 255]$, see Figure 3.3. Because the three polynomials 'flow' into each other, the color will smoothly transition to the next one when iterating over t. The formulas for the polynomials are

$$\begin{aligned} r(t) &= 9 * (1-t) * t^3 * 255 \\ g(t) &= 15 * (1-t)^2 * t^2 * 255 \\ b(t) &= 8.5 * (1-t)^3 * t * 255 \end{aligned}$$

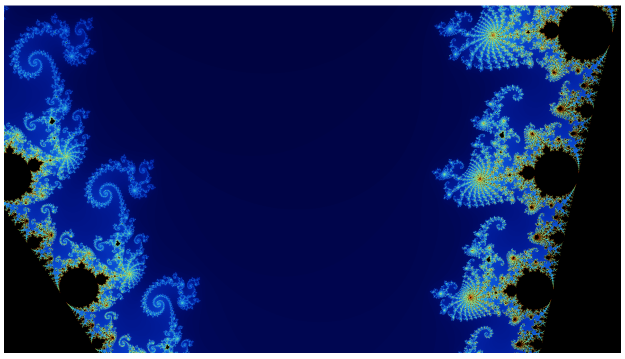


Figure 3.2: Coloring using Bernstein polynomials.

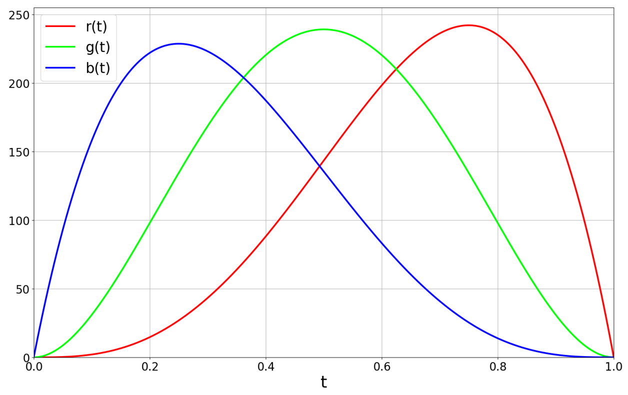
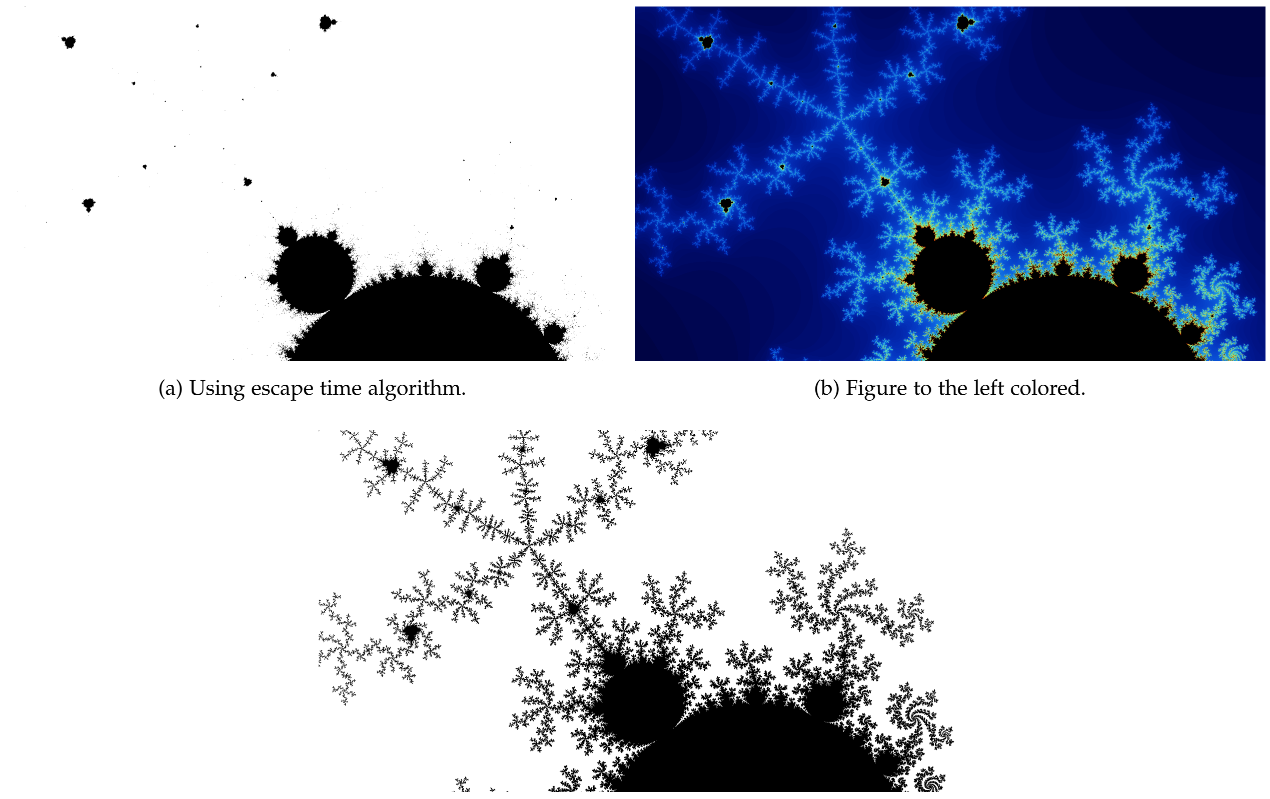


Figure 3.3: A plot of the three Bernstein basis polynomials.

Even though the color bands flow into each other nicely, banding is still visible. See Appendix B.1 on how to prevent this. Note that pixels which escape immediately ($n = 0$) and pixels which diverge ($n = \texttt{NMAX}$) are both colored black. This is not a problem, since all points which diverge immediately lie far away from the set. In Fraccert, immediately escaping points are slightly colored to remedy this problem.

3.2 Exterior distance estimation coloring

When rendering an image using the escape time algorithm, the focus of the rendered image is on the behavior of the iterative function around the set. This is partly because the detail is in the colors, but more importantly, often most pixels in our view are not in the set at all. The set has very small filaments (branches) connecting bigger "islands" of the set [Mun11]. Since these filaments are so small, they often travel between pixels. An escape time rendered image of the set, where black is in the set and white outside, will look disconnected and sparse, see Figure 3.4a.
Instead of rasterizing the domain within our view and calculating the escape velocity for each pixel to color it, we could also use the smallest distance to a point in the set as a measure to color a pixel. This measure emphasizes the shape of the actual set instead of the behavior around the set, see Figure 3.4c. The resulting pattern closely resembles the pattern made by the color bands, see Figure 3.4b.


(a) Using escape time algorithm.

(b) Figure to the left colored.

(c) Using exterior distance estimation.

Figure 3.4: Three images of the same part of the Mandelbrot set.

As described in [YD02], the exterior distance can be estimated by using the Hubbart-Douady potential

$$G(c) = \lim_{n\to\infty} \frac{1}{2^n} \ln |z_n| \tag{3.1}$$

The distance estimate is given by

$$d = \frac{G(c)}{|G'(c)|} \tag{3.2}$$

Where the derivative of $G(c)$ is

$$G'(c) = \lim_{n\to\infty} \frac{1}{2^n} \cdot \frac{|z'_n|}{|z_n|} \tag{3.3}$$

So the distance can be calculated as

$$d = \lim_{n\to\infty} \frac{|z_n| \ln |z_n|}{|z'_n|} \tag{3.4}$$

Since $G'(c)$ is a function of c, the derivative z'_n is with respect to c which gives us

$$z'_{n+1} = 2z_n z'_n + 1 \tag{3.5}$$

Note that the '$+1$' term is only applied to the Mandelbrot set. When calculating the derivative of the Julia sets with iterative function $f_c(z)$, the c in $z_{n+1} = z_n^2 + c$ is constant, so derives to 0.
The implementation of the distance estimation resembles the algorithm described in Section 3.1.1, however, in the distance estimation version, the inner loop also has to keep track of $z'_{n+1} = 2z_n z'_n + 1$, see Algorithm 2. Instead of coloring the pixels continuously as described in Appendix B.1, the pixels are colored black if they are "close enough" to the set. The value of close enough should scale with the rendered domain. When rendering the set fully zoomed out, close enough might be within 0.1 (Cartesian distance). However, if the domain is only 0.001 wide, our entire view would be black if there is a point in the set nearby. So close enough scales with ΔRe / `CLOSEENOUGH`, where `CLOSEENOUGH` can chosen by the user.

3.3 Scaling and translation

There are two ways of interacting with the view on the fractal. You can move the part of the fractal which is viewed (translation), or zoom in or out (scaling). This section describes the straightforward method of scaling and translation. More advanced methods are described in Section 4.2.

3.3.1 Translation

In order to translate our view, we only have to update Re_{min} and Re_{max} or Im_{min} and Im_{max} by the same amount. To move left or right, respectively decrease or increase Re_{min} and Re_{max}. To move up and down, respectively increase or decrease Im_{min} and Im_{max}. If we then render the fractal for the new bounds, our view will have translated. When a naive implementation is used, all pixels will be recalculated. However, only the pixels that enter our view need to be calculated and the other remaining pixels can be reused. This technique will be explored in Section 4.2.

```
input  : $\mathrm{Re}_{min}$, $\mathrm{Re}_{max}$ and $\mathrm{Im}_{max}$
output: Pointer to pixelbuffer of rendered fractal
screenWidth $\leftarrow$ getWidthPixels();
screenHeight $\leftarrow$ getHeightPixels();
pixels[screenWidth $*$ screenHeight] $\leftarrow 0$;
pixelSize $\leftarrow \Delta\mathrm{Re}/$screenWidth;
lineWidth $\leftarrow \Delta\mathrm{Re}/$LINEWIDTH;
for $y \leftarrow 0$ to screenHeight do
    for $x \leftarrow 0$ to screenWidth do
        $c \leftarrow [\mathrm{Re}_{min} + (x * \text{pixelSize}), \mathrm{Im}_{max} - (y * \text{pixelSize})]$;
        $z \leftarrow 0$ ; //For Julia sets, use $z_0$
        $z' \leftarrow 0$ ; //For Julia sets, use 1
        $n \leftarrow 0$;
        for $n$ to NMAX do
            if $|z| > 2$ then break;
            $z' \leftarrow 2 * z * z' + 1$ ; //For Julia sets, omit the $+1$
            $z \leftarrow z^2 + c$
        end
        if $n =$ NMAX then
            $d \leftarrow 0$
        else
            $d \leftarrow |z| * \ln|z| / |z'|$
        pixels[$y *$ screenWidth $+ x$] $\leftarrow$ ($d <$ lineWidth ? BLACK : WHITE);
    end
end
return pixels
```

Algorithm 2: Exterior distance estimation for calculating the Mandelbrot set.

```
input  : Center of zoom $(x, y)$ and scaleDirection (1 or -1)
output: Screen boundaries are updated
xRatio $\leftarrow x/$getWidthPixels();
yRatio $\leftarrow y/$getHeightPixels();
$\Delta\mathrm{Re} \leftarrow \mathrm{Re}_{max} - \mathrm{Re}_{min}$;
$\Delta\mathrm{Im} \leftarrow \mathrm{Im}_{max} - \mathrm{Im}_{min}$;
if scaleDirection $= 1$ then   // Zoom in
    $\Delta(\Delta\mathrm{Re}) \leftarrow (\text{SCALEFACTOR} * \Delta\mathrm{Re}) - \Delta\mathrm{Re}$;
    $\Delta(\Delta\mathrm{Im}) \leftarrow (\text{SCALEFACTOR} * \Delta\mathrm{Im}) - \Delta\mathrm{Im}$;
else   // Zoom out
    $\Delta(\Delta\mathrm{Re}) \leftarrow ((1/\text{SCALEFACTOR}) * \Delta\mathrm{Re}) - \Delta\mathrm{Re}$;
    $\Delta(\Delta\mathrm{Im}) \leftarrow ((1/\text{SCALEFACTOR}) * \Delta\mathrm{Im}) - \Delta\mathrm{Im}$;
end
$\mathrm{Re}_{min} \leftarrow \mathrm{Re}_{min} - (\Delta(\Delta\mathrm{Re}) * \text{xRatio})$;
$\mathrm{Re}_{max} \leftarrow \mathrm{Re}_{max} + (\Delta(\Delta\mathrm{Re}) * (1 - \text{xRatio}))$;
$\mathrm{Im}_{max} \leftarrow \mathrm{Im}_{max} + (\Delta(\Delta\mathrm{Im}) * \text{yRatio})$;
$\mathrm{Im}_{min} \leftarrow \mathrm{Im}_{min} - (\Delta(\Delta\mathrm{Im}) * (1 - \text{yRatio}))$;
```

Algorithm 3: Algorithm which updates the screen bounds when scaling.

3.3.2 Scaling

Scaling the view around point (x, y) can be seen as increasing or decreasing the distance between the bounds for which the fractal is rendered. Only the real part will be considered in this section, since the calculations for the imaginary part are similar. Let the real distance $\Delta\text{Re} = \text{Re}_{max} - \text{Re}_{min}$. When scaling our view, ΔRe should increase or decrease by a certain factor, the scale factor, to get the new distance $\Delta\text{Re}'$. So we have to update Re_{min} and Re_{max} such that $\Delta\text{Re}' = \texttt{SCALEFACTOR} * \Delta\text{Re} = \text{Re}'_{max} - \text{Re}'_{min}$. Note that $\texttt{SCALEFACTOR} < 1$ decreases $\Delta\text{Re}'$ and thus zooms in.

Re_{min} and Re_{max} should change in such a way, that the center of scaling is point (x, y). This can be achieved by changing Re_{min} by an amount proportional to the fraction of pixels left of x (xRatio) and changing Re_{max} by an amount proportional to the fraction of pixels right of x (1−xRatio). Both changes together should amount to the desired increase or decrease of the distance between the bounds. Now, when we use $\Delta(\Delta\text{Re}) = \Delta\text{Re}' - \Delta\text{Re}$ (i.e. the change in distance), the bounds Re_{min} and Re_{max} can be updated by the following two formulas:

$$\begin{aligned}\text{Re}_{min} &-= \Delta(\Delta\text{Re}) * \text{xRatio}\\ \text{Re}_{max} &+= \Delta(\Delta\text{Re}) * (1-\text{xRatio})\end{aligned}$$

With these new bounds, the fractal can be calculated as described before. See Algorithm 3 for a pseudo-code implementation of the scaling algorithm.

3.4 Arbitrary precision floats

In Fraccert, we use 'normal' floating point numbers with double precision by default. If we were to use a GPU, we would have been limited to single precision because most consumer GPUs have little double precision float units. Floats can only describe a finite number of real numbers. The numbers which can be accurately described depend on the float standard used. The IEEE 754-2008 [ieee08] standard (which describes the standard for floating point arithmetic) states that a float consists of three parts; one bit for the sign of the number, the exponent and the significant (sometimes called the mantissa). The significant describes an integer which is then multiplied by 10^{exponent}, like scientific notation for numbers. Note that most implementations of floats use base 2 instead of 10. Since the base of a float does not change its behavior (only the number it represents), we assume the base of a float is 10, because it is more intuitive to work with.

When calculating a very small domain of a fractal, two things can happen. This significant may not have enough bits to describe the number. This is comparable to a number being larger then `INT_MAX`. Secondly, the exponent may not have enough bits to scale the significant to a small enough value. E.g. when the exponent is seven bits, thus can scale the significant to $10^{2^6} = 10^{64}$ (not 10^{2^7} because there is a sign bit), and the domain is from $1 * 10^{-70}$ to $2 * 10^{-70}$. The latter is very unlikely, because when working with very small numbers they usually also have a large significant part (i.e. the small numbers usually have a lot of non-zero digits, like 0.1234567 and 0.1234568 instead of 0.00000123 and 0.00000124 which can easily be described as $1.23 * 10^{-6}$ and $1.24 * 10^{-6}$).

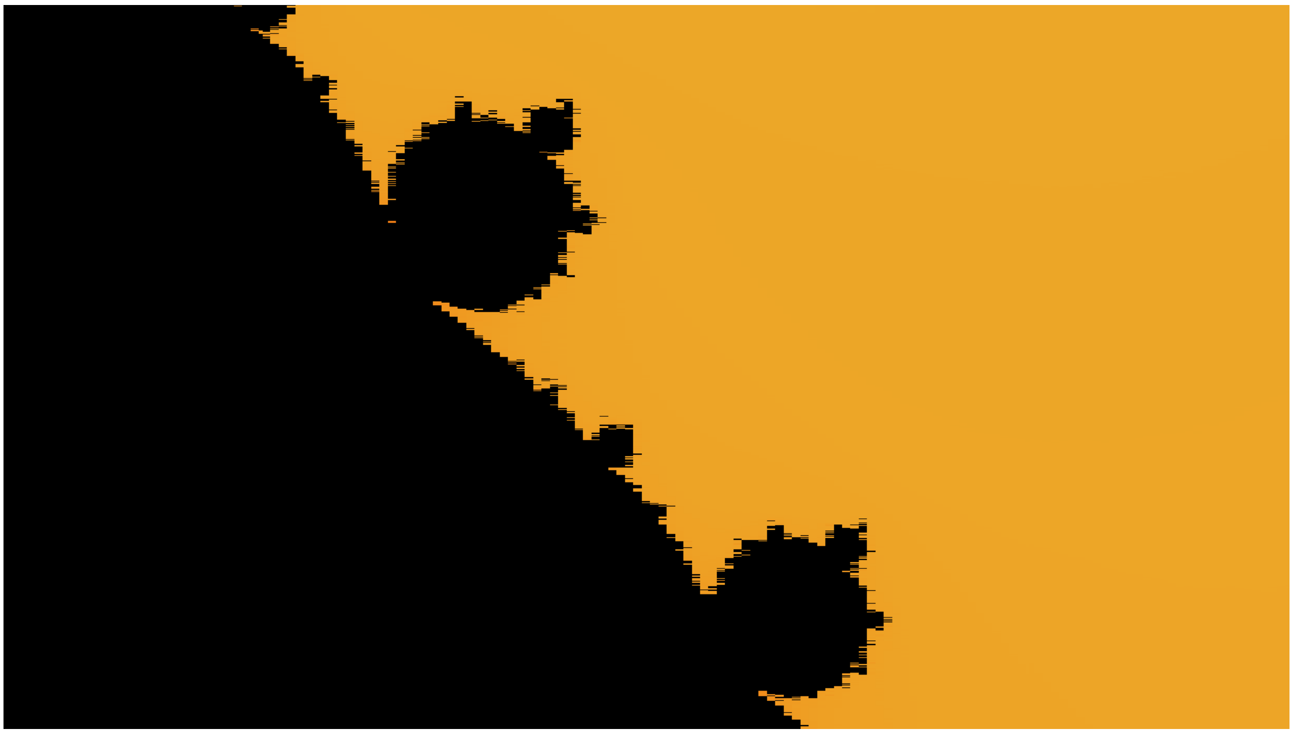


Figure 3.5: Image of the Mandelbrot set below the limit which double precision floats can describe. In Fraccert, use the command "loc limit" to get this image.

If there are not enough significant bits to differentiate two floats when rendering a fractal, some pixels get clumped together, see Figure 3.5. To achieve higher precision floats, GNU Multiple Precision arithmetic library (GMP) is used. This library was chosen, because it is present on most Linux systems (it is a dependency for the GCC compilers and FFmpeg which are present on most Linux systems) and it is one of the fastest. One of the reasons it is the fastest, is because most multiple precision libraries are built on GMP.
Using GMP floats instead of normal floats greatly reduces the speed of arithmetic as will be discussed in Section 5.7.
After zooming in, ΔRe may become smaller than a float can describe. Then, Re_{min} and Re_{max} can become the same value. When this happens, the user is unable to zoom out (but can still set the domain through the console). To prevent this, scaling (and setting the domain) should always check if the current precision can describe the domain. If not, the precision should be increased before calculating the new domain. This check is difficult to implement in a generic manner; independent of the number of bits for the precision and exponent. Another way to solve this problem is always using higher precision floats for the program state than used in calculating the fractal. This ensures that the low precision rendering artifacts always occur before the program state has too little precision to describe its state. When the user sees these artifacts the user can increase the precision used.
It is important to note that only the significant part of a GMP float scales with precision. The exponent is always a constant number of bits [gmp]. Even though this makes GMP's precision limited, the limit is nearly impossible to reach.
In Fraccert, an unsigned integer is used to describe the iteration count. On the platform on which Fraccert is developed, `UINT_MAX` is 4 294 967 295 ($= 2^{32} - 1$). Such a high value of `NMAX` is only necessary when the domain is much smaller than normal floats can describe. When rendering such a domain with normal floats

it would already take a very long time, so with GMP floats it would take an unreasonable amount of time. If an unsigned integer is not big enough, it can easily be changed to a unsigned long or unsigned long long by changing the typedef for `iter_t` in `fractal.h`. On the platform on which Fraccert is developed, `ULLONG_MAX` is 18 446 744 073 709 551 615 ($= 2^{64} - 1$). Calculating the set using such high `NMAX` and GMP floats on current computers can easily take years (if not decades) to compute. If, for some reason, this value is not high enough, the program can be changed to support GMP integers.

3.5 Julia window

When viewing the Julia sets, a second smaller window pops up with the Mandelbrot set, see Figure 3.6. This window helps exploring the resemblance between the two sets, as described in Section 2.4. This window has the same functionality as the main window and uses all the implemented optimizations. The complex parameter c used for the Julia set can be set by clicking in this window. It is also possible to drag the mouse while holding it down. This will only render the Julia set for the pixel under the cursor after the previous set has rendered. When holding 'ctrl' while dragging, every Julia set is rendered for every pixel that is dragged over. This creates a smooth animation where the Julia sets morph into each other.

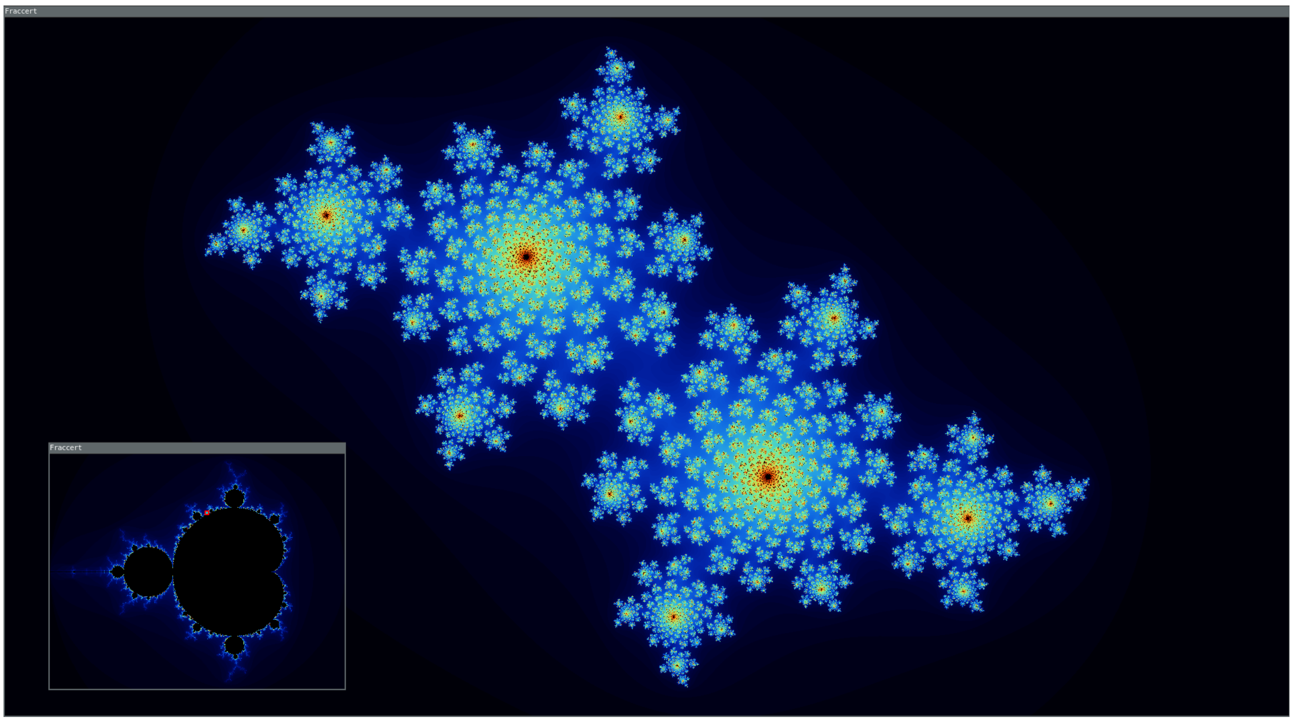


Figure 3.6: Julia window which pops up when viewing the Julia set.

3.6 Orbit plotting

Orbits are an important tool in complex dynamics. They are often used to reason about the behavior of iterations. An orbit is the path z_n travels when iterating. Since this behavior emerges from subsequent iterations, it is called dynamic behavior. The behavior of an orbit is chaotic, however, there seem to be patterns in this chaos. These patterns can be described with orbits very well. For a complex number c, the complex numbers in the neighborhood of c show similar dynamic behavior. Also, the dynamical behavior of some complex numbers can be inferred from the behavior of other complex numbers. To view the orbit of a pixel in Fraccert, right click the pixel. See Figure 3.7 for an example.

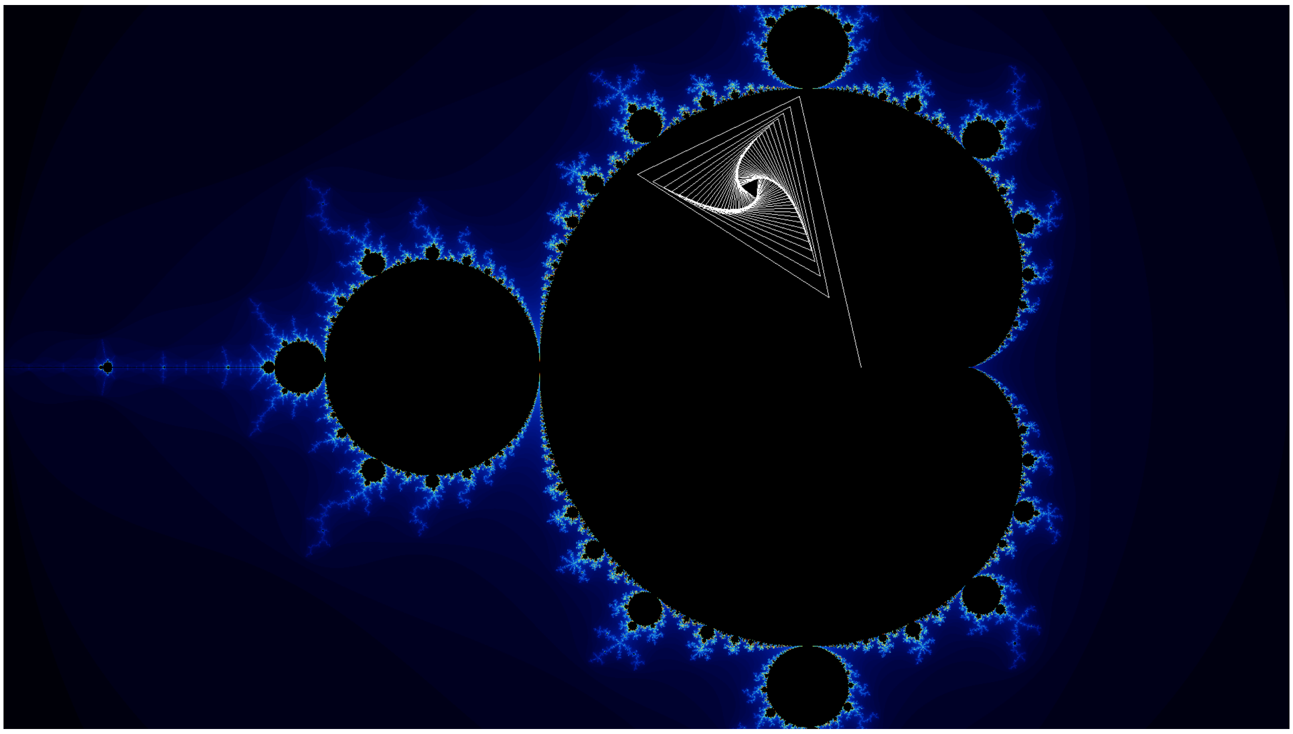


Figure 3.7: An example of an orbit.

3.7 Design choices and program architecture

The code of Fraccert was made with modularity in mind. Not only does it help maintaining the application, but it also makes implementing new features very easy, which is important, because Fraccert is mainly developed as a fractal research tool as mentioned in Section 1.2. The modular architecture helps understanding the code-base, which allows others to easily expand upon it.
Fraccert can be seen as two separate components; the front-end and back-end. The front-end handles all user interaction using a model-view-controller design pattern. The front-end consists of handling user input (controller), keeping track of the program state (model) and drawing pixels to a screen based on this state (view). The back-end is used through a 'fractal' interface. The only task of the back-end is calculating certain values for the front-end. This could be the color or the escape iteration/distance estimation for all pixels. This interface is a base class which specific fractals can inherit. Because of this interface, it is easy to add other fractals to Fraccert which can also utilize all optimizations used in the front-end.

The back-end is compiled as a library called Fracfast. It is compiled as a static library by default (for efficiency), but it can also be built as a shared library by changing a setting in the Makefile. This library file (along with the headers) can easily be used in other projects. Building the back-end as a separate library forces the back-end to be independent from the front-end. The back-end has no state; it uses the functional design paradigm. This means that Fractal class and all its child classes should have no non-const data members. Because of this, every function in the back-end is modular and has no side effects on subsequent calls to the back-end. This ensures the back-end cannot be in an unstable/corrupted state. Another way to put it, is that Fracfast is a math library. It only performs certain calculations and does not offer any interactivity.

To make the back-end as versatile as possible, it returns an array `uint32_t pixels[width*height]` with RGBA values for every pixel which has to be deleted (freed) by the receiver. This can easily be interpreted by different front-ends to produce efficient graphics. The back-end is called with at least three parameters:

1. **Domain**: The complex domain defined by $[\mathrm{Re}_{min}, \mathrm{Re}_{max}]$ and $[\mathrm{Im}_{max}, \mathrm{Im}_{min}]$.
2. **Resolution**: The number of horizontal and vertical pixels the back-end should return. The domain is rasterized according to the resolution.
3. **Data**: All required data for the back-end is sent through this void pointer. The back-end can also return extra data through this pointer.
4. **Range** *optional*: The range defines which pixels of the resolution should be calculated using x_{min}, x_{max}, y_{min} and y_{max}. This is useful when only a part of the screen needs to be rendered. When it is not provided, the full resolution will be rendered.

Passing the range to the back-end instead of calculating the smaller domain and resolution for the corresponding range in the front-end helps preventing rounding errors which would result in rendering artifacts in the resulting frame. This also makes partial rendering of a fractal easy, which helps implementing features such as multi-threaded rendering (Section 4.1) or rendering on a distributed system (e.g. multiple CPUs/GPUs or multiple computers connected by a network).

Examples of parameters which are sent as part of the data parameter are `NMAX`, `CLOSEENOUGH` and the shapes for shape checking (Section 4.4). It can also contain information about what optimizations should be used. An example of data the back-end may send back is the escape iteration for each pixel, which is useful for the deepen optimization discussed in Section 7.5.

The front-end always uses GMP floats. It would be more efficient to use normal floats until they are not precise enough anymore, however, the overhead for GMP floats in the front-end is very small. All front-end calculations are still very fast compared to the calculations the back-end has to perform. By always using GMP floats, the design of the front-end is simplified significantly. For results about the speed-down, see Section 5.7.

To implement new optimizations or fractals, it is important to be familiar with the flow of the program. See Figure 3.8 for visualization of the front-end.

The front-end only responds to user input, so we start with an input from the window or console. This input is parsed by `IOController` or `Console` respectively (the controller), which in turn calls the corresponding member function of `Program` (the model). Since input can come from two different threads, race conditions can occur. This is why all public member functions of `Program` need to lock a mutex to control this race.

Make sure when calling any public member function of `Program` from a member function that the lock is not requested twice, otherwise a deadlock will occur. `Program` will work out what fractal should be rendered next for which domain according to the user input by updating its state. Then, `Graphics` (view) is called to render the frame. All front-end rendering optimizations are performed here. `Graphics` works out how to call the back-end based on a snapshot of the current state provided by `Program` and draws the output to the screen. `Graphics` also saves the last rendered frame which can be used for optimizations.

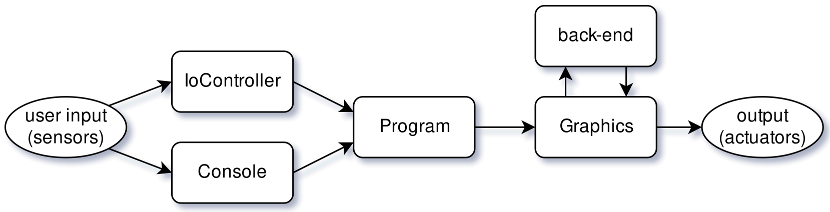


Figure 3.8: Flow of user input through Fraccert.

The main functionality of the back-end is rendering a frame (i.e. filling a pixelbuffer with colors). This can be done by calling `render`() on a `Fractal` object. This function will determine how the pixelbuffer should be calculated. It handles multi-threading (Section 4.5.2) and calls a `calcScreen`() function. The different `calcScreen`() functions provide different methods of rendering a screen. The methods can differ in the result they produce (e.g. different algorithms like escape time or distance estimation coloring, different coloring schemes like normalized or smooth coloring or normal/GMP floats) or differ in which optimizations are used. The `calcScreen`() functions determine what pixels should be calculated in which order and calls a `calcPixel`() function to calculate the measure which will be used to color the pixel. This measure is sent to a `calcColor`() function to get the color of the pixel.

Chapter 4

Optimizations

The implementation as discussed in Chapter 3 performs inadequately. As mentioned before, even though modern GPUs will render images of the Mandelbrot set nearly instantaneously, problems arise when higher precision is needed. This only makes the GPU efficient on relatively low scale factors. Also, branching is inefficient on GPUs, so some kinds of optimizations might make GPU rendering less efficient. However, for high precision CPU work, these optimizations are still very valuable.
There are a number of small optimizations which can be done with the implementation in Algorithm 1. Modern compilers implement some automatically. For instance, we could cache the square of the real and imaginary part of z, since it is used more than once. But study of the generated binary from the compiler shows this is done automatically. Also, as noted before, the tests $\sqrt{\Re(z)^2 + \Im(z)^2} > 2$ and $\Re(z)^2 + \Im(z)^2 > 4$ are equivalent. However, the latter saves an expensive square root operation.
Many optimizations described in this chapter may produce small errors. Note that most of these errors might not be a problem, because most of the rendered fractal is still accurate. When the user is exploring the fractal, speed is of high importance. Only if the user wants an accurate render, these errors become a problem. This is why all error introducing optimizations can be turned off in Fraccert.
In this chapter, we first look at optimizations which can be realized through parallelization. Then, we look at optimizations of features described in Chapter 3. Finally, we look at optimizations which rely on characteristics of the Mandelbrot and Julia sets.

4.1 Parallelization

The computation of the color of a pixel is independent from other pixels. In other words, the color of a pixel is only dependent on itself. This allows us to calculate any number of pixels in parallel. The reason GPUs can calculate the set quickly is because they can efficiently do single precision floating point operations in parallel. A GPU could calculate every pixel at the same time. However, as discussed before, a GPU has its limitations. Therefore, in the remainder of this section we will only consider CPUs.
There are two ways of parallelization on CPUs. The first is using the multiple cores on a system concurrently.

The second is by using SIMD instructions, which is described in Section 7.2. SIMD is an abbreviation for 'single instruction, multiple data', which is a very broad term to describe a class of parallel computers. There also is an extension to the SIMD instructions called fused multiplication addition (FMA). The instructions from this extension allow an addition and multiplication to be performed in one instruction. Technically, this is not SIMD, because it works on a single piece of data. But it could be considered a form of parallelization, because it performs two operations at once.
It is possible to use multiple CPU cores at the same time with multi-threading. A simple approach to split the workload is to divide the screen in n blocks, where n is the number of cores on the system. After a block has been calculated, it is written to a shared pixel buffer. When all blocks have been calculated, the shared pixel buffer can be returned from the back-end. A problem with this method is that some blocks require more work to be calculated than others when more pixels reach higher iteration counts. The calculation time bottlenecks on the slowest block.
A thread per pixel is also not optimal, because there is a constant overhead per `render()` call in the back-end and thread management also introduces overhead. It will also create many cache misses because the pixels may not be calculated in memory order. A good way to split the work over multiple threads is by assigning each thread a line (or line segment/multiple lines) of pixels to render, because the pixels are stored in memory as lines. When the number of line(segment)s is sufficiently higher than the number of threads, we circumvent the previously described bottleneck problem. As long as there are line(segment)s available, a thread that has finished calculating its current line(segment) will simply be assigned a new one.
The implementation of multi-threading is discussed in Section 4.5, because the optimization in this section requires a specific threading scheme.

4.2 Pixel reuse

When translating and scaling, many pixels on the current screen will be the same as pixels on the next screen and may be reused. The current frame has to be saved in order to implement this optimization, along with information about the frame to implement it efficiently. Fraccert only saves the domain of the last frame along with the colors of the pixels.
Reusing pixels while translating is very straightforward to implement, however, there is a problem which is easily overlooked. If the translation is not exactly n pixels (a whole number), there will be a difference in distance between two pixels on the border of the new and recycled pixels. It is almost impossible to guarantee a pixel aligned translation due to how floats work (rounding errors in float math), however, a close enough approximation can prevent noticeable rendering artifacts. Due to rounding errors, it is also possible for the ΔRe and ΔIm to change during translation. This may result in off-by-one errors when calculating the exact pixels to copy from/to, so out of bound array accesses may occur. If all the invalid accesses are avoided, the result is still correct but might be one pixel off.
Since the implementation is very straightforward, no code for translation will be provided in this thesis. To see an implementation, see the function `extendDraw()` in `graphics.cpp`. Implementing this algorithm comes down to the idea of calculating three areas of pixels. First you have to calculate which pixels are shared by

the two frames and where these pixels end up in the new frame. Then, you have to calculate which part of the new frame is still empty and fill these pixels with their corresponding colors.

4.3 Symmetry

The Mandelbrot set is symmetric around the real axis. If $\text{Im}_{max} > 0$ and $\text{Im}_{min} < 0$, then there are pixels which can be reused through symmetry. When calculating pixels, we take the complex number in the upper left corner of the pixel. When reusing pixels from above the real axis, we reflect these pixels over the real axis. As a result of this reflection, the complex number in the bottom left corner of a pixel is used instead. This could be seen as erroneous behavior, but one could argue it does not matter. The Mandelbrot set is perfectly symmetric, so one could expect the discrete approximation to be perfectly symmetric as well. This argument would make the 'erroneous' version better. Usually some form of anti-aliasing is performed on the resulting image. This would also lead to the use of a different part of the pixel. When this is used, the error introduced by symmetry checking is overshadowed by anti-aliasing. Since it is debatable if this optimization leads to imperfect images of the Mandelbrot set, there is an option to turn it off in the program.
All Julia sets (also Julia sets using other iterative functions than f_c) have symmetries, however, these are difficult to compute [Bea16]. For the Julia sets $J(f_c)$, there is a rotational symmetry around the origin of 180°. This is the same as a reflection over the real axis followed by a reflection over the imaginary axis (order does not matter). Since the reflection is over both axes, both have to be in the screen bounds. Because this barely happens, we decided to focus on other optimizations instead of implementing this for the Julia sets.
The speed-up using symmetry pasting technique is not as trivial as it may seem, see Section 5.2 for more information.
Algorithm 4 describes an efficient CPU implementation of this algorithm with pseudo-code. The function `copyRow(array, src, dest)` copies the row with index `src` to index `dest` of `array`. Here, `array`[j][i] is the same as `array`[$j*$width + i]. `copyRow()` is implemented using `memcpy()` from the standard C library, which is typically a highly optimizes routine to copy data. In lines 5–13, we first check if there is any data to copy. This is only the case if there are lines of pixels above and below the real axis. Then, we have to determine if the biggest part is under or above the real axis. We calculate this part and the other part is constructed by copying the corresponding rows. The idea behind this algorithm is that the lines are copied around a pivot, the real axis. First, the location of the pivot is calculated. Then, the lines on one side of the pivot are copied to the other side of the pivot. The actual implementation in Fraccert uses GPU acceleration, which copies the entire area and flips it as one big block of pixels, see Section 5.2 for results about the difference between CPU and GPU symmetry pasting.

```
input  : $\mathrm{Re}_{min}$, $\mathrm{Re}_{max}$, $\mathrm{Im}_{min}$ and $\mathrm{Im}_{max}$
output : Pointer to pixelbuffer of rendered fractal
screenWidth $\leftarrow$ getWidthPixels();
screenHeight $\leftarrow$ getHeightPixels();
pixels[screenWidth $*$ screenHeight] $\leftarrow 0$;
pixelSize $\leftarrow \Delta\mathrm{Re}/$screenWidth;
// Symmetry checking
$y_{min} \leftarrow 0$;
$y_{max} \leftarrow$ screenHeight;
if $\mathrm{Im}_{min} < 0$ and $\mathrm{Im}_{max} > 0$ then
    if $\mathrm{Im}_{max} + \mathrm{Im}_{min} >= 0$ then   // Most pixels above real axis
        $y_{max} \leftarrow ((\mathrm{Im}_{max} * \text{screenHeight})/(\mathrm{Im}_{max} - \mathrm{Im}_{min})) + 1$ ;  //Set $y_{max}$ to the real axis
    else   // Most pixels below real axis
        $y_{min} \leftarrow ((\mathrm{Im}_{max} * \text{screenHeight})/(\mathrm{Im}_{max} - \mathrm{Im}_{min}))$ ;  //Set $y_{min}$ to the real axis
end
// Normal calculation
for $y \leftarrow y_{min}$ to $y_{max}$ do
    for $x \leftarrow 0$ to screenWidth do
        $c \leftarrow [\mathrm{Re}_{min} + (x * \text{pixelSize}), \mathrm{Im}_{max} - (y * \text{pixelSize})]$;
        pixels[$y *$ screenWidth $+ x$] $\leftarrow$ calculatePixel($c$);
    end
end
// Copy to top half
for $y \leftarrow 0$ to $y_{min}$ do  copyRow(pixels, $y_{min} + y_{min} - y$, $y$) ;
// Copy to bottom half
for $y \leftarrow y_{max}$ to screenHeight do  copyRow(pixels, $y_{max} + y_{max} - y$, $y$) ;
return pixels
```

Algorithm 4: Symmetry checking and pasting in the Mandelbrot set.

4.4 Shape checking

Complex numbers which are part of the Mandelbrot set are the most expensive ones to compute, because for these numbers NMAX will be reached. We can prevent iterating these numbers by using geometric shapes (represented through formulas) to describe parts of the set. When iterating a complex number, we calculate if the complex number is in one of the shapes. Because we check each shape individually, this check takes longer for every shape we add. So when trying to describe the entire set with these shapes, the check will take infinitely long.

There are only two 'perfect' shapes in the Mandelbrot set, see Figure 4.1. These are the cardioid and the disk to the left of the cardioid called the period-2 bulb. They are defined by Equations 4.1 and 4.2, respectively. Here, $z[0] = \Re(z)$ and $z[1] = \Im(z)$.

$$q * (q + (z[0] - 0.25)) < z[1]^2 * 0.25, \text{ where } q = (z[0] - 0.25)^2 + z[1]^2 \tag{4.1}$$

$$z[0]^2 + (2 * z[0]) + 1 + z[1]^2 < 0.0625 \tag{4.2}$$

All other disks are slightly distorted [Vepoo]. Unfortunately, ellipses do not fit them either. It is possible to use the largest disk/ellipse which is contained by a bulb to further optimize the calculation process, but this will increase the overall computation time for only a small reward.

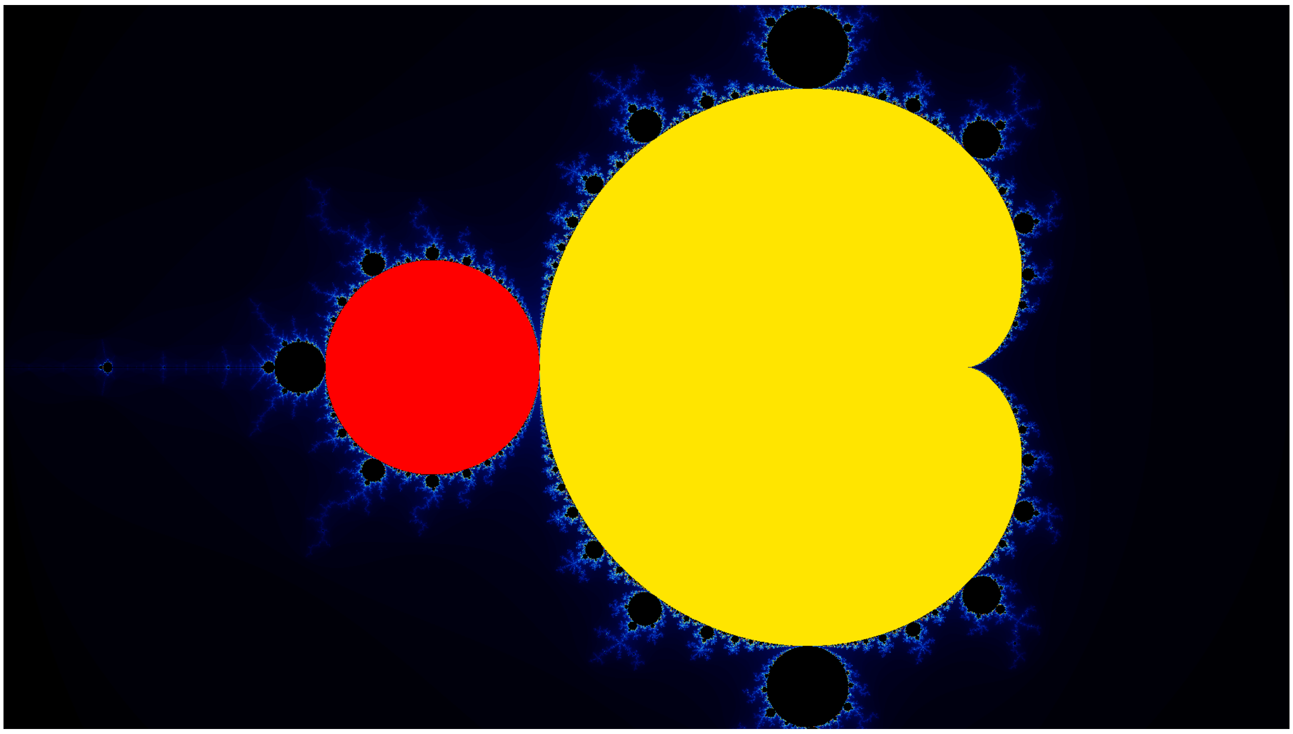


Figure 4.1: Grey shapes visualizing the period-2 bulb (red) and the cardioid (yellow).

4.5 Border tracing

The Mandelbrot set is simply connected [Hub82]. In other words, there is a path from any point in the Mandelbrot set to every other point which only goes through points in the set. Because it is simply connected, there is only one topologically distinct path between two points. This means that if there are multiple paths connecting two points, there is a continuous function which can transform these paths into each other. This implies that if a closed curve can be drawn where all points on the curve are in the set, all points inside the closed curve are in the set. This saves iterating all the points within the closed curve. Note that Julia sets where the complex parameter is in the Mandelbrot set are also connected, so can be border traced as well. If we find the perimeter of a set, we can skip calculating all pixels within the perimeter because we know they are part of the set. Unfortunately, it is only proven that the set itself is simply connected; not the color bands surrounding the set. However, it is reasonable to assume this is also the case for the color bands. Note that the color bands are not simply connected, but dually connected, since a color band always contains one hole, which is the Mandelbrot set (or a color band which escapes one iteration higher). When calculating the Mandelbrot set with $\texttt{NMAX} = 0$ and repeatedly increase `NMAX` by one, you see the pattern shown in Figure 4.2. Here, you can see that every iteration starts a new color band strictly inside the Mandelbrot set for that iteration. It might not look like this in the above images, but if you zoom in sufficiently far, you will see no color bands touching with a difference in escape iteration greater than one. See Figure 4.3 for an example of the third image zoomed in sufficiently far. For $\texttt{NMAX} < 10$, you can easily see that every color band is dually connected.

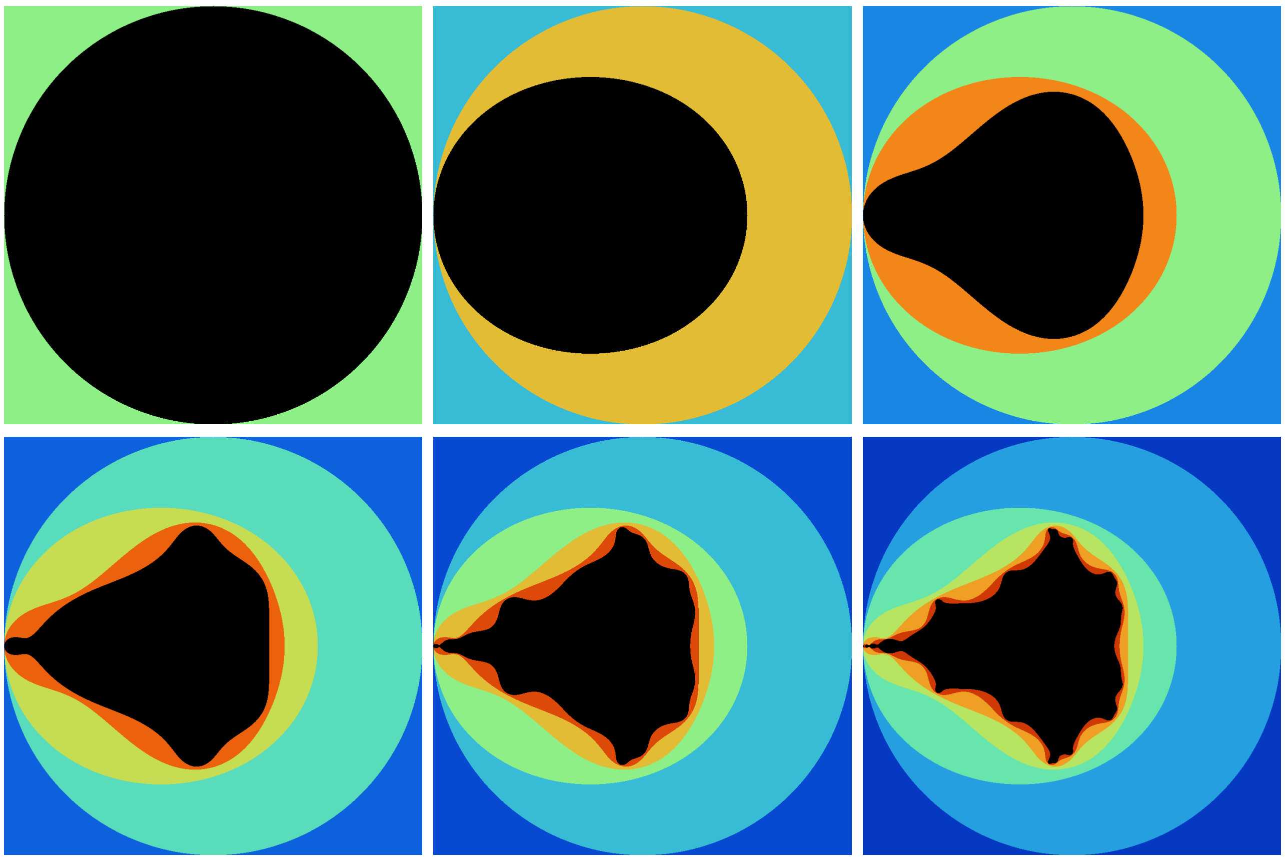


Figure 4.2: Mandelbrot set with NMAX 0 to 5.

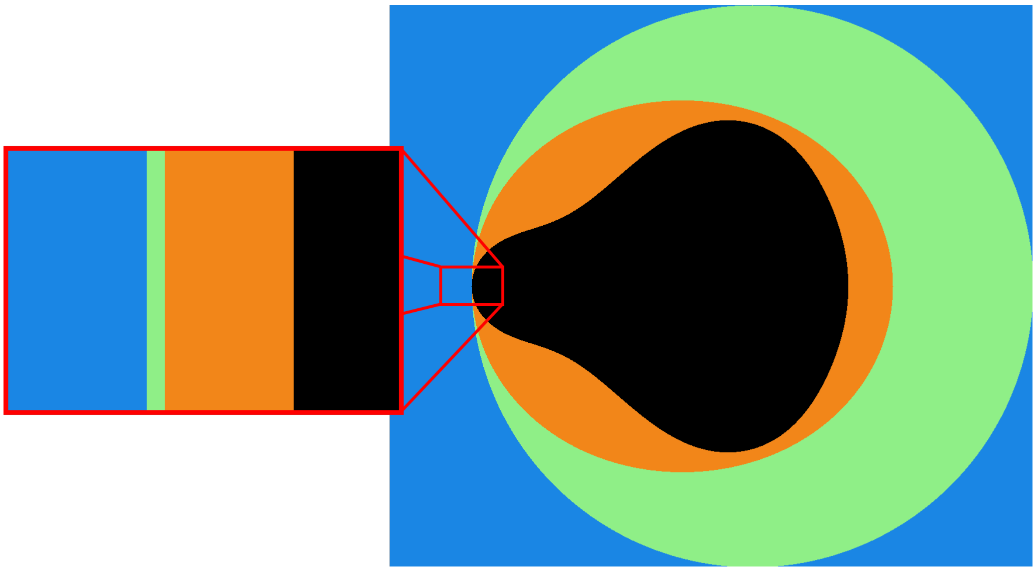


Figure 4.3: Example which shows every color band is simply connected for NMAX $= 2$.

Even though the Mandelbrot fractal shows complex behavior, it always seems to follow certain patterns (i.e. when you zoom in on a pattern, it roughly stays the same no matter how far you zoom in). So it seems reasonable to assume all color bands stay connected for any value of NMAX.

There are many methods to calculate the boundaries of color bands, however, most methods only give a description (usually represented through formulas) of the curve which describes the boundary of a color band. It is computationally expensive to determine which pixels are hit by the curve. Also, it is difficult to

implement zooming with these methods, since you always get a description of the entire curve, not only the part which is on screen. Other currently known methods perform a walk along the border, which is called border tracing.
Even though the Mandelbrot set and some Julia sets (and possibly the color bands) are connected, our discrete approximation might not be. Because of this, the image obtained with border tracing might not be accurate, see Figure 4.4 for a visual representation of a pattern which would cause an issue. Here, if we use the top-left corner of a pixel, then the border tracing algorithm would not color the top ellipse black if we were tracing the bottom black part. It would not see the little black line connecting the two parts because it is in between two pixels. However, these kinds of patterns do not seem to emerge too often. If they were to appear, the generated image would often look even better than a perfect image, because this error works as a sort of noise reduction post-processing in areas with many different colors close together. See Section 5.4 for information about the difference in the images.

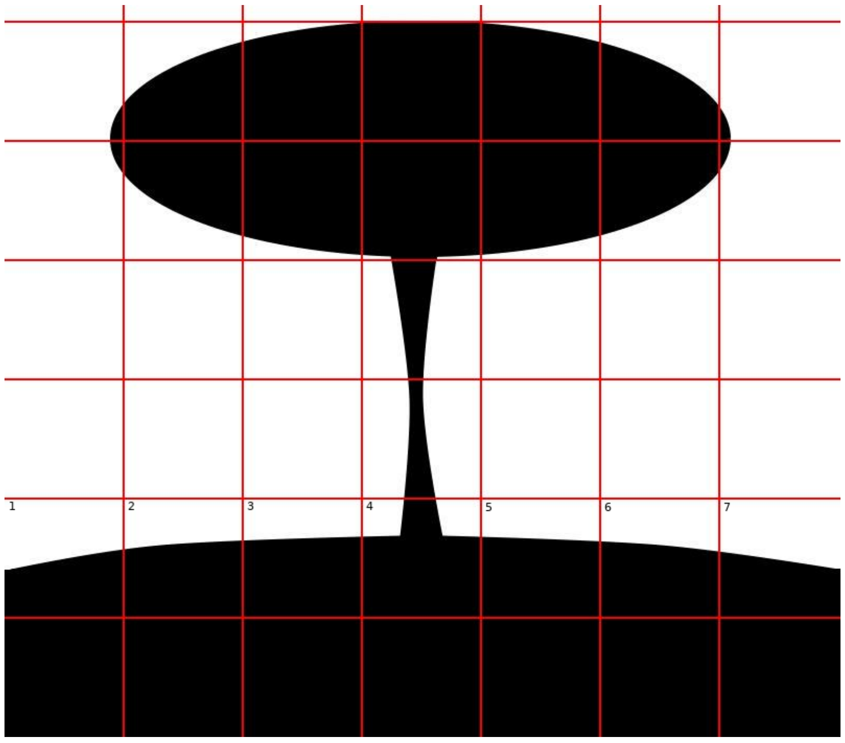


Figure 4.4: An example of a pattern on which border tracing would produce an inaccurate image.

4.5.1 Implementation

There are multiple ways to implement border tracing. We only focus on one method, because as will be discussed in Section 7.2, it allows more optimization in the future. Fraccert finds all pixels that have a neighbor pixel with a different color. It starts by putting the border of the screen in a queue called the `pixelQueue`. This is a queue of pixels for which the check, if the neighbors have a different color, is performed. After the screen border has been added, the main loop starts. The only thing this loop does, is getting the next pixel from `pixelQueue` and check its neighbors until `pixelQueue` is empty. To check if the color of a neighbor is different from the color of the pixel considered, the color of the pixel and its eight neighbors have to be calculated. If one of the neighbors pixels has a different color, this neighbor is added to `pixelQueue`. If

`pixelQueue` is empty, all borders are colored. Now, to color the remaining pixels, we iterate over the pixel buffer and assign to every uncolored pixel the color of the pixel to its left.
The alpha color channel is used to save information about border tracing. The alpha channel is normally used to indicate how opaque a pixel should be. Since we do not use the alpha value in the front-end (we do not use transparency), we do not reset it back to a sensible value for efficiency reasons. If correct alpha values are required, they can be set correctly in the loop which fills all areas inside the borders. We assume all pixels which have not been considered yet have an alpha of 0, so we have to initialize the pixelbuffer with zeros.
For more details, we refer the reader to an excerpt of the Fraccert C++ implementation in Appendix C.

4.5.2 Multi-threading

In Section 4.1, we argued that we should implement multi-threading by splitting the screen in lines. However, to utilize border tracing we need a surface to fill in, so multi-threading border tracing this way will not result in a speed-up (even a speed-down, due to overhead that border tracing introduces over brute-forcing). Border tracing always calculates the border of the screen and can at best save calculating the surface of the screen, so it is important to keep the surface to perimeter ratio as high as possible. There are two factors which determine the surface to perimeter ratio. The first factor is the ratio between the width and height of the shape. A square has the largest surface of any rectangle for a given perimeter, so we should keep the width to height ratio as close to 1 as possible. The second factor is the size of the shape. The larger the rectangle, the higher the surface to perimeter ratio becomes. In short, we want to partition the screen into as few squares as possible. However, as discussed in Section 4.1, we also want more squares than the number of threads to improve opportunities for load balancing. This trade-off is studied in Section 5.5. The optimal number of squares is dependent on the domain which we render. If there are parts of the domain which are much quicker than other parts, it is advantageous to divide the screen in more parts. However, if all blocks roughly take the same time to compute, it is advantageous to only split the screen into n parts.
Not only the number of blocks has to be determined, but the number of threads can also impact the performance. Using more threads than cores can impact the performance in multiple ways. Aside from the overhead of managing threads, using more threads than cores creates excessive context switches and may increase the number of cache misses. However, if a core is idle, then it can be switched to a different thread. Fraccert spends most time in the iteration loop, so a CPU core is barely ever idle. This trade-off is studied in Section 5.5.
The multi-threading scheme is implemented as follows. We start by dividing the screen in blocks. Blocks are expressed using the ranges which are described in Section 3.7. First we determines the number of splits which should be performed. Splitting is dividing a block in two equal sized blocks of which the width to height ratio is closest to 1. This comes down to splitting along the axis which has the most pixels. The number of blocks b can be calculated from the number of splits s as $b = 2^s$, because every split doubles the number of blocks. Since we want more blocks than threads t, we perform a minimum of $s = \lceil \log_2(t) \rceil$ splits. The ranges determined by these splits are put in a queue. Every idle thread tries to get a block from the queue until there are no blocks left. A mutex has to be used to control any race conditions that might occur while getting a block from the queue. After a block is calculated, its pixels are copied to a shared pixel buffer which is returned when all blocks are finished calculating. Note that for performance reasons we do not use a queue

in the actual implementation. We instead use an integer to the next block in an array.
In theory, it is possible to create a threading algorithm which will split blocks into smaller blocks when there is a thread free. However, it is very difficult to get consistent performance gain from this method. For consistent speed-up, we would need a measure (heuristic) of progress of a block, so we could subdivide the block which has the most work to be done left. Keeping track of or calculating this information adds much overhead and finding a good heuristic is a difficult task. Even with this information available, it is difficult to determine if subdividing will actually speed rendering up, because stopping the calculations of a block and dividing it over more threads causes overhead.

Chapter 5

Experiments and discussion

Running experiments on the Mandelbrot and Julia sets is not trivial. Some features and optimizations have different effects on different domains and values for `NMAX`. To compensate for this fact, we run tests on two domains. The first is the default Fraccert domain with a very high resolution. We will refer to this domain as 'home'. The second is the sum of nine arbitrarily chosen domains with a small $\Delta\mathrm{Re}$ and $\Delta\mathrm{Im}$ with resolutions close to screen resolutions. We will refer to this sum as 'average domain', because it reflects the speed-up you might expect on average while exploring. For a list of all used domains with corresponding resolutions and `NMAX` with pictures, see Appendix A. When testing an optimization, all other optimizations are disabled to reflect the effect of the optimization compared to brute forcing. In all experiments, we measure the time it takes for the back-end to return the pixels. All are performed using double precision floats or GMP floats where stated. Fraccert is compiled with GCC's g++ (9.1.0) with the `-O3` and `-s` flags. All tests were forced on 1 CPU with 'taskset' (except the multi-threading tests) and were given a niceness of -20 (highest priority). The niceness should not matter, because the computer did not have any other processes or services/deamons running (the only other process that was 'running' was the shell from which the tests were started, which is put to a sleep process state). All the listed results are the average of 25 tests. Before running the tests, we run our brute force algorithm on the home domain to try to get the CPU frequency up to the frequency which will be used during the other 25 tests. Also, we prevent the CPU from going to idle (core parking) by disabling it with 'cpupower'. The tests were run on an Intel Core i7 6700HQ.
In this chapter, we first look at the optimizations symmetry checking, shape checking and border tracing. Then we look at the speed-up of multi-threading. Subsequently, we check how much GMP floats effect calculation speeds. Finally, we look at the speed-up of all optimizations combined.

5.1 Brute force

We need a reference to compare our optimizations to. We use the brute force variant, because this is the most straightforward method of computing the set. It is also the least efficient method. The results are listed in Table 5.1. When we list the standard deviation in the rest of this chapter, it is always the standard deviation of the results with the optimization, since the standard deviation of the brute force results can be found here. To quickly recapitulate, the standard error σ notes that 68.27% of the results were within $\pm 1\sigma$, 95.45% within $\pm 2\sigma$ and 99.73% within $\pm 3\sigma$. We abbreviate standard deviation as 'stdev' in all tables.

domain	time	stdev
home	43.130 s	0.0011
average	19.858 s	0.0009

Table 5.1: Calculation speed with brute force.

5.2 Symmetry

The upper bound for speed-up which can be realized through symmetry pasting is 2x, because at best half of the calculations can be skipped. This does not take the overhead of checking for symmetry and pasting pixels into consideration. When rendering the Mandelbrot set with the real axis vertically centered, the difference between the theoretical speed-up and the measured speed-up is the overhead of symmetry checking and pasting. See Table 5.2 for the results. The listed standard deviation for the theoretical speed-up is from the results without symmetry pasting. The other two are from the results with their respective symmetry pasting variant. Symmetry pasting is done in the front-end, so we do not only measure the back-end speed, but also the overhead of symmetry checking and pasting in the front-end. We use a slightly altered home domain, because in this domain the overhead of symmetry pasting and checking is much larger compared to calculating the set which makes interpreting the results easier. Here, $\text{Re} = [-2, 2]$ and $\text{Im} = [-1.5, 1.5]$ with a 1600x1200 resolution and $\texttt{NMAX} = 256$. The average domain is not considered, because the domains in the average domain cannot utilize symmetry pasting. The CPU implementation is as described in Algorithm 4 in Section 4.3. The GPU implementation uses SDL's textures, which is an abstraction provided by SDL to easily use VRAM (GPU memory). While in VRAM, the GPU can efficiently perform operations like rotation, translation, mirroring (flipping), scaling and sheering. Instead of copying the pixels row by row like our CPU implementation, the GPU can work on the entire texture at once, which is more efficient. However, the pixels have to be copied to the VRAM first.

	without	with	speed-up	stdev
theoretical	294.42 ms	147.21 ms	2.000x	0.7056
CPU	294.42 ms	153.88 ms	1.913x	0.8031
GPU	294.42 ms	147.57 ms	1.995x	0.9242

Table 5.2: Calculation speed with and without symmetry pasting.

The parameters on which the test was performed are very similar to the parameters Fraccert starts with. The default resolution is lower, however, we think most user will use Fraccert full screen which would be closer to the used resolution. Higher resolutions increase the amount of work which can be omitted, but also increases the amount of data to copy. So the speed-up on higher resolutions is dependent on the number of iterations needed to render the frame. When higher values of `NMAX` are used, the user is usually not rendering parts which have the real-axis in screen, so no extra benefit will be gained.

Note that the speed-up of using a GPU for symmetry copying on the Julia sets might be higher, because the copied pixel buffer also has to be reversed. On the CPU, reversing an array is expensive. On the GPU, flipping a texture vertically is as fast as flipping it vertically and horizontally.

5.3 Shape checking

We first look at the speed-up realized through shape checking, see Table 5.3 for the results. Note that in the experiments we only check for the cardioid and period-2 bulb.

domain	without	with	speed-up	stdev
home	43.130 s	5.817 s	7.41x	0.0006
average	19.858 s	19.641 s	1.011x	0.0010

Table 5.3: Calculation speed with and without shape checking.

The speed-up on the home domain is very high. Most pixels on this domain only require few iterations to diverge and most pixels which do not diverge in `NMAX` iterations are in one of the shapes. Because of this, it poorly reflects speed-up when exploring deeper zoom levels. However, most of the time, the user starts from the home domain to find interesting parts. If the user chooses to explore near defined shapes, the speed-up becomes very high because points in the set are more expensive to compute when `NMAX` is increased.
The results of the average domain show little speed-up. Without average domains e and f, it is slightly below 1. This is because average domains e and f are the only domains which contain pixels within one of the shapes. On domains e and f, the speed-up is about 1.32x. The speed-up on all the other domains combined is 0.98x (1.02x speed-down).
Our implementation of shape checking is not optimal. If we have descriptions of many shapes, so most points in the set are described, we could skip iterating many more points. However, doing the check would take very long, because checking for any extra shape adds computation time per pixel. If the shape is out of screen bounds, then no pixel will be in this shape. If we calculate which shapes are within the screen bounds, we can limit our check to only these shapes. This increases the number of computations per screen, but will save much branching per pixel, so it moves computation away from the inner-loop. There are very efficient algorithms which can figure out which shapes are within screen bounds, because they are important for hitbox detection (checking if two or more bodies collide) which is done a lot in video games. To facilitate this, the back-end expects a list of shapes which to check for.

As with symmetry pasting, the speed-up of shape checking is not trivial. In this case, we also have to consider the time it takes to calculate if a pixel is in a shape. To test how well shape checking performs, we compare it to a theoretical speed-up. The theoretical speed-up is based on the number of iterations skipped by using shape checking, see Table 5.4 for the results. The 'measured' result is from Table 5.3. There are 11 017 252 551 iterations needed to calculate the home domain. Only 1 243 399 551 are done with per pixel shape checking. 9 773 853 of 48 000 000 pixels, which is 20.4%, are within a shape. The theoretical speedup is $\frac{11017252551}{1243399551} = 8.86$.

	without	with	speed-up
theoretical	43.130 s	4.868 s	8.86x
measured	43.130 s	5.817 s	7.41x

Table 5.4: Calculation speed of shape checking compared to theoretical speed-up.

About 1 second is spend on performing the calculations. This is very little time, especially considering that we check the equations for both shapes 48 000 000 times. When rendering a Full HD ($1920x1080$) image, we only perform these equations 2 073 600 times (23 times less). So the overhead of checking the two shapes in domains which do not contain them is very small. However, other shapes, except the one mentioned in Section 7.1, require more operations to compute. As a consequence, the overhead of checking for these shapes when they are not within screen bounds may be much higher.

5.4 Border tracing

We first check if the border tracing produces correct images. If the images are too distorted, the result is incorrect and its speed becomes irrelevant. To test the difference in images, the Mandelbrot set is computed with and without border tracing. Then, we count the number of pixels which do not have the same color (discarding the alpha channel). The results are listed in Table 5.5.

domain	#incorrect	#total	ratio	
home	64	48 000 000	$1.33 * 10^{-6}$	(approximately 1 in 750 000)
average	62	18 662 400	$3.32 * 10^{-6}$	(approximately 1 in 300 000)

Table 5.5: Difference in images with and without border tracing.

The error border tracing introduces is very minimal. There are 54 incorrect pixels are in average domain f and 8 in e, so most domains are fully correct. Since the differences are so small, it is very difficult to notice it in side by side pictures. To see the difference, use Fraccert and turn border tracing on and off. The error might even be favorable in many cases. Errors usually occur in areas with much variation in a small region of pixels. In these areas, the patterns in the chaotic behavior cannot be distinguished anymore because of the limited number of pixels to describe these areas. Especially when using coloring algorithms with strong banding, these areas will look like noise (like tv static) as can be seen in Figure 3.1 of Section 3.1.2. Border tracing fills some of these noisy areas in with one color. The error border trace introduces cannot be seen without comparing it to a brute forced image, because the patterns in the Mandelbrot set emerge from the behavior of the borders of the color bands (called lemniscates) and the borders are unaffected (only the color of the area in the borders). Because of this, the error works somewhat as a noise reduction technique which is applied in post-processing by other fractal viewers.

Next, we take a look at the speed-up realized though border tracing, see Table 5.6 for the results.

domain	without	with	speed-up	stdev
home	43.130 s	1.435 s	30.06x	0.0006
average	19.858 s	12.955 s	1.53x	0.0006

Table 5.6: Calculation speed with and without border tracing.

The speed-up of border tracing is very high on the home domain. Not only are there many points in the set of which iteration can be skipped. Most parts of the set which are skipped have circular borders. Circles have the lowest circumference to area ratio and only the circumference has to be calculated, so these shapes are optimal for border tracing. The results for the average domain may reflect the speed-up that users experience better.
The speed-up of border tracing is strongly dependent on the value of `NMAX` for a given domain. If `NMAX` is very low for the zoom level, there is little detail so every color band has a large surface. When `NMAX` is very high for the zoom level, some details may be skipped. The extra details on higher values for `NMAX` increase the number of color bands, which in turn increases the number of pixels on a border. The added detail will usually lower the circumference to area ratio, see Figure 5.1, which also impacts the performance negatively.

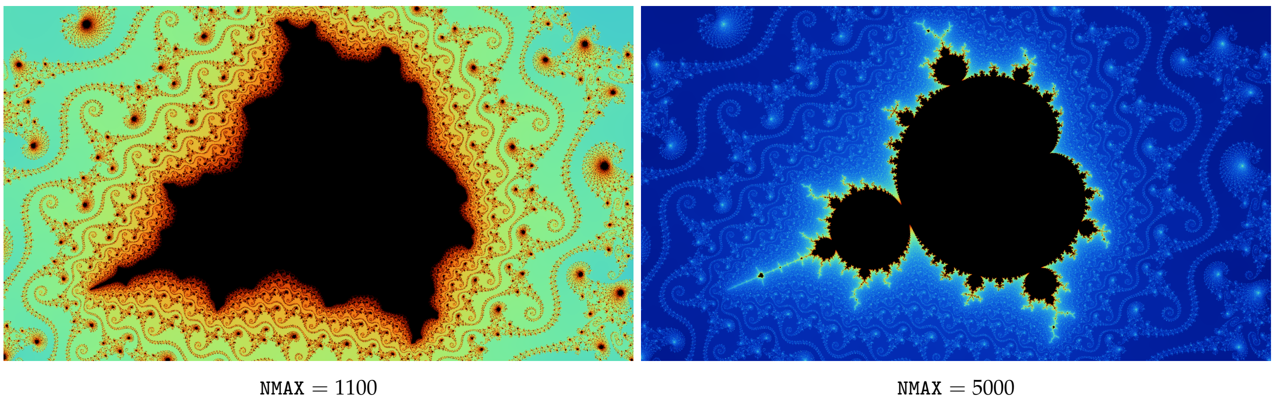


Figure 5.1: Difference in circumference to area ratio for difference choices of `NMAX`.

The current implementation of border tracing is not optimal with respect to the number of pixels that are actually calculated, because it will create four wide borders (two pixels for both colors). When tracing a border, every pixels checks all eight neighbors. By keeping track which direction the border is relative to the pixel, we can skip checking some neighbors. This will also prevent making four wide borders. However, our method of border tracing allows much easier implementation of SIMD instructions which is described in Section 7.2. Also, the four wide borders prevent some errors, because it forces more pixels to be calculated.

5.5 Multi-threading

The speed-up from multi-threading is very dependent on the domain. For instance, when testing the speed-up of the home domain with few blocks, the results seem very poor. This is because blocks close to the border of the screen finish very fast and blocks in the center take much longer, so the entire process bottlenecks on the blocks in the center. However, when zooming in, the workload of each block tends to average out. This is comparable to splitting the screen in more blocks, except that using more blocks adds a small overhead. However, this overhead is very small compared to the time it takes to render a frame.

All tests in this section were performed on a CPU with four cores which are reported as eight cores due to hyper-threading. The results may differ on non-hyper-threading cores.

As discussed in Section 4.5.2, our choice of both the number of threads and splits may affect the rendering time. Since the optimal number of threads should be trivial (the number of cores reported by OpenMP), we first test the optimal number of threads. We still test the optimal number of threads, because hyper-threaded cores do not behave the same as two cores. Some algorithms can benefit from having more threads than cores. This likely is not the case with border tracing, but it is still wise to check. The results of this test can be found in Table 5.7 and Figure 5.2. In the table, the diminishing returns of too many threads can be seen well. In the graph, you can see the behavior of increasing the number of threads. We split the results into two graphs, because of the huge difference between the results in both graphs which can be seen by the different *y*-axis scaling. We chose 8 splits, which corresponds to $2^8 = 256 = 16 * 16$ blocks. The standard deviation is so small, that the error bar which represents it becomes a dot in the graph.

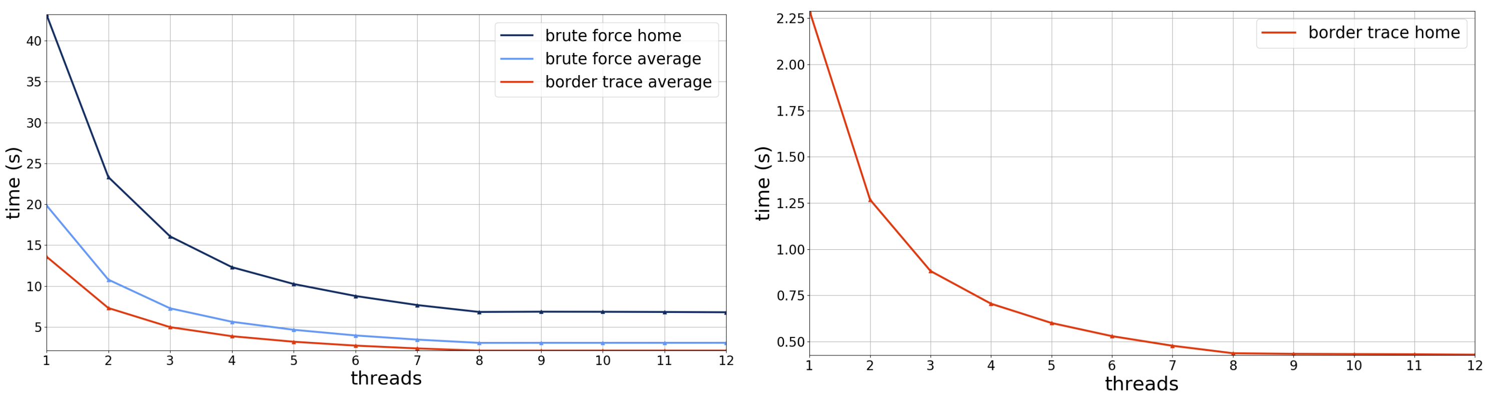


Figure 5.2: Calculation speeds for different number of threads.

threads	6	7	8	9	10	11	12	13	14	15	16
brute force home	8.783 s	7.675 s	6.834 s	6.865 s	6.855 s	6.831 s	6.796 s	6.804 s	6.796 s	6.803 s	6.798 s
stdev	0.0129	0.0235	0.0333	0.0385	0.0467	0.0392	0.0264	0.0351	0.0405	0.0283	0.0252
brute force average	3.962 s	3.452 s	3.054 s	3.060 s	3.060 s	3.062 s	3.064 s	3.064 s	3.065 s	3.063 s	3.062 s
stdev	0.0026	0.0025	0.0007	0.0049	0.0068	0.0078	0.0058	0.0053	0.0053	0.0062	0.0055
border trace home	0.529 s	0.477 s	0.436 s	0.433 s	0.432 s	0.431 s	0.429 s	0.427 s	0.427 s	0.428 s	0.428 s
stdev	0.0021	0.0008	0.0010	0.0030	0.0037	0.0036	0.0031	0.0021	0.0021	0.0039	0.0037
border trace average	2.719 s	2.374 s	2.110 s	2.110 s	2.112 s	2.112 s	2.111 s	2.112 s	2.113 s	2.111 s	2.111 s
stdev	0.0011	0.0015	0.0005	0.0035	0.0038	0.0039	0.0035	0.0029	0.0026	0.0040	0.0036

Table 5.7: Calculation speeds for different number of threads.

It seems the number of cores reported by OpenMP (eight, including hyper-threading cores) is the optimal choice. Note that we ran these experiments on a minimal system; the results may differ on a system which has to context switch constantly to facilitate other threads.

Next, we check what number of splits is optimal, see Table 5.8 and Figure 5.3 for the results. We use 8 threads because it is the best choice for most measurements according to Table 5.7 and OpenMP reports 8 cores in the used CPU. Note that as discussed before, the results on the home domain are very specific to this domain and should not be used for setting the number of splits used for any arbitrary domain. Again, the standard deviation is so small, that they become dots in the graph.

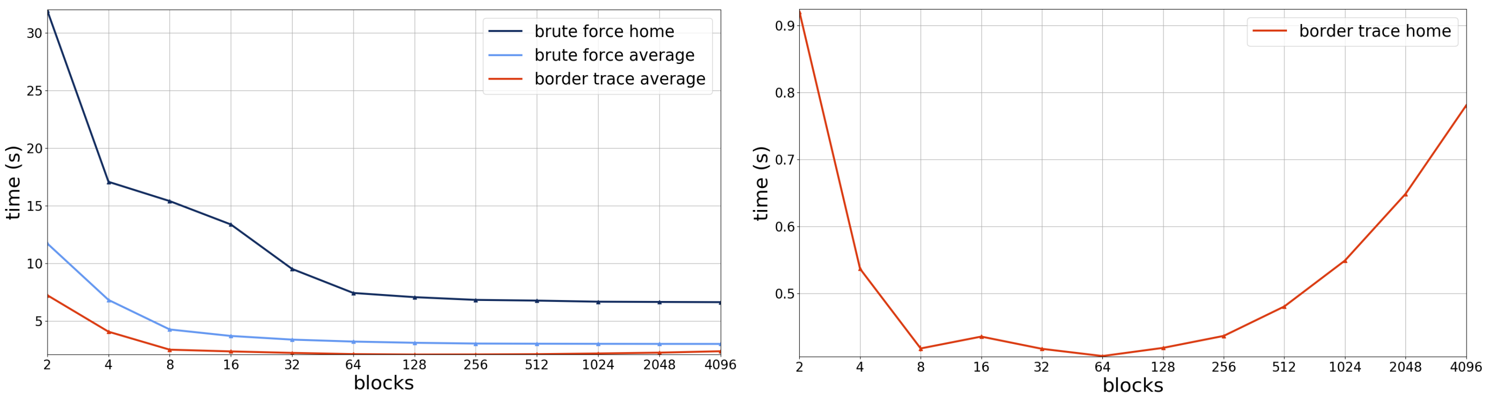


Figure 5.3: Calculation speeds for different number of splits.

splits	6	7	8	9	10	11	12	13	14	15	16
blocks	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
brute force home	7.441 s	7.078 s	6.841 s	6.780 s	6.684 s	6.662 s	6.644 s	6.640 s	6.640 s	6.639 s	6.639 s
stdev	0.0078	0.0037	0.0055	0.0012	0.0027	0.0008	0.0009	0.0002	0.0002	0.0001	0.0001
brute force average	3.226 s	3.120 s	3.054 s	3.039 s	3.030 s	3.023 s	3.021 s	3.021 s	3.022 s	3.025 s	3.031 s
stdev	0.0024	0.0009	0.0006	0.0003	0.0002	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000
border trace home	0.407 s	0.419 s	0.437 s	0.481 s	0.549 s	0.649 s	0.781 s	0.984 s	1.243 s	1.635 s	2.121 s
stdev	0.0010	0.0014	0.0010	0.0005	0.0002	0.0002	0.0001	0.0001	0.0001	0.0001	0.0001
border trace average	2.138 s	2.103 s	2.110 s	2.134 s	2.192 s	2.264 s	2.386 s	2.518 s	2.730 s	2.916 s	3.194 s
stdev	0.0013	0.0006	0.0005	0.0002	0.0001	0.0001	0.0000	0.0001	0.0000	0.0001	0.0000

Table 5.8: Calculation speeds for different number of splits.

The optimal number of splits is very difficult to determine. Not only does it depend on the domain, resolution and `NMAX`, but also on the number of cores available. On the system which performed the tests, 7 splits seemed to be optimal for the average domain with border tracing (which will be used in most settings). If a computer has more than $2^7 = 128$ cores, a higher number of splits might be faster. Note that 4 splits on the home domain with border tracing performs very poorly. This is probably due to bottlenecking on the last blocks. Also, using more than 6 splits quickly reduces the speed-up. This is because border tracing performs very well on this domain. By making more splits, we create more borders which have to be calculated which could normally be skipped.

Finally, we test the total speed-up, see Table 5.9 for the results. We use 8 threads because it is the number of cores reported by OpenMP and 7 splits because it seems the optimal choice according to Table 5.8. Note that on an aspect ratio of 16:9, the block will roughly be square for an uneven number splits. For an aspect ratio of 4:3, an even number of splits results on roughly square blocks. Because our all domain in the average domain have a 16:9 aspect ratio and we chose an uneven number of splits, they have an advantage over the home domain when border tracing is used.

domain	without	with	speed-up	stdev
brute force home	43.130 s	7.078 s	6.09x	0.0037
brute force average	19.858 s	3.120 s	6.36x	0.0009
border trace home	1.435 s	0.419 s	3.43x	0.0014
border trace average	12.955 s	2.103 s	6.16x	0.0006

Table 5.9: Calculation times with and without multi-threading using the optimal configuration of 8 threads and 7 splits.

On most domains, we see a speed-up of about 6x. Because calculating the fractals can be fully parallelized, the theoretical speed-up is 8x when using eight cores. We are still far from an 8x speed-up. We actually have 4 cores which are reported as 8 due to hyper-threading, which limits our speed-up. Even though we use many splits, we probably still bottleneck on some blocks, especially on the home domain with border tracing. Also, the splitting algorithm and thread management introduce some overhead.
Strangely, when using OpenMP's 'for' directive, which automatically splits a loop over multiple threads, we see a 1.5x speed-down, even though it could handle the required mutex more efficiently. So in our implementation, we handle locking the mutex and assigning blocks from our queue ourself.

5.6 Combining all optimizations

The speed-up of all optimizations combined can be found in Table 5.10. Here, we use symmetry pasting, shape checking with the main cardioid and period-2 bulb, border tracing and multi-threading with 8 threads and 7 splits.

domain	none	all	speed-up	stdev
home	43.130 s	0.437 s	98.70x	0.00064
average	19.858 s	2.172 s	9.14x	0.00051

Table 5.10: Speed difference between all or no optimizations.

The results are worse than expected. The results from Table 5.9 are slightly faster. In the home domain, symmetry pasting essentially adds an extra split, because only halve of the screen is rendered and the same number of splits is performed. So we should reduce the number of splits when using symmetry pasting. The average result is slightly worse, because we check for the shapes every pixel. In our results in Table 5.3, we still see a speed-up in the average domain. Since we use border tracing, some pixels in one of the shapes are skipped, so the speed-down increases slightly. We think this can fully be prevented if we implement the hitbox detection for shape checking. Note that the average domain calculates nine domains, so the average rendering time per domain is $\frac{2.172}{9} = 0.241$.

5.7 GMP

First we look at the performance in the front-end. Since we want to describe and modify the domain using GMP floats, it is important to test if this performs well. If these calculations already add a noticeable overhead, then the program can never be responsive, no matter how much the back-end is optimized. It takes < 0.003 milliseconds to perform scaling in the front-end using normal double precision floats and 0.022 milliseconds with GMP floats. The speed of the entire process (getting user input, parsing it and drawing the new screen) without the back-end time when using GMP floats is 0.25 milliseconds. This is the total latency of the front-end, which is important when reasoning about the responsiveness of the front-end. Since scaling is the most expensive front-end operation we perform and it still takes well within one millisecond, we do not need to measure the other operation speeds. The speed-down is very small, especially compared to the speed of the back-end, so we choose to always use GMP floats in the front-end. Most computer monitors already have a response time of > 4 milliseconds which most users never notice. Note that timing also introduces some overhead. The results of timing scaling using normal floats and doing nothing at all are indistinguishable. So we can only conclude that using GMP floats is fast enough. The reported speed-down is far from accurate.

Next, we will look at the performance of GMP floats in the back-end. Using GMP floats over normal floats in the back-end introduces a significant overhead as can be seen in Table 5.11. The minimum precision of a GMP float is 64 bit, which is very close to a standard double which has 53 bit precision. To see the speed difference on higher precisions, see Figure 5.4. Here, the normal float is listed with a precision of 0 to illustrate the difference between a normal float and GMP floats of different precisions. The precision of GMP floats increases in steps of 64 bit (this can be different on different architectures). The times noted in these figures are the time it took for the back-end to return a pixelbuffer after requesting it. No optimizations were used. Note that these experiments were run on a server because these experiments take a very long time to complete. This server has a slightly faster CPU than the one used in all other experiments, so the results are slightly faster than others in this chapter. This server has an Intel Xeon E5-2667 v2 CPU, uses GCC version 8.2.0 and GMP version 6.1.0. We could use taskset, but could not set the niceness below 0, which in this case would have mattered because there were other processes running on the system.

domain	normal float	GMP float	speed-down	stdev
home	43.130 s	2088.24 s	48.42x	6.374
average	19.858 s	978.846 s	49.29x	5.690

Table 5.11: Speed difference between normal floats and 64 bit GMP floats with brute forcing.

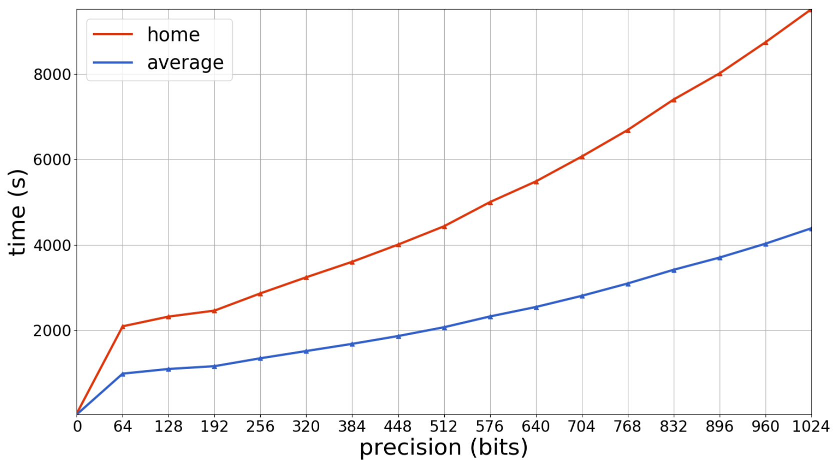


Figure 5.4: Speed difference between different precision GMP floats with brute forcing.

The speed difference between a 64 bit and 128 bit float is much less than the difference between a normal float and a 64 bit GMP float. Otherwise very high precision calculations become infeasible quickly. However, even when using the lowest amount of bits, rendering is not fast enough for interactive viewing, so optimizations become even more important, assuming they still speed rendering up. Unfortunately, some optimizations do not work as well on such small scales. One example is symmetry checking, because the real axis will almost never be in screen. Shape checking has the same problem.

Next we will look at the effect of border tracing. Note that border tracing might perform slightly worse on scales which have to be represented by GMP floats, because the set shows more detail on higher zoom levels. The results can be found in Table 5.12 and Figure 5.5. Table 5.13 shows the speed-up gained from using border trace.

domain	normal float	GMP float	speed-down	stdev
home	1.435 s	50.616 s	35.27x	0.082
average	12.955 s	552.83 s	42.67x	0.532

Table 5.12: Speed difference between normal floats and 64 bit GMP floats with border tracing.

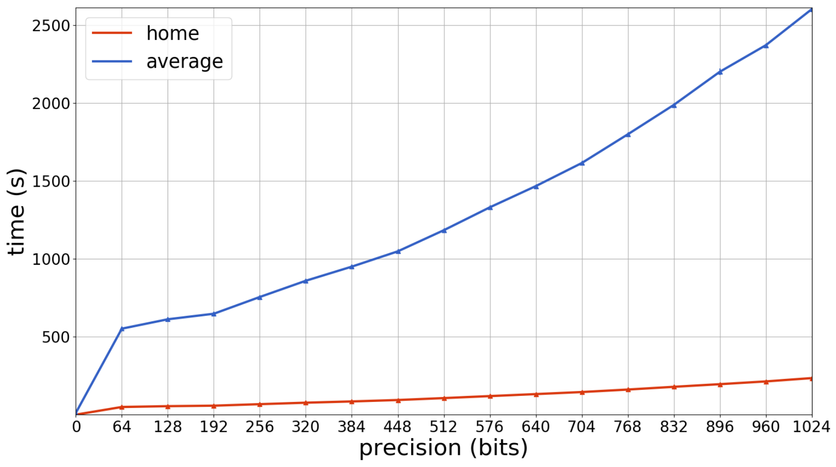


Figure 5.5: Speed difference between different precision GMP floats with border tracing.

domain	brute force	border trace	speed-up
home	2088.24 s	50.616 s	41.26x
average	978.846 s	552.83 s	1.77x

Table 5.13: Speed difference between brute forcing and border tracing using GMP floats.

We do not need GMP floats to describe the home domain; double precision floating points are precise enough. However, we can still compare the results to the normal float border trace. We see that the GMP border trace performs better when using GMP floats, especially on the home domain. This is because every pixel is more expensive to compute, so skipping a pixel saves more time. The higher the number of skipped pixels, the lower the speed-down. Even if we use multi-threading with 8 cores and assume a 6 times speed-up (based in Table 5.9), we still can not render using GMP floats within our real-time constraint.

Chapter 6

Conclusions

In this thesis, we have described methods to optimize calculating images of the Mandelbrot and Julia sets. We use symmetry and the shapes in the Mandelbrot set to skip iterating pixels. We also skip calculating pixels using border tracing. All the calculations are split over blocks which can be calculated in parallel. These optimizations allow us to calculate the set 10 to 100 times faster. At first, calculating the home domain took about 40 seconds and the average domain about 20 seconds. With all optimizations, this is reduced to about half a second for the home domain and about two seconds for the average domain. There are nine domains in the average domain, so it takes about 0.2 to 0.25 seconds to compute one domain. This is well within our set real-time constraint of one second. However, we did not test for very high values for `NMAX`. High values for `NMAX` are not a problem for most domains. Usually, most pixels escape much earlier and the iteration of many pixels in the set can be skipped with border tracing. The images calculated with GMP floats still take about one minute to calculate, which is too long for interactive viewing. However, this is much faster than before, considering that it first took 30 minutes to calculate the home domain and about two minutes to calculate the average domain.

The first goal was to research methods to generate fractals quickly. We successfully implemented most optimizations found in related work. The only big optimizations we did not implement are SIMD and perturbation theory, which are discussed in Chapter 7. We did not implement periodicity checking because of its questionable use and error prone nature. Using orbit plotting in Fraccert, it can easily be seen that barely any orbit goes though the same point twice.

The second goal was to develop a tool in which fractal optimizations could easily be implemented. Also, it should support running experiments on the implemented methods. We succeeded in doing this by creating a clear design in the back-end with an expandable interface which front-ends can use to interact with the optimizations. Dividing the back-end in four steps which correspond to composing the screen (`render`()), calculating the screen (`calcScreen`()), calculating a pixel (`calcPixel`()) and calculating a color (`calcColor`()) allows others to implement optimizations in each step of the calculation process. Also, all optimizations are implemented generically. Shapes can dynamically be added to a list of shapes to check for. Border tracing is implemented as part of the fractal interface, so others can easily use this optimization.

The third goal was creating a tool which could efficiently display the fractals. As discussed, user friendliness helps with efficiency, because it can prevent having to compute extra frames if the result is slightly different than expected. Even though Fraccert's controls feel intuitive, precise control is still difficult. The user can only easily translate and scale with a certain step size. For more precise control, the user has to provide the exact domain through the console. As will be discussed in Section B.2, we have plans for implementing better controls, however, it is not yet present. We do support some features which are absent in many other fractal viewers like orbit plotting, the ability to see the Julia sets morphing into each other and a way to view the small filaments connecting different parts of the sets.
Comparing our results to other well optimized fractal viewer is difficult. Most programs do not report the time rendering a screen took. Also, the well optimized fractal viewers usually use techniques which cause the screen not to be rendered as a whole. For instance, XaoS uses smooth scaling. Because many pixels are reused, we cannot compare XaoS directly to our results. Gnofract 4D uses anti-aliasing and successive refinement, which is explained in Section 7.8.

Chapter 7

Future work

Rendering time can still be improved, as will be discussed in this chapter. As discussed in Chapter 5, shape checking and border tracing can still be improved further. We do not discuss utilization of the Droste effect for Julia sets, because the math describing this is not defined well enough to implement this.

7.1 Symmetry/shape check improvement

Symmetry copying introduces an error as discussed in Section 4.3. When the real axis is aligned with a pixel, the top of a pixel above the real axis has the same distance from the real axis as the top of the pixel below the pixel containing the real axis. We can circumvent the error by aligning the real axis to a pixel. However, this would limit the precision of translation. Aligning the real axis would allow us to use another shape for shape checking; the line on the real axis described by the interval $[-2, 0.25]$. When the pixels are not aligned with the real axis, this shape cannot be used, because most complex numbers c diverge when their real part is in this shape ($\Re(c) \in [-2, 0.25]$) but their imaginary component is non-zero ($\Im(c) \neq 0$).

7.2 Explicit SIMD

In this section, we will only focus on the SIMD instructions on x64 architectures provided through instruction sets like AVX (Advanced Vector Extensions) and FMA. The instructions in the AVX extensions allow a single operation to be performed on multiple variables at the same time. The supported data types and instructions are very limited, but all our required double precision floating point arithmetic is supported. The most recent AVX-512 supports 512 bits registers. Since we use double precision floats, which are 64 bit, we can perform $\frac{512}{64} = 8$ calculations concurrently. Note that we are using complex numbers, so we need two floats per complex number. Coincidently, Intel uses computation of the Mandelbrot set as an example to show their AVX C++ bindings [Lom11]. Here, we can see a brute force implementation and results about the speed-up. The speed-up from SIMD instructions is less than the number of calculations which can be performed concurrently, because data has to be moved to special AVX registers and the escape checking has some extra overhead.

However, as we can see in [Lom11], the speed-up gets close to the theoretical limit.
As said before, Intel implements the brute force variant. Implementing border tracing with SIMD is much harder. Fortunately, our version of border tracing allows for an easy way to implement SIMD. We always calculate the color of the eight neighbors of a pixel, so we can calculate up to eight neighbors concurrently, given our AVX registers are big enough. The current AVX can only calculate four neighbors concurrently, but when AVX-1024 is released, we can calculate all neighbors concurrently. This will probably give less of a speed-up than using SIMD on the brute force variant, due to the random access nature of border tracing. Because of this, we should check that border trace with SIMD is indeed faster than brute force with SIMD when this is implemented.
As with AVX, the compiler might not find all cases where FMA can be used. When using FMA in the source code, we do not have to worry about supporting architectures which do not have FMA hardware, which makes the source code more readable. However, still not as readable as not using FMA functions in the source code. During compilation, the compiler can easily change all FMA operations to normal multiplications and additions. If we were to explicitly implement FMA in our source code, we would like to provide a script which can transform the source code back to source code without FMA operations to maintain readability.

7.3 Handwritten assembly

Almost all calculation time is spent in the loop which iterates a complex number, so it is important this loop is well optimized. The assembly the compiler generates may not be optimal. The instructions for this loop could be replaced by handwritten assembly, which could perform better. One disadvantage of using handwritten assembly is that the code is less portable.

7.4 GPU acceleration

In the same way we switch from normal floats to GMP floats when necessary, we could also switch between single and double precision floats. When we do this, we can perform all single precision calculations on a GPU. However, the precision limit of a GPU is reached very quickly ($\Delta\text{Re} = 8 * 10^{-5}$ with 2000 pixels). In the current state of Fraccert, there is only a small latency when rendering frames on this scale. This latency will further be reduced if SIMD is implemented or processors with higher core counts are used. However, by the time users have higher core counts or AVX-1024 is released, we might also have much higher resolution screens. Resolution increases in two dimensions (width and height), so doubling the resolution may square the calculation time, so there might still be a benefit of using GPUs on lower scales.

7.5 Deepen

When increasing `NMAX`, only points that were considered to be part of the set can get a different escape iteration, because points that escaped before `NMAX` will still escape on the same iteration. So we only have to recompute points which reached `NMAX`. When decreasing `NMAX`, we can just color all pixels black which have an escape

iteration higher than the new `NMAX`. Implementing this requires information about when a pixel escaped to be saved between two back-end calls. Since the back-end does not have a state, we would have to return a (pointer to a) second buffer of the same dimensions as the pixel buffer which contains this information. In theory, the escape iterations could be calculated from the colors of pixels, but this is very expensive, because with our coloring algorithm it would need to perform two square roots and two cube roots.
This optimization is very useful, because when zooming in, every few steps `NMAX` has to be increased to see the smaller details on the smaller domain. Also, changing `NMAX` can change the coloring of the current domain. When trying to get a nice screenshot of the set, the user might change `NMAX` much to get the most visually appealing coloring, so this optimization makes this process much faster.

7.6 Perturbation theory

Perturbation theory is a field of math which studies the approximation of problems which cannot be solved exactly. It tries to solve these problems by solving an easier problem which can be solved exactly. Then, using this easier problem, the solution of the unsolvable problem is calculated by estimating the error the unsolvable problem introduces compared to the exact solution. These estimations are usually expressed as power series, because much research has been done on perturbation in power series (i.e. the Lagrange remainder in Taylor expansions to approximate transcendental functions) and they are generally used to solve these kinds of problems with. As described in [Mar13], it is possible to estimate z_n without arbitrary precision floats from a different point y_n which is calculated with high precision. In this paper, it is postulated that computing three orders of this power series is enough to estimate z_n if the third order error is significantly smaller than the second order error. This would reduce the number of high precision floating point calculations significantly. However, not every choice of y_n will guarantee that the third order error is much smaller than the second order error, so we need a method of finding such complex number efficiently, which is a difficult problem. There are programs which successfully implement this, but most still have significant errors around certain points [HA14]. Figure 7.1 shows an example of distortion when using perturbation theory. This distortion is much less when viewing small domains which have to be described by GMP floats.

7.7 Scaling

Reusing pixels when scaling is very difficult. When using an arbitrary scale factor, at worst (e.g. with scale factor π) only one pixel can be reused (the center of scaling pixel) because none of the pixels align. Using a whole numbered scale factor (especially 2) maximizes the number of pixels which can be reused. However, such large scale factors are not user friendly when exploring, because it is difficult to see where the current frame came from with respect to the previous frame (imagine using Google Maps where you could only scale by ten steps at a time). As mentioned in Section 1.3.2, XaoS solved the pixel reuse problem very well. XaoS spends all available CPU time in rendering new pixels while reusing the others which could not be calculated in time. A heuristic determines which pixels have a higher priority of being recalculated which minimizes the amount of rendering artifacts. Since XaoS implements this so well, we instead focused on other optimizations.

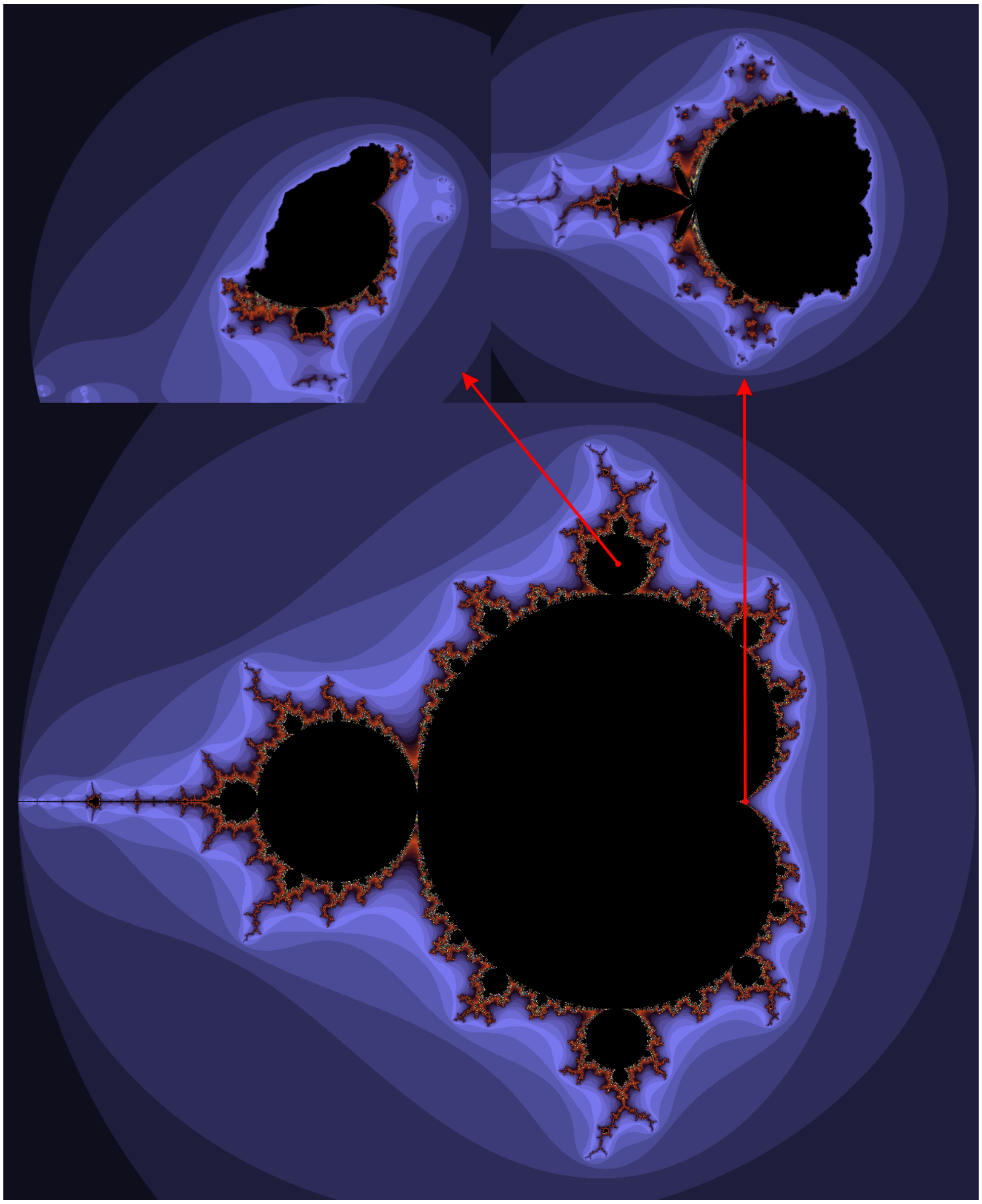


Figure 7.1: Distortion when using the depicted values of c. The three images use the same domain. Screenshots made with XaoS.

7.8 Successive refinement

While calculating the next frame, the user has to wait without feedback. There are ways by which we can provide feedback to the user about the new domain without rendering it.

Currently, scaling works in steps. While waiting for the next frame to be calculated, we can use the GPU to efficiently scale the current image to the new domain. Because the GPU can do this so efficiently, we can do

it in many steps, which results in a smooth transition to the new domain. This scaled image will have poor quality, but it does provide some feedback on the new domain. When rendering a frame takes very long, the user can see an approximation of the new frame and get the choice to cancel rendering the domain.
We could also use a technique which is called successive refinement [Mun10]. Here, we first render the new frame on a very low resolution and successively render the pixels in between the already calculated ones, effectively doubling the resolution each step, see Figure 7.2. This can also be seen as an optimization, because the user sees the requested domain faster but with a lower resolution. Our implementation of border tracing would still work efficiently if we do not clear the alpha channels used for flow control between successive steps. The earlier mentioned article [Mun10] mentions an optimization using successive refinement. This is the inverse of a border tracing algorithm, because it detects if a pixel is part of the surface of a color band. Unfortunately, this method is more error prone than border tracing. It cannot make use of SIMD as much as our border tracer.

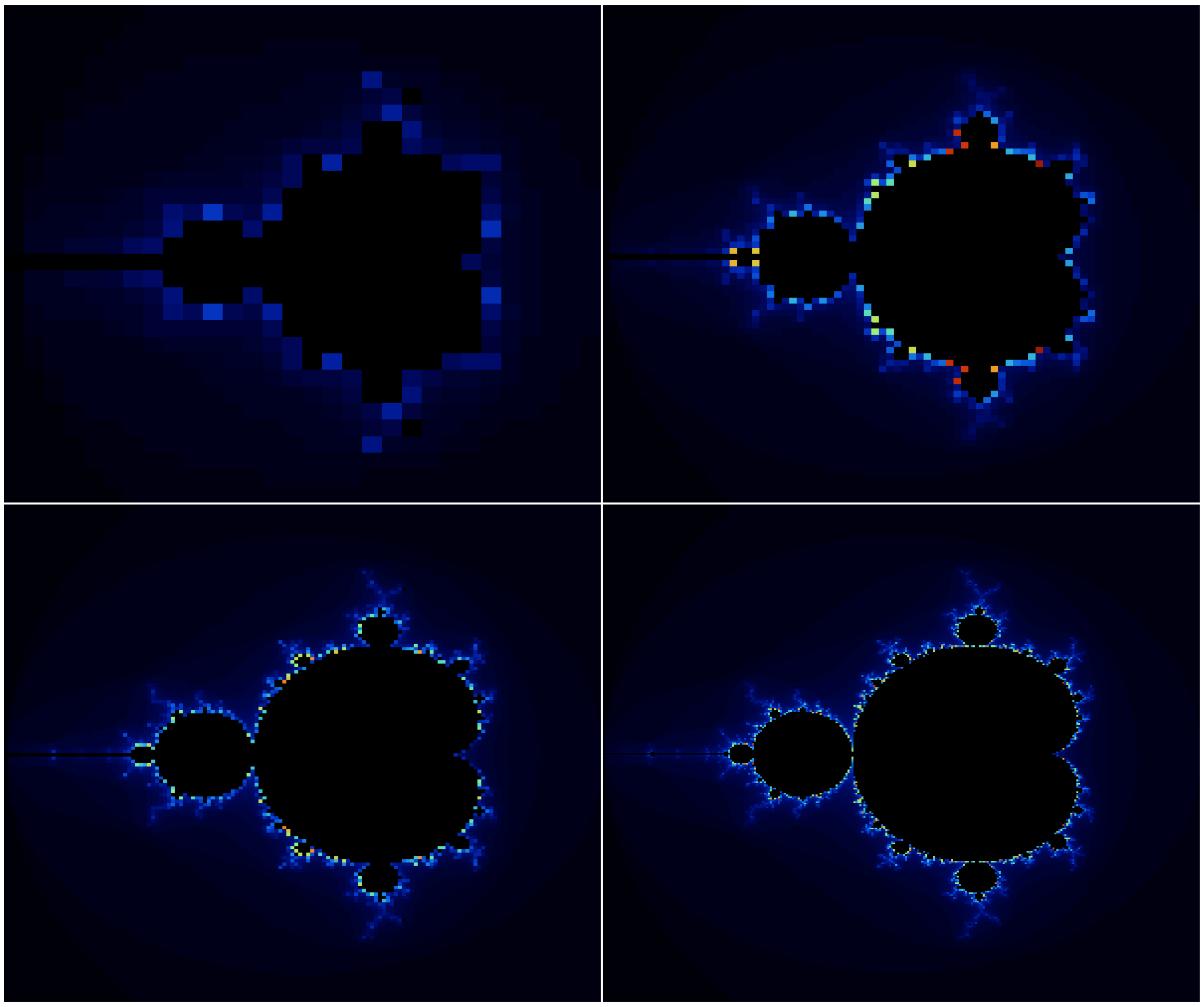


Figure 7.2: Four steps of successive refinement.

Appendix A

Experiment domains and parameters

All locations used during experiments are listed in Table A.1. All of these values can also be found in `locations.cpp`. To see any location, use the command "loc <location>" in Fraccert. For all locations in the average domain, a picture is provided in Figure A.1.

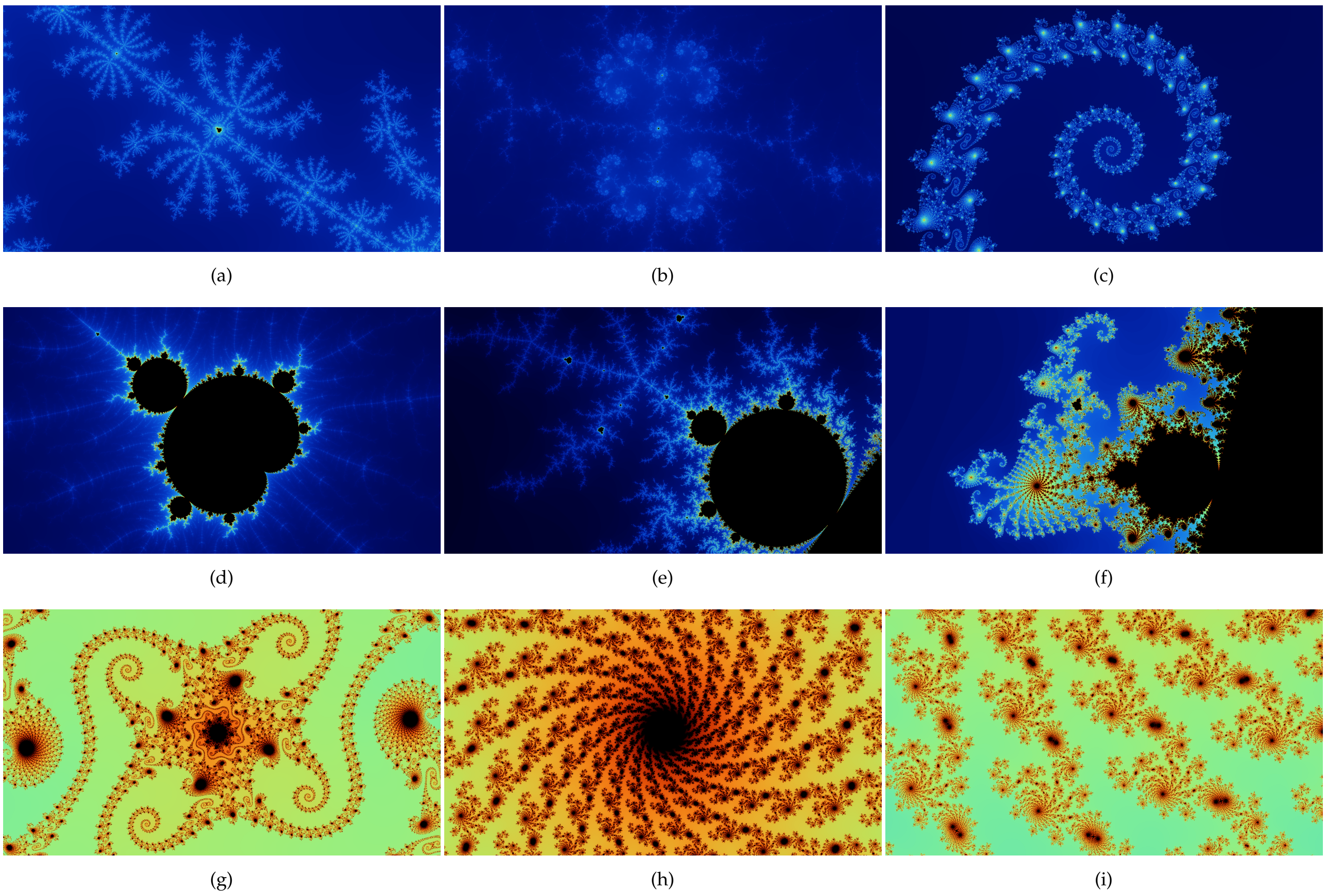


Figure A.1: Pictures of locations in Table A.1

location	Re	Im	resolution	`NMAX`
home	$[-2.0, 1.0]$	$[-1.125, 1.125]$	8000x6000	1000
sym	$[-2.0, 2.0]$	$[-1.5, 1.5]$	1600x1200	256
a	$[-0.6701, -0.6641]$	$[0.4539, 0.4572]$	1920x1080	600
b	$[-1.74858614, -1.74858479]$	$[0.01262719, 0.01262795]$	1920x1080	2500
c	$[-0.74766, -0.74728]$	$[0.08290, 0.08312]$	1920x1080	2250
d	$[-0.6701, -0.6641]$	$[0.4539, 0.4572]$	1920x1080	1500
e	$[-1.26, -1.195]$	$[0.1417, 0.1782]$	1920x1080	700
f	$[-0.753, -0.727]$	$[0.1441, 0.1588]$	1920x1080	350
g	$[-0.7475087485, -0.7475087322]$	$[0.0830715266, 0.0830715359]$	1920x1080	1000
h	$[-0.439165, -0.439089]$	$[0.574562, 0.574604]$	1920x1080	450
i	$[-0.439165, -0.43909]$	$[0.574507, 0.574549]$	1920x1080	475

Table A.1: Table of all locations.

Appendix B

Future Fraccert features and improvements

Fraccert could use some improvements. Some features are still missing and some features are not as easy to use as others. For instance, the user should be able to cancel the current calculation. We would also like to support custom iterative functions to render arbitrary fractals.

B.1 Coloring improvements

Even though our current coloring algorithm looks visually appealing, it has some problems. There is still much noise in some areas. Even though this can never fully be prevented, some coloring algorithm may have less noise. Another way of circumventing this problem is by using higher resolution screens, however, this increases rendering times much. The effect of a higher resolution screen can be mimicked through anti-aliasing techniques like supersampling.

As mentioned before, when `NMAX` is higher than the number of colors, some color bands get the same color. There are multiple ways to prevent this problem. One of the easier methods is performing $n_{\text{col}} = n_{\text{div}} \mod k$, where n_{col} is the number which is used by the chosen coloring algorithm, n_{div} is the escape iteration and k is a number, like the number of colors in the color space. This way, the color space cycles after k colors. Note that with our smooth coloring method, pixels outside the set become black if $n_{\text{div}} \mod k = 0$. This can be prevented by using slightly different equations for values $n_{\text{div}} > k$.

When using coloring methods where the color is chosen based on n_{div}, you will always see banding, because n_{div} is not continuous. Using gradient coloring methods eliminates this problem. If a color is first chosen with an escape time algorithm, we can then add a gradient to the color band which slowly transitions between the two neighboring color bands. For an example, see Figure B.1. As seen in Section 3.2 the bands seem to follow the pattern made by the set itself. If this is the case, the distance estimate could directly be used for gradient coloring.

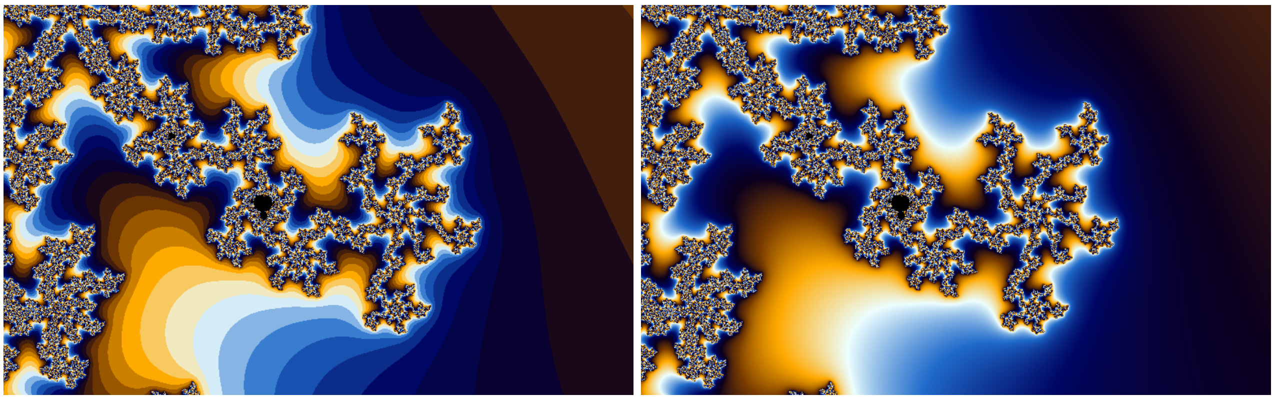


Figure B.1: Difference between escape time coloring and gradient coloring.
Source: `https://en.wikipedia.org/wiki/Mandelbrot_set#Continuous_(smooth)_coloring`

B.2 Precise translation and scaling

When translating or scaling in Fraccert, it moves big steps at a time. Even though this is fine for most cases, it would be nice to be able to easily change the domain in smaller steps. Currently, it can be done through the console, however, this is not user friendly. It would be more convenient to be able to use the mouse for these actions, since it offers precise control. Translating could be done by dragging the screen. Scaling could be done by selecting a rectangle which should become the new screen bounds. One problem is that the user can select a rectangle of any aspect ratio. Gnofract 4D only lets the user select a rectangle with the same aspect ratio as the screen. The initial click sets the upper left corner of the new view. This makes selecting the exact domain which you want to view difficult. Usually, people put the object of focus in the middle. By letting the user click on the middle of the new domain and then letting him drag the mouse while holding the mouse button down allows for comfortable precise scaling. While the user is dragging the mouse, a rectangle could appear on screen which indicates what the new domain will be when the mouse button is released.
A problem of letting the user translate by dragging is that new pixels should be calculated while dragging to see what the new domain will be. Calculating every row of pixels which enters the screen is inefficient, because we cannot border trace on a row of pixels. Since we would need to use pixel reuse to be fast enough for smooth transitioning, we can only change the domain in steps the size of a pixel, which limits the precision of translating.

B.3 Symmetry pasting in back-end

We already use GPU acceleration in symmetry pasting. Since we use the GPU through an abstraction provided by SDL, we do the pasting in the front-end. It would be better to perform symmetry checking in the back-end, so it does not have to be implemented by everyone using the back-end. We do not perform it in the back-end, because we do not want a dependency on a graphics/multimedia library in a math library. Also, SDL has to be initialized, which breaks with our functional design. When we implement symmetry pasting, we would probably use a framework like OpenCL, because it is supported by most GPUs.

Appendix C

Border trace implementation

```cpp
void Fractal::calcScreen(const Domain& domain, const Resolution& res, const Range& r, Shapevector&↩
     shapes, uint32_t* pixels) const {
    const double ps = (domain.rMax − domain.rMin) / (double)res.w;

    // Set border trace struct up
    BorderTrace bt;
    bt.pixels = pixels; bt.pixelSize = ps; bt.data = &shapes;
    bt.rMin = domain.rMin; bt.iMax = domain.iMax;
    bt.w = res.w; bt.h = res.h;
    bt.xMin = r.xMin; bt.xMax = r.xMax; bt.yMin = r.yMin; bt.yMax = r.yMax;
    bt.dX = r.xMax − r.xMin; bt.dY = r.yMax − r.yMin;

    // Border trace
    edgeInQueue(bt);
    while(!bt.pixelQueue.empty()) {
        checkNeighbors(bt, bt.pixelQueue.front());
        bt.pixelQueue.pop();
    }
    fillEmptyPixels(bt);
}

void Bordertrace::addQueue(BorderTrace& bt, const unsigned int pixel) const {
    if(bt.pixels[pixel] & QUEUED)
        return;

    bt.pixelQueue.push(pixel);
    bt.pixels[pixel] |= QUEUED;
}

void Bordertrace::edgeInQueue(BorderTrace& bt) const {
    // This function is only called at start of border trace, so clear queue.
    bt.pixelQueue = std::queue<unsigned int>();

    for(unsigned int y = bt.yMin; y < bt.yMax; y++) {
        addQueue(bt, y * bt.w + bt.xMin);
        addQueue(bt, y * bt.w + bt.xMin + (bt.dX - 1));
    }
    for(unsigned int x = bt.xMin + 1; x < bt.xMax - 1; x++) {
        addQueue(bt, x + (bt.yMin * bt.w));
        addQueue(bt, x + (bt.yMin * bt.w) + ((bt.dY - 1) * bt.w));
    }
}

uint32_t Bordertrace::getColor(BorderTrace& bt, const unsigned int pixel) const {
    if(bt.pixels[pixel] & COLORED)
        return bt.pixels[pixel] & COLOR;

    const unsigned int x = pixel % bt.w,
                       y = pixel / bt.w;

    double c[2];
    c[0] = bt.rMin + (x * bt.pixelSize);
    c[1] = bt.iMax - (y * bt.pixelSize);

    bt.pixels[pixel] = calcPixel(c, bt.data) | COLORED;

    return bt.pixels[pixel] & COLOR;
}

void Bordertrace::checkNeighbors(BorderTrace& bt, const unsigned int pixel) const {
    const unsigned int x = pixel % bt.w,
                       y = pixel / bt.w;

    // Calculate current pixel
    const uint32_t pixelColor = getColor(bt, pixel);

    // Bools for existence of left-, right-, up- and down-neighbor
    // Cache the results, because they are used often
    const bool rightExists = x < bt.xMax - 1,
               leftExists = x > bt.xMin,
               downExists = y < bt.yMax - 1,
               upExists = y > bt.yMin;

    // First calculate 4 the neighbors and check if they are different
    bool rightDifferent = false, leftDifferent = false, downDifferent = false, upDifferent = false↩
        ;
    if(rightExists)
        rightDifferent = getColor(bt, pixel + 1) != pixelColor;
    if(leftExists)
        leftDifferent = getColor(bt, pixel - 1) != pixelColor;
    if(downExists)
```

```
        downDifferent = getColor(bt, pixel + bt.w) != pixelColor;
    if(upExists)
        upDifferent = getColor(bt, pixel - bt.w) != pixelColor;

    // Check neighbors of the neighbors which are different
    if(rightDifferent)
        addQueue(bt, pixel + 1);
    if(leftDifferent)
        addQueue(bt, pixel - 1);
    if(downDifferent)
        addQueue(bt, pixel + bt.w);
    if(upDifferent)
        addQueue(bt, pixel - bt.w);

    // Same for diagonals
    bool rdDifferent = false, ruDifferent = false, ldDifferent = false, luDifferent = false;
    if(rightExists && downExists)
        rdDifferent = getColor(bt, pixel + bt.w + 1) != pixelColor;
    if(rightExists && upExists)
        ruDifferent = getColor(bt, pixel - bt.w + 1) != pixelColor;
    if(leftExists && downExists)
        ldDifferent = getColor(bt, pixel + bt.w - 1) != pixelColor;
    if(leftExists && upExists)
        luDifferent = getColor(bt, pixel - bt.w - 1) != pixelColor;

    if(rdDifferent)
        addQueue(bt, pixel + bt.w + 1);
    if(ruDifferent)
        addQueue(bt, pixel - bt.w + 1);
    if(ldDifferent)
        addQueue(bt, pixel + bt.w - 1);
    if(luDifferent)
        addQueue(bt, pixel - bt.w - 1);
}

void Bordertrace::fillEmptyPixels(BorderTrace& bt) const {
    unsigned int pixel;
    for(unsigned int y = bt.yMin; y < bt.yMax; y++) {
        for(unsigned int x = bt.xMin + 1; x < bt.xMax; x++) {
            pixel = y * bt.w + x;
            if(!(bt.pixels[pixel] & COLORED))
                bt.pixels[pixel] = bt.pixels[pixel - 1];
        }
    }
}
```

Bibliography

[Bea16] A.F. Beardon. Symmetries of Julia sets. 2016.
`https://vdocuments.mx/symmetries-of-julia-sets.html`.

[BM09] Ari Ben-Menahem. *Historical Encyclopedia of Natural and Mathematical Sciences*. Springer-Verlag Berlin Heidelberg, 1st edition, 2009.

[Cre25] Hubert Cremer. Über die Iteration rationaler Funktionen. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 33:185 – 210, 1925.

[Cro95] Richard M. Crownover. *Introduction to Fractals and Chaos*. Jones & Bartlett Publishers, 1995.

[Cro05] Donald D. Cross. Algebraic solution of mandelbrot orbital boundaries. 2005.
`http://cosinekitty.com/mandel_orbits_analysis.html`
Accessed on 27-06-2019.

[fra] Fractint official site.
`https://fractint.org/`
Accessed on 27-06-2019.

[gmp] GMPs official documentation.
`https://gmplib.org/manual/Floating_002dpoint-Functions.html`
Accessed on 28-05-2019.

[gno] Gnofract 4D official site.
`https://edyoung.github.io/gnofract4d/`
Accessed on 27-06-2019.

[HA14] Claude Heiland-Allen. Perturbation glitches. 2014.
`http://mathr.co.uk/blog/2014-03-31_perturbation_glitches.html`
Accessed on 20-06-2019.

[HOP92] Dietmar Saupe Heinz-Otto Peitgen, Hartmut Jürgens. *Chaos and Fractals*. Springer-Verlag New York, 1992.

[Hub82] Adrien Douady, John Hamal Hubbard. Itération des polynômes quadratiques complexes. 1982.

[ieee08] IEEE 754-2008 - IEEE standard for floating-point arithmetics. 2008.
`https://ieeexplore.ieee.org/document/4610935`.

[Jul18] Gaston Julia. Mémoire sur l'itération des fonctions rationnelles. *Journal de Mathématiques Pures et Appliquées*, 1:47 – 246, 1918.

[Lei89] Tan Lei. Similarity between the Mandelbrot set and Julia sets. 1989.
`http://www.math.univ-angers.fr/~tanlei/papers/similarityMJ.pdf`.

[Lom11] Chris Lomont. Introduction to Intel Advanced Vector Extensions. 2011.
`https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf`
Or shorter version:
`https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/`.

[Man77] Benoît B. Mandelbrot. *Fractals: Form, chance, and dimension*. W.H.Freeman & Company, 1977. Translation of Les objets fractals. Forme, hasard et dimension from French from 1975.

[Man82] Benoît Mandelbrot. The fractal geometry of nature, 1982.
`https://users.math.yale.edu/~bbm3/web_pdfs/encyclopediaBritannica.pdf`.

[Mar13] K. I. Martin. Superfractalthing maths. 2013.
`http://superfractalthing.co.nf/sft_maths.pdf`
Accessed on 26-03-2019.

[Mat78] Robert W. Brooks, J. Peter Matelski. The dynamics of 2-generator subgroups of PSL(2, c). *Riemann Surfaces and Related Topics*, pages 65 – 71, 1978.

[Mun10] Robert P. Munafo. Successive refinement. 2010.
`http://mrob.com/pub/muency/successiverefinement.html`
Accessed on 20-06-2019.

[Mun11] Robert Munafo. From the Mandelbrot set glossary and encyclopedia. 2011.
`https://www.mrob.com/pub/muency/filament.html`
Accessed on 21-03-2019.

[nvi] Documentation of CUDA SDK.
`https://docs.nvidia.com/cuda/cuda-samples/index.html#mandelbrot`
Accessed on 27-06-2019.

[Pet91] Tim Wegner, Mark Peterson. *Fractal Creations*. Waite Group Press, 1991.

[qba] Boundary tracing method in qbasic.
`https://web.archive.org/web/20150220012221/http://www.reocities.com/CapeCanaveral/5003/mandel.htm`
Accessed on 27-06-2019.

[Rue12] Franciska Ruessink. Boundary trace floodfill. 2012.
`https://www.codeproject.com/tips/461694/boundary-trace-floodfill`
Accessed on 27-06-2019.

[Sil13] Paul Silisteanu. The Mandelbrot set in C++11. 2013.
`https://solarianprogrammer.com/2013/02/28/mandelbrot-set-cpp-11/`
Accessed on 29-10-2018.

[Vep00] Linas Vepstas. Mandelbrot bud maths. 2000.
`http://linas.org/art-gallery/bud/bud.html`
Accessed on 12-11-2018.

[Wil] Lucas Willems. Mandelbrot set symmetry proof.
`https://www.lucaswillems.com/en/articles/3/mandelbrot-set-symmetry`
Accessed on 27-06-2019.

[xao] XaoS official site.
`http://matek.hu/xaos/doku.php`
Accessed on 27-06-2019.

[YD02] Dan J. Sandin Yumei Dang, Louis H. Kauffman. *Hypercomplex Iterations, Distance Estimation and Higher Dimensional Fractals*. World Scientific, 2002.