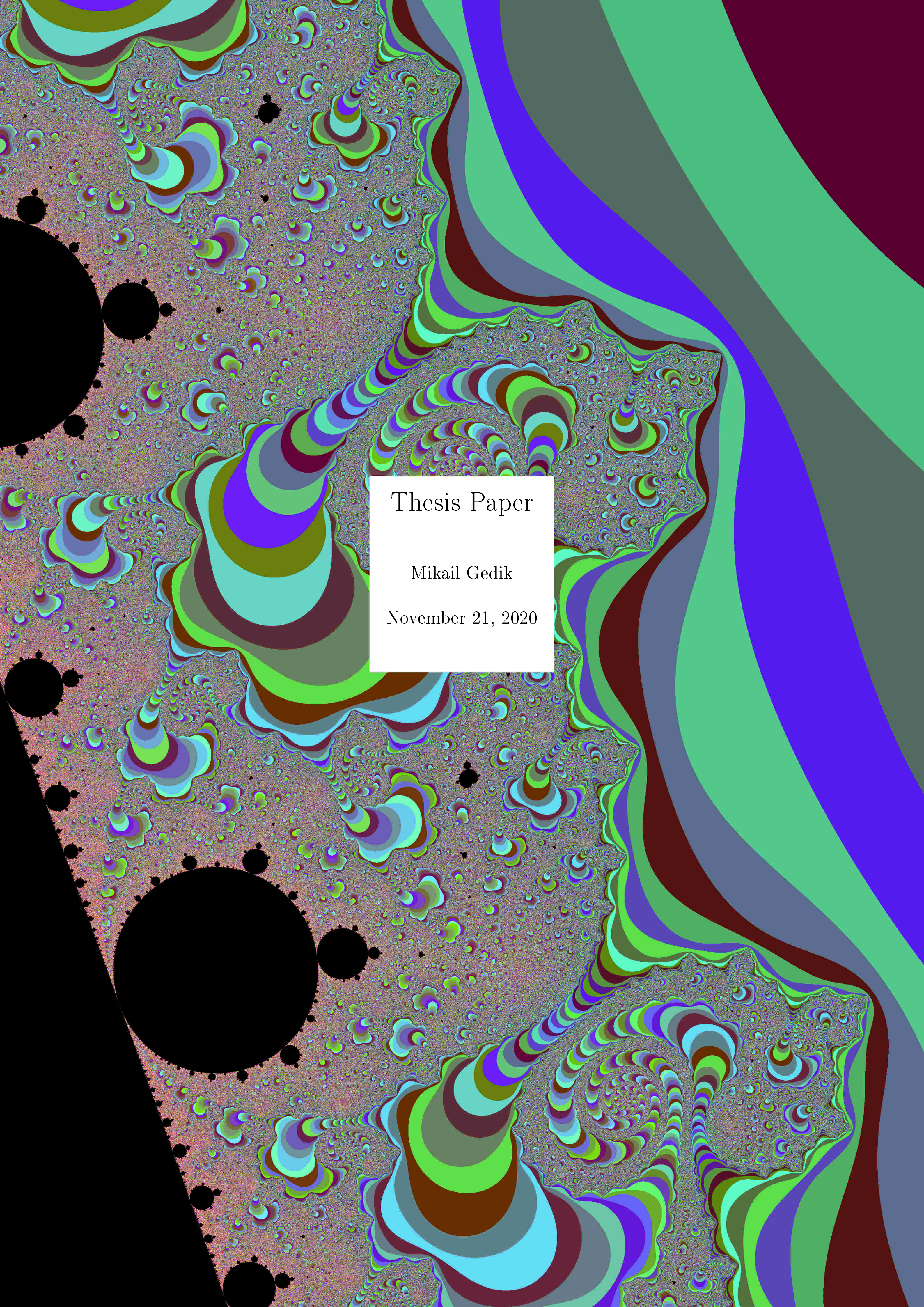


Thesis Paper

Mikail Gedik

November 21, 2020

Contents

List of Figures

1 Abstract

This paper tackles the coding of a program in Java and OpenCL capable of calculating a few selected fractals and shines light on the core aspects I have implemented, such as how the data is calculated and stored. The paper will first explain what a fractal is, albeit only briefly since they do not have to be discussed in depth for my use case. The next section shortly describes the functionalities and the structure of my program, grouping it into logical components depending what they are responsible for. The last section is dedicated to bugs, known issues and their workarounds, limitations and possible improvements of my program, followed by a reflection on my work.

2 What are Fractals?

Mathematicians have longtime relied on classical algebra to describe as well as research sets and functions. Nonetheless, there have always been limitations to algebra: a minimal level of regularity is required. In the past, sets not fitting this property were labeled pathological, irrelevant and not payed further attention to. This decision however was found to be a fallacy, as irregular functions and sets provide a better representation of natural phenomenons and structures (for example coast lines, cloud borders, turbulence in fluids, snow flakes (see figure 2 (page 2))) than normal shapes described by classical geometry. These irregular sets and functions are known as fractals. While they were first conceptualized by Felix Hausdorff in 1918, the term fractal (from Latin fragmented, broken) was only coined in 1975 by mathematician Benoît B. Mandelbrot. A factor in this long time span was the invention of computers, which made the exploration of fractals much easier due to their impressive computation power. Mandelbrot used fractals as a tool to examine the stock market, but they were also found to be useful in various fields like physical chemistry, fluid mechanics and physiology[Fal93; Bri].

2.1 The Sierpiński Triangle

The properties of fractals will be illustrated with the aid of the Sierpiński triangle (figure 1 (page 2)). The structure, while seeming complex at first glance, reveals a rather simple creation process. As illustrated in figure 3 (page 3) the construction starts with a filled triangle which is split into three smaller triangles by cutting out an triangular upside-down hole. The resulting triangles are once again cut out, creating more triangles. This process is repeated infinitely on any newly created triangles.
The first notable property is the perfect self similarity. An infinite amount of scaled down copies of the fractal can be found in the original structure. This property is also found in other fractals, including the Mandelbrot or Julia set, although the copies found often only resemble the original approximately. Some fractals display a statistical self similarity, meaning that they statistically show the same properties at many scales.
The next property is indirectly a predecessor of the aforementioned self similarity. Fractals have to contain a certain micro structure. The Sierpiński triangle can be magnified infinitely without ever showing a final structure which is not anymore fractured. Additionally, the local geometry cannot be described by classical (euclidean, meaning shapes like circles, lines, spheres, cuboids and such) geometry; the structure has by far not enough regularity.
The definition of the Sierpiński triangle is, contrary to its appearance, quite simple. It is furthermore recursive, i. e. the same process is repeated multiple (infinitely in this case) times. Both these characteristics are also often properties of other fractals.
Lastly, the Hausdorff dimension is usually greater than its topological dimension. Its mention, although the subject is out of this paper's scope, is still due. Broadly speaking, the Hausdorff dimension is a real number (contrary to the topological dimensions, which are always integers) which describes how much space a set fills up [Fal93, Einleitung].

2.2 The Mandelbrot Set

Although the most part of my thesis paper is dedicated to computer science the necessity arises to explain the most used fractal in my work. The Mandelbrot set was named after Benoît B. Mandelbrot, and is the first one to be called a fractal. In terms of properties, it is related to the Julia set, which is not a part of my work, although my program has the ability to display them. The Mandelbrot set $\mathbb{M}$ is an, unlike the previously mentioned geometric Sierpiński triangle, algebraic fractal in the complex plane $\mathbb{C}$. There are multiple definitions for the

Figure 1: Sierpiński triangle [Sie]

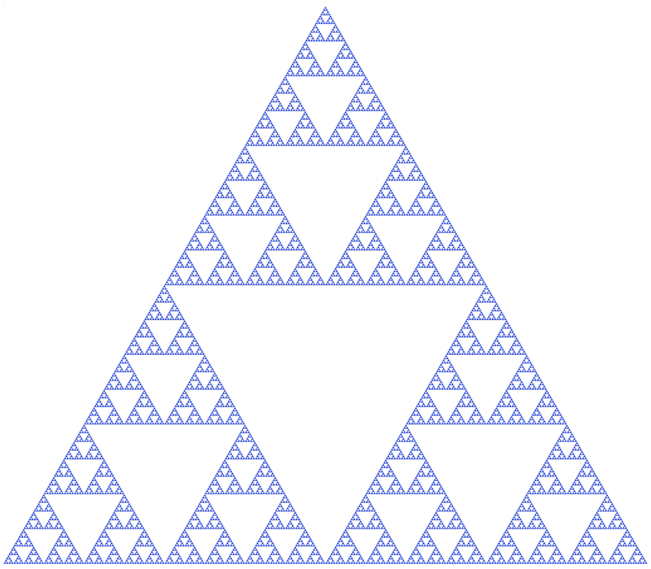
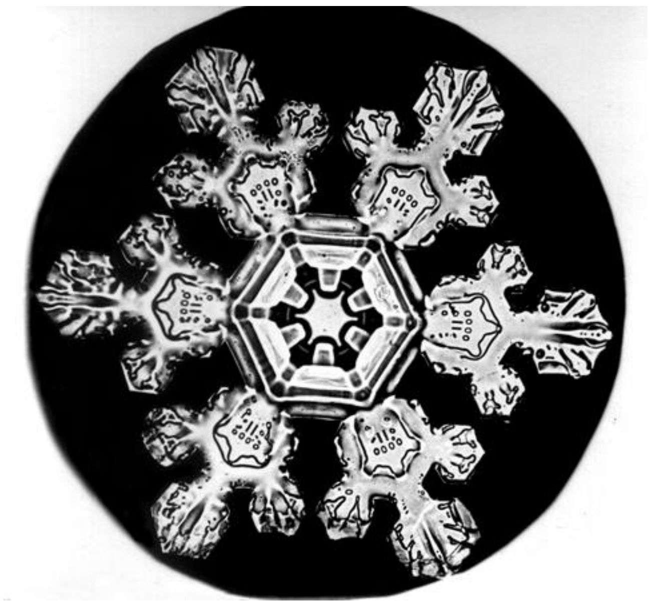


Figure 2: Each branch of the snowflake creates new smaller branches [Wik]



set, but the for my work relevant is as follows:

$$\begin{aligned} f_c(z) &= z^2 + c \\ f_c^n &= f_c \circ f_c \cdots \circ f_c \\ \mathbb{M} &= \{c \in \mathbb{C} : f_c^n(0) \not\to \infty \text{ for } n \to \infty\} \end{aligned}$$

Each point $c \in \mathbb{C}$ has a corresponding function $f_c(z) = z^2 + c$. Now let $z_0 = 0$, $z_1 = f_c(z_0)$, $z_2 = f_c(f_c(z_0))$, or generalized $z_n = f_c(f_c(\dots f_c(0))) = f_c^n(0)$. The example below may be easier understood with the recursive definition $z_{n+1} = z_n^2 + c$. z_k is called the kth iteration of a point. If $|z_n|$ grows to infinity for $n \to \infty$, c is not part of the set. On the other hand, if $|z_n|$ tends to a finite number, c is part of the set.
Table 4 (page 3) shows the first ten iterations of two examples. It is trivial that $z_1 = c$, since $z_0 = 0$. While z_n diverges quickly for $c = 0.9 - 0.2i$, it remains small and does not tend to infinity for $c = 0.3 + 0.2i$. For other c it may take hundreds of thousand of iterations to diverge. It can be shown that, without providing proof since it is out of this paper's scope, if $|z_k| > 2$ for any k then $\lim_{n\to\infty} z_n = \infty$, meaning that further iterations are not necessary to determine that the point is not part of the set. Theoretically, a point has to go through an infinite amount of iterations to prove that is part of the set and does not diverge. To conquer this obviously unaccomplishable task, programs usually only calculate a limited amount of iterations and give up as soon as that maximum is reached. Raising the limit creates more detail in the fractal at the cost of calculation time, but this is only necessary when magnifying the set [Fal93, Chapter 14.2].

Figure 3: Creation of the Sierpinski triangle

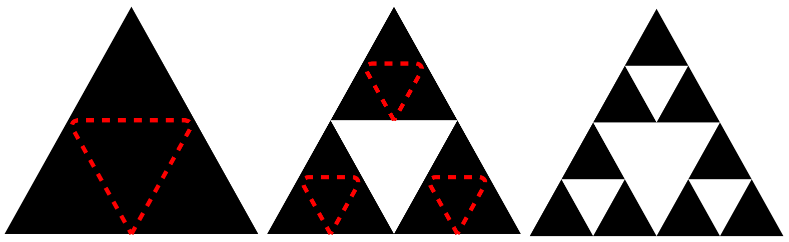


Figure 4: Iterations of different c

	$c = 0.3 + 0.2i$
z_1	$0.3 \; - 0.2i$
z_2	$0.35 \; - 0.32i$
z_3	$0.3201 \; - 0.424i$
z_4	$0.222688 \; - 0.471445i$
z_5	$0.12733 \; - 0.40997i$
z_6	$0.148137 \; - 0.304403i$
z_7	$0.229284 \; - 0.290187i$
z_8	$0.268363 \; - 0.33307i$

	$c = 0.9 - 0.2i$
z_1	$0.9 \; + 0.2i$
z_2	$1.67 \; + 0.56i$
z_3	$3.3753 \; + 2.0704i$
z_4	$8.00609 \; + 14.1764i$
z_5	$-135.974 + 227.196i$
z_6	$-33128.1 - 61785.2i$
z_7	$-2.71994 \times 10^9 + 4.09366 \times 10^9 i$
z_8	$-9.35996 \times 10^{18} - 2.2269 \times 10^{19} i$

3 The Program

This section represents the core of my paper. It starts by listing the capabilities of my program without giving technical details to the underlying model, which in turn is described as next. Lastly, the implementation is discussed.

3.1 Program Functionalities

My program is able to calculate and render fractals. Before starting the calculation engine, parameters of the calculation can be set, for example the maximal amount of iterations a point can go through before assuming that it is part of the Mandelbrot set. When the user has started the render engine, he can move around and zoom into the fractal. Should he find an interesting spot, he can export the location as an image and save it to the disk. It is also possible to create an animation and export it as a video. The export dialog sports the option to change the resolution. The raw calculation data may also be saved to, or loaded from, the disk.
Users familiar with OpenCL can also modify the calculation and render kernel in order to change the fractal and coloring function. Depending on how many iterations are requested by the user massive computing power is needed. Should the local power not suffice to achieve the result in useful time, it is possible to harness the computing power of multiple computers by connecting them to each other. One machine then acts as the "master" and coordinates all the other machines ("slaves"), while they in turn calculate the fractal.

3.2 Structure

My program is separated in two halves, the GUI and core. While the latter is autonomous, the former is dependent on the core, as it must know its interface, which is the connecting bridge between them. The GUI can request or order a command through a command string or an integer, i. e. a character sequence or a number. The core answer these with a command result, which may also be an image.
Because the core is detached from the GUI, I may also develop multiple GUIs for different needs. The GUI in the diagram below shows a possible version of window-oriented GUI, which is relatively modern and user-friendly, but for example useless in a command line environment. Thus I see the need to develop different GUIs.

3.2.1 Core

The core consists of multiple components or modules, which are tightly bound together. Each of them is responsible for a certain task and should not do anything else. The modules are only capable of communicating to each other and are isolated from the 'outside', with the exception of the connector and file handler. The latter two interchange data with the UI or underlying file system.

3.2.1.1 Settings The settings for calculation are either used directly for computation or as parameters for the program itself. Variables used during the computation could be for example the C in the Julia set, numerals

Figure 5: The Mandelbrot set

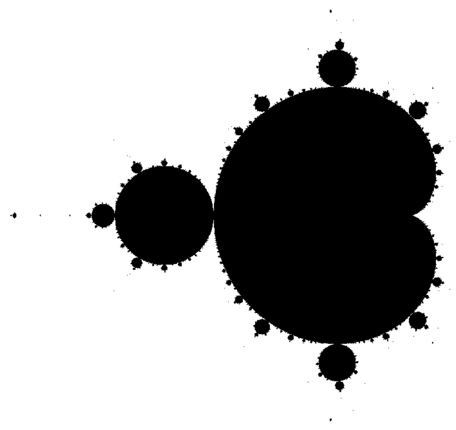
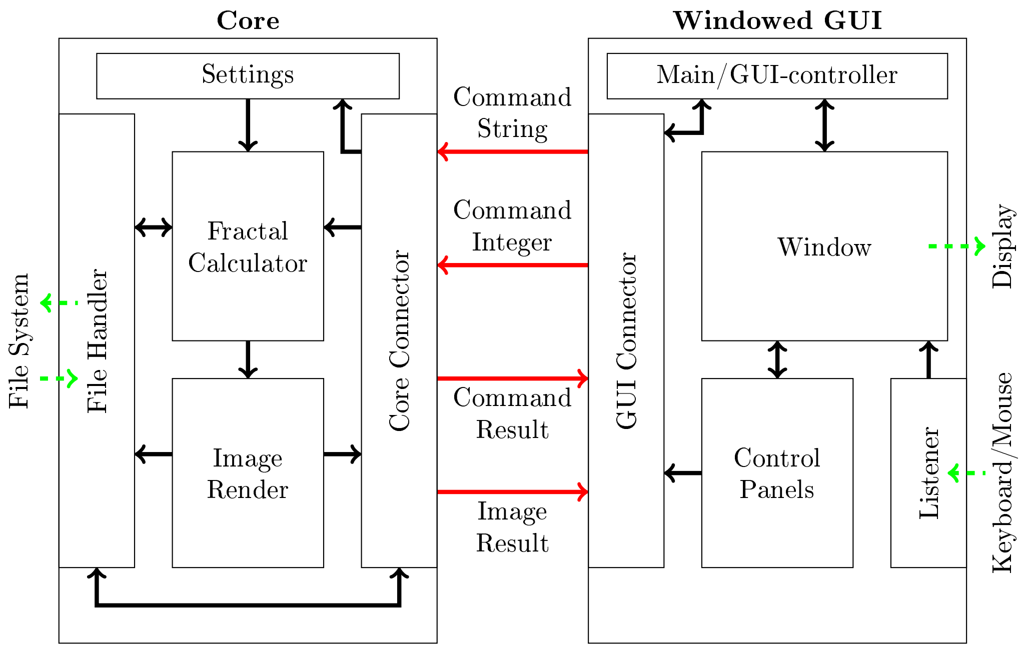


Figure 6: Program structure



such as the maximal number of iterations allowed during computation or the color function used when coloring the calculated set. Other more technical information regarding the system, for instance available RAM, CPU name or the version of the program, may also be stored there. Although the word settings suggest that these values are changeable too during runtime, they are not meant to change. To see the difference between mutable and immutable values, changeable ones are prefixed with `setting`, while the others are prefixed with `value`.

3.2.1.2 Fractal Calculator Upon receiving the command from the interface, the fractal calculator starts to calculate a set of points according to the parameters fetched from the settings module. After having created all necessary information to start the rendering process, the calculation's result is given either to the image render module or to the file handler. The fractal calculator is the core part of my thesis paper and I will spend most of my time tinkering with and finding optimizations for it.

3.2.1.3 Image Render Module The image render module creates images from the data calculated. A major advantage of this module is the buffering of data; Since the user will zoom in only gradually, new images created are often based on already used data. To make use of this, previously generated images are buffered and reused if possible.

3.2.1.3.1 Color Function The color function maps the values of calculated points to colors. The simplest coloring is the in-out one, showing if a point has diverged or not. A more complex coloring method involves the amount of iterations a point had to take for its divergence, resulting in a much more colorful plot, and also showing the Mandelbrot's branches clearly.

3.2.1.4 File Handler The file handler abstracts the saving and reading process of fractals, videos or images. This module is the smallest one, since most of its implementation is already provided by the default library of Java.

3.2.1.5 Data Model The data model is the one of the most complex parts of my project. One of the toughest nuts to crack was the intelligent allocation and distribution of data, which was necessary to have low access time when searching specific points or areas in the dataset. To achieve this, the data is logically split in multiple layers: the levels and clusters. A level is container for multiple clusters, which in turn contain the data. All levels cover the same area, but with a different density of clusters, resulting in a different precision. One more caveat was RAM usage; My first algorithm was simpler and less sophisticated, and started using a massive amount of RAM to precache not yet calculated points. Although this made fetching data faster, it was unusable since the application was bound to crash after zooming in only a little bit.

3.2.1.6 Connector The connector manages calls from the UI to the backend, instructing the fractal calculator or image render module on what to do. Although this part is omittable since all the components could be called directly by the UI, I have added it because it abstracts and facilitates the usage of the core. Through this abstraction I have been able to make changes to the core without altering the UI.

3.2.2 GUI

The GUI (graphics user interface) is the mediator between the core and the user. It makes the program's capabilities, which are invoked through methods and functions, available to humans, only capable (without considering voice commands or touch screens) to interfere with the computer by mouse and keyboard. The GUI takes commands, which are issued by mouse clicks or keyboard strokes, and forwards them to the core. These commands usually yield a result, which is reported back to the GUI and shown to the user. For all this, a window is needed.

3.3 Implementation

This section sums up the central algorithms and structures which build my program.

3.3.1 Helper Classes

These structures do not contain any important algorithms or functionalities, but are used as wrappers for simple data. Since they are used often, I must make them trivial before being able to explain the more complex aspects of my program.

Figure 7: Apples(red) in a box (black), as seen from above

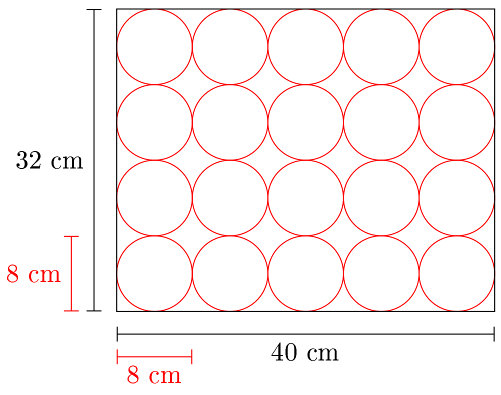
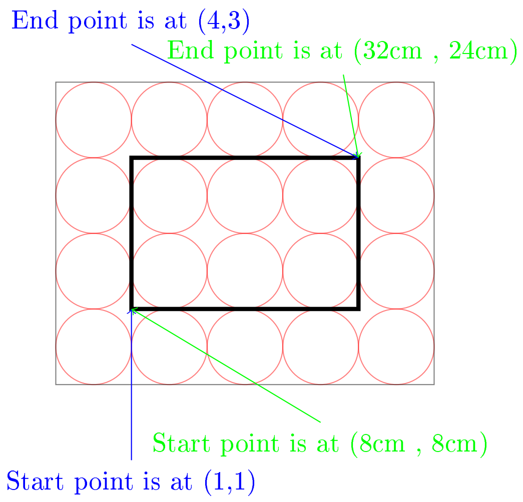


Figure 8: Mesurements for normal region (green) vs. logic region(blue)



3.3.1.1 Region A the `Region` class is a wrapper for a clip of the calculated set. It is defined by a start and end point. The coordinates are real floating point numbers, internally stored as primitive `doubles`. To avoid unintentional changes, the fields of the class are marked as final. Thus, there are only getter methods for the fields. To simplify the usage further, there are also methods to calculate the width and height of the clip.

3.3.1.2 Logic Region The `LogicRegion` class has the same functionalities as the `Region` class, but uses a different approach to store the data. Explaining the alternative method will be in vain without having read the section 3.3.2 (page 6) regarding the data management, but may still be explained by using an example. Imagine a box filled with apples. The box and apples both have fixed dimensions, as shown in figure 7 (page 6). The box and apples are an analogy to my program, which uses levels and clusters. An area of this box can now be defined using two different methods. The more intuitive way would to make the measurements in centimeters, i. e. the green arrows; the region starts at the point $(8\text{cm}, 8\text{cm})$ and ends at the point $(24\text{cm}, 32\text{cm})$. No matter how big or small the apples are, the region stays the same. Another way to define the area, illustrated by blue arrows, employs another metric. Instead of centimeters apples are used, thus defining the start at $(2\text{apples}, 2\text{apples})$ and the end at $(5\text{apples}, 4\text{apples})$. Note that the apple at $(5\text{apples}, 4\text{apples})$ is not in the clip, although it is used to define the clip. This is because when referring to an apple, the reference point is always the lower left corner. Although it might seem unfit to define it this way, there is another reason to do this. To get the dimensions, one can now simply subtract the start from the end point, which would otherwise give a wrong result $((5\text{apples}, 4\text{apples}) - (2\text{apples}, 2\text{apples}) = (3\text{apples}, 2\text{apples}))$.
The logic region facilitates the access to clusters stored in a level by being able to directly access their position instead of having to deal with their absolute position. The implementation of this class is the same as the `Region`, but it uses the `int` instead of the `double` type.

3.3.1.3 Screen The `Screen` class represents a rasterized image with fixed dimensions. Internally, a `int[]` array stores all the data. The class provides various methods used in basic image manipulation. I had initially written this class long ago for other projects, but since I needed a class with these functionalities I have recycled it and added it to this project.

3.3.2 Data Management

Data calculated can be stored in many ways, from simple arrays to complex databanks. To understand the approach I have taken on storing data, I will have to explain the way computers store memory.

3.3.2.1 Accessing RAM It has become trivial knowledge that computers work with zeroes and ones, but to understand the problems faced in storing data, I have to explain the principles a little further. To remember the values that my program has calculated, it stores them in the RAM. The RAM (random access memory) is essentially a gigantic collection of cells, each able to hold a single byte (one byte corresponds to eight bits, or eight zeroes or ones). To gain access to the memory, every cell has its own address, which is (on modern operating systems/computers) a 64 bit number. Since a 64 bit number is quite long, they are usually represented in hexadecimal format. In figure 9 (page 7) there is a schematic drawing of five memory cells, each labeled with

Figure 9: Memory cells, with addresses in hexadecimal format

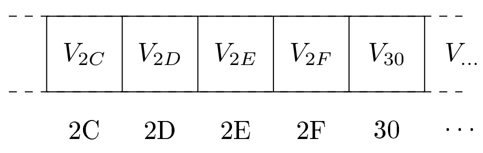
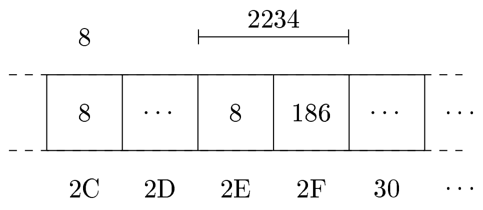


Figure 10: Memory cells storing a numbers



their address (to make the numbers shorter, the addresses' length is only 16 bit). Each of these cells can hold a value V between zero and 255 (that being the highest number a byte can represent). How these cells are exactly accessed, set up and managed is quite interesting and complicated, but I will not discuss this in detail since it is not relevant for my work.

3.3.2.1.1 Storing numbers

Storing an number is quite simple. Figure 10 (page 7) shows two numbers, one eight and one sixteen bit, stored in the RAM. The eight bit number can have values from 0 to 255, since it cannot represent more values. As bigger numbers are required, the system reserves multiple cells and treats their value as a single one. Storing the number 2234, $(00001000\ 10111010)_2$ in binary format, requires two cells. The number is split between the cells, one holding the value $(00001000)_2 (= (8)_{10})$ and the other $(10111010)_2 (= (186)_{10})$. Nowadays computers usually use 32 or 64 bit numbers, requiring four or eight bytes per number.

3.3.2.1.2 Storing lists

The next step to storing single numbers is storing a list of numbers. Lists, often called arrays, are nothing more than many numbers lined up behind each other. Figure 11 (page 7) shows a list with five 1 byte elements. In computer science, list usually start at the index zero instead of one because it usually is easier to access them, as I will shortly explain. The zeroth element has the address $2E$, the first $2F$ and so on. Because each element's address can be easily expressed by the previous', it is only necessary to remember the zeroth's. Assuming A_0 is the zeroth's address, the nth elements' can be determined using the formula $A_n = A_0 + n \times s$, where s is the amount of bytes each element needs, in this case one. Giving an example, the third elements address A_3 is equals to $2E + 3 \times 1 = 31$. [How]

3.3.2.1.3 Storing an Image in RAM

Since the RAM is a one dimensional space, allocating one dimensional objects like numbers, lists or similar things is quite easy. It suffices to know the size of the elements to access one. Storing multidimensional objects requires two map them to a single dimension.
In case of an image, each pixel has two indexes denoting the locations on the abscissa and ordinate axes. The pixels are thus describe by two identifiers, making them two dimensional. To store them in RAM, the two indexes have to be converted into a single identifier. Figure 12 (page 8) shows this transition. On the left, each pixel $P_{x,y}$ is described using two indices x, y, while on the right the pixels P_i are described by the single index i. The goal is to find a way to transform the indices x, y to a single index. By defining $i = x + 3 \times y$, both indices are mapped to a single one.

Figure 11: Memory cells storing an array

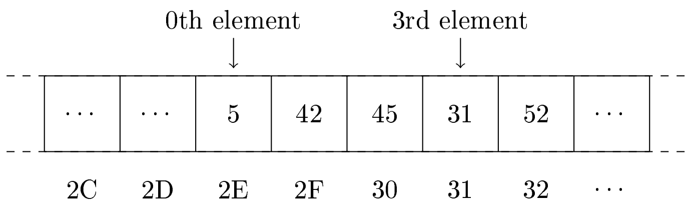


Figure 12: Mapping a two dimensional image to a single dimension

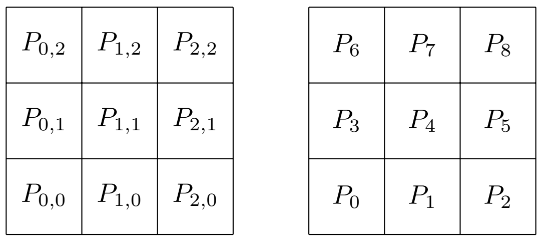
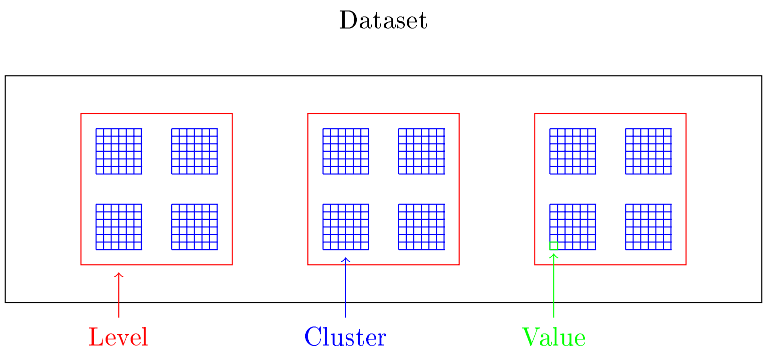

$P_{0,2}$ $P_{1,2}$ $P_{2,2}$
$P_{0,1}$ $P_{1,1}$ $P_{2,1}$
$P_{0,0}$ $P_{1,0}$ $P_{2,0}$
P_6 P_7 P_8
P_3 P_4 P_5
P_0 P_1 P_2

Figure 13: The structure of the dataset



3.3.3 The Dataset

The dataset is responsible for the storage, access and management of values calculated by my program. Implementation wise, the central parts are located in the `DataSet` class, which contains all methods relevant for the management. The classes `Level`, `Cluster` and `Value` help sorting the data in logically aligned subsets. Figure 13 (page 8) shows the components hierarchically. The dataset contains levels and these contain cluster which are made of values.

3.3.3.1 Basic understanding All values calculated are stored in the `Value` class. This class is a wrapper for a single value. The job of the dataset is to store these in a way that they are quickly accessible. The first viable solution is to store them in a single big list. This however has the drawback that the entire list has to be searched for a value. The solution to this was to group values in an area together to form a so-called cluster. These are then again sorted by their precision into different levels. Figure 14 (page 8) shows how the data is stored, taking the actual position of the data into account.

3.3.3.1.1 Value The value wraps a single value. It contains no other data, not even the location which the data corresponds to. These are determined by the position of the cluster and the index of the value in the cluster.

Figure 14: The actual of the dataset shown with the Cartesian coordinate system, taking position into account The dataset/level is black, clusters are blue and values are green

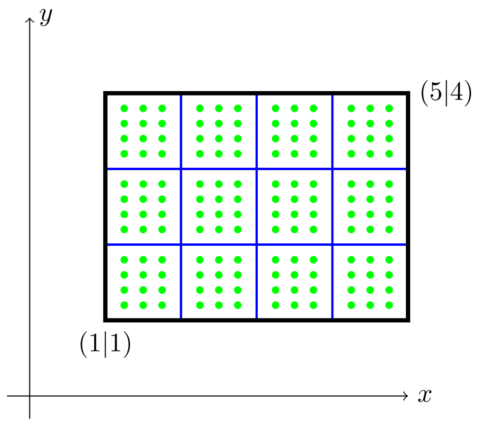
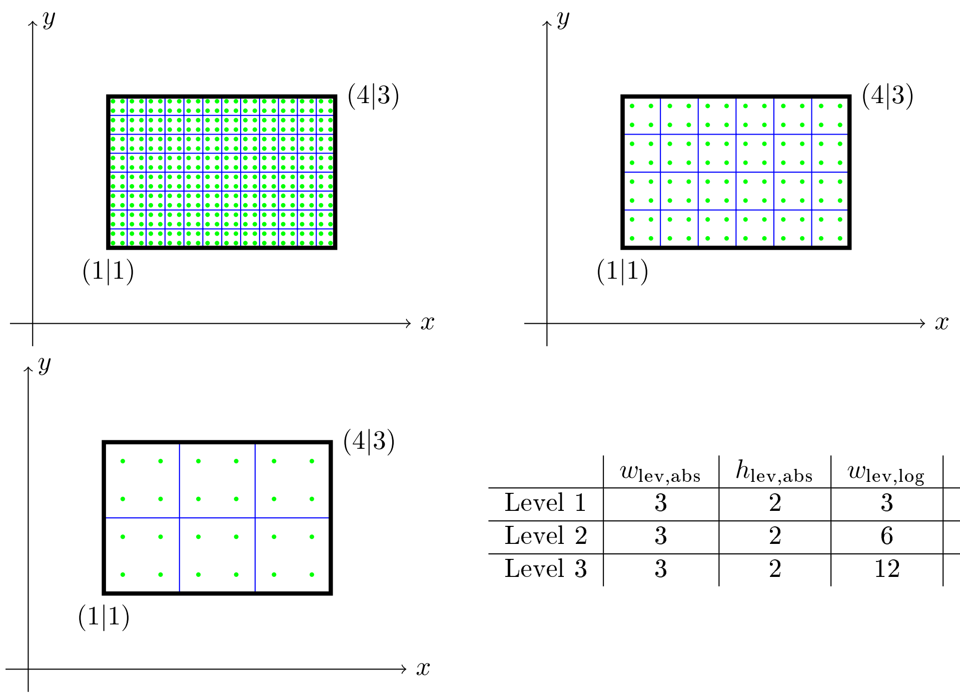


Figure 15: Different levels in the same dataset



	$w_{\text{lev,abs}}$	$h_{\text{lev,abs}}$	$w_{\text{lev,log}}$	$h_{\text{lev,log}}$
Level 1	3	2	3	2
Level 2	3	2	6	4
Level 3	3	2	12	8

Figure 16: Function Signature to create images

```
public Screen createScreen(int minPixelWidth, int minPixelHeight, Region region,
              int threads, long maxWaitingTime)
```

3.3.3.1.2 Cluster A cluster is a collection of values. As seen in figure 14 (page 8), a the values contained are in a rectangle. The width can be measured in two ways: either the absolute width $w_{\text{cls,abs}}$, referring to the width of cluster's area, or the logical width $w_{\text{cls,log}}$, expressing the width in amount of values. In the figure, $w_{\text{cls,log}} = 3$ and $w_{\text{cls,abs}} = 1$ can easily be interpreted. The same rule can be applied to the height, expressed by $h_{\text{cls,log}}$ and $h_{\text{cls,abs}}$.
In a dataset all clusters are required to have the same logical dimensions. The absolute width and height, as well as the position, can vary.

3.3.3.1.3 Level The level is a list of clusters. Similar the aforementioned, it has two different ways to measure its dimension. An absolute, referring to area of the set it covers ($w_{\text{lev,abs}}$ and $h_{\text{lev,abs}}$), and a logical, involving the amount of clusters it contains. The main difference is that all levels in a dataset have the same absolute dimensions. In figure 15 (page 9), three levels of a dataset are shown. All span from the point (1|1) to (4|3). However, the amount of clusters they fit is different. The table shows the values that change with the depth d, which describes how many values a level contains.

3.3.3.1.4 Dataset The dataset stores all levels, clusters and values and provides methods to access them. It defines all the parameters used, such as $h_{\text{lev,abs}}$ or $w_{\text{cls,log}}$.

3.3.3.2 Implementation

3.3.4 Image Generation

A less vital but still important aspect of my work is the visualization of the data. It is crucial to consider the way the data is stored when performing access-intense tasks, such as this. A more naive approach of mine search the whole dataset for each pixel colored, resulting in abysmal performance. By using a more sophisticated algorithm, the data can be directly mapped to the image without searching long for it. This section will go sequentially through all the steps taken.

Figure 17: The dataset region(black) and the requested region (red)

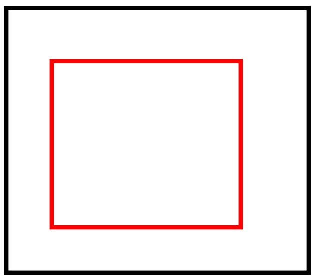
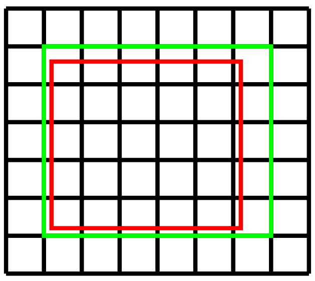


Figure 18: The level, its clusters and the logic region in green



3.3.4.1 Image Parameters To know what type of image the program has to create, parameters have to be passed to the function called. As seen in figure 16 (page 9), the first two refer to the (minimal) image constraints in pixels, `region` describes the region of the fractal to render. `threads` indicates the maximal, although not minimal, amount of CPU threads used to create the image, and `maxWaitingTime` indicates how long the image creation can take before it is aborted to avoid blocking the application in case of an error, infinty loop or deadlock (although neither is likely to happen at all).
The returned `Screen` has at least the dimensions given by the parameters, but is very likely to surpass them. Due to the mapping of the pixels the actually displayed region might vary by at most a single pixel, but I have deemed that negligible.

3.3.4.2 Preparing the rendered Area In figure 17 (page 10), the starting point of the problem is visible. A clip of the dataset has to be mapped onto the red region. Using the given dimensions of the red area, the necessary precision of the datapoints can be derived. Through the precision, we can calculate the necessary level and get the smallest logic region which can fit the red area inside it (figure 18 (page 10)). The logic region is needed to know from which level and place (index to be exact) in the dataset the data has to be queried, rending a long search for the datapoints redundant. The precision of the level is always better or equal to the precision of the requested image.

3.3.4.3 Image allocation Instead of allocating an empty image with the specified dimensions and filling it directly with data, the allocated image has the dimensions of the logical area. I have illustrated this in figure 19 (page 11). The logic cluster width and height are both 5, the logical area has the width and height of 3. The requested area lies unaligned with the rest in the middle, with a width of 5 and a height of 6. As seen in the image, the region where data is fetched from has a greater precision as the red requested area; The values in the dataset lie closer to each other than in the area.
Following this example, the allocated image width is the logical cluster width times the logical region width ($4 \times 5 = 20$), and the image height is the logical cluster height times the logical area height ($4 \times 6 = 24$). The dimensions are measured in pixels. After it has been created, it can be immediately filled with data , which is a rather easy process since the pixels are by definition perfectly aligned with the data. In my program, this interim result is called `ImageResult` and is cached for later usage(see section 3.3.4.4 (page 10))
After the image has been populated by the values, it can be cropped to the requested area. But as seen in figure 20 (page 11), the cropped image's properties are not as requested. The image is wider and taller than asked for, but this is a rather minor problem since it does not degrade the quality and can be easily corrected by scaling the image down. A bigger problem regarding the accuracy is that the returned clipping is slightly bigger than requested.

3.3.4.4 Buffering As mentioned above, generated `ImageResults` are cached for further usage. The reason becomes quite clear when thinking of the usual navigation in fractals. The user will often slowly zoom into a

Figure 19: The logical area (green) with clusters and values (blue), and the requested region (red) with pixels

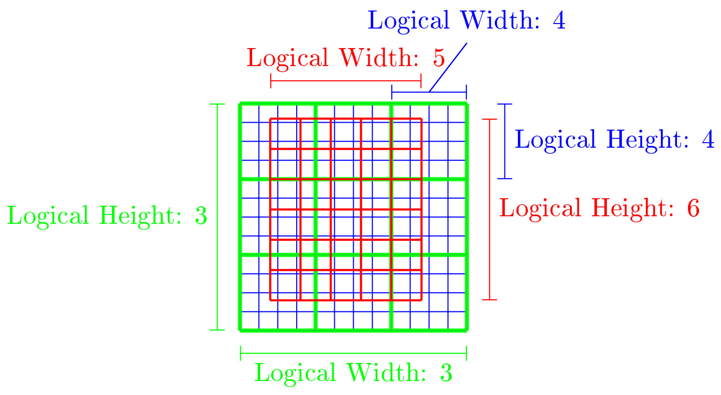
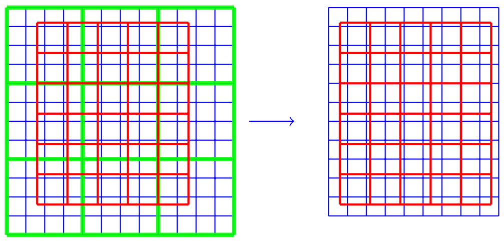


Figure 20: Cropping the image to the area given by the parameters



section of the set. This means for the render engine that it will create many images with a nearly identical area, thus being in the same logical area. Hence having a buffer filled with previously generated `ImageResults` can reduce the rendering time by much. Searching the buffer for a match takes place after knowing the logical area an image needs to be rendered. It is then checked against every entry in it, and if a suitable `ImageResult` is found, it is taken and used instead of creating a new one from scratch. If this is not the case, a new `ImageResult` has to strenuously be created and is afterwards added to the buffer.
An issue encountered with buffering is the increasing RAM usage. Since the buffer contains only data which can be recreated at medium cost, it is not necessary to let it grow infinitely. Limiting its size leads to less RAM usage, but potentially a higher performance impact.
The buffer is only useful in cases where the roughly the same area is rendered often, such as zooming in or moving our a little. As soon as the area is moved around or zoomed in too much, thus requiring a new logical area or new depth respectively, a new `ImageResult` has to be created.

3.3.5 GUI

4 Workflow and Major Version History

5 Reflection

Programming an application of this size has come with many unexpected challenges which are tied to the nature of computer science. These challenges have, when not accounted for, caused either several delays in my time schedule or forced me to not implement a feature. One of the most banal and trivial hardships encountered are logical errors in the code, causing either simple misbehavior or runtime crashes. These can be prevented

References

[Bri] *title.* URL: `https://www.britannica.com/science/fractal`.

[Fal93] Kenneth J Falconer. *Fraktale Geometrie : mathematische Grundlagen und Anwendungen.* ger. Heidelberg [etc.: Spektrum, Akademischer Verlag, 1993. ISBN: 3-86025-075-2.

[How] *title.* URL: `https://computer.howstuffworks.com/c23.htm`.

[Sie] *title.* URL: `https://upload.wikimedia.org/wikipedia/commons/thumb/4/45/Sierpinski_triangle.svg/1000px-Sierpinski_triangle.svg.png`.

[Wik] *title.* URL: `https://en.wikipedia.org/wiki/Snowflake`.