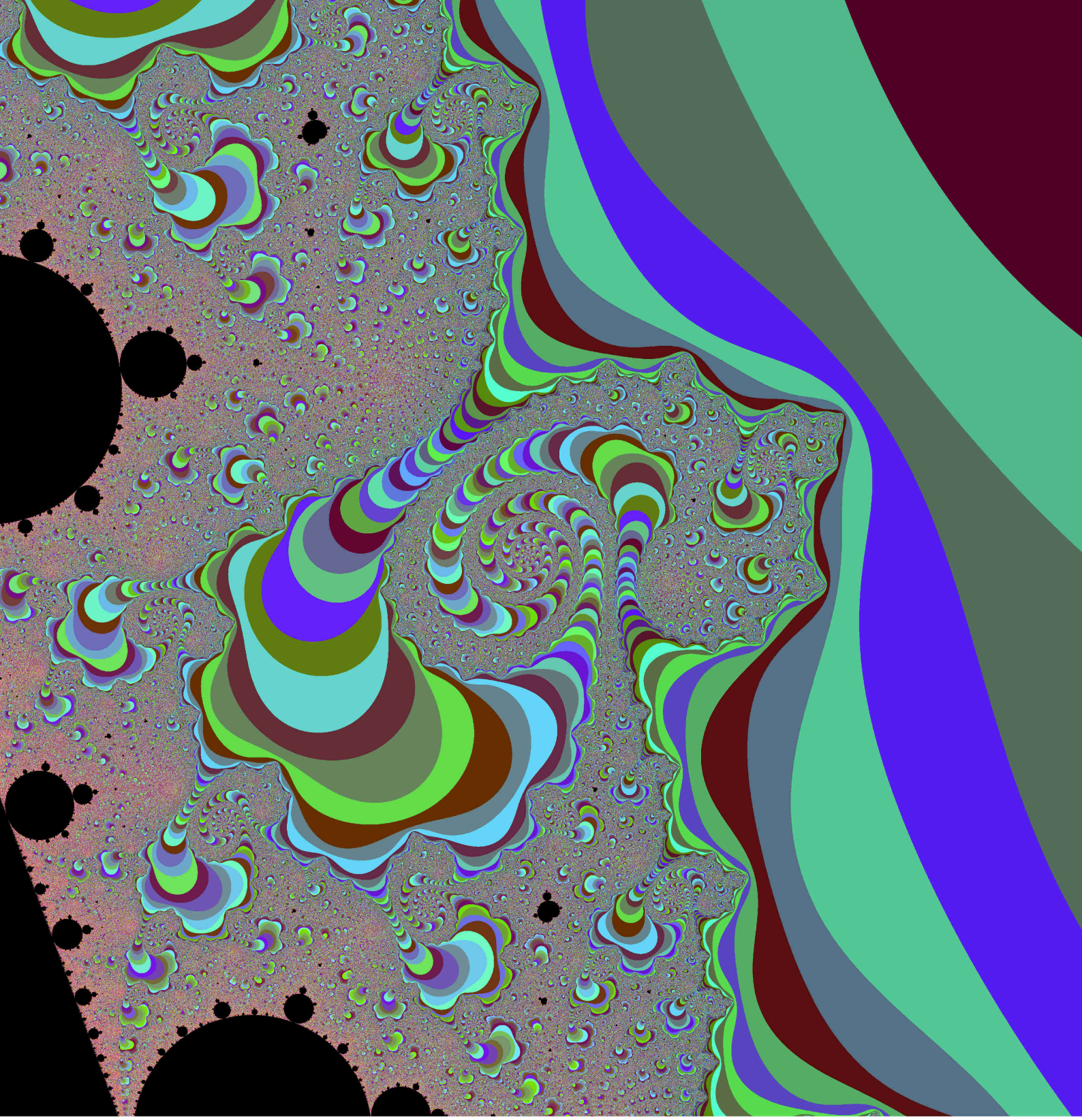


Programming of a Tool for Fractal Visualization in Java

Mikail Gedik, M6f

Maturitätsarbeit and der
Kantonsschule Zürich Nord

Niederweningen, November 2020
Betreuungsperson: Peter Holzer

Contents

List of Figures

1 Abstract

This paper describes the coding of a program in Java and OpenCL capable of calculating a few selected fractals and casts 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 2 (page 1) shortly describes the functionalities and the structure of my program, breaking it down into logical components depending what they are responsible for. The last section is dedicated to bugs, known issues and their workarounds, limitations and possible ways of improving of my program, followed by a reflection on my work.

2 What are Fractals?

Mathematicians used to rely exclusively on classical algebra to describe as well as research sets and functions. As classical algebra is only able to describe regular sets and functions, any irregular set could not by described and ended up being labeled as pathological, irrelevant and not being paid any further attention. 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 now known as fractals. While they were first conceptualized by Felix Hausdorff in 1918, it was only in the 1970s, after the advent of computers with their impressive computational power, that the exploration of fractals became much easier. The term fractal (from Latin *fragmented, broken*)) was coined in 1975 by mathematician Benoît B. Mandelbrot. Mandelbrot used fractals as a tool to examine the stock market, but they have proved 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 utilizing 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 *ad infinitum* 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 often only resemble the original approximately. Some fractals display a statistical self similarity, meaning that the copies 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 microstructure. The Sierpiński triangle can be magnified infinitely without ever showing a final structure which is not fractured itself. Additionally, the local geometry cannot be described by classical Euclidean (meaning shapes like circles, lines, spheres, cuboids and such) geometry; the structure is by far not adequately regular.
Contrary to its appearance, the definition of the Sierpiński triangle is thus quite simple. Furthermore, it is a recursive. Both these properties are often also characteristic of other fractals.
Although it lies outside the scope of this paper, the Hausdorff dimension is also worth mentioning. Broadly speaking, the Hausdorff dimension is a real number (as opposed to the topological dimensions, which are always integers) which describes how much space a set fills up and it is usually greater than its topological dimension [Fal93, Einleitung].

2.2 The Mandelbrot Set

Although most of my senior thesis revolves around computer science the necessity arises to explain the fractal crucial 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 also not a part of my work, although my program has the ability to display it. The Mandelbrot set $\mathbb{M}$ is an, unlike the previously mentioned geometric Sierpiński triangle, the Mandelbrot set $\mathbb{M}$ is an algebraic fractal in the complex plane $\mathbb{C}$. There are multiple definitions for the set, but the most relevant for my work is as follows:

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

Figure 1: Sierpiński triangle [Sie]

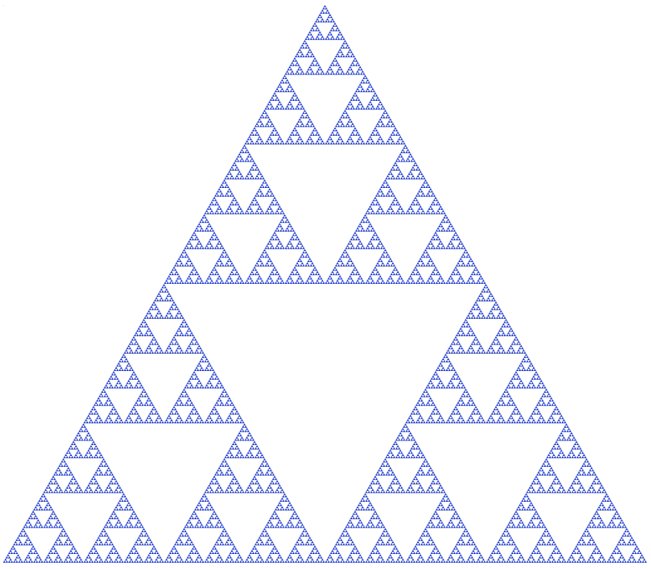
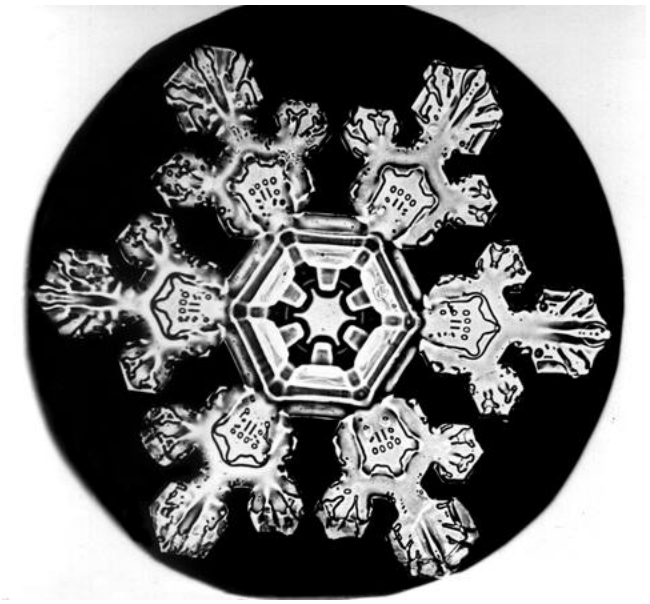


Figure 2: Each branch of the snowflake creates new smaller branches [Wik]



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 more generally $z_n = f_c(f_c(\dots f_c(0))) = f_c^n(0)$. The example below may be more easily 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 any other c it may take hundreds of thousand of iterations to diverge. Whilst no proof will be provided, since it lies outside the scope of this paper, it can be shown that $\lim_{n\to\infty} z_n = \infty$ if $|z_k| > 2$ for any k, meaning that further iterations are not necessary to determine that a point is not part of the set. Theoretically, a point has to go through an infinite amount of iterations to prove that it is part of the set and does not diverge. To complete 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].

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

Figure 3: Creation of the Sierpinski triangle

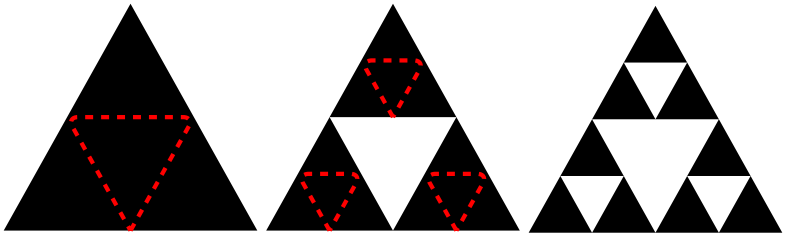


Figure 4: Iterations of different c

	$c = 0.3 + 0.2i$
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	$-9.35996 \times 10^{18} - 2.2269 \times 10^{19} i$

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. Once the render engine has been started, the user can move around and zoom into the fractal. Should an interesting spot be identified, the user 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 offers 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

Programs are usually programmed in a divide-and-conquer manner. Instead of implementing the project as a whole, it is split in smaller non autonomous sub modules. This has several advantages. Because the code is grouped in smaller modules, the project is easier to maintain over time as specific pieces of code can be found easier when the need to edit them arises. Also, other coders will understand the code better if it is structured practically.
Another perk is the ease of debugging. The modules are like links tied together in a chain. When the program has to do something, the first link receives an order to do what it was programmed to and passes an interim result to the next link, which does another calculation composing a new interim result. This process is repeated until the end of the chain is reached where the final result is created. A single faulty link out of thousands will likely screw up the entire procedure. The search for the culprit, alias debugging, is much easier when the code is not written in a single big block, because the interim results can be inspected, betraying any faulty links. It is also possible to test a single modules (unit testing) instead of the whole program.
Figure 6 (page 4) shows how the code is grouped and structured in smaller components, as well as how they interact with each other. The harshest division is between the core, which encapsulates the logic to calculate, store and render data, and the GUI (graphical user interface) which offers a window for the user to interact with. This is because a programmer may wish to use the calculation capabilities in his own program and no GUI is needed. This way the GUI can be easily omitted.
The next section will only briefly describe how the single modules work, starting with the core components and then moving on to the less complex GUI. An in-depth discussion will follow in section 3.3 (page 5).

Figure 5: The Mandelbrot set

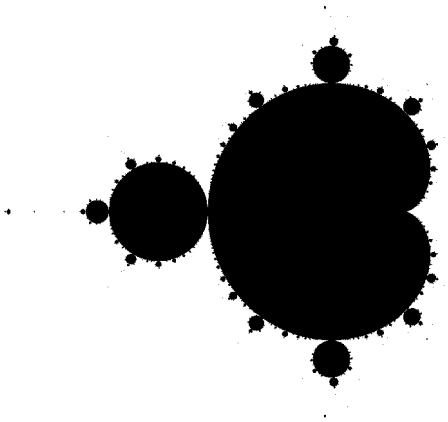
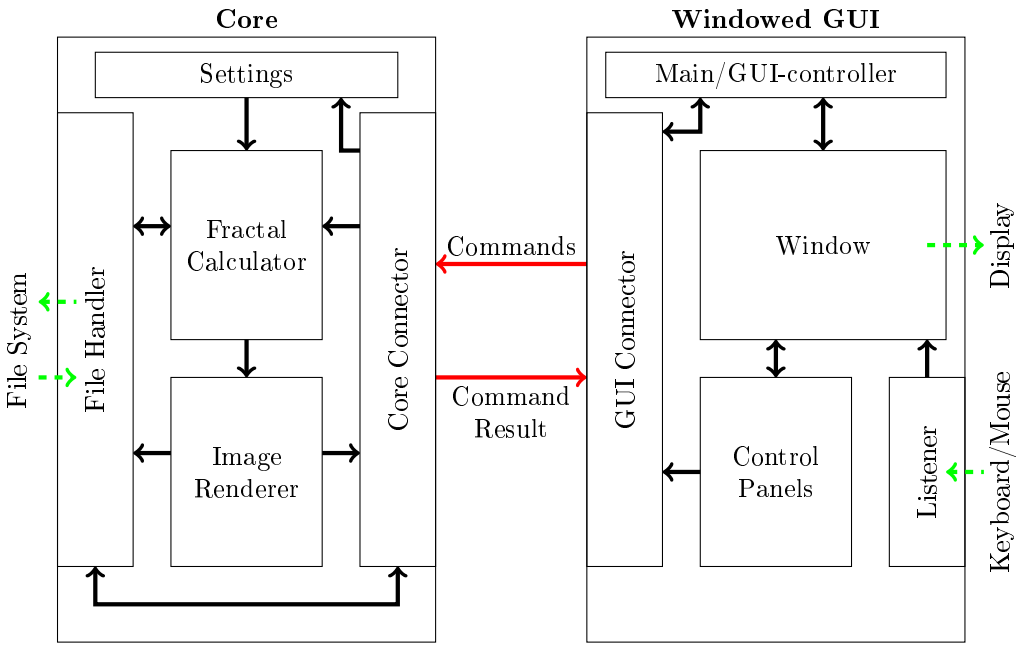


Figure 6: Program structure



3.2.1 Core

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

3.2.1.1 Settings The settings are configurations used either directly during computation or as parameters for the program itself. Variables used during the computation could be for example the maximal number of iterations allowed or the color function used when rendering the data. The settings can only be changed before the actual fractal calculator is started.

3.2.1.2 Database The data model is the one of the most complex parts of my project, as the data has to have low access times to lots of data. To achieve this, the data is logically split multiple times. The topmost containers are called levels and sort the data by precision (more to that in section 3.3 (page 5)). They do not contain the data directly but instead are made up by clusters, which store small chunks of data sorted by location. The size of these chunks is determined by the settings.

3.2.1.3 Fractal Calculator The calculator determines if points are part of the set, and if not how many iterations it takes to be sure that the point is not in the set. It contains the logic to harness the power graphic cards provide using OpenCL and can even connect to other computers to use their computing power as well.

3.2.1.4 Renderer The renderer converts raw data to an image. The amount of iterations a point has gone through before it escaped the set is mapped to a specific color depending on the color function defined.

3.2.1.5 Connector The connector conduces all the components and provides the API to control the core, meaning that the GUI can issue commands only through the connector. This adds an extra level of abstraction, as the GUI does not have to worry about how the core calculates data and only has to request an image.

3.2.2 GUI

The GUI is the face my program has when utilized. Although most of my efforts have gone towards developing the core, a stable GUI still had to be developed. The application consists of a simple main window and an occasional popup when user input such selecting a file is required.

3.2.2.1 Login Upon starting the application, a login screen is displayed. The user can adjust the settings by opening the settings dialog and edit the calculation and render kernel if he has the knowledge to. It is also possible to start the application as a slave.
When the start button is pressed, the user can select which computer parts are used to render the fractal. Usually, only installed GPUs will show up as an option, but machines with OpenCL support on the CPU will also show a CPU. If the program accepts external slaves they will also show up as options. After confirming the selection the program changes to the master view. If the slave option was selected, the program attempts to connect to the specified master and shows the slave view, which is empty.

3.2.2.2 Master View The master view consists of the fractal in the middle and a menu bar on top containing entries to export images or videos and save calculated data. By left clicking and dragging the mouse the user is able to move around. Dragging using the right mouse button selects an area which will be magnified. It is also possible to zoom using the mouse wheel. Alternatively, the viewport can also changed via the menu bar.

3.3 Implementation

This section discusses the functionalities in greater depth, although without going into the Java specific implementations, i.e. no classes or class methods will be mentioned.

3.3.1 Core

3.3.1.1 Dataset The connector passes the data generated from the calculator to the dataset where it is stored. The data consists of points (or mathematically speaking complex numbers) which each have a value corresponding to the amount of iterations the program has applied to a point. To facilitate further use, the point and its value are from now on grouped in a structure called entry. Table 7 (page 6) shows three entries with values. The -1 denotes that the point did not diverge and is thus in the Mandelbrot set.

Figure 7: A table containing entries

Point	Value
$0.3 + 0.3i$	-1
$0.4 + 0.2i$	31
$0.4 + 0.5i$	7

3.3.1.1.1 General Understanding This section gives an insight on how the data is sorted in the dataset without going into mathematical details, which will done in the following sections 3.3.1.1.3 (page 6) and 3.3.1.1.2 (page 6). Figure 8 (page 7) illustrates the separation of points of the set (green) into clusters (red) and levels (blue). The topmost square shows the entire "raw" dataset and the three following squares each show a level with clusters.
To understand the model further the purpose of the data has to be considered, which is generating images. Broadly speaking, every pixel represents a point in the dataset. Because all pixels have the same distance to their neighboring pixels all points needed have the same distance to their neighboring points. It is thus sensible to sort points by their distance to their neighbors. This makes it possible to quickly query the necessary points when an image is rendered instead of having to search the entire dataset. This is put to practice by the so called levels.
Furthermore, the image often only shows a part of the area the dataset spans over, meaning that only the points in that part of the area are needed. The image creation process will thus mostly need a part a level's points. Since the points needed are also always close to each other, it makes sense to group the points by their location in small chunks, called clusters in my program. Their advantage is that they can be queried much faster than single values.
I take the limitations of this dataset model beforehand: It can only store points which are in the bounds of the dataset and lie on an invisible grid. The dataset requires that data added fits into the present structure. It might seem a bit baffling that it is only capable of storing certain points, but for the purpose needed this actually facilitates the image generation process because the data can be directly mapped to an image, see section 3.3.1.3 (page 9).

3.3.1.1.2 Levels As previously mentioned, a level is a subset of a dataset whose points all have the same distance to their neighbors. This distance is called precision p. Naturally the clusters inherit this property from the level they are in. As seen in the image, levels are identified as L_d where $d \in \mathbb{N}_0$ is the depth of the level. The depth is another way to describe p, but the link between them can only be defined exactly later as soon as the levels and clusters are more fully explained.
The dataset has a fixed dimensions and a fixed position. In Figure 8 (page 7) the dataset has a width and height of 4 and its center lies on the axes' origin. All levels inherit this properties from the dataset. The position is named $(l_x), (l_y)$. While it might seem intuitive that the position denotes the center of the levels and dateset it is more useful to store the lower left corner. In this example the position is thus $(-2, -2)$. The width and height are denoted as $l_{a,w}$ and $l_{a,h}$, where the $_a$ stands for *absolute*. However the width and height can also be described by the amount of clusters a level contains: $l_{l,w}$ and $l_{l,h}$, where $_l$ stands for *logical*, define the amount of clusters the level has horizontally and vertically. The amount of clusters a level has is thus equals to $l_{l,w} \times l_{l,h}$. *Logical* means that the measure is an actually "invented" used to describe levels, whereas *absolute* refers to the classic width and height of a area covered by the level.
Next the definition for $l_{l,w}$ and $l_{l,h}$ is due. As seen in Figure 9 (page 7), the logical width and height are multiplied by a factor for each increment of d. It can thus be written

$$l_{l,w}(L_1) = f \times l_{l,w}(L_0)$$

$$l_{l,w}(L_k) = l_{l,w}(L_0) \times f^k$$

$$l_{l,w}(L_1) = f \times l_{l,w}(L_0) = f^2 \times l_{l,w}(L_0)$$

where $f \in \mathbb{N} \setminus \{0, 1\}$ describes by which factor $l_{l,w}$ grows with each level. f, $l_{l,w}(L_0)$ and $l_{l,w}(L_0)$ are all given by the settings upon initializing the database. It should be trivial that the formulas for $l_{l,h}$ are the same as $l_{l,w}$. The clusters can be uniquely identify by the level they are in and their position. They thus store the depth of the level they are in and an index marking their position in the level. Figure 9 (page 7) shows how the clusters are numbered. If k and h are the row and column of a cluster $C_{h,k}$, the index i of a cluster C_i can be described as $i = h + k \times l_{l,w}$. It should be trivial that $0 \leq h < l_{l,w}$ and $0 \leq k < l_{l,h}$ since the clusters must be inside the level.

3.3.1.1.3 Clusters The clusters group points in an area into small chunks. Figure 10 (page 8) shows that their width and height are defined as $c_{a,w}$ and $c_{a,h}$. The absolute position (c_x, c_y), meaning the lower left

Figure 8: An exemplary dataset with L_0 to L_2

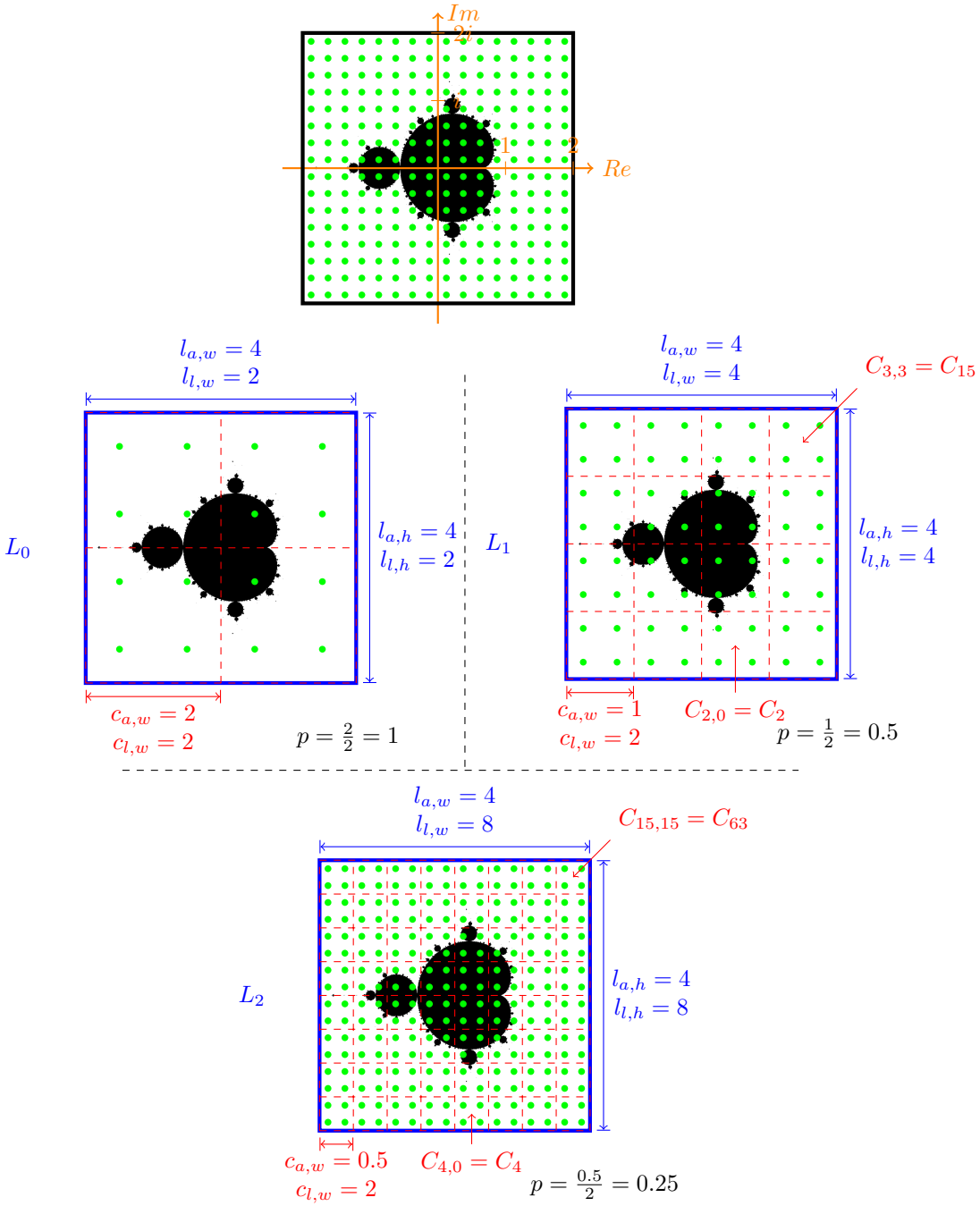


Figure 9: The relationship between different levels with $f = 2$

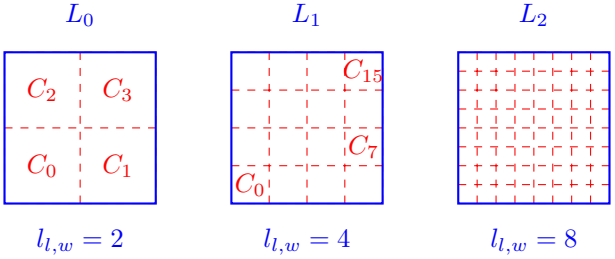
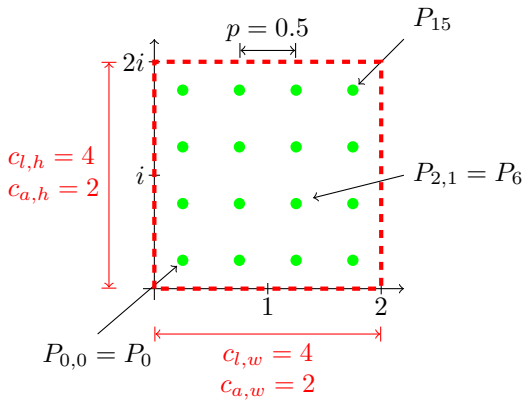


Figure 10: Points in a cluster



corner, of a cluster can be described as $c_x(C_i) = l_x + (i \mod l_{l,w}) \times c_{a,w}$ and $c_y(C_i) = l_y + (i \mod l_{l,h}) \times c_{a,h}$. These two formulas first get the row and column of the cluster and then transform them absolute coordinates. The logical behind it should be trivial.
Analogous to the levels, clusters have too logical dimensions. $c_{l,w}$ and $c_{l,h}$ refer to the amount of points they are wide and high. As seen in Figure 10 (page 8), clusters index the points they contain as well. This procedure is analogous to the indexing of clusters. The index i of a point P_i can be described as $i = h + k \times c_{l,w}$, where h is the column and k is the row of the point do describe. The position $(p_x|p_y)$ of a point i can be described as $p_x = c_x + (i \mod c_{l,w}) \times p$ and $p_y = c_y + \lfloor (i/c_{l,w}) \rfloor \times p$.

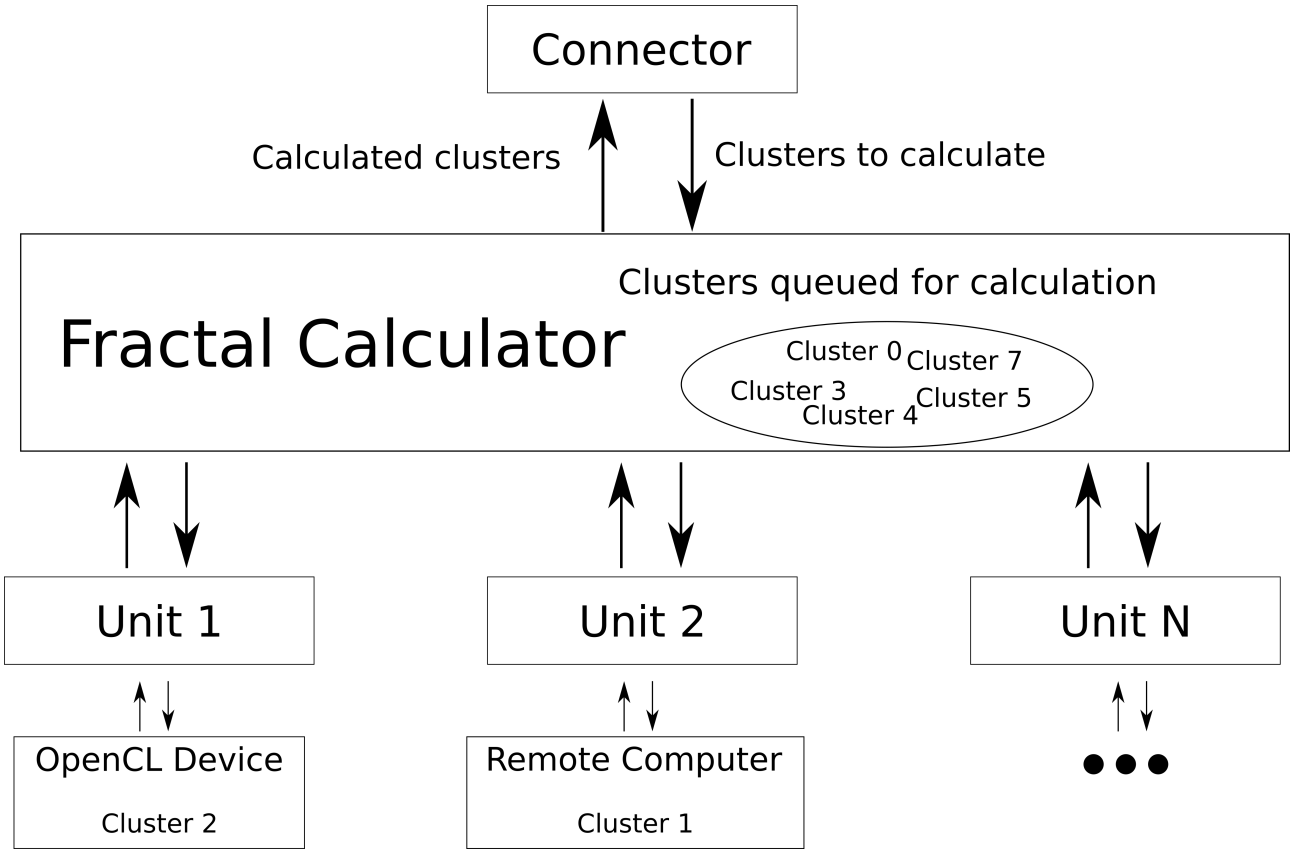
3.3.1.2 Fractal Calculator The workload which is done by my program is highly parallelizable: each point can be calculated on its own, without any dependence on other points. Such task are usually not performed on a CPU, since they are specially optimized for fast serial computations and suffer from the inability of only taking little advantage of parallization. The component suited for that is the GPU, which can usually compute hundreds if not thousand of operations simultaneously. It is also possible to distribute the work among multiple devices, should as system have more than one GPU for example.
Java however has no native way of accessing GPUs, and I thus had to fall back onto another programming language, OpenCL. OpenCL (Open Computing Language) is able to address GPUs and use them for computations. The caveat however is that OpenCL works on a C based design, meaning that I as a programmer have to worry much more about coding than I would in Java, where some work is done by the running JVM, especially regarding memory management. It is to note that not every CPU or GPU has proper drivers supporting OpenCL, so my program may not run at all on some system.
Figure 11 (page 9) shows the calculator from the inside. The connector sends the clusters which need to be calculated to the calculator manager, which distributes them on the calculator units.

3.3.1.2.1 Calculator Units To facilitate the usage of OpenCL devices, each of them is wrapped by a calculator unit. The calculator unit takes care of the manual management required by OpenCL, such as initialization of the device, sending a signal to start the calculation to the device and querying the result back to Java once the calculation has completed.

3.3.1.2.2 Calculator Units Management To start the calculation a list of clusters to create is passed to the calculator. If only one unit exists, which is the common case on computers, it can sequentially get every cluster, compute it and mark it as done. However, things get messier if multiple units exist. Assume that two units try to simultaneously get the next cluster to calculate. This would inevitably lead to a crash or unintended program behavior. To eliminate that risk the calculator has to synchronize access to and from the calculator units.

3.3.1.2.3 Remote Calculator Units If the power of the current machine is not sufficient for the calculations, the work can be split over other machines via Internet. One computer then acts as the "master", which organizes the calculation. All other machines are "slaves" and connect to the master to offer him their services. From the calculators perspective the slaves are calculator units, and thus sees no difference between a remote machine and for example an installed GPU. The units wrapping remote computers have of course a different implementation than the normal ones. They have to handle the connection and make sure that data is sent and received correctly.
One central thing to consider when using remote connections is the latency. While commands can be sent almost

Figure 11: Schematic of the calculator



instantly to a GPU it may take many milliseconds to send it to another machine over the Internet, which is a lot on a computer that can perform hundreds of operations in just a picosecond. It is thus crucial to send as little data as possible. This can be achieved by buffering data, which means that the program avoids sending many small datapackets but instead sends few big ones.

3.3.1.3 Image generation

3.4 Internal Process Course

4 Reflection

I would like to use this last section of my work to add my personal thoughts to my senior paper and to neatly close the discussion around my work.

4.1 Limitations, Bugs and Workarounds

Programs naturally come with hidden bugs and limitations. As my program became bigger more bugs to be squashed arose. I have due to time issues not been able to tackle every one and thus am obliged to shortly describe the bugs. One bug is also induced by factors outside of my range such as driver support.

4.1.0.1 OpenCL on Macs Apple decided to stop supporting OpenCL on its devices [App]. Although the program detects the GPU and CPU as OpenCL devices, only the latter works.

4.1.0.2 Magnification Limit Every cluster C_i has an index i to store their location. This index is stored in a Java `int` which can usually only assume values up to $2^{64} - 1$. If the program tries to allocate a cluster with a higher index the program crashes due to a numerical overflow. This happens when $l_{l,w} \times l_{l,h}$ is greater than the aforementioned value because $i = a + b \times l_{l,w}$ where $0 \leq a < l_{l,w}$ and $0 \leq b < l_{l,h}$.
If due to this it is no possible to render a specific area the following workaround is recommended: restart the application, but set the levels' position and dimension to the previously unavailable area.

4.1.0.3 Memory Constraints

As the program calculates new values it stores them in RAM and VRAM, which both are limited in size. Without freeing any RAM during runtime the program is bound to try to allocate more RAM when no more is available, leading to a crash caused by an `OutOfMemoryError`.

4.1.0.4 Floating-Point precision

Floating point math on computers is if no special workarounds are used always tied to the machine precision, meaning that small rounding error occur during calculations. While the error is negligible on normally used numbers it can have devastating effects when the numbers become too small to be represented correctly. In my program this happens when the precision p becomes smaller than 10^{-13}, causing the values to be miscalculated.

4.2 Workflow

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.

4.2.0.1 Structure Design

The design as it is present now was the result of a long journey. At the very beginning it consisted of a simple program that calculated points of the Mandelbrot set, wrote the result into a file and then quit. It was all in just one file, without any logical structure. Because of the small size changes were easily introduced. However, new functionalities naturally increased the programs complexity by many times. The code was grouped in multiple modules which each took over a function. The advantage of modules is the ability to make internal changes to one without affecting other modules. This however does not hold true for any changes to the overall model, since this affects all modules. This happened quite often as I made many model changes due to me having taken either shortsighted decisions or simply making a faulty designs, costing me much time. The most extreme cases were the redesign of the database and the introduction of OpenCL.
The first database created was similarly to the current cluster based, but had serious design flaws related to memory usage as it would reserve lots of RAM for data that was not guaranteed to be calculated. Because all other crucial components depend on the database's implementation, they were all forced follow the changes made there.
Since I had never programmed in OpenCL before I had no knowledge what kind of structure would suit the addition of OpenCL supported calculations. Upon its introduction to my code is was still only halfway aware of how worked, making the design choices based solely by best guesses. Since OpenCL does not use the same RAM the program the additional challenge arose manually manage, i.e. allocate and free, RAM in a C like style, which is ironic since one of the reasons I coded in Java is to not have to worry about memory allocations and deallocations.
Another time stealing mistake of mine was the opposite making a design not future proof. Thinking too big I made parts of the model more complex than they ultimately needed to be. An example is the way a points value is represented. Whereas it is just a simple `int` in the final version it used to be more class, which made it more cumbersome to perform operations on it and also resulted in a slight performance penalty.

4.2.0.2 Squashing bugs

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 by using assertions. They are checks done at runtime to detect any illegal or senseless program states. If the check fails an `AssertionError` is thrown to let the programmer know that there are issues in his code. I have used these extensively in combination with the OpenCL API.
Assertions alone however are not enough. Sometimes it is useful to pause the execution and inspect the program state manually. This can be done by attaching a debugger to the running program. The debugger is able to pause the program execution at any moment and lets the programmer see the current state. It is also possible to set breakpoints, which cause the debugger to automatically pause execution when such a point is reached. Furthermore, it also pauses when an exception is about to be thrown. This was the most practical use for me since I was able to examine the cause leading to the exception.
The Java debugger however has its limits: It does not work on OpenCL code. This was quite inconvenient because the entire JVM crashes without any useful information if faulty parameters are passed to OpenCL functions, making the search for bugs quite cumbersome.

4.3 Possible enhancements

The following sections shortly describe what features I would have added to my program if I have had any more time. Neither of them are crucial for my program but each of them would nonetheless bring more stability or comfort.

4.3.0.1 OpenGL Although the Java Swing Toolkit which is responsible for the GUI has longtime been replaced by newer alternatives I have decided to use it because it is the only type of GUI of which I am able to program. As of now, the image data has to be passed from OpenCL to the Java Swing application, resulting in a high latency and waste of resources. Had I had more time I would have made effort to implement an OpenGL GUI, which has features to work directly with OpenCL, saving valuable computing power. Another caveat is that since OpenCL works with native machine code it is more responsive and faster than the in any Java code.

4.3.0.2 Memory to Disk As mentioned in section 4.1.0.3 (page 10), the memory is a main cause of trouble when the application runs for long a duration. A possibility to combat this problem could be to offload unused data onto the disk, which as - in comparison to RAM - more than enough free space. The caveat here is to determine which data will probably not be used in the near future.

4.3.0.3 Dynamic Iterations A question left unanswered in my senior paper is the amount of iterations a point should go before it is considered part of the Mandelbrot set. The general rule of thumb is that the smaller the clip of the fractal is i.e. the closer the points to calculate are the more iterations are necessary to see a clear picture. In my program however the amount of iterations per point is indifferent to the distance between the points d. Dynamically changing the maximal amount of iterations would drastically speed up the calculation of clips with a lower resolution and increase the quality of clips which have zoomed in greatly.
This feature is already for the most part implemented in the program making it possible to give levels which are deeper more iterations. However I have not made any efforts to improve this feature because I simply lacked the time to investigate about how to determine the appropriate amount of iterations for a point.
The default iteration model is named “static” and uses the same amount of iterations for every level. A not thoroughly tested variant named “antiProportional” can be selected in the settings, which increases the amount of iterations anti proportionally.

4.4 Conclusion

As described section 4.3 (page 10) I was not able to implement every aspect I wanted to be included. However I can proudly present a program that meets all basic requirements I have given, such as being able to harness the power of a graphics card or exporting videos. Looking back at my work I can say that I made the mistake of investing too much time in the program and by far not enough in the paper. I had not thought that writing down my ideas in an understandable way was going to take way less time. Lastly I would like to thank Mr. Holzer for giving advice, guiding and supporting me through my first greater piece of work.

References

[App] *OpenGL, OpenCL deprecated in favor of Metal 2 in macOS 10.14 Mojave*. 2018. URL: https://appleinsider.com/articles/18/06/04/opengl-opencl-deprecated-in-favor-of-metal-2-in-macos-1014-mojave.

[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.

[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.