

# Thesis Paper

Mikail Gedik

October 17, 2020

# Contents

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>What are Fractals</b>	<b>1</b>
<b>3</b>	<b>The Mandelbrot Set</b>	<b>1</b>
<b>4</b>	<b>The Program</b>	<b>2</b>
4.1	Program Functionalities . . . . .	2
4.2	Structure . . . . .	2
4.2.1	Core . . . . .	2
4.2.2	GUI . . . . .	4
4.3	Implementation . . . . .	4
4.3.1	Helper Classes . . . . .	4
4.3.2	Data Management . . . . .	4
4.3.3	Image Generation . . . . .	5
4.3.4	GUI . . . . .	6
<b>5</b>	<b>Workflow and Major Version History</b>	<b>6</b>

## List of Figures

1	Sierpi?ski triangle [[3]] . . . . .	1
2	Each branch of the snowflake creates new smaller branches [4] . . . . .	2
3	Creation of the Sierpinski triangle . . . . .	2
4	Program structure . . . . .	3
5	Memory cells, with addresses in hexadecimal format . . . . .	4
6	Memory cells storing a simple string . . . . .	4
7	Function Signature to create images . . . . .	5
8	The dataset region(black) and the requested region (red) . . . . .	5
9	The level, its clusters and the logic region in green . . . . .	5
10	The logical area (green) with clusters and values (blue), and the requested region (red) with pixels	6
11	Cropping the image to the area given by the parameters . . . . .	6

# 1 Abstract

This paper tackles the calculation of a few selected fractals and shines light on the core aspects I have implemented and furthermore optimized to use all the resources provided by the computer. My journey begins at a single threaded Java program and ends in a multi threaded C application able to make use graphics cards. Additionally, I will make an easy-to use yet powerful UI, which will enable even tech-unfamiliar people to use my software.

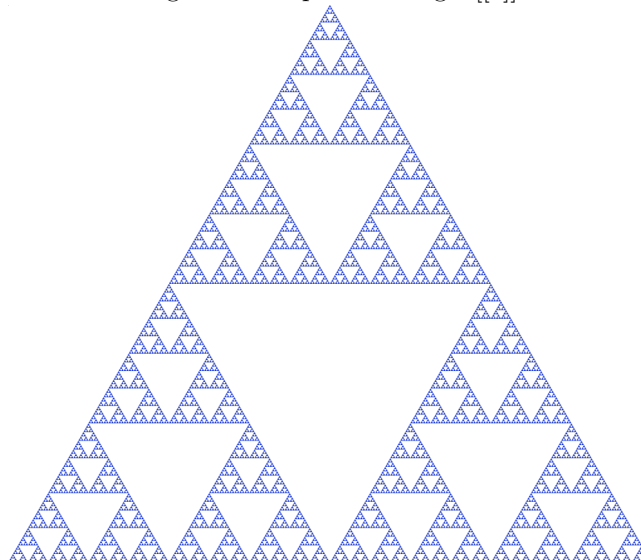
# 2 What are Fractals

Fractals are complex geometric shapes with special properties. But in contrast to normal finite Euclidean shapes (such as the circle, sphere, cube etc.), fractals are infinite. The angles and the lines of a cube are indifferent from the magnification. Fractals have the property that no matter how much they are magnified or zoomed into, the edges are never smooth but rough. New levels of detail will appear. Surprisingly, some fractals can even contain themselves. This property (well seen in the Sierpinski triangle, figure 1) is called self-similarity. Although the self-similarity in this example is perfect, many fractals contain non-perfect copies of themselves. This can often be observed in nature, for example tree branches or snowflakes (figure 2).

There are different types of fractals: geometric, algebraic and naturally occurring. Geometric and algebraic fractals are created by repeating a process over and over again. The Sierpinski triangle repeatedly cuts out the center piece of each black triangle 3. The algebraic ones iterate have to iterate over an equation to determine its shape.

While they were first conceptualized by Felix Hausdorff in 1918, the term fractal (from Latin fragmented, broken) was only coined in 1975 by mathematician Benoit B. Mandelbrot. A factor in this long time span were 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 were also found to be useful in various fields like physical chemistry, fluid mechanics and physiology. [1], [1], [2].

Figure 1: Sierpinski triangle [[3]]



# 3 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 Benoit 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, albeit my program has the ability to display them. The Mandelbrot set is an algebraic fractal in the complex plane  $z = a + bi$ . To find out which points are part of the set, we have to repeatedly apply the function  $z_{n+1} = z_n^2 + c$  to every point  $z$  in the plane. If the point diverges (or is known to diverge) as  $n$  approaches infinity, it is said to be outside the set. The most interesting points lie at the boundary of the set, as the edge is the most interesting part of any fractal.

Figure 2: Each branch of the snowflake creates new smaller branches [4]

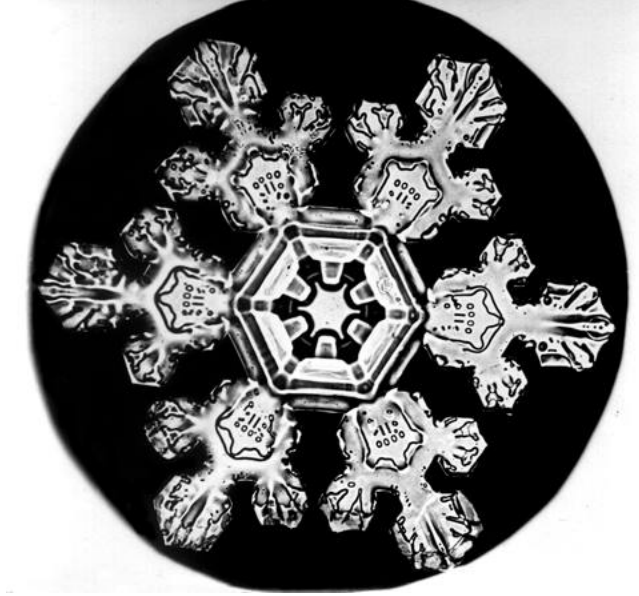
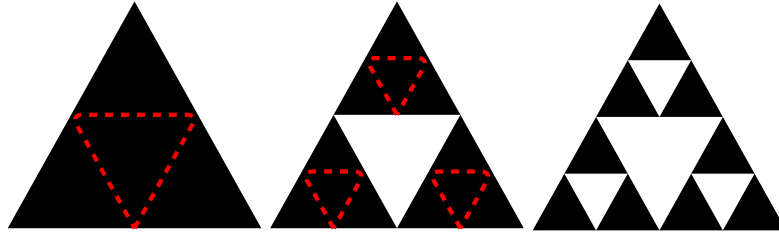


Figure 3: Creation of the Sierpinski triangle



## 4 The Program

This section represents the core of my paper. The capabilities, structure and implementation are discussed in depth

### 4.1 Program Functionalities

#### 4.2 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.

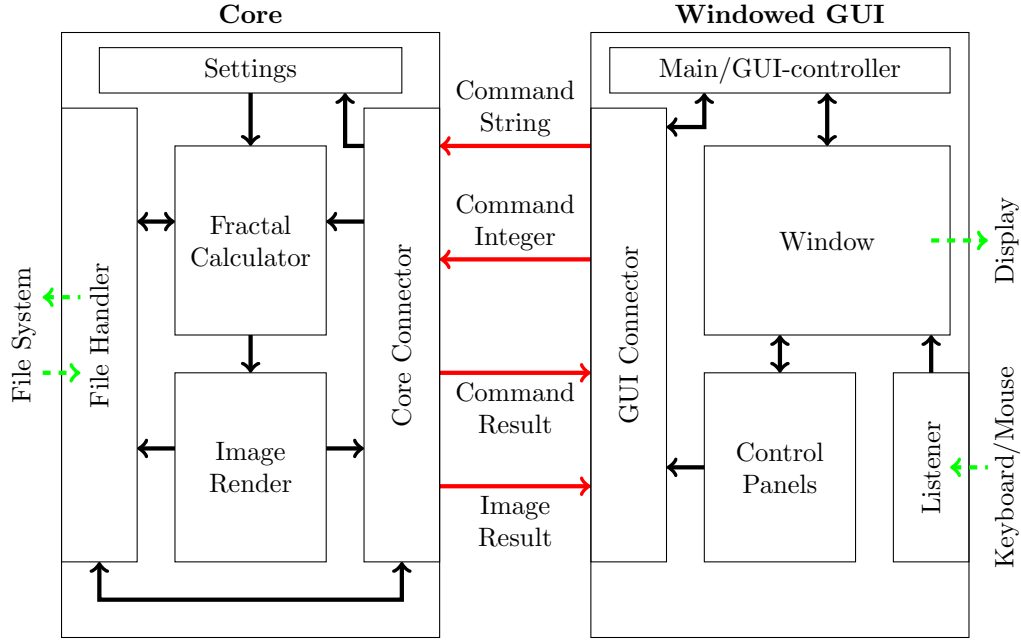
Because the core is detached from the GUI, I may also develop multiple GUIs for different needs. The GUI in the diagram below shows a possible version of window-oriented GUI, which is relatively modern and user-friendly, but for example useless in a command line environment. Thus I see the need to develop different GUIs.

##### 4.2.1 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.

**4.2.1.1 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 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

Figure 4: Program structure



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**.

**4.2.1.2 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.

**4.2.1.3 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.

**4.2.1.3.1 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.

**4.2.1.4 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.

**4.2.1.5 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.

**4.2.1.6 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 omissible 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.

Figure 5: Memory cells, with addresses in hexadecimal format

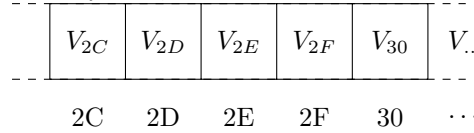
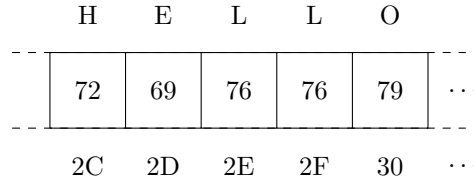


Figure 6: Memory cells storing a simple string



## 4.2.2 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.

## 4.3 Implementation

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

### 4.3.1 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.

#### 4.3.1.1 Region

#### 4.3.1.2 Logic Region

#### 4.3.1.3 Screen

### 4.3.2 Data Management

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

**4.3.2.1 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. In since a 64 bit number is quite long, they are usually represented in hexadecimal format. In figure 5 (page 4) there is a schematic drawing of five memory cells, each labeled with 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).

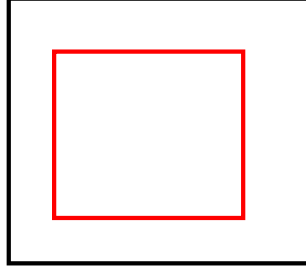
After having set up the cells, which I will not go into detail since it is not relevant for my work, values can be passed to the cells. In figure 6 (page 4), the string *HELLO* is stored (each letter corresponds to a numeric value, i. e. A equals 65, B equals 66 etc.). Since the characters are stored sequentially, it is not of importance to know the address of each letter; knowing the address  $A_{\text{start}}$  of the first suffices, as the following letter's address  $A_{\text{second}}$  can be calculated as follows  $A_{\text{second}} = A_{\text{start}} + 1$ .

Lists behave quite similarly to [5]

Figure 7: Function Signature to create images

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

Figure 8: The dataset region(black) and the requested region (red)



**4.3.2.2 Storing an Image in RAM** Since the RAM is a one dimensional space, allocating one dimensional objects like numbers, strings or similar things is quite easy. It suffices to know the length of the objects to know how much RAM has to be allocated, and figuring out which elements of the

### 4.3.3 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.

**4.3.3.1 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 7 (page 5), 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.

**4.3.3.2 Preparing the rendered Area** In figure 8, 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 9, page 5). 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, rendering a long search for the datapoints redundant. The precision of the level is always better or equal to the precision of the requested image.

Figure 9: The level, its clusters and the logic region in green

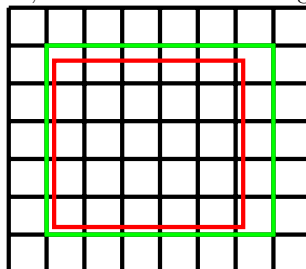


Figure 10: The logical area (green) with clusters and values (blue), and the requested region (red) with pixels

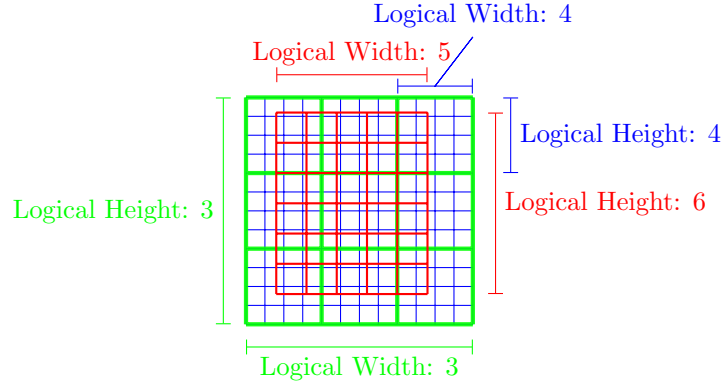
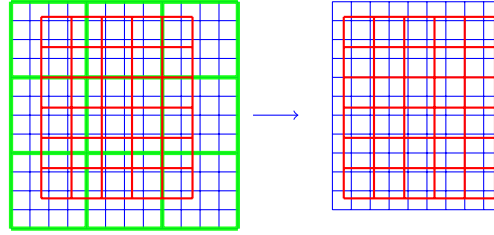


Figure 11: Cropping the image to the area given by the parameters



**4.3.3.3 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 10 (page 10). 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 4.3.3.4, page 6)

After the image has been populated by the values, it can be cropped to the requested area. But as seen in figure 11 (page, 6), 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.

**4.3.3.4 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 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.



#### 4.3.4 GUI

## 5 Workflow and Major Version History

## References

- [1] *title*. URL: <https://fractalfoundation.org/fractivities/FractalPacks-EducatorsGuide.pdf>.
- [2] *title*. URL: <https://www.britannica.com/science/fractal>.
- [3] *title*. URL: [https://upload.wikimedia.org/wikipedia/commons/thumb/4/45/Sierpinski\\_triangle.svg/1000px-Sierpinski\\_triangle.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/4/45/Sierpinski_triangle.svg/1000px-Sierpinski_triangle.svg.png).
- [4] *title*. URL: <https://en.wikipedia.org/wiki/Snowflake>.
- [5] *title*. URL: <https://computer.howstuffworks.com/c23.htm>.