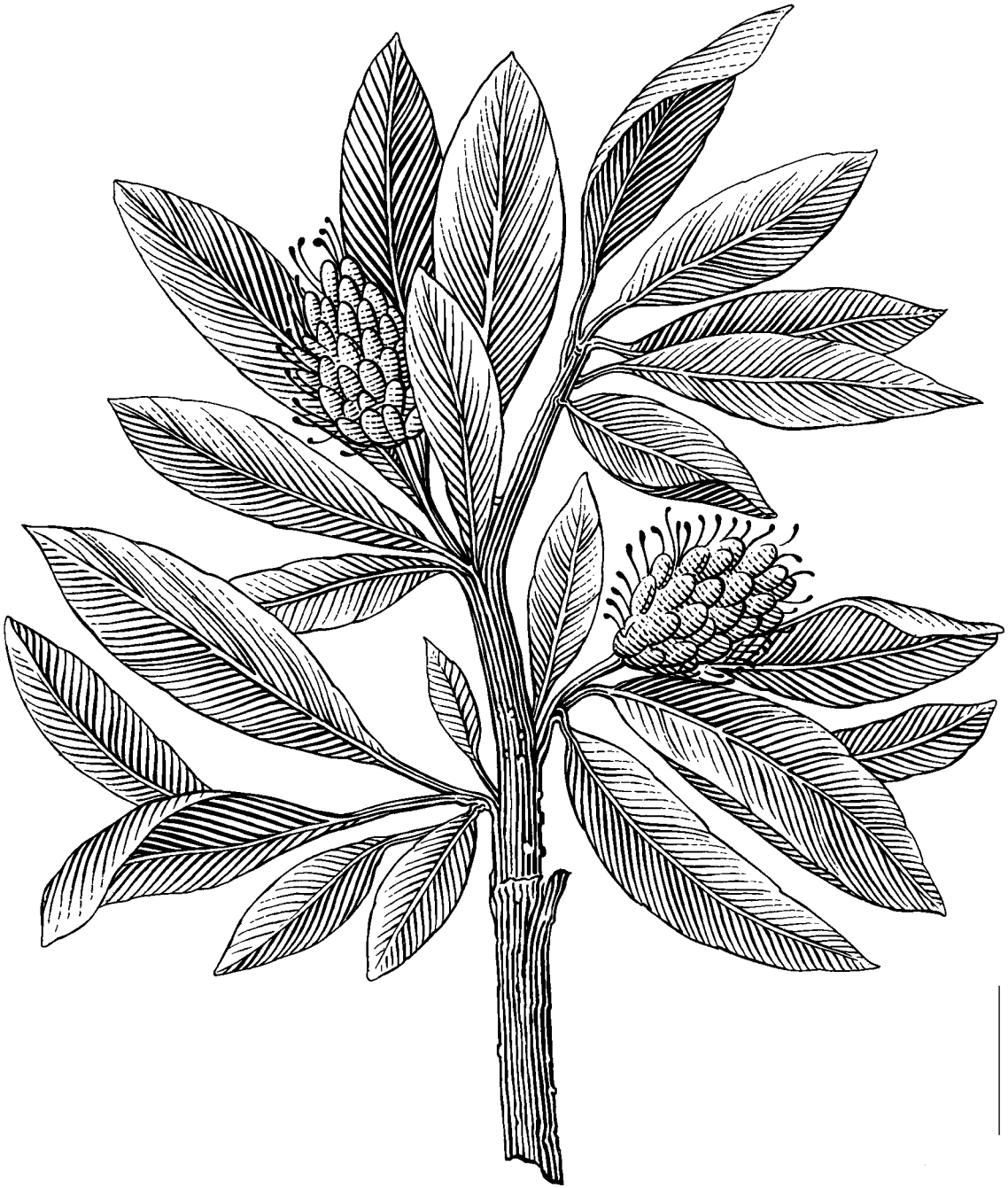
Linnæus University
School of Computer Science, Physics and Mathematics

Degree project

Creating Music Visualizations in a Mandelbrot Set Explorer



Author: Christian Knapp
Date: 2012-08-03
Subject: Computer Science
Level: Bachelor
Course code: 2DV00E

Abstract

The aim of this thesis is to implement a Mandelbrot Set Explorer that includes the functionality to create music visualizations.

The Mandelbrot set is an important mathematical object, and the arguably most famous so called fractal. One of its outstanding attributes is its beauty, and therefore there are several implementations that visualize the set and allow it to navigate around it.

In this thesis methods are discussed to visualize the set and create music visualizations consisting of zooms into the Mandelbrot set. For that purpose methods for analysing music are implemented, so user created zooms can react to the music that is played.

Mainly the thesis deals with problems that occur during the process of developing this application to create music visualizations. Especially problems concerning performance and usability are focused.

The thesis will reveal that it is in fact possible to create very aesthetically pleasing music visualizations by using zooms into the Mandelbrot set. The biggest drawback is the lack in performance, because of the high computation effort, and therefore the difficulties in rendering the visualization in real-time.

Keywords: Mandelbrot set, music visualization, GPGPU programming

Contents

List of Figures

List of Tables

1 The problem

This section will explain the motivation behind the topic of this thesis, as well as outline the basic problem and structure of the thesis.

1.1 Problem and Motivation

The Mandelbrot set is a mathematical object, a set of complex numbers to be precise, that was discovered just about 30 years ago, and is ever since playing a major role in today's mathematical research.[10] Making it imaginable for everyone just was possible by the development of powerful computers, that were able to render the set. Basically it is decided if a complex number is part of the set by observing the influence of that number on a dynamic, mathematical process (this will be revealed in detail in section 2.1), and the set can then be represented by coloring the complex number being part of the set in a complex plane. [10] These visualizations of the set offer interesting and aesthetically appealing images. A simple visualization of the set can be seen in figure 1.1, a more colorful one in higher magnification in figure 1.2.

There are several existing tools allowing it to explore the Mandelbrot set, these tools are consequently called "Mandelbrot Set Explorers", or more general "Fractal Explorers" (see section 3.1 for an overview). They help users to get an idea of the infinity of details and complexity of the structures that are generated by a basically simple process. It became a hobby of many people to create images of some exceptionally beautiful parts of the set. But many of these existing tools lack at least in one attribute, may it be performance, offered features, or simply usability for the non-advanced user.

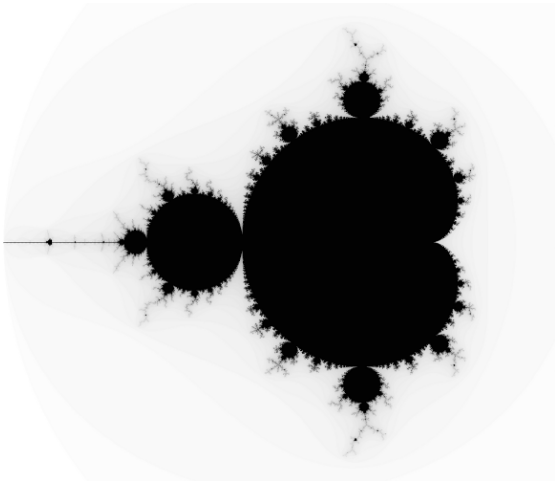


Figure 1.1: Entire Mandelbrot set, simple black to white coloring

Moreover, when people are seeing the zooms or creating videos of zooms into the set, it is quite natural that they connect it with music, because it immediately reminds one of the typical music visualizations such as MilkDrop[15] or its open reimplementation ProjectM[16]. Mandelbrot set zooming videos are typically rendered in advance, because even today, when zooming far into the set, computational power is a limiting factor and advanced and deep zooms can take hours and days to be rendered.

The next step of connecting these Mandelbrot zooms with music is to render it real-time and synchronize it with an arbitrary piece of music. Today's hardware is at least strong enough to handle this real-time calculation of the set up to a certain threshold. But this job is not trivial, because it needs a high amount of optimizations and some tradeoffs to reach real-time rendering.

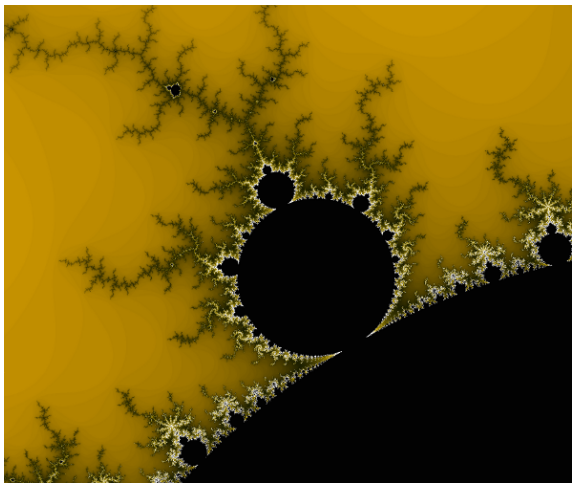


Figure 1.2: Mandelbrot set at magnification 10 with goldish coloring

1.2 Goal

The first aim of this bachelor's project is to create a Mandelbrot Set Explorer with focus on easy usability and on good support to create customizable images, especially by offering a good editor to modify the used coloring.

Easy usability implies an intuitive way of interaction with the set to navigate around it and to reach interesting locations in the set. It also means that for users, who are not experts in any related field, an easy interface should be provided that also gives them feedback about what they are causing with their actions. But that does not mean, that only few features should be offered to keep the tool simple, because advanced users should also find pleasure in using advanced interactions.

High customizability means, that it should not only be possible to change the location in the fractal, but especially the colors should be adaptable in an easy and intuitive way and a high variety of coloring modes should be provided to create different kinds of images, even if they show the same part of the same object.

Besides these two attributes, the tool should also satisfy other basic requirements, such as a convenient performance to make it possible to explore the set if someone does not own a high-end computer. The current development in computer hardware leads to a more and more parallel approach, which is of high benefit for the purpose of this thesis. The calculation of the Mandelbrot Set is highly parallelizable, and can therefore benefit a lot from multi-core CPUs, and especially from general purpose GPU programming. That is why the option will be provided to use the GPU for accelerating the calculations.

But it does not end there. The final aim of the project is the possibility to create zooms into the set, that can then be stored and played back. The zooms should again be highly customizable, with color changes on the way and different other features.

These replays should then be connectable with music. The zooms should have some parameters, e.g. speed of the zoom, used colors and speed of color rotations. These parameters then change according to some characteristics of the played music. For instance the colors of the set could change on every beat that occurs in the song.

1.3 Restrictions

The application developed in the course of this bachelor's project could develop to a large tool rich of many features. Looking at some commercial fractal explorers it becomes clear, that it is impossible to reach the amount of features they offer in an application that is written in a short time period by one single person. So it should also be clear that this tool will not compete with professional tools and also some free tools that are being developed for a long time.

The performance of current high-end computers is raising exorbitantly fast, so the possibility of the real-time rendering of the set is given. But still, the effort to be made is extremely high, so it cannot be guaranteed that real-time rendering is possible for high magnifications or on older hardware.

The application will for the moment be restricted on calculating the Mandelbrot set and no other fractals, to keep the complexity low.

The report will not focus on the theoretical background of the Mandelbrot set, but give a short introduction into the topics of fractals in section 2.1 to make the idea clear. The same is the case for music analysis, an overview of the possible techniques and an explanation of the used ones will be given in 2.3, but the topic will not be discussed in detail. The thesis should not focus on mathematical theory, but on the implementation of this particular application.

The implementation of the application is a first prototype implementation, because the implementation effort will be big. Due to that the fulfilling of the aimed features will be measured only by providing the application prototype that will show that the features are implemented. Concerning usability there will be no usability testing yet because of the restricted scope of the report.

1.4 Structure of the thesis

This thesis should provide an overview over some basic underlying concepts of the topic, and it should give a deeper explanation about the implementation process of the application and arguments about decisions that were made during this implementation process.

This first section is an introduction to the problem and how the idea about this project had been developed.

The second section is the theoretical part, that explains some concepts around which this thesis is built, namely fractals, music analysis and GPU programming.

The third section will give an overview about related works, in particular the section will introduce existing Mandelbrot Set Explorers and compare them in consideration of features that will be important for this project's implementation.

The fourth section points out basic requirements that should be fulfilled in the course of the implementation. Moreover, it should give a brief overview about the applications architecture and the used software technologies.

The fifth section discusses details about the implementation. It gives an overview about the main implementation features, how they were implemented, what deci-

sions where made during the implementation process and why this decisions were made.

The sixth section is a short discussion about the outcomes of the project.

And finally the seventh section gives an outlook on what could be attached in the future to this work.

2 Theoretical Backgrounds

This section will introduce the reader to the main theoretical backgrounds of the thesis, namely the theory of fractals and the Mandelbrot set in particular, some concepts about GPGPU programming, and possible ways to analyse music.

2.1 Mandelbrot Set Theory

This section introduces the basic theory about fractals in general and the Mandelbrot set in specific.

2.1.1 Feedback Processes

For developing the further principles of fractals, it is necessary to understand a basic underlying concept, named feedback processes.

A feedback process describes an outcome of a process depending on its previous state. One iteration step processes an input value to generate an output, and this output is reused as input for the next iteration step, as shown in figure 2.1.

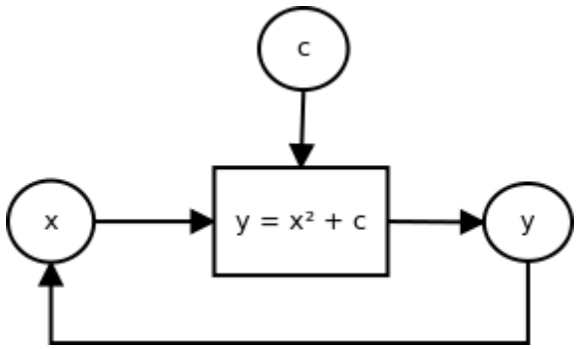


Figure 2.1: Iteration process as feedback machine

These one-step iteration processes do not sound complicated intentionally. But even if the iteration process is simple, the behaviour can get complex.

If an iteration process is carried out with a computer, it has to be considered that computers can only represent numbers with a certain amount of decimal places, every following digit gets trunced off. If the computer can represent, say 10 decimal places, one would probably think that this is precise enough. But that is not the case, because even the smallest deviation at some point of the iteration process can grow to a massive deviation at a later point. [10, p.49-53]

2.1.2 Self-Similarity

A basic characteristic of fractals is self-similarity, more or less pronounced. In principle self-similarity can be described with the example of a cauliflower, shown in figure 2.2. It contains branches, which themselves look much like the whole cauliflower, only smaller. These branches can again be decomposed into smaller pieces with the same property. On a cauliflower, this self-similarity stops at some point when the branches get to small. Strict self-similarity in its mathematical meaning would imply, that this decomposing process can be done infinitely many times. But also without that attribute, self-similarity can be useful to examine natural structures. [10, p. 215]

Figure 2.2: Cauliflower with natural self-similar structures

A more mathematical explanation of this attribute can be formulated as follows. Two objects are similar, if they have the same shape, regardless of their size. So corresponding angles must be equal, and corresponding line segments must all have the same factor of proportionality. [10, p. 138] In nature, self-similarity is never perfect, because the magnitude will be bound and a smaller part of an object will never be exactly the same as the whole.

2.1.3 Basic Fractals

Now that some basic terms that are necessary to talk about fractals are clear, some first examples of fractals can be revealed.

Cantor set An important fractal, which is dating back to the 19th century, is the Cantor set. It is basically an infinite set of points in the interval $[0, 1]$. [10, p. 67]

The set is constructed by starting with the interval $[0, 1]$ and taking away (cutting out) the open middle interval $(1/3, 2/3)$. That leaves the two intervals $[0, 1/3]$ and $[2/3, 1]$. Now this step is repeated on the two remaining intervals, so their middle thirds get cut out and four intervals of length $1/9$ remain. The first steps of this process are shown in figure 2.3. Carrying out this removal steps infinitely often leads to the Cantor set. [10, p. 68]

Figure 2.3: Construction process of a Cantor set

Is the Cantor set self-similar? It is, because e.g. taking a look at the interval $[0, 1/3]$ shows that this part is a scaled down version (by $1/3$) of the whole Cantor set in $[0, 1]$. [10, p. 75]

The Koch Curve The Swedish mathematician Helge von Koch introduced the now called Koch curve in 1904, and it is geometrically constructed by starting with a straight line, which is called the initiator. This line is partitioned into three equal parts, the middle third is replaced by an equilateral triangle and its base is taken away, leaving a figure made out of four lines. This procedure is repeated on each of these four line segments. Self-similarity is built into the construction process. The first steps of the process are illustrated in figure 2.4. [10, p. 90]

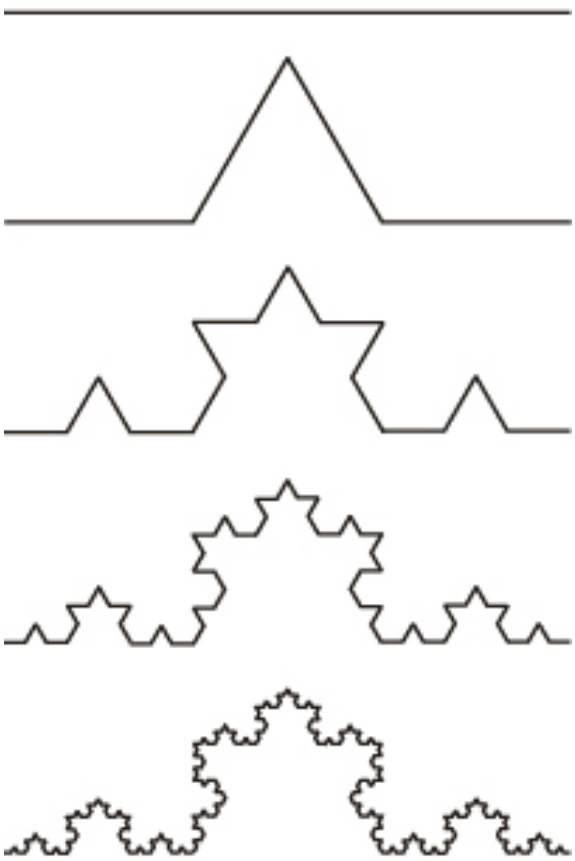


Figure 2.4: Construction process of a Koch curve

The Koch curve represents self-similarity in its purest form. Scaling up one of the four equal parts of the Koch curve by the factor three results in the exact same curve as before. [10, p. 145]

2.1.4 Limit

Many fractals are obtained by a construction process, and ideally this process should never terminate. Any finite stage may have fine structure, but is always far from the true fractal, and that fact is important to keep in mind. So the fractal only exists as an idealization. Or, in other words, fractals are limits. [10, p. 147]

This property is highly related to the concept of limits e.g. of the geometric series. The sum of the geometric series $1 + q + q^2 + q^3 + \ldots$ has the limit $1/(1-q)$ (if $q < 1$). But in fact the sum S_n (the sum of the first n elements of the geometric series) will always be different from the limit, no matter how large n is. But in a finite accuracy computer this difference is indistinguishable if n is large enough. [10, p. 148f]

Also the construction progress shows high analogies. A starting value, here 1, is scaled down by the scaling factor (which is q) and added (with geometrical objects this is interpreted as a union of sets). This infinite construction leads to a new number which represents this process, and which is the limit of the geometric series. Exactly analogous to that, the construction of a fractal leads to a new geometric object. [10, p. 149]

This now examined strong link between the geometric series and the Koch curve helps to understand and to provide evidence for the existence of fractals. Another approach of doing this are fractals as solutions of equations, this is explained in detail in [10, p.168-178].

2.1.5 "How long is the coast of Britain?"

Benoit Mandelbrot wrote an article in 1967 with the theme "How long is the coast of Britain?".[7] This article refers to the problem of measuring a coastline, which will always get different results, depending on the granularity of the measurements. The smaller the scale, the longer the resulting measures will be. But how do these values (scale of the 'compass' and measured length) correspond to each other? To find that out, a log/log diagram is drawn with the inverse compass setting ($1/setting$) on the horizontal axis, which represents the precision of the measurement. The vertical axis is the logarithm of the length. A line (approximated by the method of least squares) is laid through the measured points. The line is described by $u = d * log(1/s) + b$, and the slope d of the line is the key to the fractal dimension (the term will be explained in 2.1.6) of the underlying object. In the case of Britain's cost line, this is about 0.36. [10, p. 184,p. 192-195]

If the measured length is called u and the precision $1/s$, then the power law of the growth of the measured length of Britain's coastline can be formulated like this:

$$u \propto \frac{1}{s^0.36}$$

[10, p. 198f]

Now the same process can be applied to a pure mathematical situation, e.g. the Koch curve. For the compass settings $s = 1/3, 1/9, 1/27, ...$ can be chosen, because it fits best to the length of the segments of a Koch curve. The used logarithm should be log_3, to get convenient numbers. The growth law is then $log_3(u) = d * log_3(1/s)$, therefore $d = log_3(4/3) = 0.2619$. [10, p. 201]

2.1.6 Fractal Dimension

The term of dimension usually raises intuitively the thought about line, plane and space as 1st, 2nd and 3rd dimension. But the term can get much more complex than that, and was discussed a lot especially in the turn from 19th to 20th century. Mathematicians came up with a bunch of different notions of dimension, which are all somehow related, but fit differently well to specific situations. [10, p. 202]

For the sake of this thesis, Mandelbrot's fractal dimension is of most importance, which was already built up in 2.1.5, and can now be defined exactly.

Self-similarity dimension When observing self-similarity of fractals, there is usually a scaling factor s, which is characteristic for fractal structures, and a number of scaled down pieces a into which the structure is divided. [10, p. 203]

Now a power law relation between these number of pieces a and the reduction factor s should be formulated:

$$a = \frac{1}{s^D}$$

, where $D = 1$ for a line, $D = 2$ for a square, and $D = 3$ for the cube, representing exactly the numbers familiar for (topological) dimensions. [10, p. 204f]

For the Koch curve on the other hand, the results is $4 = 3^D$, or $log(4) = D*log(3)$, or

$$D = \frac{log(4)}{log(3)} = 1.2619$$

. This is the self-similarity dimension of the Koch curve. A general self-similarity dimension is calculated by

$$D_s = \frac{log(a)}{log(\frac{1}{s})}$$

. [10, p. 205]

This calculated dimension of 1.2619 should sound familiar. It is exactly the fractional part of the length measuring for the Koch curve in 2.1.5. Intuitively it seems like the self-similarity dimension should be the length measuring slope plus one, $D_s = 1 + d$. And this is, in fact, correct, as it is proved in the given reference. [10, p. 205ff]

2.1.7 Attractors

The term 'attractor' will be needed for developing the ideas of Julia sets (explained in section 2.1.10). As it is often with new terms, the easiest way to introduce them is by giving an example.

One can think of an experiment, where three strong magnets are placed somewhere on a surface. Above this surface, a pendulum is fixed, consisting of the string and a metallic ball at its end. If the pendulum is now moved to some position and gets released there, it will swing for a while until it stops exactly above one of the three magnets, depending on where the pendulum was released. The three positions on which the pendulum can stop are called attractors.

To speak in more mathematical terms but stay informal, a variable moving according to the dictates of a dynamical system evolves to a so called attracting set. In practice smaller sets will be obtained, because some parts of an attracting set may not be attracting. These smaller distinct sets are called attractors. [2]

2.1.8 Basin Boundaries

For a system with several coexisting attractors (see pendulum experiment), where depending on the starting value one of the attractors is approached, the term of "final state sensitivity" is used. A set of initial values which lead to one specific attractor is called a basin of attraction (see figure 2.5). So there must be a boundary of these corresponding basins of attractions, and such boundaries often are fractals. The complexer this boundary is, the more problematic it gets because of the limited possibility for numerical representations. [10, p. 757]

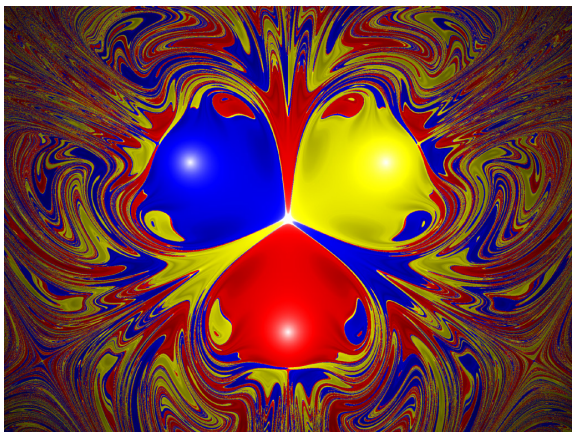


Figure 2.5: Basins of attraction in pendulum experiment [10, p.711]

2.1.9 Prisoners versus Escapees

The simplest nonlinear iteration procedure in complex numbers is $z \to z^2$. Geometrically this means, the corresponding length of z is squared, the angle $arg(z)$ of z is doubled (modulo 2π). [10, p. 789]

It is quite obvious that the points inside of the unit circle lead to a sequence that converges to the origin, the points exactly on the unit circle lead to a sequence that remains on the unit circle forever, and the points outside lead to sequences that escape to infinity. [10, p. 789]

The complex plane of initial values can be subdivided into two subsets. The first one collects points for which the iteration escapes and is called the “escape set E". The iteration of all other initial values remains in a bounded region forever, these points are collected in the “prisoner set P". [10, p. 789f]

2.1.10 Julia Sets

In the previously introduced iteration $z \to z^2$, the prisoner set P is the disk around zero with radius 1, and the escape set E is the outside of that disk. The boundary between E and P is the unit circle, and is called the “Julia set" of the iteration. The Julia set is invariant under iteration, i.e., for initial values in the Julia set, the iteration generates only points which again lie in the Julia set. [10, p. 789f]

The iteration process has two fixed points, 0 (attracting) and 1 (repelling). We can interpret the interior of P as the basin of attraction for the point 0, and the escape set E as basin of attraction of the point at infinity. [10, p. 790]

In this simple example, the Julia set is a circle, hence a geometrical object from classical Euclidean geometry. But this is just an exception, most Julia sets are in fact fractals. The iteration process should now be changed to $z \to z^2 + c$, where c is some complex parameter. To visualize these fractals a way needs to be found to find the escape set E and visualize it, the remaining points will then be the prisoner set P, and the boundary between them the Julia set. [10, p. 791]

A first example: $c = -0.5 + 0.5i$. This leads to a set of points escaping to infinity (escape set) and a set of points converging to $z \approx -0.408 + 0.275i$. So again, there are two basins of attraction, but it is not zero which is one of the attracting points. The Julia set of this example has clearly self-similar structures, no matter how far it is magnified. [10, p. 792]

How can it be decided if a point escapes to infinity or not? Observation shows,

that points z_k from an orbit will escape to infinity with certainty once their absolute value is large enough (because a square of a large number will make the constant c rather insignificant). But how large must it be? There is in fact an optimal answer to this question, which helps a lot for the computations. This threshold number $r(c)$ can be calculated as the maximum of the absolute value $|c|$ and 2:

$$r(c) = max(|c|, 2)$$

. Thus, if $|z_k|$ exceeds $r(c)$ in absolute value, the iteration will escape to infinity for sure. [10, p. 793f]

In practice there is the problem, that it may take a long time until a certain orbit escapes a disk of radius $r(c)$, even though if it will escape at some point. So the computation needs to be limited by a maximum number of iterations, if the iterated point does not exceed $r(c)$ in absolute value during these iterations, then it must be assumed that the initial point belongs to the prisoner set. So it is important to keep in mind, that the precision of the algorithm is limited. [10, p. 794]

By visualizing this process, different results can be observed. Connected Julia sets can appear, which are the common boundary of two basins of attraction, or a prisoner set with no interior points is seen, which is equal to the Julia set. Hence it can be distinguished between prisoner sets that are connected (see figure 2.6) and those that are a dust of points (see figure 2.7). This observation is important for the understanding of the Mandelbrot set. [10, p. 798]

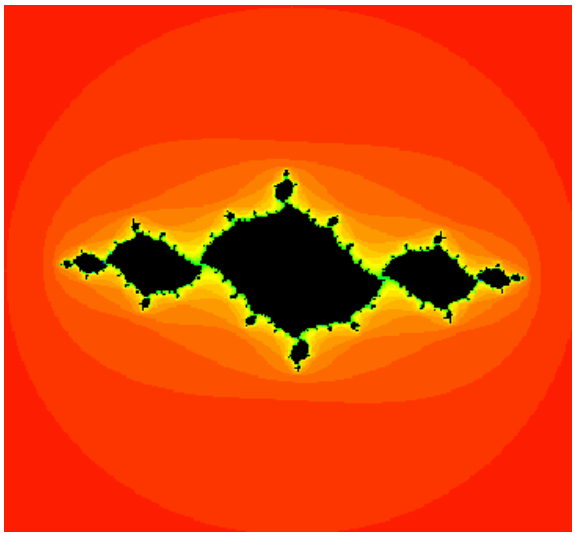


Figure 2.6: Julia set - connected

2.1.11 Mandelbrot Set

The Mandelbrot set is certainly the most popular fractal, and has been a major research object since the first experiments to show this extraordinary mathematical objects by Benoit Mandelbrot around 1980. [10, p.841]

The occurrence of highly performant computers made it possible to show this fascinating fractal to everyone, instead of it just being imagined in genius minds like the one of Benoit Mandelbrot. The Mandelbrot set is a glimpse to everyone what mathematicians sometimes call the aesthetics of mathematics. [10, p. 842]

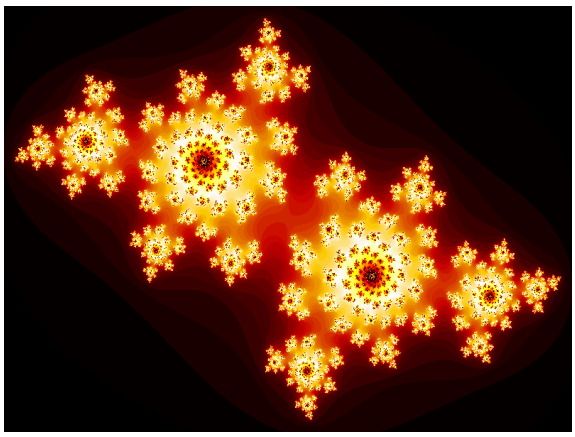


Figure 2.7: Julia set - dust of points

The Mandelbrot set is not only aesthetically pleasant, but it offers a high amount of mathematical backgrounds. But since this report focuses on the visual representation of the set and its use as an artistical object, its mathematical properties are only touched on the surface.

Metaphorically speaking the Mandelbrot set can be described as an infinite picture book, each page holds the image of one particular Julia set J_c, and the page numbers are the complex parameter c belonging to the Julia sets. This infinite book can be organized in two chapters: the first for all connected Julia sets and the other for those that are totally disconnected. This was the way how Mandelbrot discovered the Mandelbrot set in 1979:

$$M = \{c \in \text{Complex plane C } | J_c \text{ is connected}\}$$

. [10, p.843] So any point in the Complex plane, interpreted as a parameter c for the iteration of $z \to z^2 + c$, corresponds to a Julia set. The point is colored black, if the Julia set is connected (it is a 'piece'), and white if the set is disconnected (it is a 'dust'). [10, p. 843f]

But how can it be decided computationally whether a parameter c belongs to the Mandelbrot set or not?

Proposition 1. *The prisoner set P_c is connected if and only if the critical orbit $0 \to c \to c^2 + c \to \ldots$ is bounded*

[10, p.844]

This proposition provides the alternative definition for the Mandelbrot set, Mandelbrot himself used

$$M = \{c \in C | c \to c^2 + c \to \ldots \text{ remains bounded}\}$$

as the definition for M in his 1979 experiments. [10, p.844] This definition is similar to that of the Prisoner set P_c,

$$P_c = \{z_0 \in C | z_0 \to {z_0}^2 + c \to \ldots \text{ remains bounded}\}$$

. [10, p.845] The Julia set is part of the plane of initial values z_0 which have orbits that reside in the same complex plane. The Mandelbrot set however is the plane of parameter values c, it is not appropriate to plot any orbits from the iteration of $z \to z^2 + c$ in this plane. [10, p. 844f]

As one of the characteristics of the Mandelbrot set, everything outside of a disk of radius 2 is not part of the Mandelbrot set, because, if $|c| > 2$, the critical point escapes to infinity for sure, and the Julia set is a dust. [10, p. 845]

Given the parameter c, how can be computationally decided whether the orbit of c is bounded or not, i.e., whether $c \in M$? This leads to the same problem as deciding if z_0 is in the prisoner set or not, so theoretically infinitely many iterations could have to be conducted, but in practice only a limited amount of iterations for a sufficiently accurate approximation is carried out.

Figure 2.8 shows a visualization of the set with a blueish coloring (where colors are according to the amount of iterations needed to decide whether $c \in M$ or $c \notin M$).

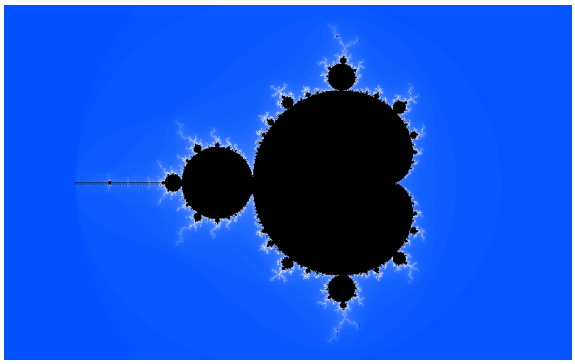


Figure 2.8: Entire Mandelbrot set with blueish coloring

2.1.12 Characteristics of the Mandelbrot Set

The complexity of the Mandelbrot set is in an altogether different class compared to that of Julia sets. On the one hand, the Mandelbrot set has a solid interior without any structure, and on the other hand it is bordered by a complex boundary with an infinity of different shapes. [10, p. 855]

An interesting feature of the Mandelbrot set are the small buds which are lined up along the one big, heart-shaped, central region. The buds have a meaning for the associated Julia sets. The big heart-shaped region is in fact the set of all (complex) parameters c for which one of the two fixed points of $z \to z^2 + c$ is attractive. At the left end of the heart-shaped region, at $c = -0.75$, there is a bud. For the parameters in this bud neither one of the two fixed points of $z \to z^2 + c$ can be attractive because c is outside of the heart-shaped center of M. The iteration of the orbit can either diverge or it is dominated by the attractive orbit of period 2:

$$0 \to -1 \to 0 \to \dots$$

All initial values of the interior of the prisoner set are attracted by this orbit, and the Julia set is the boundary of this basin of attraction. The next big buds attached at the edge of the heart-shaped center of M correspond to period-three behaviour, then there are buds which house parameters belonging to attractive cycles of period 4, and so on. Figure 2.9 shows a Mandelbrot sets with numbered buds, the numbers correspond to the periodicity of the orbits of Julia sets in this bud. [10, p. 855, 862ff, 866]

There are many more mathematical observations about the set that can be made. Many of these observations are explained in detail in literature [10] and [1]

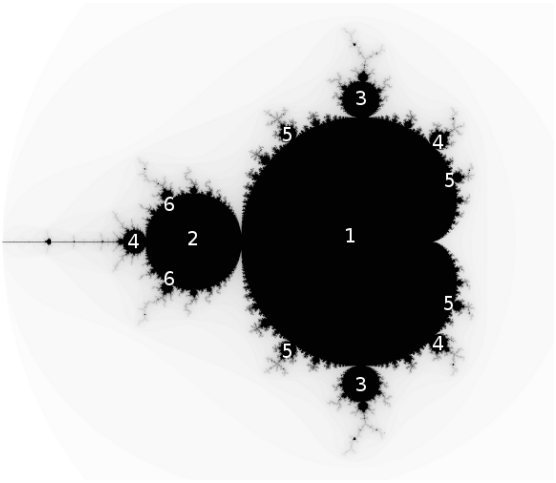


Figure 2.9: Mandelbrot set with numbered buds

2.2 General Purpose GPU Programming

It should be clear that the decisive factor in the development of the music visualization is to get the calculation of the Mandelbrot set as fast as somehow possible to accomplish real-time zooming. To achieve this it can be of huge benefit to use a GPU for calculations, because in the last years GPUs got capable of handling not only 3D-rendering but a wide field of heavy calculations. Especially for highly parallelizable calculations, as the Mandelbrot set calculation, the GPU is the perfect tool to use.

To understand why the GPU is so suitable for this purpose, it is necessary to take a short look on the basic underlying architecture of a modern GPU. After that, a look on the programming model will reveal how it is possible to actually use a modern GPU for general purpose calculations. And finally the actual programming interfaces will be discussed, which allow it to 'speak' to different GPUs in a high-level programming language.

2.2.1 GPU Architecture

Modern GPUs are highly parallel programmable processors, not only powerful graphics engines as they used to be. GPUs are designed for applications with particular characteristics, such as large computational requirements and high parallelizability. Some years ago, the GPU was a highly specialized processor, built around the graphics pipeline to perform graphic calculations. But since then, both the hardware and the programming interfaces developed to make them more suitable for general purpose applications. [9]

The graphics pipeline had a list of triangles in a 3-D world coordinate system as input. These geometrical primitives got processed in several steps to finally map them onto the screen. This was done by fixed-function operations. The key step in the evolution to general purpose processing was the replacement of these fixed-function operations by user-specified programs. So today we have a processor as a programmable engine surrounded by supporting fixed-function units, instead of them being the central and only part of a GPU. [9]

The programmable units of the GPU process multiple elements in parallel with a single program. Each element must be independent from the others. Even though only a single program is used, different elements may take different paths. But taking

a different branch in the program is a major deficit to the performance because of the GPUs architecture, so it should be avoided. That is why elements get grouped into blocks which are proceeded in parallel. If elements inside one of these block branch into different parts of the code, the different branches get executed on all elements of the block. So it has to be taken into consideration that elements of one block should preferably follow the same code branch to optimize the performance of the calculations. [9]

The computing on a modern GPU is not structured in terms of graphic calculation, but adapted a more general approach that is still closely connected to the former graphical structure, but more beneficial for general purpose applications. It exploits data parallelism and provides a balance between generality and also restrictions to ensure a good performance. [9]

The basic steps are the following:

- Step 1: Defining a computation domain as a grid of threads.
- Step 2: A single process multiple data general-purpose program computes the value of each thread.
- Step 3: The computation of the value is done by a combination of math operations and global memory access, and is stored in a resulting buffer in global memory.

2.2.2 Programming Interfaces

GPGPU programming used to be done through the graphics API. Therefore, calculations needed to be adapted exactly to the structure of graphic calculations. This was time consuming and needed a deep understanding of the underlying hardware structure. To avoid it, it was necessary to introduce APIs that can be easily used in higher level languages and are abstracted of the graphic computation structure. This led to the development of the stream programming model, which structures programs in a way so they match the parallel processing resources and the memory system of the GPU hardware. A stream program consists of a set of ordered sets of data (streams) and the functions which are applied to the elements (kernels), and produce streams as the output. [9]

The first widely used GPGPU programming systems got developed by the two big GPU producers, AMD and NVIDIA. NVIDIA provides a higher level interface than AMD does, namely CUDA. CUDA code adapts the syntax of C and is overall closely related to C. It compiles offline, and it exposes two levels of parallelism, data parallelism and multi-threading. [9]

The first open standard for parallel programming on modern computer hardware was OpenCL (Open Computing Language). It has the great benefit of running on a wide range of different hardware architectures.

CUDA and OpenCL have a similar functionality, and porting applications between them is quite simple. So a comparison between their performance should be undertaken. The transfer of data to and from the GPU is faster on CUDA. When it comes to kernel execution time, CUDA is also consistently faster than OpenCL. So, when trying to reach a performance as high as possible, CUDA seems to be the better choice. On the other hand there is the fact that OpenCL code will run on almost every modern system, while CUDA code needs a specific hardware. [6]

Besides the choice of the right programming interface, there are other things to be taken under consideration to benefit as much as possible of GPGPU processing: [9]

- Emphasize parallelism: The efficient use of GPU depends on a high degree of parallelism. CUDA e.g. prefers to launch thousands of threads at one time to optimize the performance. But on the other hand, the use of shared resources between the threads should be minimized, as well as synchronization between them.
- Minimize SIMD divergence: It is important to avoid branches in the code inside data blocks.
- Maximize arithmetic intensity: The number of numeric computations should be as high as possible compared to memory transactions, because the GPU can play out its advantages on its strong floating-point unit.
- Exploit streaming bandwidth: Despite the fact that maximizing arithmetic intensity is desirable, the GPU has also a high bandwidth on its on-board memory, which can be made use of when a lot of memory accesses are necessary.

2.3 Music Analysis

To develop a good method for analysing played music and adapt graphical visualizations to it, some basics about music theory and signal processing need to be understood.

The signal that reaches the human ear when one listens to music has a certain energy, and the higher this energy value is, the louder the music seems. The rhythm of music is determined as the (more or less) periodical succession of beats. Beats again are determined as energy peaks, i.e. the energy level at some point is considerably higher as it was over a time period before. [11]

2.3.1 Energy Analysis

The energy level of an audio signal gives us already some crucial information for sound analysis, since beats can basically be determined by analyzing the energy level. On the other hand it can happen as well, that the overall energy level will be loud in some music genres and the energy level at moments where humans would 'feel' a beat is not higher compared to a time interval before. This can happen because unlike the human ear, the energy level is not distinguishing between different frequencies. For that reason, it can be necessary to introduce the beat detection based on frequency bands. [11]

2.3.2 Frequency Based Beat Detection

If simple energy analysis fails to find beats in an audio signal, advanced techniques to make the beat detection dependent on the signals frequencies can be used. In that way it can e.g. be possible to distinguish between different instruments.

To get the audio signal from time to frequency domain, the fast Fourier transform is being used. In that way, a bunch of values describing the energy level in a certain frequency band are obtained. So now different frequencies can be distinguished, and

beats in a certain frequency band can be found. This for instance makes it possible to detect the drumming pattern of a song, even if it is not as loud as e.g. a guitar. [11]

2.3.3 Beat Spectrum

The beat spectrum is a quite new and advanced method for automatically characterizing rhythm and tempo of audio. It is a measure of acoustic self-similarity as a function of lag time. Peaks in the beat spectrum show major rhythmic components. The lag time of the corresponding peak shows the repetition time, and the amplitudes of different peaks reflect the strengths of their rhythmic components. The beat spectrogram is a graphical illustration of rhythmic variation over time, and makes changes in tempo or time signature visible. [5]

The beat spectrum is calculated from three principle steps.

1. The audio gets parametrized, which results in a sequence of feature vectors.
2. The similarity between all pair wise combinations of feature vectors gets calculated, which is then embedded into a two-dimensional representation, the similarity matrix (example of a similarity matrix for Bach's Prelude No. 1 in C Major, BVW 846, performance by Glenn Gould, in figure 2.10).
3. The beat spectrum is determined by finding periodicities in the similarity matrix (result for Gould Prelude shown in figure 2.11).

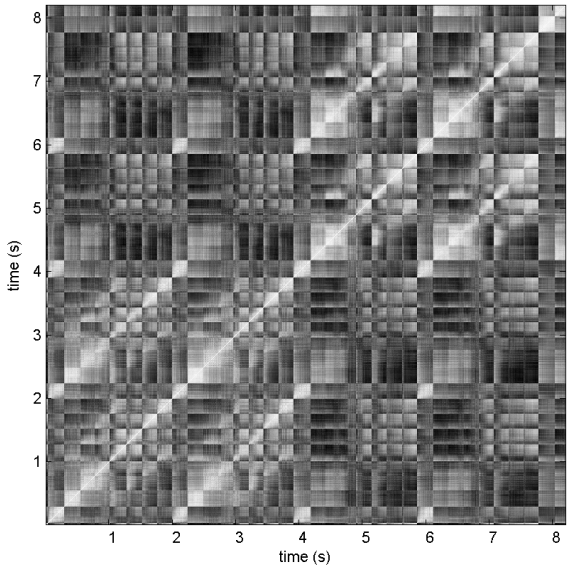


Figure 2.10: Similarity matrix of Gould Prelude [5]

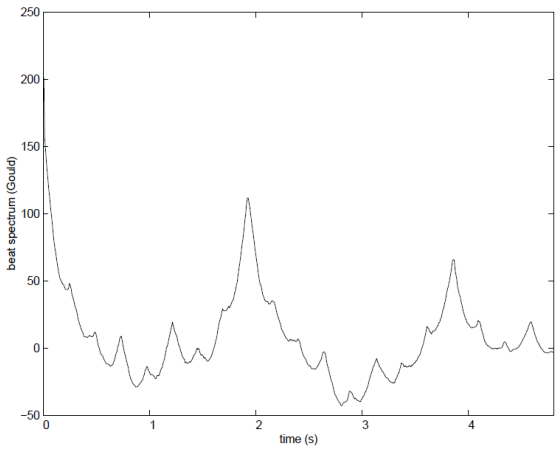


Figure 2.11: Beat spectrum of Gould Prelude [5]

3 Related Work

As related work to this report are seen existing applications that are similar to the one that will be developed for the purpose of this report, because the focus of the thesis lies on implementation work.

3.1 Mandelbrot Set Explorers - Comparison

Numerous Mandelbrot Set Explorers can be found around the Internet, some of them are simple and were written for test purposes or own use, while others are fairly advanced and often not only include the Mandelbrot set but also several other fractals which can be displayed. The different applications vary a lot in offered features, performance, appearance, price, and other important attributes. An overview over the most popular and some not so well known but interesting fractal explorers will be given, and the good and bad parts are pointed out to learn from them for the implementation of the application for this thesis.

Some of the tested explorers are not free, but all of them have a free trial version. The tests are always done with the trial version of these non-free explorers. All of them are only limited by watermarks in the fractal pictures and by time based limits, not by functional limits.

Name	Platforms	Version	Last release	Price
Fractal Science Kit	Windows	1.20	20-10-2011	29.95$
JM's Mandelbrot Explorer	Windows	1.21	26-05-1999	Free
Ultra Fractal	Windows	5.04	11-08-2010	39-139$
XaoS	Win, Linux, Mac	3.5	17-07-2009	Free
Ultimate Fractal	Windows	2.1	19-10-2011	38$

Table 3.1: An overview over the tested fractal explorers

3.1.1 Fractal Science Kit [17]

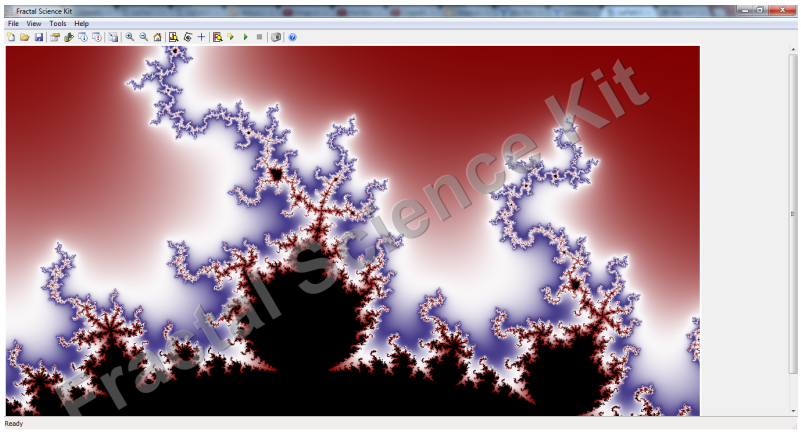


Figure 3.1: Fractal Science Kit 1.20 - Main View

The Fractal Science Kit is a fractal explorer, that implements the basic features an explorer should have. The main view of it is shown in figure 3.1.

First of all, it is possible to save and load image properties, but also many other things like own fractal equations and so on. It is also possible to take snapshots of the fractal image.

The tool has basic navigation features, it is possible to jump to a certain location and rotate the fractal. But all that has to be done through a menu, there is no possibility at all to interact directly with the image without clicking a menu button first. There is no panning at all, and zooming can only be done by a box which outlines the area to zoom into. The zoom jumps in, there is no smooth movement. All this makes it annoying to navigate around the set and is not convenient at all.

What is also quite annoying is the fact that it is not possible at all to change the images size and resolution, both is fixed. Also, there is no possibility to change the colors of the fractals, the color set is predefined and cannot be changed by any means.

The feature which makes this tool exceptional, is the sheer versatility of options to represent a lot of different kinds of fractals, and define own fractals as well. So this tool is really more of a creator than an explorer, and should therefore rather be measured by its huge variety of fractals and the possibility to manipulate them. But still, better navigation and the possibility to manipulate the color plates would make this tool much more user friendly and would make it more appealing.

3.1.2 JM's Mandelbrot Explorer [18]

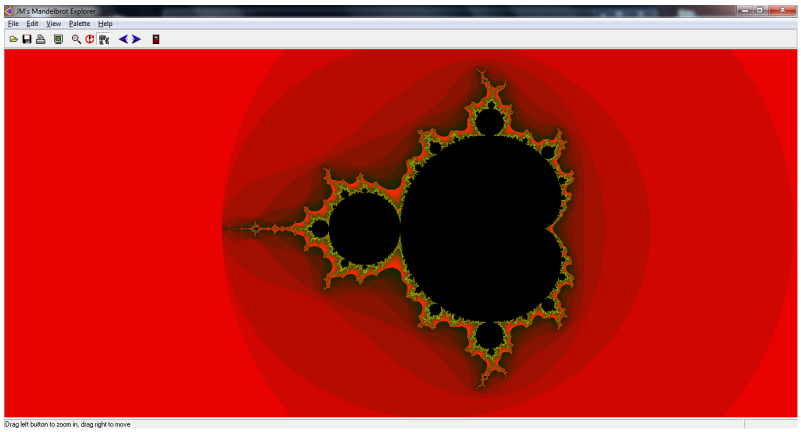


Figure 3.2: JM's Mandelbrot Explorer 1.21 - Main View

JM's Mandelbrot Explorer is a small and fairly old fractal explorer with a narrow band of features. Figure 3.2 shows the main view of it.

It is possible to save and load locations, images and color plates.

This tool has the distinction that it has a great usability in navigating the Mandelbrot set. The navigation is easy and is done on the set image itself. By left clicking it is possible to draw up a rectangle on the image that specifies the area that should be shown. Then the zoom to this window is smooth with coarse image representation on the way in. The zoom is quick and without big delays. A right click is used for panning. All in all the way of navigation is user friendly and convenient, even though it would be nice to have some additional navigation features. The worst thing missing is the possibility to jump to a location that is entered by the user directly, even though it is well possible to save and load locations where the

viewport is at that moment. Also, the users get no feedback about their location in the set.

Image size and resolution change automatically when the window size changes. A color plate editor is built into the program. At first it can be hard to use it, because it is not obvious what the different options cause in the color plate. But after a short time it is clear what is happening, and then the editing of colors works well.

The tool lacks in options. There is almost nothing that can be adjusted, not even the maximum iteration count (which leads to a limited reachable magnification). There are some few options of fractal equations to use, but the Mandelbrot set is in fact the only one of them which is really interesting.

This tool is good for beginners in the world of fractals who just want to jump around the Mandelbrot set with a tool that is easy to learn. But it is not the right thing for people who have high expectations in possible options and adjustments to make or who want to go deep into the set.

3.1.3 Ultra Fractal [19]

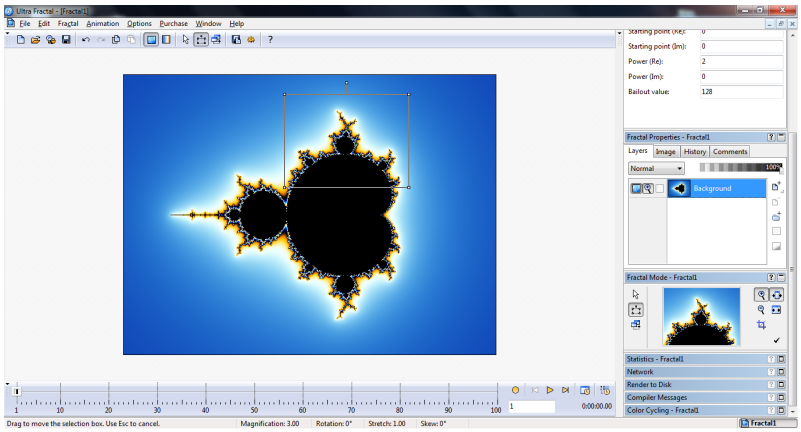


Figure 3.3: Ultra Fractal 5.04 - Main View

Ultra Fractal is an advanced and professional fractal explorer, seen in figure 3.3.

First, besides being possible to save and load positions and create images, it is also possible to write own transformation algorithms, fractal algorithms, and even coloring algorithms.

The zooming is done by opening a zooming window on the fractal image (which can also be rotated), and clicking again to smoothly zoom in to this window. A right click opens a context menu which makes it possible to zoom back out, to change colors, to change to full screen, and some more. By using shortcuts it is also easily possible to pan/skew/rotate the image directly. The users can also write in a location to jump to that place.

The resolution of the image can be changed manually. The feedback given to the user is great, also with displayed calculation time.

The color plate editor of Ultra Fractal is exceptionally good (shown in figure 3.4). It is possible to create own color plates in an easy and intuitive way, it is quickly understood how it works. Only one small critic is that the set with the new colors cannot be seen simultaneously, but the user has to switch between color plate mode and fractal mode to see the result of his manipulations.

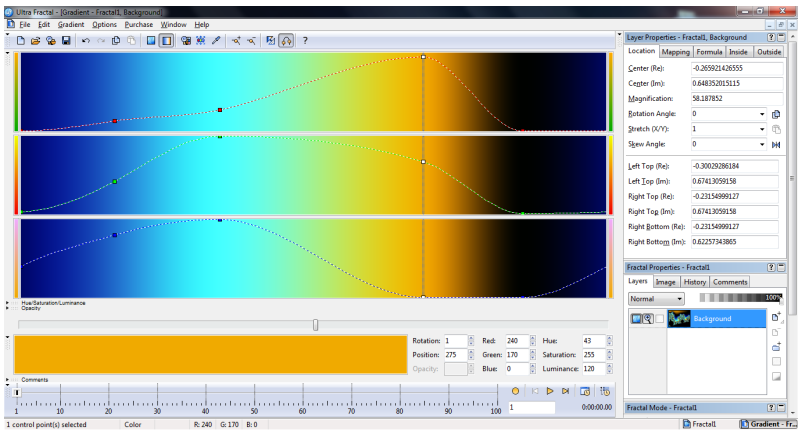


Figure 3.4: Ultra Fractal 5.04 - Color Editor

The settings offer a lot of different possibilities to adjust the tool to the own needs. And as mentioned in the beginning, especially the possibility to code own transformations, fractals and coloring methods is a great way to get deeper into the topic of fractal visualizations.

A special feature of Ultra Fractals, which is of special interest for this project, is the possibility to create animations. On a time line it is possible to record and play animations of zooms into the sets by setting key values. The problem with it is, that it is not intuitive at first, and it is sometimes not that easy to create animations as wanted. But it is already a great tool for animating predefined zooms and play them back.

All in all Ultra Fractals is clearly a tool for professional or at least experienced users with all its possibilities to extend the program with own code. But it can also be a tool for beginners who want to experience fractals, because the tool is fairly simple in usage.

3.1.4 XaoS [20]

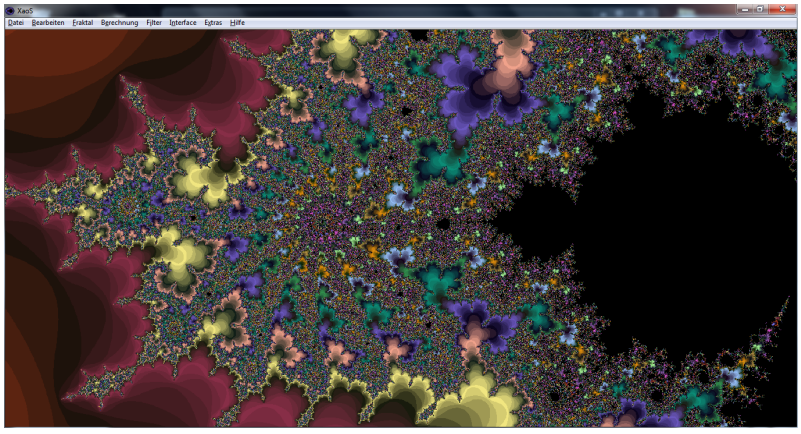


Figure 3.5: XaoS 3.5 - Main View

XaoS is a free and simple fractal explorer, which offers all the basic features and has some nice aspects. Figure 3.5 shows the main view of the application.

Locations can be loaded and saved, images created, and even zooms can be recorded and played.

The navigation is done by clicking somewhere on the fractal image to slowly and smoothly zoom into the set in that direction, as long as the left mouse button is held down. By holding the right mouse button down, the zoom goes out instead of in. It is also possible to jump to certain positions, but no panning can be done.

The menus have a clear structure, everything can be found in a quick and intuitive way. But the menu has to be opened every time something should be done, so at least some clicks are inevitable for almost every action.

Size and resolution of the image change automatically. User color plates do exist, but they just make it possible to choose one of some different algorithms to create the plate and some offsets. Besides that, it is possible to pick random plates (which works well), and even the equation to calculate the coloring of a pixel out of different variables (like iteration count) can be changed by the user.

There are several options to adjust the fractals. Several different equations can be chosen out of, and several interesting filters can be laid over the fractal image. Apart from that, there are a few other options to modify calculation and appearance.

As mentioned before, it is possible to record zooms. A special feature is to load and pre-calculate these zooms, so they can then be played without more delays.

XaoS is a nice tool, that is extremely easy to handle. It may have not as many features as some other tools, but at least the features are easy to find and it is easy to understand what they cause. Also, the zooms look good and the overall appearance of the tool and the fractal image is appealing. This program shows how a great tool can be created without making it complicated to use it.

3.1.5 Ultimate Fractal [21]

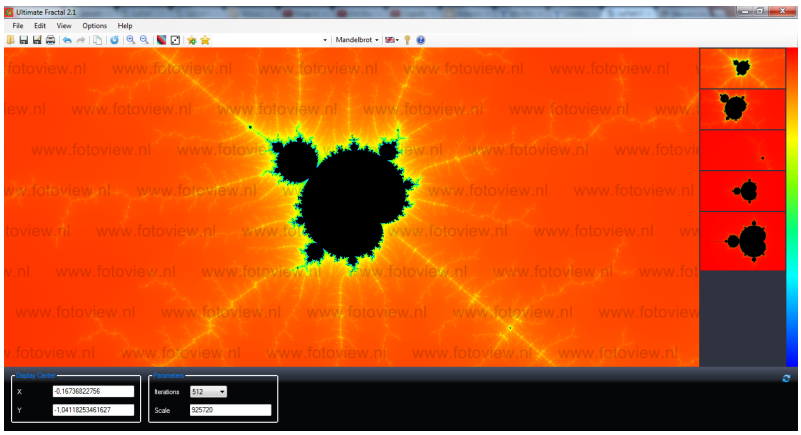


Figure 3.6: Ultimate Fractal 2.1 - Main View

Ultimate Fractal is an advanced but easy to use fractal explorer for a lot of different fractals, figure 3.6 shows the main view of the application.

Parameters and images can be saved and loaded. The tool has also 'back'- and 'forward'-options.

Zooming in is done by opening a zoom window. The zoom is a jump and no smooth zoom. By clicking the right mouse button, a jump zoom out of the fractal occurs. There is no panning, but easy and direct jumping to certain positions. Resolution of the set is changed automatically when resizing the window, but no full screen mode is available.

The tool has some predefined color gradients, as well as an own color plate editor. The editor (see figure 3.7) is really easy and intuitive to use and makes it easy to create nice own color plates, as well as saving or loading them. One disadvantage is again, that the colors of the fractal images are not updated simultaneously while changing the color set, it is necessary to first close the editor to see how the fractal changes.

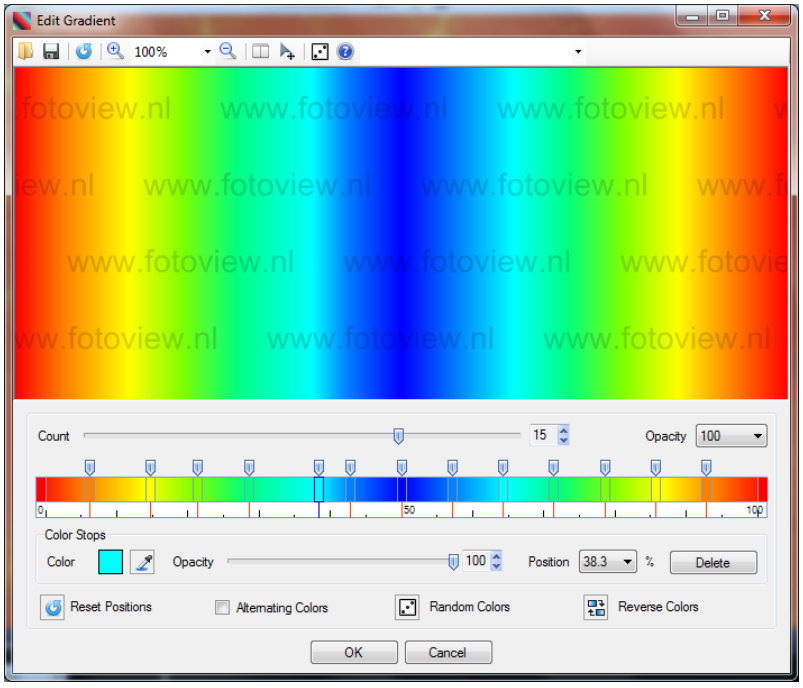


Figure 3.7: Ultimate Fractal 2.1 - Color Editor

There are several different fractals and equations that can be loaded, but not many program options besides that. Because of the not so overwhelming amount of options, the menu structure stays quite clear.

Ultimate Fractal is a quite nice explorer, but it still could need some inspiration of tools like XaoS which have higher functionality, nicer appearance, run on different platforms, and even are completely free.

3.2 Conclusion of Mandelbrot Set Explorers

Table 5.1 should give a brief overview over the functionality of the tested explorers, without giving a qualitative assessment.

The observations made about existing fractal explorers influence the requirements for the thesis project. E.g. the interaction benefits highly from direct interaction with the the fractal on the viewport, and by smooth zooms instead of simple jumps into or out of the set, and the coloring of the set should be easily customizable.

Explorer	Mult. Fractals	Viewport Interact.	Smooth Zoom	Color Editor	Create Zooms	Outstanding Feature
Fractal Sc. Kit	Yes	No	No	No	No	Create Fractals
JM's M. Expl.	No	Yes	Yes	Yes	No	None
Ultra Fractal	Yes	Yes	Yes	Yes	Yes	Own Algorithms
XaoS	Yes	Yes	Yes	No	Yes	Usability
Ultimate Fractal	Yes	Yes	No	Yes	No	None

Table 3.2: Conclusion of the tested fractal explorers

4 Requirements and Software Structure

Section 3 revealed what can be necessary for a convenient user experience and what should be avoided. According to these findings, this section formulates the requirements for the application, consisting of functional and non-functional requirements. Moreover, after defining these requirements, the basic architecture of the application is constructed.

4.1 Feature list (Functional Requirements)

1. Display the Mandelbrot set

   The obvious first feature of the application should be the displaying of the Mandelbrot set.

2. Navigation

   The navigation around the set should be interactive. That means, the users should be able to directly interact on the set image with their mouse and keyboard, and immediately see the result of their actions. In that way, the users will quickly internalize the ways of navigation that are provided by the tool.

   (a) Panning

   Panning is the navigation on the set, that leaves the magnification level the same, but changes the real and imaginary centre of the visualization.

   (b) Zooming

   Zooming into the set should be done in an intuitive way, which is easy to handle for the user. One possible way is clicking on a position in the set, and then jumping in by a certain level, e.g. doubling the magnification. To make it more interactive, the zoom should not jump, but when the user is clicking on a position in the set and keeping the mouse button down, the magnification level should increase (or decrease) slowly, so that it looks like a smooth movement into the set, that stops on releasing the mouse button.

3. Basic Settings

   The application should provide some basic settings which allow the users to directly influence the appearance of the displayed set.

   (a) Location

   The user can change the location in the set, which consists of the complex number in the centre of the visualization and the magnification level. By these parameters, the exact current viewport can be obtained.

   (b) Iterations

   The appearance of the set will change according to the maximum number of iteration steps to carry out. The more iterations can be done, the more accurate the picture will be because less mistaken prisoners will be added to the prisoner set. So the users should have the possibility to modify this value on their own.

4. Saving/Loading Positions

   To make it easy to find and share interesting locations in the set, the users would want to save certain locations and load them later, to not have to remember the locations themselves. So there should be an easy way to save a location, to see all saved locations, to load one of the saved locations and to remove locations that are not needed anymore. It would also be convenient to make it possible to export/import locations and not just save/load them internally, so users could exchange locations with each other and do not have to insert the locations manually.

5. Color Sets

   For the calculation of the set the number of iterations is evaluated that are needed, until the absolute value of the complex number exceeds a certain limit. That means, the calculation just results in a raster of integer-values which represent this iteration counts. Hence, arbitrary color values can be mapped to these integers. To make the set visually appealing, the chosen color mapping is important and also editable by the user.

   (a) Pick Predefined Color Set

   The users should be able to pick one of some predefined color sets to ensure a nice appearance of the set.

   (b) Create Color Sets

   To make the appearance more customizable it should be possible for the users to create their own color mappings. That is done by a convenient color plate editor, where the user can create own color transitions by setting color keys and giving them a distinct color. Every color in between those values is interpolated. Color keys should be moveable interactively. To give an immediate preview of how the modification of the colors affects the set, the displayed set should be redrawn on the fly in the new colors when the plate was modified.

   (c) Save/Load Color Sets

   To not have to recreate the own color sets every single time, the users should have the possibility to save and load their created plates. Also they should be able to export and import them, so they can be exchanged with other users of the program.

   (d) Color Rotation

   The users can let the colors of a color plate rotate, and also let this happen automatically, for example when zooming into the set. This causes a color movement effect, which will be important for the music visualization.

6. Zooms

   To make it possible to play the music visualization, the users must first be able to record zooms into the set.

   (a) Pick a zoom

   A set of predefined zooms into the set should be provided, so the users have some visually appealing zooms ready.

(b) Create a zoom

The users should be able to create nice looking zooms into the set to their own preferred locations in an easy way. For that, the principle of the color plate editor is adapted, so that the users can specify key frames which hold a certain location. At the points in time between these keys, the locations (i.e. position, magnitude, maximum iterations) are interpolated.

(c) Save/Load zooms

Created or modified zooms should be storable and loadable at a later moment. Moreover, it should be possible to export and import them, so users can share them with other users of the tool.

(d) Play zoom

Zooms get played with a default speed level and some settings, e.g. if color rotation should be carried out while zooming.

(e) Play music visualization

As before a zoom gets played, but now the zoom should change some of its parameters according to a music piece which is chosen by the user. The length of song and zoom can be adjusted, or the zoom is simply looped.

7. Settings

The tool should be customizable, so the users can change it to their own needs and also adapt it to their used hardware, so they can get the best possible performance out of the program. Some parameters that should be modifiable are:

- Used algorithm
- Used arithmetic
- Used zoom modes
- Algorithmic details

4.2 Non-Functional Requirements

This section formulates the non-functional requirements for the application.

4.2.1 Performance

The performance of the application is a crucial element in the development. The main feature of the application, creation and playback of music visualizations, requires real-time rendering. Therefore, the application should be able to render at least 25 frames per second, which means it can spend up to 40 ms on calculating and drawing one frame. Since the calculation is highly time consuming, this is not an easy task, but the main concern in the development process.

Generally speaking, the calculation effort increases when zooming deep into the set. This is due to the fact that higher magnifications need a higher iteration threshold until a point gets considered inside the set to create a sufficiently accurate image. A higher iteration threshold means a higher computation time, except if

all the points escape quickly, but the positions where that is the case do usually not look as interesting and will therefore usually not be the first choice to go to for the users. Moreover, if we go deeper into the set, we need a consecutively more precise arithmetic, which is the second big reason for a major slowdown when the magnification level increases.

It is difficult to give a threshold to which the performance should be raised. That is due to

- different calculation times in different positions of the set
- a variety of different algorithmic optimizations, which have different influence on the performance, depending on the position and magnification of the fractal
- and a wide range of different consumer hardware on which the application should run.

A minimum requirement can be formulated:

The application should be able to render arbitrary zooms in low magnification levels on fairly modern hardware.

This requirement is still vague, because neither the term low magnification levels, nor fairly modern hardware can be defined exactly. The point of this minimum requirement should be clear though.

4.2.2 Accuracy

The calculation of the Mandelbrot set in the application includes several tradeoffs in accuracy. By the nature of the used algorithm, it cannot be decided for sure if a point really lies inside the set, or if it would escape at some point, because the iteration process cannot be iterated infinitely often, but has to have a threshold at which the iteration process is stopped and the point is considered to be inside the set. The higher this threshold, the higher the accuracy of the algorithm because there are less points that are mistakenly assumed to be inside the set. But the effort gets high to increase this threshold, just to eliminate some small mistakes.

Moreover, there is a loss of accuracy because of the computer's arithmetic. Since the computer is not able to exactly represent floating point values, there is an inevitable mistake in calculations, and therefore a loss of accuracy. If the magnification of the set gets higher, this loss gets higher as well, and as soon as the precision of the parameters exceeds the precision of the used arithmetic, the accuracy is lost completely.

A third source of loss of accuracy are optimization algorithms for the calculation of the Mandelbrot set. Some of these optimization algorithms have a high loss of accuracy, some do not lead to a loss at all. But since performance requirements need to be fulfilled as well, a tradeoff between accuracy and performance has to be done.

And at last, there is the boundary of resolution, which makes it only possible to display the set with a certain amount of detail, even if the accuracy of the calculation would have been higher than that. That leads to the point where it is unnecessary and just wasted computation time if the set is calculated more accurately, than the displaying device can represent it.

This application should lay its focus more on performance than on accuracy, but some level of accuracy has to be maintained. It can be said, that the accuracy of the calculations should generate fractal images, that are not or just in small

details distinguishable from the perfect image by the human eye. The fact of limited resolution actually helps a lot to fulfil this requirement even if the focus lies on performance.

4.2.3 Usability

The usability of the application is an important factor, that is often neglected by too technically or mathematically thinking computer scientists.

This application should provide a high level of usability to a wide range of users. The users are not expected to be experts or even advanced in neither of the touched fields, may it be computer science, mathematics or physics (signal processing and music theory). So everyone who has basic knowledge about how to use modern computer applications should be able to create his or her own customized music visualization after a short time of exploring and accustoming to the applications mechanisms.

But, also the usability is highly related to the performance the application can provide. If users have to wait for seconds to render each single image, they would not be happy with the application and quickly drop it. Therefore, performance has a high impact on usability.

4.2.4 Scalability

As already mentioned in the performance requirement, it is difficult or even quite impossible to set exact aims which should be reached considering performance of the application. Same is the case for scalability. It is obvious because of the need of a real time rendering of the set, that the application is not likely to run on older hardware. But users with hardware that is not performant enough, even for real time rendering in slow magnifications, should still be able to use the application for the exploration of the Mandelbrot set. Therefore, the application should provide mechanisms to lower the accuracy of calculations, so the performance can increase considerably to make the tool conveniently usable for users with slow hardware.

4.2.5 Platform (In)-Dependency

The application should run on current versions of the most common desktop platforms. That includes Linux, Microsoft Windows and Apple MacOS. The application should run on 32 bit as well as on 64 bit operating systems.

4.3 Architecture

Figure 4.1 outlines the basic architecture of the application. The application will be discussed in more detail in the following sections.

4.3.1 Model-View-Controller

The used architectural pattern is the Model-View-Controller (MVC) pattern. It strictly assigns three different roles to three parts of the application, which interact in a specific way. The model holds the application data and implements the business logic. The view is responsible for the visual representation of the program, and is the interface with which users directly interact. It has access to the model, to get

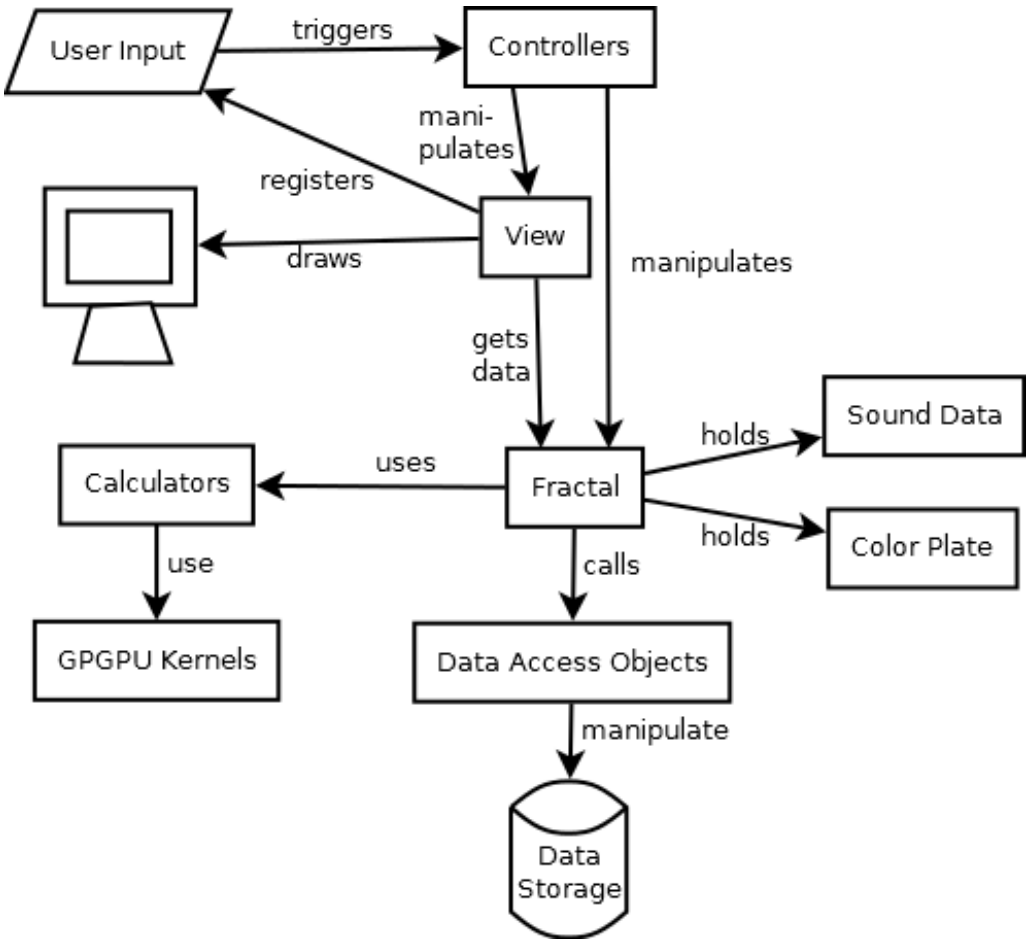


Figure 4.1: Basic application architecture

the data that should be displayed. The controller at last deals with the user inputs, reacts on them, and manipulates model and view according to it.

4.3.2 Data Storage Layer

The first layer of the application is the data storage layer. It consists of some kind of storage unit. So called "Data Access Objects" (DAOs) are used to manipulate the data, i.e. to read and write from and to the storage unit.

4.3.3 Model

The main component of the model in the application is the 'Fractal' class. It is the main hub of the application by accessing the data layer and holding the necessary data for the rendering of the set. It provides methods to navigate the set, i.e. to manipulate its parameters. It also provides methods to actually calculate the set through access to the 'Calculator'-classes. Some of the calculators execute code on the GPU, the so called "GPGPU Kernels". At last the 'Fractal' class also holds the sound and color mapping data and the methods for manipulating them.

4.3.4 View

The view consists of the classes which deal with the visual representation of the application. These classes create what the users will see on their screen and what

they will interact with. It draws the fractal images on the screen, and surrounds it with all the user interface components. The data it needs to display all this is obtained by the model.

4.3.5 Controllers

The controller classes are, so to speak, the nervous system of the application, because they react to impulses from outside (from the users) and trigger reflexes according to the signals they get. Depending on the input, the controller classes react by manipulating the model and the view of the application.

4.4 Technologies

The application uses a variety of technologies to fulfil the stated requirements. This section lists these used technologies, explains how they are integrated into the application and gives arguments about why a certain technology is preferred over others.

4.4.1 Java [22]

The application is mainly written in the programming language Java. The reason for choosing Java bases on different reasons.

- A fast high programming language is needed to solve this complex and calculation intensive problem.
- The author has the most experience in Java out of all high programming languages.
- The development in Java is platform independent, so the application can be developed for several different platforms, usually without changing the code.
- Java applications benefit from the improvements of the underlying Java Virtual Machine, without being modified themselves.

On the downside the main argument would be that Java is often considered to be slow and memory inefficient in execution. But in fact, when it comes to numeric computations and multi-threading, Java already reaches a performance close to C. [12] Therefore, this slight disadvantage is acceptable and Java is used as the main language for developing the application.

4.4.2 XML [23]

Application data has to be stored somehow, and there are many different ways of doing this. For instance, the application could use a database and choose between a huge variety of different database management systems. But since the application is not going to deal with a huge amount of stored data, a semi-structured data storage on file basis, namely XML, is more adequate because of advantages such as less data overhead and easy exchangeability of files.

4.4.3 JDOM [24]

A Java-representation of the XML data is required, for that matter the JDOM-API is used. There is no need for a SAX parser, because the amount of data is comparably low. So JDOM can manipulate the data that is stored in XML files by using Java code, and offers a representation of the XML data in Java so it can be processed further.

4.4.4 CUDA [25]

CUDA is one of the used technologies by the application for calculating the fractal. CUDA is a library of NVIDIA, that allows it to programmers to use a NVIDIA GPU (of a newer generation) for executing general purpose processing tasks. This application uses CUDA for executing the operations that are necessary to obtain the raster of integer values that visually represent the current set. CUDA code is written in a C-like language and compiled for a certain hardware architecture.

4.4.5 JCuda [26]

JCuda is an API written for Java for integrating CUDA programming into Java, i.e. CUDA code can be executed from inside a Java application. JCuda offers support for different platforms like Windows, Linux and MacOS, so the multi platform approach of the application can be retained. In the application JCuda is used for setting up the necessary CUDA environment, transferring data to the GPU, start the calculations, and finally retrieve the results from the GPU again.

4.4.6 OpenCL [27]

OpenCL is the alternative to CUDA programming, to run GPGPU calculations on many different supported hardware architectures. OpenCL code does not only run on GPUs manufactured by NVIDIA, but also on others such as AMD's or Intel's GPUs. The usage is similar to CUDA, the code is again written in a language close to C.

4.4.7 JOCL [28]

JOCL is a Java API to enable OpenCL programming in Java, just as JCuda is for CUDA. Also JOCL offers libraries for several platforms, and it has the same tasks as JCuda, the usage is just slightly different.

4.4.8 Minim [29]

Minim is a strong audio library for Java. It is usually integrated in the programming framework 'Processing'[1], but it can be integrated as well in pure Java.

The good thing about Minim is that it provides easy ways to perform simple audio analysis. It is one of the few good audio libraries for Java, so the range of suitable APIs is not wide. Minim can do simple signal processing with fast Fourier transforms, and provides possibilities for playing audio files and getting their meta data. Moreover, it includes classes which can be used for doing simple energy based audio analysis or a bit more advanced frequency band beat analysis.

[1] `http://processing.org/`

4.4.9 Swing [30]

Swing is the graphics API that is part of the Java Runtime Environment since version 1.2. It brings implementations of all the important components in modern client applications, and a variety of layout managers to assemble the UI components. Swing is based on the Abstract Window Toolkit (AWT), and looks much more modern than AWT does.

The main competitor for Swing is the Standard Widget Toolkit (SWT), but Swing was chosen over SWT. It is much closer to Javas "Write once - run everywhere" principle, because SWT has much more issues in adapting it for different platforms. [4]

Swing may also be used for drawing the fractal, by manually painting every single pixel with a certain color value.

4.4.10 OpenGL [31]

The Open Graphics Library (OpenGL) is a specification for developing 2D- and 3D-graphic applications platform independently. In this application it is used for drawing the fractal. The benefit of using OpenGL instead of Swing is the huge performance gain, because OpenGL directly calls corresponding operations on the GPU, which makes it much faster than the drawing methods used by Swing. [14]

4.4.11 JOGL [32]

JOGL basically wraps OpenGL into a Java library. Most of the functions that are available for OpenGL, are directly translated into Java and can be accessed with the exact same function call. Only some adaption were done because of some Java characteristics, such as simulating pointers because Java hides pointers from the developer.

5 Implementation Process

This section goes deeper into the implementation of the application. In particular, it presents the implementation process of the major algorithms and functionalities. In an own subsection the big issues that occured during the implementation process are discussed in detail.

5.1 Implementation Details

Each of the following subsection contains a brief description about the implementation of a part of the application. Furthermore, it reveals the motivation behind certain decisions made during the implementation process.

5.1.1 Mandelbrot Set Calculation

Iteration The basic underlying algorithm that is going to be used in the program is the "Distance Estimator Method" [1, p.196ff] with small simplifications, because it creates appealing images of the set and can be optimized in different ways. The idea is to map a pixel in the viewing plane to a corresponding point in the complex plane, which will be C for the iteration of $Z_{n+1} = Z_n^2 + C$ with $Z_0 = 0$. Then this point gets iterated, until its distance from the origin exceeds a critical boundary or until it reaches a defined maximum number of iterations. If the boundary is exceeded, it is assumed that the value diverges, and the number of iterations that were needed is used to color the pixel in a certain color. But if the iteration continues until the maximum number of iterations, it must be assumed that the value will not diverge and therefore the point is in the Mandelbrot set and gets colored accordingly.

The basic algorithm to calculate and draw the Mandelbrot set does not implement any speed improvement or other optimizations. It is a simple raster scan, that iterates over every pixel of the viewport and iterates the orbit of the corresponding point.

There are several methods to optimize the calculation of the Mandelbrot set. The most suitable ones will be discussed in the following.

Mirror Symmetry Maybe the simplest and most obvious optimization is using the mirror symmetry of the Mandelbrot set at the real axis. The set has several symmetries in it, but only the symmetry on the real axis is perfect. So if the displaying window contains the real axis, it would be enough to calculate the bigger of the two parts (either above or below the real axis), and then mirror the values on the real axis.

There is one problem with this optimization. Because the rendering happens at pixel level and the value of the complex plain gets calculated for pixels, it is highly unlikely that the real axis will be exactly on one pixel line, and therefore one cannot just mirror the image on that pixel line. Nevertheless, if the image is simply mirrored on the pixel line closest to the real axis, the mistake would be neglectable and since it is not the aim to calculate as accurate as possible, it would be reasonable to accept this small error.

Orbit Detection As it has been discussed in 2.1.12, a lot of values inside the set which do not diverge (and will therefore be iterated until the full maximal iteration count), will end up in a cycle of reoccurring values. Especially in the big buds - they

will be inside the display window quite often - the periodic cycles are quite small. The main bud for example has the periodicity of two. To exploit this characteristic, a certain amount of last calculated values of the orbit can be stored and new values can get checked if they coincide with one of the stored values, which would mean there is a cycle (even if it would not be a real cycle due to inaccurate arithmetic, the computer will not break out of this cycle anymore and therefore calculate the value wrong anyways). So the calculation for this point can immediately be stopped and the point is assumed to be inside the set.

Successive Refinement Successive Refinement is a popular and efficient method to improve the speed of the Mandelbrot set calculation. Moreover, it has progressive calculation inherent, so without additional effort (except for the rendering) a progressive drawing method can be obtained, hence a user can already see a coarse approximation of the set which gets improved step by step until it eventually becomes pixel (or even subpixel) accurate.

The following implementation ideas for the successive refinement algorithm base on [8] and [1].

The basic progressive calculation is simple to implement and to understand. The pixel raster first gets grouped into so called chunks of several pixels, for example starting with blocks of the size of 8x8 pixels. Then the usual iteration method is executed for one pixel in each of these chunks (which will be the top left one), but the whole chunk will get colored according to the outcome of the calculation for the one pixel. This results in the Mandelbrot set in a much lower resolution, which already provides a glimpse on the look of the final image. In the next step, the chunk sizes is cut in half, e.g. 4x4 pixels, and the iteration process is repeated, which leads to a refined picture of the set. This method is repeated, until the chunks are in the size of only one pixel.

This is no speed improvement, but it will increase the calculation time significantly by a factor of about 2. The first logical step to improve the performance (and approximately reach the performance of the naive algorithm) is to avoid to calculate the pixels again which were already evaluated in the former step to determine the value for the parent chunk. To do so, in each step it is checked if the quotient of the x and y-value of the the pixel to evaluate and the chunk size is divisible by two, hence

```
if (((x / chunkSize) % 2 == 0)) &&
  ((y/chunkSize) % 2 == 0)) {
    "skip evaluation and use parent chunk value"
}
```

, because if this equation is true, the value was already calculated before and that value can be adopted.

But how can this method lead to a real performance improvement? This refinement method can be used to exploit a characteristic of a calculated coarse pixel map to save certain calculations in the next refinement step with lower chunk size. The short explanation is that for all neighboring chunks of some chunk should be checked if they all have the same 'value' (iteration escape count) and if so, the iteration can be skipped and the neighbors value can be adopted.

The first step to implement that is to save the values that are calculated in one refinement step and carry them on to the calculation with the half size chunks. So

in the next refinement step, the 'parent' chunk for this new chunk first needs to be obtained.

```
if (x \% (2*chunkSize) == 0)
  parentX = x;
else
  parentX = x - chunkSize;
```

Analogous this works for the vertical direction. This works because if the top left corner of a chunk was already calculated in the step before, the parent position is the same as the current one. But if this is not the case, the current chunk size needs to be subtracted to get to the position of the top left corner of the parent chunk.

Carrying out this test makes sure that pixels that were already evaluated in the parent step are not evaluated again. Now a new test is added, which examines if in the former refinement step all the neighbors of this chunk had the same value, and in that case adopt this value for the current chunk without iterating it. The distance of the neighboring parent chunks to this parent chunk is the current chunk size times two.

So in the end, if a chunk was already calculated in the former refinement step that value is adopted, or if its parent neighboring chunks all had the same value, this value is adopted. If these two tests fail, the usual iteration is carried out and the resulting value is adopted to the whole chunk. This is repeated until the chunks are of the size of one pixel.

This method works especially well on low magnitudes. For example if the whole fractal is on the viewport, in the main bud the iteration process will only be done a few times and then this values will be adopted over and over again because all the neighboring chunks have the same value. Also in the outer region, the same value appears over a big range of the viewport and therefore a lot neighboring tests will succeed. In higher magnitudes, especially on the edges of the set, the method will not be so much of an improvement, because the values differ a lot from pixel to pixel. But the calculation time is usually still improved significantly, so it is definitely well worth the implementation effort.

Again this algorithm leads to a loss of accuracy, especially narrow features such as cusps can be missed with this method. But the mistakes are comparably small to the huge performance gain, and can be neglected as well.

Multi-Threading So far the implementation of the successive refinement method does only explicitly make use of one thread and therefore one CPU core. But the algorithm is easy to split up in several pieces, by just cutting 'slices' out of the viewport and assign each of them to an own thread. It must be considered that the algorithm uses its neighboring chunks for the calculation of each chunk, that means that the slices will not be calculated completely independent from each other. In fact, only the chunks on the edge of the slices can be dependent of their neighboring pieces. One possibility to deal with that, is to simply evaluate the chunks on the edge of a piece every single time, and do not check the parent map for equal neighbors. That results in a small performance loss, but ensures the correct calculation of the set. Another possibility is to synchronize the threads, so that after each refinement step the threads are waiting for their siblings to finish the step as well, and then go on with the next refinement step.

GPGPU Processing The performance of calculations with higher magnitudes will not be sufficient for real-time rendering, even on current consumer CPUs. So GPU calculations could be the solution for a highly parallelizable problem like this one.

The first step was to implement the naive calculating algorithm in CUDA, so it can be executed on NVIDIA's graphic processors which support CUDA calculating (that is the case for almost every GeForce chip of the 8000-generation or newer). In a next step, the algorithms are also implemented in OpenCL, which is quite easy because syntax and functionality of both CUDA and OpenCL are similar.

The calculation cannot be executed with double arithmetic on every graphic processor, but this and many other issues in the process of porting the algorithms to the GPU will be discussed in 5.2.

The performance gain by porting this simple algorithm to the GPU is huge. But the result is still not really satisfying, because the algorithm was not yet optimized in any way and has therefore a lot of potential in performance increases. The first optimization to be made was implementing the successive refinement method for the GPU. But this led to several problem, which will be discussed in detail in section 5.2.4.

Calculation Method Comparison After discussing several different calculation methods, table 5.1 should now give a brief tabular overview about the differences between the seven implemented calculators.

Calculation Method	Algorithm	Calculation Hardware	CPU Threads	Arithmetic
Basic Calculator	DEM/M [1, p. 197f], Simplified	CPU	1	Double
Successive Refinement	Successive Refinement [8]	CPU	1	Double
Multi-Threaded SR	Successive Refinement [8]	CPU	1-8	Double
CUDA	DEM/M [1, p. 197f], Simplified	GPU (NVIDIA)	-	Float/ Double
CUDA SR	Successive Refinement [8], Modified	CPU+GPU (NVIDIA)	1	Float/ Double
OpenCL	DEM/M [1, p. 197f], Simplified	GPU	-	Float/ Double
OpenCL SR	Successive Refinement [8], Modified	CPU+GPU	1	Float/ Double

Table 5.1: Overwiew of implemented calculation methods

5.1.2 Drawing

In low magnifications the drawing of the set with a simple method of drawing 1x1 rectangles in AWT is a bottle neck. The drawing of the whole panel in an average resolution can already take more than 100 ms, which is weigh too much considering the targeted total rendering time (calculation plus drawing) of 40 ms. So there clearly is a need for improvements in the drawing of the set.

The obvious way of improving the performance of drawing, is to use OpenGL for it. Instead of using simple integer RGB values, it is necessary to cast the values for the color components into byte values. After that, they get stored in a flat array, which lines up the rows from the bottom to the top of the screen, so they have to be flipped because they are stored from top to bottom. But including all this additional effort, the drawing by using OpenGL only takes about 10 ms.[2]

5.1.3 Navigation

The two basic navigation features of the application are zooming and panning.

One way of zooming into or out of the set is to react on a click into the viewport by centering the set at that position and changing the magnitude by a certain amount. In that way, the users are able to jump to desired positions which seem interesting to them. This is fairly easy to implement, by figuring out the clicked complex number (which can be calculated out of the clicked x- and y-position and the current viewport parameters). Then this complex number is moved to the centre of the viewport, and the magnitude is adjusted, depending on the chosen extent of the jumps and if it is a jump in or out. The drawback of this method is the waiting time for the users until the new set is calculated, during which they do not have a feedback if the position is really the one they want to go to.

An improvement to provide a more intuitive way of zooming is to not jump directly to a position, but to fluently approach a position which is clicked by the users, as long as they hold the mouse button pressed. This is done in a separate thread, so at the same time mouse events can be registered. The user clicks with his or her left mouse button at some point in the viewport, and keeps the button pressed. The clicked position is saved in the zooming thread, and this point will be continuously approached by the zoom. The pace of how quick this point should be approached can be set by a parameter. When the mouse pointer is moved, the approached point changes right away by setting the new position as the approached one in the zooming thread. This works analogous for zooming out by holding down the right mouse button. Moreover, before starting the zooming process, the minimum chunk size can be set up to a higher value, which means that the calculation will be much faster, but only create the set in a lower resolution. But for the images during the zoom it is most important to see a glimpse of where one is zooming, it is less important to see a detailed image. After the zoom is done, the minimum chunk size has to be set down to 1 again, and the set must be calculated once more to show the fully detailed image.

The users could also want to not change the magnitude of the set, but just change the position at the same magnitude. This is referred to as panning. A way to pan is to use the arrow keys of the keyboard, which changes the center of the set by a certain degree. For instance if a user presses the left arrow key, the center of the real part is decreased and the piece of the set which is displayed moves to the left.

A certainly more convenient way for the users to pan, is to use the mouse for it. It should be possible to click on a position in the set, and 'drag' the set around. So the users can 'grab' the set at some position, and drag this position around the viewport, e.g. drag it to the center of the viewport.

[2]Times were measured on a system with an Intel Core 2 Duo CPU and NVIDIA GeForce 8600M GT GPU

5.1.4 Jumping to a Location

Besides the interactive navigation on the displayed set, the user may also want to jump to a certain location in the set by entering some specific data about this location.

The part of the fractal that is shown in the viewport is dependent on some parameters, which namely are the real values on the left and on the right edge, the imaginary values on the top and the bottom edge and the magnification of the set. The centre of the viewport is represented by a complex number, which can be calculated out of the minimum and maximum values.

The users should be able to insert the fractal parameters as they wish. For that matter, a form was implemented which allows the users to insert the complex number to be placed at the centre of the viewport (imaginary part and real part), the magnification, and finally the count of maximum iterations (see 5.1).

Figure 5.1: Location toolbar

5.1.5 Saving/Loading Locations

Saving locations means, that the users should be able to store a location where they are currently at in the viewport, so that they can easily load this location later.

The implementation of the DOM-Parser is short and simple, because the underlying storage XML files are fairly small, and the saving and loading algorithms are self explanatory.

After first having a simple drop down list of locations to choose one of them, it became obvious that this is not really convenient. Especially, there is no way to identify the saved locations just by a simple name given to the location. So an own window was created for loading positions, by showing a list of all saved locations, and by selecting one of them, a thumbnail of the locations image is displayed as well as the details about this particular location. It that way, the users can see right away, which location they are about to load (see figure 5.2).

The main implementation issue was that every thumbnail needs his own data of what should be drawn, and for that a new fractal object has to be created with the according parameters. These fractal objects also need the right color plate, because the locations are saved with a maximum iteration parameter, so when the thumbnail is displayed, first a color plate with the size of the according maximum iteration parameter must be created.

5.1.6 Taking Snapshots

A nice feature for users is to take snapshots of the currently drawn image. To do so, the current image must be drawn into a buffered image instead of the drawing panel. Then this image can be written somewhere to the file system in a chosen format. The users can also decide the resolution of the saved image (see figure 5.3). If this resolution differs from the one of the currently drawn image, the image must be recalculated with the new parameters. After doing so, the picture can be saved, and the fractal parameters can be set back to their former values.

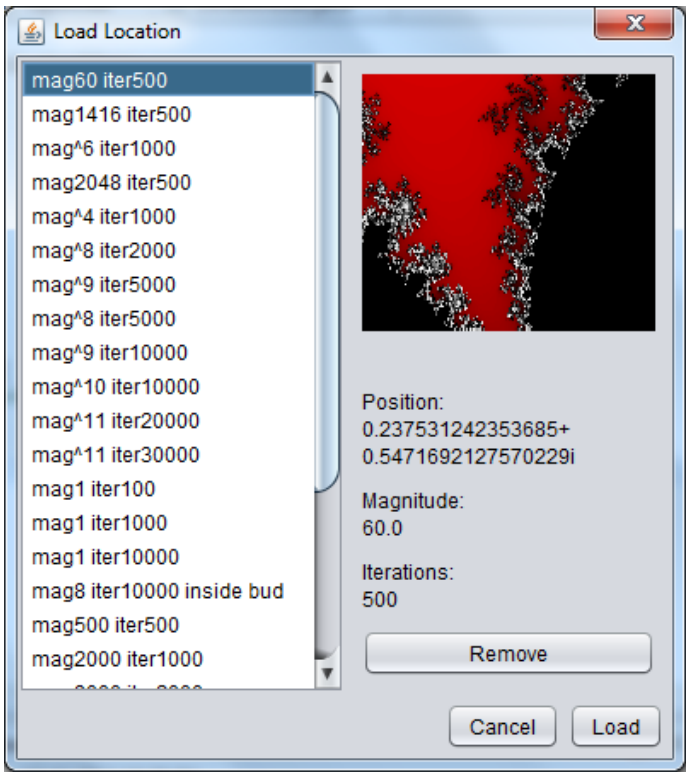


Figure 5.2: Load location dialog, showing preview of selected location

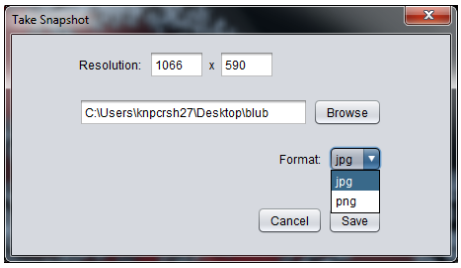


Figure 5.3: "Capture Image" dialog

5.1.7 Creating Color Plates

A central feature of the program is to create own color plates to make the color mapping highly customizable and therefore let it up to the users in which coloring the set appears and to make the zooms look as appealing as possible.

In principle, a color plate that is usable in the application is an array with the size of the maximum number of iterations before a point is considered to be inside the set. Each element of the array is a color value. For every pixel in the viewport there is a calculated escape iteration count, so for each of these pixels a color value is looked up in the array.

But saved color plates should be independent of the number of maximum iterations, because that changes frequently. For that matter, color plates are made up of color keys, which hold a color value as well as a relative positioning value from 0 to 1, where 0 represents the first element in the color plate array, while 1 represents the last one. The color of a key gets mapped to the final color plate array by using its relative position value to place it in the corresponding element of the array. All array elements in between are calculated by linear interpolations of the two enclosing

color keys, with each color component (red, green, blue) interpolated separately. So a color plate consists of at least two color keys, one at the beginning and one at the end, and can have arbitrarily many more in between.

If the users want to create a color plate, an editor is opened in a new window (see figure 5.4). There, the users see a transition stripe of the current coloring. This stripe illustrates how the color values are assigned to the number of iterations that a point needs to escape. The left side of the stripe shows the color that is linked to the value of 0 needed iterations, while the right side shows the colors assigned to the highest escape values, and the very last element on the right shows the color assigned to values which do not escape in the specified number of maximum iterations (this is the color of the interior of the set).

The user now has the possibility to add additional color keys in between of the two default keys. Then the interpolation is carried out between each neighboring pair of keys. New keys can be added by clicking somewhere on the color stripe. The position inside the stripe of already existing color keys can be changed by drag and drop. A key can be selected by clicking the circle on the bottom of it, and then the keys color can be manipulated or the key can be removed.

The implementation of the color plate editor ensures, that changes in the plate will immediately affect the currently drawn fractal. So the users can understand right away how their actions on the color keys of a plate affect the image.

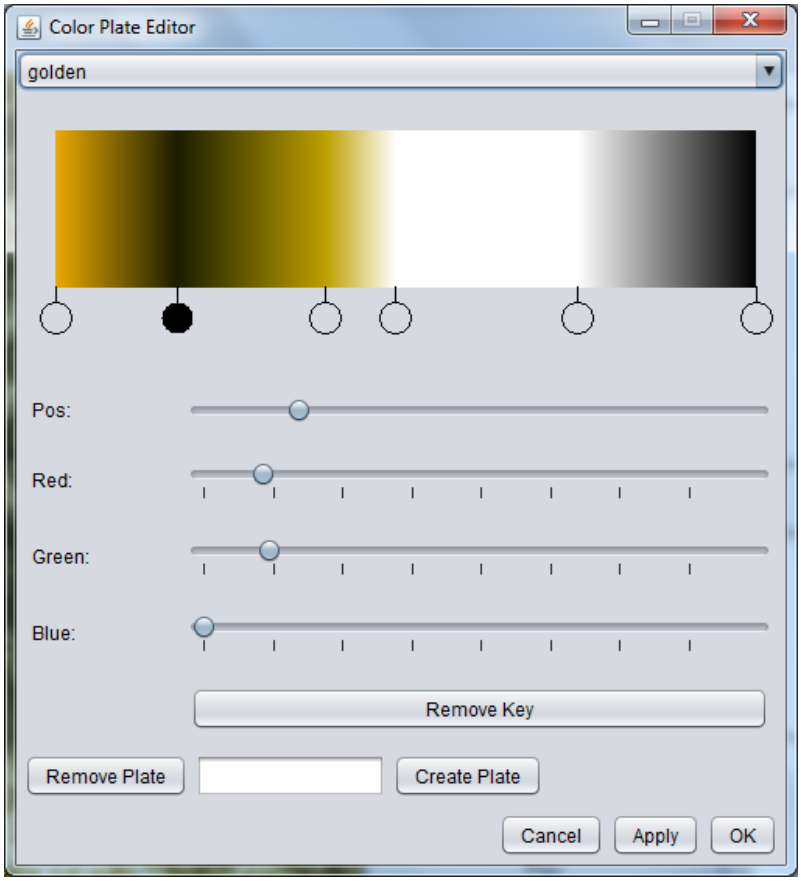


Figure 5.4: Color editor showing current color transition and color keys

There are some cases which need special treatment. One is, if either the key at the beginning or at the end of the transition stripe is moved. Then a new key at that position must be created, because there must necessarily be keys at these positions.

Another case is, when only one color plate is left and the user wants to delete that one. This would cause a series of exceptional behaviours in the application that would have to be handled. The easy way to avoid that is to simply prevent the user from deleting the last color plate. Since this is not really a drawback for the user, this is the way that is used in the application.

5.1.8 Creating Zooms for Music Visualizations

Since also not experienced users should be able to create their own zooms for a music visualization, the procedure of creating these zooms should stay as simple as possible, but should still give the chance to create zooms in exactly the way users want them to look.

The basic model to store zooms is quite similar to the one used for color plates. To make it easy to change the length of zooms, which can be of benefit to adjust their length to the length of a piece of music, the saved zooms should be independent of the total length. Therefore zoom keys are being used, where every zoom key contains a location in the set and a relative time value from 0 to 1. There is one key at the very beginning with time value 0 and one at the end with time value 1. Additional keys can be added in between, and the locations in between of the keys are interpolated.

The implementation first consists of buttons to create zooms, remove them, save modified zooms and load previously created ones. The loading of zooms looks similar to the loading of locations, the users can choose a zoom from a list and see a preview of it through several thumbnails at certain points in time of it (see figure 5.5). Also they see the important information about it. The button to remove zooms simply removes the currently loaded zoom. The saving button saves all the changes that were made to a zoom since it was last loaded. The button to create a zoom opens a simple form where the users can insert basic information about it. They can also choose a template for this new zoom, i.e. they choose a zoom that was created before, and the new one will start out with the exact zoom keys of that old one. If they want to create a zoom from the scratch, a default zoom with only a key at the beginning and the end is created.

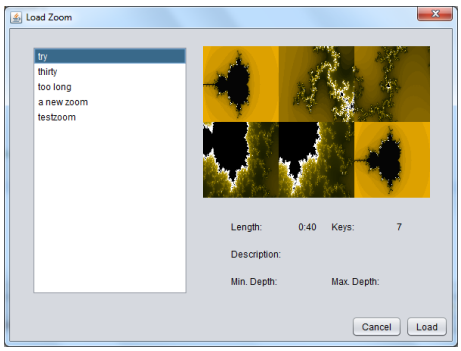


Figure 5.5: Load zoom dialog with preview of selected zoom

The user interface for creating zooms is a panel placed on the right side and looks as in figure 5.6. When a zoom is loaded, a timeline on the right shows the position of the zoom keys. The timeline looks similar to a ruler, with ticks on every second of the zoom and a long tick every five seconds. The length of the whole zoom can easily be edited, and is therefore not of big importance. But it is useful for testing a

zoom with a certain length to see how fast the movements appear. On the position of zoom keys, a circle is attached to the according position on the ruler, which is calculated from the relative time value of the key. The keys can, analogous to the keys in the color plate editor, be dragged and dropped to change their time value, and they can be clicked to select them. Selected keys can be deleted. In the fractal viewport the location of the selected zoom key is rendered. The zoom will be on exactly this location at this time. The location can be changed by usual navigation, and by clicking the "set key"-button, this new location is set for the selected key. By clicking somewhere on the timeline, where no key is already existing, a new key is created with the respective time value and the location that is currently set in the viewport.

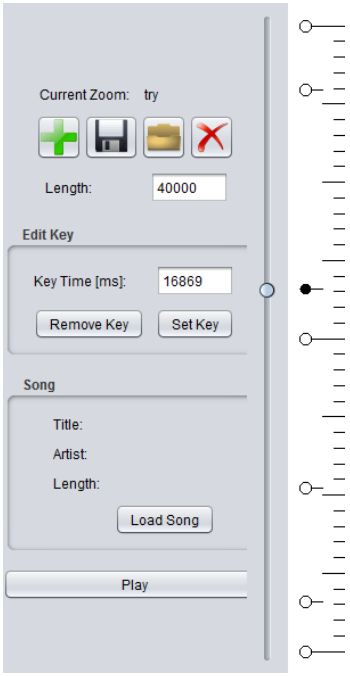


Figure 5.6: Zoom panel with timeline

There are special cases that are similar to the ones of color plates. If the key at the beginning or the end of the timeline is moved, a new one must be created on that position because there must necessarily be a key at these positions. Also, same as with color plates again, there must be always at least one zoom left, the last one cannot be deleted.

An important tool for creating and playing back zooms is the time slider. It is a simple slider next to the timeline, with the same length and value range as the timeline. When moving the slider, the time value is updated and the location is rendered at which the zoom would be at that exact point in time at which the slider resides. So the users can easily jump to an arbitrary time in the zoom to see the location rendered at that time. This slider triggers all the necessary actions to change the time values, render the fractal and redraw everything.

The interpolations between zoom keys work quite simple. The positions, i.e. the real and imaginary part of the complex number in the centre of the viewport, can be interpolated linearly, while the interpolation of the magnitude should be exponential.

5.1.9 Playback Zoom

The next step was the implementation of the playback of the zoom with or without music. The playback is carried out by an own thread. This thread basically only changes the position of the time slider, the listener of the slider takes care of the rest. So the playing thread has a loop in which it increases the value of the time slider. Since the aim is to have 25 frames per second, one loop iteration should take 40 ms, and increase the sliders value by 40. Now, the rendering will usually not take up exactly 40 ms. So if it takes less the thread should just sleep for the rest of the time until it reaches a total iteration time of 40 ms. If it takes longer, the total spent time in this iteration step is added to the slider value, so the zoom does not get delayed.

It is now possible to play back a zoom, but not yet any music to it. The zooming panel should provide the possibility to load a song from the users file system. Users can choose a song, and then they have the choice if they want to adjust the songs length. If they do not want that to happen, there are two possibilities left:

- The song is shorter than the zoom, in that case the zoom will just stop when the end of the song is reached.
- The song is longer than the zoom, the zoom will be looped and finally stop when the song comes to its end.

If a song is loaded and the user clicks the play button, the zoom is started at the time value of the current position of the time slider. Also the song starts at the time that is given by the time slider. So the slider not only moves the time position of the zoom, but also the one of the loaded song.

5.1.10 Rotation

Before the implementation of the music analysis and the synchronization of the zoom to it, the rotation of the fractal should be implemented. That will be of importance to change the fractal according to the music that is played.

The rotation of the set needs some mathematical considerations. What will change comparatively to the current version of the algorithms is the mapping from a pixel in the viewport to the according complex number. This mapping was straight forward until now. The value of the first pixel in the top left corner was already given by the minimum real and the maximum imaginary value of the current section of the fractal. Then, when increasing the x-value, also the real value is increasing. The factor of this increase is the total real range divided by the total pixel width of the viewport. The increasing of y-values works analogous.

This changes now as the fractal should be rotateable. When the x-value is changing, not only the real value will change, but also the imaginary value. The same is true when changing the y value. The factors by which the real and imaginary value change on each increase of x and y should be calculated like this:

$$dXReal = \frac{realRange}{fractalWidth} * \cos(\frac{rotation}{180} * \pi)$$

$$dYReal = \frac{realRange}{fractalWidth} * \sin(\frac{rotation}{180} * \pi)$$

$$dXImag = -\frac{imagRange}{fractalHeight} * \sin(\frac{rotation}{180} * \pi)$$
$$dYImag = \frac{imagRange}{fractalHeight} * \cos(\frac{rotation}{180} * \pi)$$

That is not all, because the starting value for the top left corner pixel is now not given any more by the minimum real and maximum imaginary value, but must be calculated according to the current rotation value.

$$startRe = realCenter - fractalWidth/2 * dXReal - fractalHeight/2 * dYReal$$

$$startIm = imagCenter + fractalWidth/2 * dXImag + fractalHeight/2 * dYImag$$

Then, for each pixel, the according complex number can be calculated as follows:

$$cRe = startRe + x * dXReal + y * dYReal$$
$$cIm = startIm - x * dXImag - y * dYImag$$

5.1.11 Music Analysis

The approached final feature of the application is the music visualization, consisting of a zoom into the Mandelbrot set. It is not desirable to simply play back a recorded zoom, but the zoom should change according to the playing music. Therefore, an analysis of the music has to be carried out, and a good way to map the data gained from the music analysis to the played zoom needs to be found.

There are several ways for analysing music, which are explained in the theoretical section 2.3. To keep the complexity reasonable, an API is used to do a simple energy and beat detection. The API will make it possible to gain this two parameters:

- Energy Level: A low energy level is related to quiet parts in the music. Naturally, if the music is quiet, it is desirable to let the zoom look quiet and slow as well. In contrary, if the energy level is high, the zoom should look dynamic and energetic.
- Beat Detection: Beats can be detected for different frequency bands, which results in several beat values. Each of them is a simple boolean value, that reveals if a beat occurs in one particular frequency band in one particular moment. It is desirable that something happens with the zoom when a beat occurs, especially on base frequency levels.

Now, what could be changed in a zoom? There are several parameters:

- Location: The location in the set at a moment in time. The locations change over time and form a zooming path.
- Location change speed: The velocity in which the location changes over time.
- Rotation: The rotation of the set, between 0 and 360 degrees, at a moment in time.
- Rotation change: The pace in which the set is rotating over a period of time.

- Color Plate: The current color mapping at a moment in time, represented by a simple RGB color value.
- Color change: The colors can change over time in different manners. For instance, the color transition stripe could be 'rotated', which means that the array which represents the mapping of iteration steps to colors is seen as a ring, and the colors get shifted inside this ring over time.

As locations are picked when creating a zoom, this parameter should stay fixed. This has the practical reason that the user will want to create a zoom which moves to interesting locations in the set, so the actual zoom playback should stick to these predefined path. An automatically generated path would most likely not show an interesting zooming path.

So there are two parameters to change over time, namely rotation and colors. Since there are also two parameters that can be easily gained from the playing music by using simple analysing methods, it is a perfect match. Only the best way to map the parameters to each other has to be figured out.

First a closer look is taken on the input parameter characteristics. The rotation of the fractal can have an arbitrary value between 0 and 360. The highest value is the same as the starting value (because 360 degrees is equal to 0 degrees). The color change can also be seen as iteration, but here the color values in a (imaginary) ring get iterated. This rotation can be represented in relative values between 0 and 1, where 0 is no rotation, and 1 is a full rotation around the ring and therefore recreates the initial state of the array. In using these relative values, the color rotation is independent from the size of the color array.

Next a look is taken on the music parameters. The energy level is given as a parameter between 0 and 1, so it is easy to map this parameter to one of the two input parameter. The beats are represented as booleans. A way to map this to one of the input parameters could be to execute a 'jump' in the rotation value when a beat occurs.

So, should a level change cause a change in the rotation of the fractal and a beat cause a change in the color rotation, or the other way around? Both makes sense, because a high energy level as well as frequent beats should cause 'high activity' of the zoom, so a fast rotating fractal as well as fast rotating colors. It is more or less arbitrary which way to map it, and it will depend on the kind of music or on personal taste, which way looks better.

The problem here is, that rotating the colors can look quite exhausting to the eyes, especially if this rotation is fast or simply if the rotation speed changes a lot (which happens when reacting to music). For that reason, it should be up to the user if the color rotation should really be reacting on music or just be static (with a rotation speed also defined by the user).

The way the color rotation reacts to the detected beats is by rotating faster right at and after a beat, and then decay again to a slower rotation. This is calculated by the exponential decay function

$$N(t) = N_0 e^{-\lambda t}$$

, where N_0 is the initial rotation speed right at the beat, λ is the decay factor (so if λ is big, the decay is quick and the rotation speed will get slower quickly after a beat), and t is the time since the last beat.

Figure 5.7 shows the application interface during the execution of a music visualization.

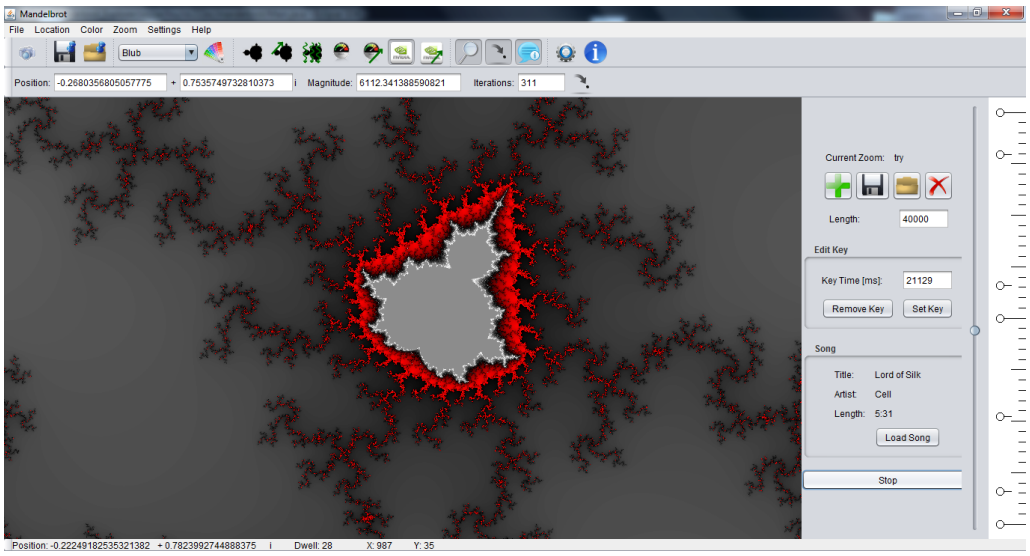


Figure 5.7: Playing visualization, reacting on music

5.1.12 User Interface

The tool should provide an easy to use interface to the user, so it does not take a long time to understand in which way they can explore the set and create music visualizations. The main view of the application can be seen in figure 5.8.

The Mandelbrot set image should take up the biggest part of the frame. A standard menu bar should be provided for interaction. A toolbar should appear right under the menu bar, to provide the most important actions to the users, so they only have to use the menu bar for less frequently needed actions. A second toolbar should show information about the part of the set that is shown at the moment, and should provide the possibility to quickly change this location by entering a new position. A status bar on the bottom should show some feedback about where the user hovers the mouse over in the set, e.g. the corresponding complex number and the iteration steps that where needed to escape the set for that number. Moreover, it should show what the tool is currently up to, e.g. if it is calculating a new location, if it is drawing, or if it is in idle mode. Finally, at the right side of the main view of the Mandelbrot set, the users can open an interface for creating and playing the music visualizations.

There are some more things that have to be taken care of: If the window is resized, the resolution of the viewport and therefore the resolution of the fractal image changes. It is important that in this case, if the ratio between width and height changes, the fractal should not be skewed, but should keep its right ratio.

When a toolbar or the zooming panel on the side are hidden, the resolution of the fractal must be adjusted immediately, without losing its correct ratio.

The users can switch into a full screen mode, which lets them solely see the fractal image without any disturbing UI elements around it. This mode is especially suitable for playing the music visualizations. To obtain a full screen mode in Java, there have to be done some steps, which involve recreating the entire JFrame.

5.2 Issues

In this section some major issues that where occurring during the development process are discussed. The focus is not on discussing small bugs in the code, even

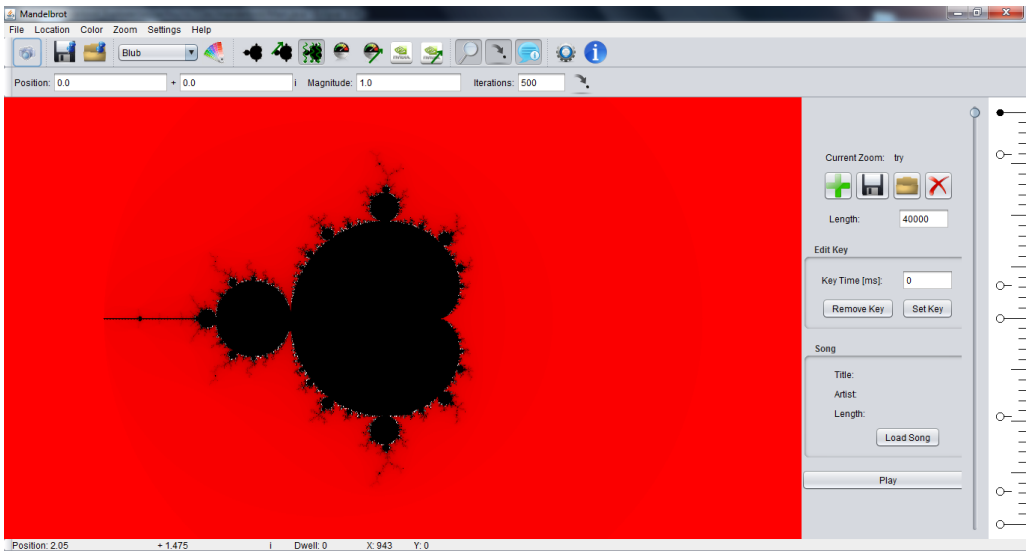


Figure 5.8: Main view of the application with all UI elements

if they were quite cumbersome to fix, but it should be on bigger issues that either delayed the development a lot, threatened the fulfillment of requirements or even are still unresolved.

5.2.1 Exact Playback Speed of Recorded Zooms

As described in 5.1.9, the zooms are played back with aimed 25 frames per seconds, which means a new frame has to be rendered every 40 ms. If the rendering is quicker, the thread is waiting for some time to fill up 40 ms. If the rendering needs more than 40 ms, then the time value is increased by the time that was needed instead of 40 ms. So by that, the zoom should always be in sync with the played music. That is especially important when adjusting the zooms length to the length of a song, because the users would expect the zoom to end exactly at the end of a song. And it is also important if a user tailors a visualization exactly for a particular song.

But in fact, zoom and song are not always in sync. The reason for this is, that Javas 'sleep' function for threads is not exact. So even if the mistake that is made for every rendered frame is little, in the end of a long song it can add up to a significant factor.

A possible solution for this would be to resynchronize song and zoom, say once every second.

5.2.2 CUDA Contexts

If some code in CUDA is written, that should be carried out on the GPU, the application first has to do some initializations. These includes creating a so called "CUDA context". This context will be associated with the calling thread. So if another thread wants to run CUDA code, it has to get its own context. This was an issue during the implementation process because different threads, i.e. the main thread, zooming thread, or music visualization thread, have to run CUDA code if a CUDA calculator is selected.

Since CUDA 2.0 it is possible to assign existing CUDA contexts to another host thread, without having to use an own context. Therefore the easiest way to fix the issue was to create a singleton that creates a CUDA context only the first time it

is called. From then on, before every action that is run on the GPU, the CUDA context is pushed to the current thread and when the action is finished, the CUDA context is popped back off the thread. In that way, the context is always assigned to the proper host thread and does not have to be recreated (which would be more time consuming).

5.2.3 GPU Arithmetics

GPUs from older generations (generally speaking about 5 years or older) do not support 64 bit floating point arithmetic (corresponds to the double type in Java and C), but only 32 bit arithmetic (corresponds to the float type in Java and C). Since the threshold of how far the zooms into the set get calculated accurately is bounded by the precision of the used arithmetic, this is a big issue in using the GPU for calculating the set. With float values a magnification level of about 10^5 can be reached, while double values allow magnification levels of about 10^{14}. Also, the algorithms written in CUDA and OpenCL must be adjusted to the supported arithmetic.

The users are allowed to change the used arithmetic for calculations on the GPU, so if their GPU would not support double arithmetic but can still run CUDA or OpenCL code, the GPU can be used for calculations, but the maximum magnification level is lower.

Since this boundary is fairly low, a third mode besides the two for using float and double arithmetic was implemented. This mode simulates double values by using two float values. [13]

It needs to be noted that using double arithmetic instead of float arithmetic slows down the application considerably, and simulating double arithmetic is again slower than native supported double arithmetic.

5.2.4 Slow Successive Refinement on GPU

As mentioned in section 5.1.1, after porting the naive calculation algorithm from Java to CUDA and OpenCL, the performance gain was big but not satisfying because of missing optimizations.

So the next obvious step was to implement the successive refinement algorithm on the GPU to gain a better performance. The simple way of doing that was to straight forward port the algorithm from its Java version to CUDA-code, which can be seen in appendix A.5.

But apparently this code leads to a slower execution time than the naive algorithm on the GPU. This in a first moment seemingly strange behaviour becomes clear when looking at the architecture of the GPU, as it was already explained in the theory in 2.2.1. The data domain is split in small blocks, usually of the size of about 8x8 elements. If there are branches inside the CUDA-code, and the execution of the code on one of these blocks leads into one of the branches, this branch is executed for all of the elements in this block. In the algorithm above, there are the main branches of either evaluating a value, or adopting a previously calculated value. The computational effort for these two branches differs extremely. But by looking at the successive refinement algorithm it gets obvious that there will be most likely at least one of the elements in a 8x8 block going to the expensive evaluating branch, so this is also executed for all other elements in the block. That means, for every refinement step almost every single element in the big raster is evaluated, but most

of the results are dropped because the evaluating branch is just executed because of the 'penalty', as explained before. Therefore, the more refinement steps are done, the slower the algorithm will be. A lot of the executed calculations would not be necessary to do at all!

To show that this is feasible, a simple test with this version of the algorithm can be done. If the algorithm was executed with a starting chunk size of 16x16 on a particular location, it needed about 160 ms of calculation time in average.[3] Now, if the former argument was correct, the performance should be increased considerably by lowering the starting chunk size.

- 16x16: 160 ms
- 8x8: 150 ms
- 4x4: 120 ms
- 2x2: 90 ms
- 1x1: 55 ms

Considering the general overhead, this is clear evidence that each refinement step needs approximately the same calculation time, and therefore the above argument is feasible.

Assuming the argument was true, a modification was made to the algorithm to workaround this problem. This can be done by splitting the successive refinement algorithm into two parts, one done by the CPU and one done by the GPU. The decision making which chunks have to be evaluated in an iteration step can be done by the CPU, because this process includes a code branch that decides if heavy computation has to be done or not. After that, all position that have to be evaluated are added to an array, which is then sent to the GPU. The GPU does the heavy computations, i.e. the iteration process for the given positions, and sends the result back. In that way, the GPU can strongly parallelize the computation and has no branches. Furthermore, only the necessary information is transferred from CPU to GPU memory instead of the whole pixel array. This should result in a storng performance gain.

In fact, a performance gain is achieved compared to the first 'naive' version of the successive refinement algorithm on the GPU, but it is still slightly slower in most locations than the basic algorithm.

5.2.5 Grabbing Sound Signal from Soundcard

In the beginning the plan for the visualization playback was to let users play songs from an arbitrary source, and the application will pick up the sound signal somehow to analyse it and synchronize the visualization to it.

But that turned out to be a major issue, because it is not easily possible to grab the signal that is played on the sound card, especially in Java. In fact, there is no library for Java that can do this job.

[3] All the tests in this chapter were done on an Intel Core i7 (quad-core) CPU and a NVIDIA GeForce GT 630M GPU

6 Results

This section evaluates if the requirements that were raised for this bachelor project were fulfilled and to which degree the outcome of the application development was satisfactory.

6.1 Functional Requirements

The functional requirements that have been formulated for the application are widely fulfilled.

The aspired navigation features are fully implemented in the application. The navigation in the explorer is easy to understand and straightforward to use. The application gives feedback about the location where the user currently is, and the users can change several settings to adjust some navigation details to their wishes. By showing thumbnails when the users want to load a previously saved location they can immediately recognize locations, which is a big advantage in comparison to other Fractal Explorers, because giving a name to a location gives no guarantee that the location can be found again easily.

The color plate editor is a convenient way to create own color mappings. Not only can users completely adjust the appearance of the set by setting the color keys, but the automatic interpolation between color keys creates a smooth transition of colors instead of distinct jumps. And by immediately seeing the effect of their actions to the set it is easy to understand how to create own plates and what is of benefit for the appearance of the fractal.

The creation of own zoom paths into the set is a crucial feature, that needs to be as easy to use as possible but at the same time give free hand on creating completely customized zooms. This tradeoff was reached by again using the principle of key values, so the users can add interesting locations in the set at a good point in time, while the application takes care of the interpolation in between.

The visualization that reacts on songs in the background could have been a whole own topic for a thesis, because the range of possibilities is almost endless. The decision to only change rotation of the set and the rotation of the color plate is a good way to make it predictable for the users how the zoom they create will look "in action", but still react to the music to a degree that the zoom looks different to every different song that is played.

The functional requirements that were not fulfilled are the import and export features for locations, color plates and zooms, because time was running out and this feature seamed to be the most reasonable to cut first, because it is not a central feature that impacts other features as well.

6.2 Performance

From the beginning of the project on it was obvious that besides usability and music analysis, the performance would be the crucial point in the development process. Even if the application would make it possible to create the most amazing music visualizations ever, there would be no point in it if almost nobody can play them on their hardware because it is not fast enough for real-time rendering, and therefore the animations are not displayed fluently.

The performance that was reached is good enough to render simple zooms, i.e. zooms that do not go far into the set but stay more on the 'surface', in real-time

on modern consumer hardware, especially if a GPU is available to support the calculations. So the main aim of showing the beauty of music visualizations created in a Mandelbrot Set Explorer was reached.

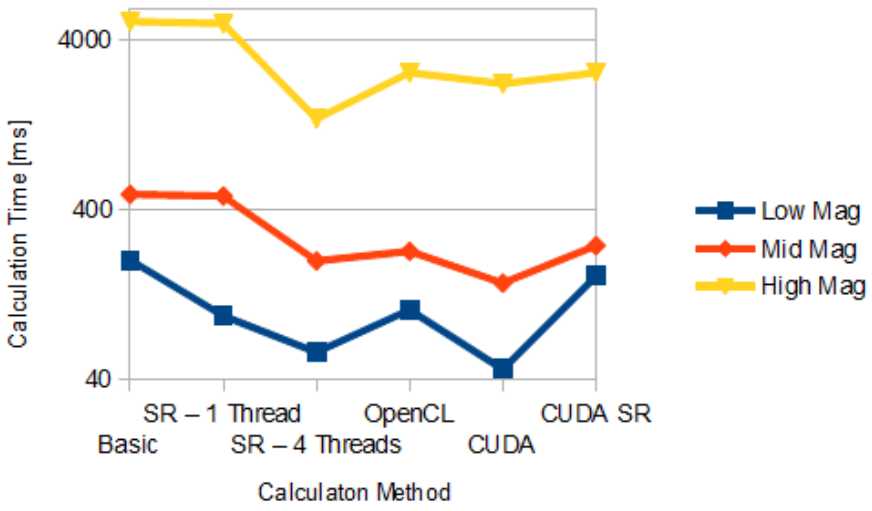


Figure 6.1: Performance comparison between calculation methods

Figure 6.1 shows a performance comparison of the implemented calculation methods ('SR' stands for "Successive Refinement", and note that the time scale is logarithmic). Some important observations in this comparison are:

- Switching from the naive distance estimator method to successive refinement has a huge performance boost in low magnifications, but only a small one in higher magnifications (see 6.2.2).
- The performance gain from using more than one thread varies considerably at different locations in the set (see 6.2.1).
- OpenCL is much slower than CUDA, especially in low magnifications. This is mainly due to an issue with the context initialization of OpenCL, which creates a lot of overhead, but also in general the execution time with OpenCL is a bit slower than with CUDA.
- Successive refinement on the GPU is unexpectedly. This is discussed in detail in 6.2.2.

6.2.1 Workload Distribution on Multiple Threads

As discussed in 5.1.1, the successive refinement algorithm can easily be parallelized by breaking down the pixel raster into parts, that can be assigned to different threads and be calculated independently. But the question is, how to distribute the work between threads to get a good work balance. That is an important factor, considering that the rendering of the whole fractal will not be finished until every single thread has done its work, and therefore the slowest thread determines the overall calculation time.

This distribution of workload is simply done by splitting the viewport into equal pieces and assign one piece to each of the threads. But this leads to scenarios like

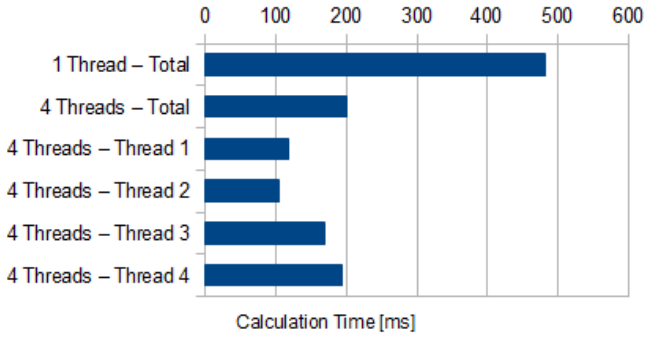


Figure 6.2: Example for bad workload distribution between threads

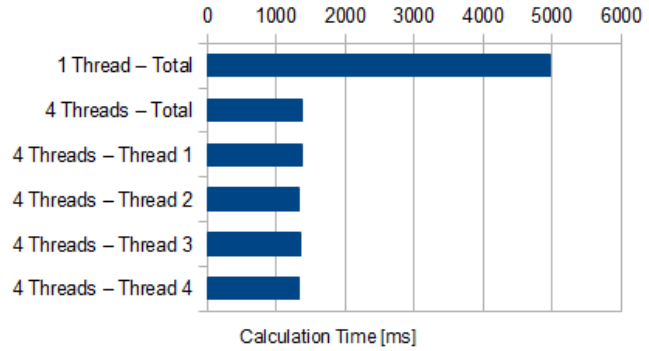


Figure 6.3: Example for good workload distribution between threads

follows. There are four parts of the set, assigned to four different threads. What if three of them contain complex numbers which have quickly diverging orbits, while one of them is full of numbers diverging slowly? One thread is carrying almost the whole workload, and while three threads will be finished after a short time, one thread can be busy for a long time. This is illustrated in figures 6.2 and 6.3. First, 6.2 shows the location at $0.23853497490613537 + 0.5544634809846665i$ and magnitude 21800, and the workload balance is bad here. The overall computation time when using four threads is depending on the computation time of the slowest thread, in this case thread 4, while thread 2 is finished a lot earlier. On the other hand, figure 6.3 shows the location at $0.3680814529993473 - 0.14978321260314592i$ at magnitude $2.11854039 * 10^9$. The workload in this case is split up almost equally between all of the four threads, and therefore the overall computation time is actually only about 1/4 of the computation time of one thread alone.

6.2.2 Successive Refinement and Beyond

The highest expectations regarding performance were put in the implementation of the successive refinement algorithm on the GPU, but as already partly discussed in 5.2.4, this was a bit of a disappointment. One of the reasons for that is simply the characteristics of the successive refinement method regardless of if calculation is done on a CPU or GPU. Especially in higher magnifications, but in general if the fractal image has a lot of small details and no big monotonic areas (like the inside of the set, or areas quite far outside, but not the edge of the interior of the set), there is almost no performance gain (or even none at all) of the successive refinement

method, because in the end almost every pixel of the fractal needs to be evaluated. In addition to that, there have also been done overhead calculations, e.g. to decide if a point should be evaluated or not. This altogether leads to a only slightly faster or even poorer performance for the successive refinement method than for the naive calculation method.

Besides that general drawback of successive refinement, there is also some specific drawback for the used variation of the method to do heavy calculations on the GPU. First, it must be checked which points have to be evaluated, and this is done on the CPU. Data structures are generated for it, and a check has to be done on each point of the current iteration step. Then all this data has to be packed into arrays, sent to the GPU, and after that the results are read back from the GPU. All of that has to be done for every chunk iteration step, and already consumes a significant amount of time.

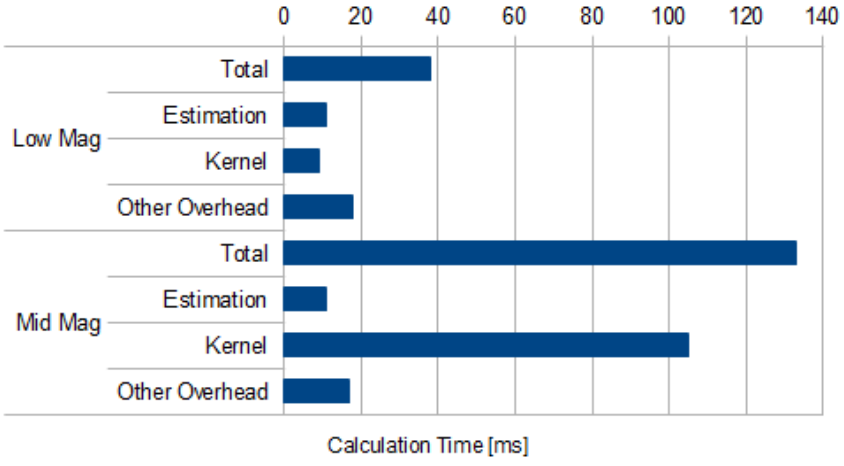


Figure 6.4: Distribution of Calculation Time for Successive Refinement on GPU

Figure 6.4 shows how the overall computation time is distributed between the parts of the algorithm for two different locations, one at the magnification level of 1, and the second on the magnification level of 21,800. For high computation times, the overhead of estimating which pixels have to be evaluated in an iteration step and the other overhead (mainly data movements from and to the GPU) make just a small part of the whole calculation time, most of the time is spent on actually executing the kernel on the GPU. But this high computation times usually occur on high magnification levels, where the successive refinement method does not lead to a high performance gain anyways. In the more important magnification levels, i.e. the ones that should still be calculated in real time, the overhead is clearly to big with around 30 ms. So it would be necessary to minimize this overhead.

Clearly, for higher magnitudes and detailed images, a more suitable calculation algorithm must be found to increase the performance and allow real-time calculations. These algorithms could especially have the need for estimations instead of (quite) exact calculations.

6.3 Platform Independency and Scalability

Platform independency is widely reached because it is only necessary to adapt some libraries to the operating system in use and to recompile the CUDA code for the

desired platform, but there is no need to change any of the code.

By the usage of the MVC (Model-View-Controller) pattern, it would be quite easy to adapt the application to even more platforms, so it could e.g. be possible to write the application for a smartphone or tablet, thanks to their quickly rising performance, especially if GPGPU computing should be introduced for this platforms soon.

The scalability on the other hand is not good enough yet. There especially can occur problems with the drawing of the set, that produces artifacts while zooming if the calculation is not quick enough. The simple reason for it is that drawing and calculating is not synchronized, which is beneficial to the performance, but can lead to such artifacts in the drawing process, e.g. it is even possible that a new drawing iteration is started before the last one is even finished.

7 Future Work

There are plenty of things to do, which would make this application more valuable. Some of them are major improvements requiring big effort and good consideration. Others are just minor improvements and could simply not yet be implemented because time was running out.

7.1 Small Improvements and Fixes

Several small improvements could be done to improve the application. The most important of them are listed here.

- As mentioned in section 5.1.11, the mapping of the parameters gained from the played music and the input parameters of the zoom can be mapped to each other in different ways. It would be nice if this choice of how to map them would be up to the users, so zooms get even a bit more customizable.
- When the smooth zooming navigation is used in the explorer, the speed of zooming is dependent on the performance of the user's hardware. This should be normalized, so that the zoom's speed is the same no matter how fast the users hardware is.
- The amount of CPU threads used for calculating the fractal in multi-threading mode should be determined automatically instead of letting the users do the work to choose the right amount of threads for their system.
- Another navigation method that could be useful is the box zoom. Users would be able to draw a rectangle on the viewport over a part of the fractal that seems interesting to them. Then a smooth zoom into the set is executed, that ends with the area inside the rectangle filling the entire viewport.
- Instead of just using rotations to react on music when playing back recorded zooms, the rotations could also be implemented for the explorer as a navigation feature. This would need several adjustments for the other navigation features and how to deal with the parameters of the fractal, that is why this feature was not yet implemented.
- The issue of synchronizing sound and animation was previously mentioned in 5.2.1. This could be fixed somehow, i.e. by using another waiting method or by do synchronizations from time to time.
- As described before, using double arithmetic instead of float arithmetic is necessary to get to a higher magnitude, but results in much poorer performance. An easy remedy to deal with this problem would be to simply use float arithmetic up to a magnitude where the calculation is still accurate enough, and then switch to double arithmetic (if the hardware allows it).
- When using the successive refinement method, the size of the chunks can already be changed for the zooming process in the explorer. The next step would be to automatically determine if a zoom is choppy because the performance is too low, and in that case to increase the minimum chunk size. On the other hand, if computation time is actually wasted because the calculation time is

much less than allowed, the minimum chunk size should be decreased to make the image more precise.

- When users create or edit a color plate, they can select keys and then change this key's color. A more convenient way of doing this instead of using sliders for the color components would be a color palette from which they could pick up a color.

7.2 Usability Testing

An important future work is to make usability testing to see if the application is actually convenient in usage for people with different backgrounds and interests and adapting the applications user interface and mechanics according to the results of such testings.

7.3 Import/Export features

Importing and exporting locations, color plates, and zoom records is a feature that was in the projects requirements 4.1, but could not be fulfilled due to lack of time.

7.4 Automatic Change of Iteration Threshold

When zooming into the set, the threshold for the maximum number of iterations needs to be increased continuously to still give an accurate image of the Mandelbrot set. This increase has to be done manually. A future work to do would be to automate this process by finding an algorithm for determining an appropriate value for the maximum iterations automatically if the user wishes so.

7.5 Algorithmic Optimizations

There are still several optimization that could be done on the calculation algorithms to increase its performance.

Mirror Symmetry and Orbit Detection were already explained in 5.1.1, but could not be implemented due to lack of time.

There are also other optimization algorithms that could be implemented, more about algorithms can be found in [1].

7.6 Fair Distribution of Work between Threads

As described in 6.2.1, the application does not yet implement any intelligent algorithms to balance the workload between multiple threads, which can lead to a big performance loss. Possibilities to do so would be e.g. that chunks of the whole fractal are continuously assigned to one of the threads as soon as the thread is done with its previous work, until every chunk has been calculated.

7.7 Reusing Parts

When users are panning on the viewport, they would usually just move the fractal by small steps. That means, that most of the new piece that has to be calculated, was already calculated for the former piece and could simply be reused. This would give a big performance improvement for the Mandelbrot Set Explorer, but also for parts of zooms which only consist of panning.

7.8 Pre-Rendering in Idle Mode

As mentioned in section 5.1.9, if the calculation of the set takes less than 40 ms, the thread sleeps for the rest of the time and therefore wastes precious CPU time.

An interesting approach would be the one to already start to calculate the location of the next step, if the previous one was finished in less than 40 ms. So if the zoom is still in low magnifications, but gets deep later, the algorithm could already calculate and store images for later so it can save time when calculation time gets really scarce. This was not done in the current implementation because it would raise the complexity of zooming algorithms a lot.

7.9 Anti-Aliasing

Another possibility to do something useful in that previously mentioned, wasted CPU time could be anti-aliasing, to make the fractal look nicer. Instead of simply running a standard anti-aliasing algorithm, it would also be possible to calculate the iteration value not just for every pixel, but also for subpixels, and then painting the pixel in an interpolated color value of the subpixel values.

7.10 Own Arithmetics

At the moment, the maximum magnification level is bounded by the used double values, which save a 64 bit representation of floating point numbers. Instead, an own arithmetic could be implemented that allows a higher precision. The simplest way of doing that would be to use two double values to simulate a higher precision arithmetic. The big problem is, that the calculation effort in high magnifications is already high, and that using higher precision values would make the performance extremely poor on most systems. But if a really high performing system is available, the users should be able to use it.

It could also be worth to examine if there is another arithmetic in general that fits this particular problem well and would be worth implementing to increase performance and/or accuracy.

7.11 Other Coloring Methods

The color in which a pixel of the viewport is painted, is determined by a direct mapping from an iteration count to a color. That is how most Mandelbrot Set Explorers do the color mapping. But there would be other possibilities as well, loads of different ways to map an iteration count to a color would be imaginable. By implementing some of them, the user would have even more freedom in customizing the fractal.

7.12 Improve Music Analysis

The current music analysis algorithm analyses the played audio file in real-time on energy and frequency bands. The analysis could still be improved a lot. There are a big variety of different algorithms, and a lot of time could be spent to find the most suitable ones and adapt them to the needs of the application.

One step would be to implement the self-similarity analysis, as explained in the theoretical section 2.3.3. That would make it necessary to analyse the whole file

before it is played. But this would also have the advantage, that the whole computation time during the playback of the visualization could be used for the rendering of the zoom, instead of also spending time on analysing the audio signal in real-time.

7.13 Read Played Music from Sound Card

As described in 5.2.5, reading a sound signal from the sound card is not that easy. But if a possibility to do that would be found, that would make it possible to not just load a song and play the visualization for it, but it would allow it to the users to use their own arbitrary music player. This would especially be desirable, because a lot of people today stream their music directly in the Internet and do not even have it as files on the local file system or audio-CDs.

7.14 More Fractals

Finally, an obvious future work that would most likely not even need a whole lot of implementation effort would be to extend the application from only rendering the Mandelbrot set to rendering a variety of different fractals, or make it even possible for the user to insert his or her own formula for the iteration process to create own fractals.

8 References

References

[1] Michael F. Barnsley, Robert L. Devaney, Benoit B. Mandelbrot, Heinz-Otto Peitgen, Dietmar Saupe, and Richard F. Voss. *The Science of Fractal Images*. Springer-Verlag, 1988.

[2] J.-P. Eckmann and D. Ruelle. Ergodic theory of chaos and strange attractors. *Reviews of modern physics*, 57:617–656, 1985.

[3] Jianbin Fang, Ana L. Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. *In 2011 International Conference on Parallel Processing (ICPP'11)*, pages 216–225, 2011.

[4] Barry Feigenbaum. Swt, swing or awt: Which is right for you? Online at `http://www.ibm.com/developerworks/grid/library/os-swingswt/`, February 2006. Visited on 21 Apr 2012.

[5] Jonathan Foote and Shingo Uchihashi. The beat spectrum: A new approach to rhythm analysis. *Multimedia and Expo, 2001. ICME 2001. IEEE International Conference on*, pages 881–884, August 2001.

[6] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of cuda and opencl. Report number: arXiv:1005.2581, May 2010.

[7] Benoit Mandelbrot. How long is the coast of britain? statistical self-similarity and fractional dimension. *Science*, 156:636–638, May 1967.

[8] Robert Munafo. Mu-ency - the encyclopedia of the mandelbrot set. Online at `http://mrob.com/pub/muency.html`. Visited on 23 Apr 2012.

[9] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96:879–899, May 2008.

[10] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals - New Frontiers of Science*. Springer-Verlag, 1992.

[11] Eric D. Scheirer. Tempo and beat analysis of acoustic musical signals. *The Journal of the Acoustical Society of America*, 103:588–601, 1998.

[12] Aamir Shafi, Bryan Carpenter, Mark Baker, and Aftab Hussain. A comparative study of java and c performance in two large-scale parallel applications. *Concurrency and Computation: Practice and Experience*, 21:1882–1906, February 2009.

[13] Andrew Thall. Extended-precision floating-point numbers for gpu computation. *ACM SIGGRAPH 2006 Research posters*, 2006.

[14] Yi-Hsien Wang and I-Chen Wu. Achieving high and consistent rendering performance of java awt/swing on multiple platforms. *Software: Practice and Experience*, 39:701–736, March 2009.

[15] http://sourceforge.net/projects/milkdrop/. Visited on 3rd July 2012.

[16] http://projectm.sourceforge.net/. Visited on 3rd July 2012.

[17] http://www.fractalsciencekit.com/. Visited on 3rd July 2012.

[18] http://www.chaucery.com/jme/fractals/. Visited on 3rd July 2012.

[19] http://www.ultrafractal.com/. Visited on 3rd July 2012.

[20] http://wmi.math.u-szeged.hu/xaos/doku.php. Visited on 3rd July 2012.

[21] http://www.fotoview.nl/uf_eng.htm. Visited on 3rd July 2012.

[22] http://www.oracle.com/technetwork/java/javase/overview/index.html. Visited on 3rd July 2012.

[23] http://www.w3.org/XML/. Visited on 3rd July 2012.

[24] http://www.jdom.org/. Visited on 3rd July 2012.

[25] http://www.nvidia.com/object/cuda_home_new.html. Visited on 3rd July 2012.

[26] http://www.jcuda.org/. Visited on 3rd July 2012.

[27] http://www.khronos.org/opencl/. Visited on 3rd July 2012.

[28] http://www.jocl.org/. Visited on 3rd July 2012.

[29] http://code.compartmental.net/tools/minim/. Visited on 3rd July 2012.

[30] http://docs.oracle.com/javase/tutorial/uiswing/. Visited on 3rd July 2012.

[31] http://www.opengl.org/. Visited on 3rd July 2012.

[32] http://jogamp.org/jogl/www/. Visited on 3rd July 2012.

A Source Code

This appendix section contains parts of the source code. It includes the most important parts of the business logic. Some initializations and other lines which are not directly of importance for understanding the logic may have been left out to increase the readability of the code.

A.1 Determining Complex Numbers for Pixel

```
dXReal = realRange / fractalWidth * cos(rotation/180*PI);
dYReal = realRange / fractalWidth * sin(rotation/180*PI);
dXImag = -imagRange / fractalHeight * sin(rotation/180*PI);
dYImag = imagRange / fractalHeight * cos(rotation/180*PI);

startRe = realCenter - fractalWidth/2 * dXReal -
  fractalHeight/2 * dYReal;
startIm = iamgCenter + fractalWidth/2 * dXImag +
  fractalHeight/2 * dYImag;

cReal = startRe + x*dXReal + y*dYReal;
cImag = startIm - x*dXImag - y*dYImag;
```

A.2 Basic Calculator

```
for(y=0; y<fractalHeight; y++) {
  for(x=0; x<fractalWidth; x++) {
    cReal = startRe + x*dXReal + y*dYReal;
    cImag = startIm - x*dXImag - y*dYImag;

    // calculate point and if it is inside set
    zRe = cReal;
    zIm = cImag;

    pixelMap[x][y] = maxIterations;
    for(iter=0; iter<maxIterations; iter++) {
      zReNew = zRe*zRe;
      zImNew = zIm*zIm;

      // check if calculated complex number is
      // bigger 4 and therefore an escaper
      if(zReNew + zImNew > 4) {
        pixelMap[x][y] = iter;
        break;
      }
      zIm = 2*zRe*zIm + cIm;
      zRe = zReNew - zImNew + cRe;
    }
  }
}
```

A.3 Successive Refinement

```
for(chunkSize=maxChunk; chunkSize>=minChunk;
  chunkSize=chunkSize/2) {
  for(y=0; y<fractalHeight; y+=chunkSize) {
    for(x=0; x<fractalWidth; x+=chunkSize) {
      cReeal = startRe + x*dXReal + y*dYReal;
      cImag = startIm - x*dXImag - y*dYImag;
      calculateChunk(x,y,cReal,cImag,chunkSize);
    }
  }

  copyPixelMapToPixelMapParent();
}
```

```
/**
 * Calculate and fill a chunk of pixels.
 * @param x Starting x value.
 * @param y Starting y value.
 * @param cRe Real part of adding constant
 * @param cIm Imaginary part of adding constant
 * @param chunkSize Size of the chunk in pixels
 */
calculateChunk(x, y, cRe, cIm, chunkSize) {
  if (x%(2*chunkSize) == 0)
    parentX = x;
  else
    parentX = x - chunkSize;

  if (y%(2*chunkSize) == 0)
    parentY = y;
  else
    parentY = y - chunkSize;

  if (chunkSize == maxChunk)
    evaluate = true;
  // if a chunk was already calculated in parent step
  else if ((x == parentX) && (y == parentY)) {
    evaluate = false;
  } else if (equalNeighbors(parentX,parentY,chunkSize*2)) {
    evaluate = false;
  } else {
    evaluate = true;
  }

  if (evaluate) {
    // calculate point and if it is inside set
```

```
    double zRe = cRe;
    double zIm = cIm;

    for(iter=0; iter<getMaxIterations(); iter++) {
      zReNew = zRe*zRe;
      zImNew = zIm*zIm;

      // check if calculated complex number
      // is bigger 4 and therefore an escaper
      if(zReNew + zImNew > 4)    {
        break;
      }
      zIm = 2*zRe*zIm + cIm;
      zRe = zReNew - zImNew + cRe;
    }
  } else {
    iter = pixelMapParent[parentX][parentY];
  }

  // fill chunk
  for (xFill=x; (xFill<(x+chunkSize)) &&
    (xFill<fractalWidth); xFill++)
    for (yFill=y; (yFill<(y+chunkSize)) &&
      (yFill<fractalHeight); yFill++)
      pixelMap[xFill][yFill] = count;
  }
```

A.4 CUDA Basic Algorithm

A.4.1 CUDA Code

```
// Calculate the Mandelbrot Set with double arithmetic
__device__ inline int CalcMandelbrotTrueDouble(
  const double xPos, const double yPos,
  const int maxIter)
{
  double y = yPos;
  double x = xPos;
  double yy = y * y;
  double xx = x * x;
  int i = 0;

  while ((i < (maxIter-1)) && (xx + yy < 4.0f)) {
    y = x * y * 2.0f + yPos;
    x = xx - yy + xPos;
    yy = y * y;
    xx = x * x;
    i++;
  }
```

```
  return i;
}
```

```
// The Mandelbrot CUDA GPU thread
// function with double arithmetic.
extern "C"
__global__ void MandelbrotTrueDouble(
  const double dXReal, const double dYReal,
  const double dXImag, const double dYImag,
  const double startRe, const double startIm,
  const int maxIterations, const int fractalWidth,
  const int fractalHeight, int* pixelMap) {

  const int ix = blockDim.x * blockIdx.x + threadIdx.x;
  const int iy = blockDim.y * blockIdx.y + threadIdx.y;

  if ((ix < fractalWidth) && (iy < fractalHeight)) {
    // Calculate the location
    const double xPos = startRe + ix*dXReal + iy*dYReal;
    const double yPos = startIm - ix*dXImag - iy*dYImag;

    // Calculate the Mandelbrot index for
    // the current location, write it in flat array
    pixelMap[ix+iy*fractalWidth] =
      CalcMandelbrotTrueDouble(xPos, yPos, maxIterations);
  }
}
```

A.4.2 Java Code - JCuda

```
doInitializations();

hostOutput[] = new int[fractalWidth*fractalHeight];

// Allocate device memory for the array pointers, and copy
// the array pointers from the host to the device.
CUdeviceptr deviceOutput = new CUdeviceptr();
cuMemAlloc(deviceOutput,
        fractalWidth * fractalHeight * Sizeof.INT);

calculateStartingValues();

// Set up the kernel parameters: A pointer to an array
// of pointers which point to the actual values.
Pointer kernelParameters = Pointer.to(
  Pointer.to(new double[]{(double)dXReal}),
  Pointer.to(new double[]{(double)dYReal}),
  Pointer.to(new double[]{(double)dXImag}),
```

```
    Pointer.to(new double[]{(double)dYImag}),
    Pointer.to(new double[]{(double)startRe}),
    Pointer.to(new double[]{(double)startIm}),
    Pointer.to(new int[]{getMaxIterations()}),
    Pointer.to(new int[]{getFractalWidth()}),
    Pointer.to(new int[]{getFractalHeight()}),
    Pointer.to(deviceOutput)
);

// Call the kernel function.
cuLaunchKernel(function,
    gridSizeX, gridSizeY, 1,       // Grid dimension
    blockSizeX, blockSizeY, 1,       // Block dimension
    0, null,                // Shared memory size and stream
    kernelParameters, null // Kernel- and extra parameters
);

// this waits for the conclusion of GPU calculation
cuCtxSynchronize();

// copy the array pointers from the device to the host.
cuMemcpyDtoH(Pointer.to(hostOutput), deviceOutput,
    fractalWidth * fractalHeight * Sizeof.FLOAT);
cuMemFree(deviceOutput);

// this copies the flat result array to the pixel raster
unflatten(hostOutput);
```

A.5 CUDA Successive Refinement Algorithm

A.5.1 CUDA Code

```
// The Mandelbrot CUDA GPU thread function,
// calculates a list of points
extern "C"
__global__ void calculateList(const double dXReal,
    const double dYReal, const double dXImag,
    const double dYImag, const double startRe,
    const double startIm, const int maxIterations,
    const int fractalWidth, const int fractalHeight,
    const int chunkSize, const int evalCount,
    int *evaluateMap, int *resultMap) {

    int num = blockDim.x * blockIdx.x + threadIdx.x;
    int ix = evaluateMap[num*2];
    int iy = evaluateMap[num*2+1];

    if (num < evalCount) {
        double xPos = startRe + ix*dXReal + iy*dYReal;
```

```
    double yPos = startIm - ix*dXImag - iy*dYImag;

    int i=0;

    double y = yPos;
    double x = xPos;
    double yy = y * y;
    double xx = x * x;

    while ((i < (maxIterations-1)) && (xx + yy < 4.0f)) {
      y = x * y * 2.0f + yPos;
      x = xx - yy + xPos;
      yy = y * y;
      xx = x * x;
      i++;
    }

    resultMap[num] = i;
  }
}
```

A.5.2 Java Code - JCuda

```
/**
 * Calculate the set for a chunk size, double arithmetic.
 * @param chunkSize Chunk size
 */
public void calculateFlat(int chunkSize) {
  evaluateList = new ArrayList<EvaluatingPoint>();

  for (int y=0; y < getFractalHeight(); y+=chunkSize)
    for (int x=0; x < getFractalWidth(); x+=chunkSize)
      calculateChunk(x, y, chunkSize);

  int hostOutput[] = new int[evaluateList.size()];

  // Allocate device memory for the array pointers, and copy
  // the array pointers from the host to the device.
  CUdeviceptr deviceInput = new CUdeviceptr();
  cuMemAlloc(deviceInput, evaluateList.size()*2 *
    Sizeof.INT);
  cuMemcpyHtoD(deviceInput, Pointer.to(createEvaluateMap(
    evaluateList)),evaluateList.size()*2 * Sizeof.INT);

  CUdeviceptr deviceOutput = new CUdeviceptr();
  cuMemAlloc(deviceOutput, evaluateList.size() *
    Sizeof.INT);

  double dXReal=(getMaxRe()-getMinRe())/getFractalWidth()*
```

```
    Math.cos((float)rotation/180f*Math.PI);
double dYReal=(getMaxRe()-getMinRe())/getFractalWidth()*
    Math.sin((float)rotation/180f*Math.PI);
double dXImag=-(getMaxIm()-getMinIm())/getFractalHeight()*
    Math.sin((float)rotation/180f*Math.PI);
double dYImag=(getMaxIm()-getMinIm())/getFractalHeight()*
    Math.cos((float)rotation/180f*Math.PI);

double startRe = getReCenter() - getFractalWidth()/2 *
    dXReal - getFractalHeight()/2 * dYReal;
double startIm = getImCenter() + getFractalWidth()/2 *
    dXImag + getFractalHeight()/2 * dYImag;

// Set up the kernel parameters: A pointer to an array
// of pointers which point to the actual values.
Pointer kernelParameters = Pointer.to(
    Pointer.to(new double[]{(double)dXReal}),
    Pointer.to(new double[]{(double)dYReal}),
    Pointer.to(new double[]{(double)dXImag}),
    Pointer.to(new double[]{(double)dYImag}),
    Pointer.to(new double[]{(double)startRe}),
    Pointer.to(new double[]{(double)startIm}),
    Pointer.to(new int[]{getMaxIterations()}),
    Pointer.to(new int[]{getFractalWidth()}),
    Pointer.to(new int[]{getFractalHeight()}),
    Pointer.to(new int[]{chunkSize}),
    Pointer.to(new int[]{evaluateList.size()}),
    Pointer.to(deviceInput),
    Pointer.to(deviceOutput)
);

// Call the kernel function.
int blockSizeX = 256;
int gridSizeX = (int)Math.ceil(
    (float)evaluateList.size() / blockSizeX);
cuLaunchKernel(function,
    gridSizeX,  1, 1,       // Grid dimension
    blockSizeX, 1, 1,       // Block dimension
    0, null,                // Shared memory size and stream
    kernelParameters, null // Kernel- and extra parameters
);

cuCtxSynchronize();

// copy the array pointers from the device to the host.
cuMemcpyDtoH(Pointer.to(hostOutput), deviceOutput,
    evaluateList.size() * Sizeof.INT);

cuMemFree(deviceOutput);
```

```
  cuMemFree(deviceInput);

  resultBufferToPixelMap(hostOutput,chunkSize);

  for(int y=0; y<getFractalHeight(); y+=chunkSize)
    for(int x=0; x<getFractalWidth();x+=chunkSize)
      parentMap[x][y] = pixelMap[x][y];
}
```

```
/**
 * Calculate and fill a chunk of pixels.
 * @param x Starting x value.
 * @param y Starting y value.
 * @param cRe Real part of adding constant
 * @param cIm Imaginary part of adding constant
 * @param chunkSize Size of the chunk in pixels
 */
protected void calculateChunk(int x, int y, int chunkSize) {
  boolean evaluate;
  int parentX, parentY;

  if (x%(2*chunkSize) == 0)
    parentX = x;
  else
    parentX = x - chunkSize;

  if (y%(2*chunkSize) == 0)
    parentY = y;
  else
    parentY = y - chunkSize;

  if (chunkSize == getMaxChunk())
    evaluate = true;
  else if ((x == parentX) && (y == parentY)) {
    // if a chunk was already calculated in parent step
    evaluate = false;
  } else
  if (equalNeighbors(parentX, parentY, chunkSize * 2)) {
    evaluate = false;
  } else {
    evaluate = true;
  }

  int count=0;

  if (evaluate) {
    count = -1;
    evaluateList.add(new EvaluatingPoint(x,y));
  } else {
```

```
    count = parentMap[parentX][parentY];
    // fill chunk
    for (int xFill=x; (xFill<(x+chunkSize)) &&
      (xFill<getFractalWidth()); xFill++)
      for (int yFill=y; (yFill<(y+chunkSize)) &&
        (yFill<getFractalHeight()); yFill++)
        pixelMap[xFill][yFill] = count; }
}
```

A.6 OpenCL

A.6.1 OpenCL Code

```
kernel void Mandelbrot(
  const double dXReal, const double dYReal,
  const double dXImag, const double dYImag,
  const double startRe, const double startIm,
  const int maxIterations, const int fractalWidth,
  const int fractalHeight, global int *pixelMap) {

  const int ix = get_global_id(0);
  const int iy = get_global_id(1);

  if ((ix < fractalWidth) && (iy < fractalHeight)) {
    // Calculate the location
    const double xPos = startRe + ix*dXReal + iy*dYReal;
    const double yPos = startIm - ix*dXImag - iy*dYImag;

    double y = yPos;
    double x = xPos;
    double yy = y * y;
    double xx = x * x;
    int i = 0;

    while ((i < (maxIterations-1)) && (xx + yy < 4.0f)) {
      y = x * y * 2.0f + yPos;
      x = xx - yy + xPos;
      yy = y * y;
      xx = x * x;
      i++;
    }

    // Calculate the Mandelbrot index for the current
    // location, write it in flat array
    pixelMap[ix+iy*fractalWidth] = i;
  }
}
```

A.6.2 Java Code - JOCL

```
/**
 * Calculate the set with true double arithmetic.
 */
public void calculateTrueDouble() {
  // setup
  CLContext context = CLContext.create();

  try {
    CLProgram program = context.createProgram(new
      FileInputStream("res/mandelbrot_kernel.cl")).build();
    // select fastest device
    CLDevice device = context.getMaxFlopsDevice();
    // create command queue on device.
    CLCommandQueue queue = device.createCommandQueue();

    CLBuffer<IntBuffer> clOutputBuffer =
      context.createIntBuffer(fractalWidth * fractalHeight,
      Mem.WRITE_ONLY);

    // get a reference to the kernel function with the name
    //'VectorAdd', map the buffers to its input parameters
    CLKernel kernel = program.createCLKernel("Mandelbrot");

    double dXReal =
      (getMaxRe() - getMinRe()) / getFractalWidth() *
      Math.cos((float)rotation/180f*Math.PI);
    double dYReal =
      (getMaxRe() - getMinRe()) / getFractalWidth()*
      Math.sin((float)rotation/180f*Math.PI);
    double dXImag =
      -(getMaxIm() - getMinIm()) / getFractalHeight() *
      Math.sin((float)rotation/180f*Math.PI);
    double dYImag =
      (getMaxIm() - getMinIm()) / getFractalHeight()*
      Math.cos((float)rotation/180f*Math.PI);

    double startRe = getReCenter() - getFractalWidth()/2 *
      dXReal - getFractalHeight()/2 * dYReal;
    double startIm = getImCenter() + getFractalWidth()/2 *
      dXImag + getFractalHeight()/2 * dYImag;

    kernel.putArg(dXReal).putArg(dYReal).putArg(dXImag).
      putArg(dYImag).putArg(startRe).putArg(startIm).
      putArg(getMaxIterations()).putArg(getFractalWidth()).
      putArg(getFractalHeight()).putArg(clOutputBuffer);

    queue.put2DRangeKernel(
```

```
        kernel, 0, 0, fractalWidth, fractalHeight, 0, 0).
        putReadBuffer(clOutputBuffer, true);

      unflatten(clOutputBuffer.getBuffer());

    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      // cleanup all resources associated with this context.
      context.release();
    }
}
```

A.7 Creating a Color Plate

```
Color[] colors = new Color[size];

// delta red/green/blue, marks the step
//sizes in each pixel between two key values
float dr=0,dg=0,db=0;
// float red/green/blue, marks the current floating
// value of a color to calculate it precisely
float fr=0, fg=0, fb=0;

// iterate over color keys
for (int key=1; key<colorKeys.size(); key++) {
  delta_red = (red_right - red_left) /
    ((pos_right - pos_left) * size);
  delta_green = (green_right - green_left) /
    ((pos_right - pos_left) * size);
  delta_blue = (blue_right - blue_left) /
    ((pos_right - pos_left) * size);

  // starting color values for the interval
  r = red_left;
  g = green_left;
  b = blue_left;

  // counting the pixel steps between two keys
  step = 0;

  // iterate over values between (including)
  // last and (excluding) current color key
  for (i=pos_left*size; i<pos_right*size; i++) {
    // calculates the right position cosidering
    // the rotation of the color plate
    colors[(i + (rotation * colors.length)) %
      colors.length] = new Color(r,g,b);
    // adding delta-values to color-values
```

```
    r += delta_red;
    g += delta_green;
    b += delta_blue;
  }
}
```

A.8 Jump To Time

```
/**
 * Jump to a point in time of the current zoom
 * record.
 * @param time Time factor between 0 and 1
 */
public void jumpToTime(float time) {
  // we are on the first key
  if (previous == null) {
    real = next_real;
    imag = next_imag;
    mag = next_magnitude;
    iter = next_iterations;
  // we are on the last key
  } else if (next == null) {
    real = previous_real;
    imag = previous_imag;
    mag = previous_magnitude;
    iter = previous_iterations;
  // we are somewhere in between
  } else {
    real = linInterpol(previous_time, next_time,
      time, previous_real,next_real);
    imag = linInterpol(previous_time, next_time,
      time, previous_imag, next_imag);
    mag = linInterpol(previous_time, next_time,
      time, previous_mag, next_mag);
    iter = linInterpol(previous_time, next_time,
      time, previous_iter,next_iter);
  }

  jumpTo(real, imag, mag);
  // set max iterations and create new
  // color plate if the size has changed...
  int oldIter = getMaxIterations();
  setMaxIterations(iter);
  if (oldIter != iter) {
    colorPlate.applyColorPlate(iter);
  }
}
```

A.9 Approaching Zoom

```
/**
 * Zooming in or out to a certain direction
 * @param clickX X position clicked
 * @param clickY Y position clicked
 * @param stepSize Zooming step size, zoom in if
 * positive, zoom out if negative, pan if 0
 */
public void zoomDirection(int clickX,
  int clickY,double stepSize) {

  // values dX and dY are between -0.5 and 0.5
  dX = clickX/fractalWidth - 0.5;
  dY = -(clickY/fractalWidth - 0.5);
  realCenter = realCentre + dX*realRange/10;
  imagCenter = imagCentre + dY*imagRange/10;

  jumpTo(realCenter, imagCenter,
    magnitude + magnitude * stepSize);
}
```

A.10 Analyse Music

```
public class Song {
  public Song(File songFile) {
    this.minim = new Minim(new ProcessingDummy());
    this.songFile = songFile;
    // load song
    this.song = minim.loadFile(songFile.getAbsolutePath());
    AudioMetaData meta = song.getMetaData();
    this.artist = meta.author();
    this.title = meta.title();
    this.length = meta.length();

        // initialize detection
    beatDetect = new BeatDetect(song.bufferSize(),
      song.sampleRate());
    // use fast fourier transform
    beatDetect.detectMode(BeatDetect.FREQ_ENERGY);
    // says that detection should not listen to beats
    // for 200 ms after a beat (saves resources)
    beatDetect.setSensitivity(200);
  }

  public boolean play() {
    if (song != null) {
      song.play();
      listener = new Listener();
```

```
      song.addListener(listener);
    } else
      return false;
    return true;
  }

  public boolean pause() {
    if (song != null) {
      song.pause();
      song.removeListener(listener);
    } else
      return false;
    return true;
  }

  public boolean stop() {
    if (song != null) {
      song.close();
      song.removeListener(listener);
    } else
      return false;
    return true;
  }

  /**
  * Jump to a position of the song.
  * @param time Time in milliseconds.
  */
  public boolean jumpTo(int time) {
    if (song!=null) {
      song.cue(time);
    } else
      return false;
    return true;
  }

  /**
  * This class listens to audio and analyzes it.
  * @author Christian Knapp
  *
  */
  class Listener implements AudioListener {
    @Override
    public void samples(float[] samp) {
      // sound is mono
      this.samples(samp,null);
    }
    @Override
    public void samples(float[] sampL, float[] sampR) {
```

```
      level = song.mix.level();

      // detect beat (only uses left channel)
      beatDetect.detect(sampL);

      if (beatDetect.isSnare())
        snare = true;
      else
        snare = false;
      if (beatDetect.isKick())
        kick = true;
      else
        kick = false;
      if (beatDetect.isHat())
        hat = true;
      else
        hat = false;
    }
  }
}
```

A.11 Play Zoom

```
public void run() {
  this.playing = true;
  this.time = mainFrame.getZoomPanel().getSTime().
    getValue();

  mainFrame.getZoomPanel().getBPlayRecord().setText("Stop");
  mainFrame.getZoomPanel().repaint();

  long start, duration;

  // decay factor
  int a=Settings.getInstance().
    getColorRotationDynamicDecayFactor();
  // decay time in ms (after that time the value
  // decreased by the decay factor)
  int decay=Settings.getInstance().
    getColorRotationDynamicDecayTime();
  // duration of full rotation (at kick) in ms
  int rot=Settings.getInstance().
    getColorRotationDynamicSpeed();
  // time since last kick
  int kickCount = decay;
  // some precalculations
  final double calcPart1 = -Math.log(rot);
  final double calcPart2 = Math.log(a) / decay;
```

```
fractal.getColorPlate().setRotation(0);
// set standard color rotation (will only be changed
// if color rotation is set to dynamic)
// supposes every step takes 40 ms
fractal.getColorPlate().setRotationStep(
  1f/(float)Settings.getInstance().
  getColorRotationSpeed() * 40);

// PLAY!
fractal.getSong().jumpTo(mainFrame.getZoomPanel().
  getSTime().getValue());
fractal.getSong().play();

while (playing) {
  start = System.currentTimeMillis();
  mainFrame.getZoomPanel().getSTime().setValue(time);

  if (fractal.getSong().isKick())
    // 1 instead of 0 because of characteristic
    // of decay function
    kickCount = 1;

  if (Settings.getInstance().isColorRotationDynamic()) {
    // calculate exponential decay function
    // y(t) = e^(-ln(rot) + ln(decfac) / dectime * (1-t))
    // suppose that every step takes 40ms...
    fractal.getColorPlate().setRotationStep(
      (float)(Math.exp(calcPart1 + calcPart2 *
      (1-kickCount)) * 40));
  }

  fractal.getColorPlate().rotate();
  fractal.getColorPlate().applyColorPlate();

  // fractal rotation
  fractal.getCalculator().setRotationStep(
    fractal.getSong().getLevel()*20);
  fractal.rotate();

  duration = System.currentTimeMillis() - start;

  if (duration < 40) {
    Thread.sleep(40 - duration);
    duration = 40;
  }
  time += duration;
  kickCount += duration;

  time = time % fractal.getZoomRecord().getLength();
```

```
    }

    this.playing = false;

    if (paused) {
      fractal.getSong().pause();
    } else {
      fractal.getSong().stop();
      time = 0;
    }

    fractal.getColorPlate().setRotation(0);
    fractal.getCalculator().setRotation(0);
    fractal.getColorPlate().applyColorPlate();
    mainFrame.getZoomPanel().getSTime().setValue(time);

    c.getMainFrame().setIdleAfterZooming();
    mainFrame.repaint();
  }
```

B Raw Data

This appendix shows the raw time measurements that were made with the application and used for analyses in this thesis. Each test was executed five times, and then the average of this five times was calculated. That is due to inaccuracies in time measurements because the operating system in the background may be busy with own tasks and therefore occupy CPU time, so it is better to have several measures.

B.1 Low Magnitude

Table B.1.

- Position: -0.5
- Magnitude: 1
- Pixel Resolution: 800x500
- Maximum Iterations: 500

B.2 Medium Magnitude

Table B.2.

- Position: $0.23853497490613537 + 0.5544634809846665i$
- Magnitude: 21800
- Pixel Resolution: 800x500
- Maximum Iterations: 1,000

B.3 High Magnitude

Table B.3.

- Position: $0.3680814529993473 - 0.14978321260314592i$
- Magnitude: $2.11854039 * 10^9$
- Pixel Resolution: 800x500
- Maximum Iterations: 10,000

Basic	#1	#2	#3	#4	#5	Avg.
Total	199	211	195	212	191	202

Successive Refinement	#1	#2	#3	#4	#5	Avg.
Total	86	90	99	105	95	95
Chunk 16x16	4	3	4	4	3	4
Chunk 8x8	4	4	6	6	6	5
Chunk 4x4	10	11	9	14	9	11
Chunk 2x2	12	14	20	16	16	16
Chunk 1x1	34	32	32	34	32	33
SR Evaluation Only	31	30	35	37	34	33

Successive Refinement - 2 Threads	#1	#2	#3	#4	#5	Avg.
Total	62	52	53	56	65	58

Successive Refinement - 4 Threads	#1	#2	#3	#4	#5	Avg.
Total	52	58	57	68	54	58
Thread 1	28	23	28	30	26	27
Thread 2	44	41	48	30	40	41
Thread 3	47	45	51	45	41	46
Thread 4	28	22	26	25	24	25

Successive Refinement - 8 Threads	#1	#2	#3	#4	#5	Avg.
Total	49	48	52	51	48	50

CUDA	#1	#2	#3	#4	#5	Avg.
Total	54	48	46	42	40	46
Kernel Only	32	31	31	31	31	31

CUDA - Successive Refinement	#1	#2	#3	#4	#5	Avg.
Total	169	146	158	170	170	163
Chunk 1 Estimate	11	11	11	10	11	11
Chunk 1 Kernel	9	9	9	9	9	9
Chunk 1 Total	38	37	38	38	39	38

OpenCL	#1	#2	#3	#4	#5	Avg.
Total	109	110	94	94	109	103
Kernel Only	47	47	31	31	47	41

Table B.1: Raw performance measurements at low magnitude

Basic	#1	#2	#3	#4	#5	Avg.
Total	491	499	492	495	490	493

Successive Refinement	#1	#2	#3	#4	#5	Avg.
SR 1 Thread - Total	483	475	483	474	496	482
SR Chunk 16	8	6	8	7	8	7
SR Chunk 8	14	16	10	14	20	15
SR Chunk 4	28	23	29	23	31	27
SR Chunk 2	89	87	88	88	88	88
SR Chunk 1	323	321	322	322	324	322
SR Evaluation Only	426	422	426	419	441	427

Successive Refinement - 2 Threads	#1	#2	#3	#4	#5	Avg.
Total	338	336	329	331	316	330

Successive Refinement - 4 Threads	#1	#2	#3	#4	#5	Avg.
SR 4 Threads - Total	208	198	199	204	191	200
Thread 1	126	118	119	114	114	118
Thread 2	115	105	106	100	95	104
Thread 3	175	168	168	173	162	169
Thread 4	202	194	193	197	185	194

Successive Refinement - 8 Threads	#1	#2	#3	#4	#5	Avg.
Total	154	149	160	144	156	153

CUDA	#1	#2	#3	#4	#5	Avg.
Total	148	148	148	147	147	148
Kernel Only	139	139	139	138	138	139

CUDA - Successive Refinement	#1	#2	#3	#4	#5	Avg.
Total	236	243	245	253	255	246
Chunk 1 Estimate	11	11	11	11	10	11
Chunk 1 Kernel	105	105	106	105	105	105
Chunk 1 Total	133	133	133	133	133	133

OpenCL	#1	#2	#3	#4	#5	Avg.
Total	219	234	234	218	234	228
Kernel Only	156	156	171	156	156	159

Table B.2: Raw performance measurements at medium magnitude

Basic	#1	#2	#3	#4	#5	Avg.
Total	5117	5097	5125	5163	5154	5131

Successive Refinement	#1	#2	#3	#4	#5	Avg.
SR 1 Thread - Total	5023	4968	4942	4986	4978	4979
SR Chunk 16	50	39	20	46	39	39
SR Chunk 8	66	61	61	64	60	62
SR Chunk 4	245	239	235	241	240	240
SR Chunk 2	953	954	953	955	955	954
SR Chunk 1	3687	3660	3662	3665	3666	3668
SR Evaluation Only	4949	4912	4890	4943	4926	4924

Successive Refinement - 2 Threads	#1	#2	#3	#4	#5	Avg.
Total	2600	2597	2605	2605	2607	2603

Successive Refinement - 4 Threads	#1	#2	#3	#4	#5	Avg.
SR 4 Threads - Total	1394	1356	1356	1364	1377	1369
Thread 1	1389	1351	1347	1360	1370	1363
Thread 2	1331	1339	1305	1310	1348	1327
Thread 3	1358	1349	1350	1352	1363	1354
Thread 4	1353	1308	1333	1312	1371	1335

Successive Refinement - 8 Threads	#1	#2	#3	#4	#5	Avg.
Total	947	901	913	947	910	924

CUDA	#1	#2	#3	#4	#5	Avg.
Total	2204	2205	2201	2225	2205	2208
Kernel Only	2193	2195	2191	2189	2205	2195

CUDA - Successive Refinement	#1	#2	#3	#4	#5	Avg.
Total	2558	2544	2560	2559	2575	2559
Chunk 1 Estimate	10	15	16	15	31	17
Chunk 1 Kernel	1797	1810	1794	1794	1810	1801
Chunk 1 Total	1827	1825	1826	1825	1856	1832

OpenCL	#1	#2	#3	#4	#5	Avg.
Total	2564	2563	2560	2563	2560	2562
Kernel Only	2494	2493	2490	2493	2490	2492

Table B.3: Raw performance measurements at high magnitude

Linnæus University
School of Computer Science, Physics and Mathematics

SE-391 82 Kalmar / SE-351 95 Växjö
Tel +46 (0)772-28 80 00
dfm@lnu.se
Lnu.se/dfm