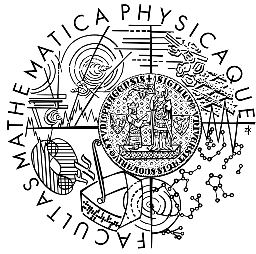


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

BACHELOR THESIS

Antonín Teichmann

Real Time Visualization of Chaotic Functions

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Mgr. Oskár Elek, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, June 26, 2019 signature of the author

I want to thank my supervisor, Oskár Elek, for introducing me this amazing topic. Thank you for all the endless fractal talks. Thank you for the great supervision and for all the feedback. Thanks for the freedom I was given when discovering the topic. Thank you for guiding me through the swamp of chaos by tracing the paths of our fractal thoughts.

I want to thank Adam Hanka for the huge support he gave me when finishing the thesis. Thank you for all the feedback. Thank you for challenging my ideas. Thank you for the amount of great practical advices you gave me, including advices about punctuation, relative clauses and Oxford Comma. Thank you for the time you spent re-reading my writings. And mostly, thank you for allowing me to meet Oskár and this topic.

I want to thank Jan Gocník for his tremendous moral support. Thank you for the big patience you had with me. Thank you for the energy you gave me and for all the times you supported me when I was down. Thank you for all the help with debugging undebuggable. And last but not least, thank you for lenting me your CUDA-equipped machine, that allowed *chaos-ultra* to emerge. Without you, I would not have been able to produce a thesis of this extent and quality. As a thank you, I dedicate one fractal in the implemented program to you.

I want to thank my family, and do it in the language they understand: Moje široká rodino, děkuji za vaši obrovskou podporu při psaní této práce. Děkuji za trpělivost, kterou jste se mnou měli během času, kdy jsem na práci pracoval. Děkuji vám za okamžiky, kdy jste chápali, že nemůžu být s vámi, i když bych chtěl. Moc si vaší podpory vážím. Konečně, děkuji svojí mamince za to, že mě chápe a podporuje.

And finally, I would like to thank Jonáš Klimeš and Michal Petřík for giving me the time flexibility that allowed me to finish the thesis.

Title: Real Time Visualization of Chaotic Functions

Author: Antonín Teichmann

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Oskár Elek, Ph.D., Department of Software and Computer Science Education

Abstract: Fractals are a fundamental natural structure that has fascinated the scientific community for a long time. To allow for better understanding of fractals, visualization techniques can be used. The focus of this thesis is real-time rendering of fractals that are similar to the Mandelbrot set or the Newton fractal. Detailed exploration of these fractals is complicated due to their recursive-manner which leads to the fact that rendering them is computationally demanding. Existing solutions do not work in real-time or have low visual quality. We want to change that and allow high-quality real-time rendering. During our analysis of the problem, we generalize fractals to chaotic functions. To achieve high-quality rendering with low overhead, we introduce a method for adaptive super-sampling of chaotic functions. To achieve real-time performance, we show how to use sample reuse, foveated rendering, and other techniques. We implement a parallel, GPU-based, high-quality renderer that runs in real-time and produces visually-attractive views of given fractals. The program can visualize any given chaotic function. This way, we open the realm of real-time visualization of chaotic functions to the public and lay a basis for future research.

Keywords: real-time rendering, the Mandelbrot set, fractals, Escape-time fractals, adaptive super-sampling

Contents

1 Introduction **5**

1.1 Motivation . . . 5
1.2 Aims . . . 8
1.2.1 Research aims . . . 8
1.2.2 Implementation aims . . . 8
1.3 Related work . . . 9
1.3.1 Software for fractal rendering . . . 10
1.3.2 Papers on fractal rendering and chaos prediction . . . 11
1.4 Goals . . . 12
1.5 Expectations on the reader . . . 13
1.6 Thesis structure . . . 14

2 Mathematical background on fractals **15**

2.1 General definitions . . . 15
2.2 Fractals . . . 15
2.3 Chaotic functions . . . 16
2.4 Continuous escape-time fractals . . . 17
2.4.1 Maps . . . 17
2.4.2 Complex maps . . . 17
2.4.3 Complex quadratic map . . . 18
2.4.4 Mandelbrot set . . . 18
2.4.5 Julia set and Julia fractals . . . 19
2.4.6 Newton fractal . . . 19
2.4.7 Other examples . . . 20

3 Problem analysis **23**

3.1 Rendering algorithms . . . 23
3.1.1 Making a fractal from the Mandelbrot set . . . 23
3.1.2 Julia fractals . . . 25
3.1.3 Newton fractal . . . 26
3.1.4 Rendering a fractal as a whole . . . 28
3.2 Generalization . . . 29
3.3 Chaotic functions . . . 30
3.3.1 Formal definition of the rendering problem . . . 31
3.3.2 Performance-increasing heuristics . . . 32
3.4 Supersampling . . . 32
3.4.1 Sample distribution . . . 32
3.4.2 Sample composition . . . 33
3.5 Fractal coloring . . . 34
3.6 GPGPU technology . . . 35
3.7 Backend and Frontend . . . 35
3.8 Programming languages . . . 36
3.9 Genericity . . . 37
3.10 Designing a graphical user interface . . . 38
3.10.1 Class architecture . . . 39

3.11 Number precision 40

4 Methods of achieving real time performance **41**
4.1 Adaptive super-sampling 42
4.1.1 Algorithm for adaptive supersampling 44
4.2 Interactive mode and progressive visualization 45
4.2.1 Fixed resource budget per frame 46
4.2.2 Progressive visualization 47
4.3 Sample reuse 47
4.4 Foveated rendering 50
4.5 Combining the methods into a toolset 52
4.6 Conclusion 53
4.7 Other possible methods 53

5 Experiments **55**
5.1 Measurements methodology 55
5.1.1 Hardware 55
5.1.2 Comparing specific chaotic functions 55
5.1.3 Choosing comparison method 56
5.2 Comparing *XaoS* and *chaos-ultra* 56
5.3 Adaptive supersampling 57
5.4 The real-time heuristic toolset 60
5.5 Rendering review 61

Conclusion **65**

Appendices **71**

A Development documentation **73**
A.0.1 CUDA terminology 73
A.0.2 Fractal representation 74
A.1 Program architecture 74
A.2 CUDA backend 75
A.2.1 Data structures 75
A.2.2 Fractal interface 76
A.2.3 Fractal specific parameters 76
A.2.4 Specific fractals implementation 77
A.2.5 Implementing the real-time heuristics 78
A.2.6 API of the module: Available kernels 79
A.2.7 Compilation and build 80
A.3 Java-Cuda mapping 80
A.3.1 Module object hierarchy 81
A.3.2 Cuda Kernel 81
A.3.3 Class CudaFractalRenderer 83
A.4 Java renderer 83
A.4.1 Supporting multiple fractals 84
A.4.2 Interface FractalRenderer 85
A.4.3 Class GLRenderer and OpenGL loop 87
A.4.4 Model 89

A.4.5 Class RenderingController 89
A.5 Graphical user interface 90
A.5.1 Swing GLCanvas 90
A.5.2 JavaFX and Swing integration 91
A.5.3 FXML 92
A.5.4 FX Presenter 92
A.6 Developer README 93

B User documentation **95**
B.1 Technical requirements 95
B.2 Installation guide 95
B.3 Navigating the user interface 95
B.4 Fractal change 96
B.5 Changing the color palette 96
B.6 Image export 97
B.7 Troubleshooting 97

C Adding a custom fractal **99**

D Electronic attachments **101**
D.1 Source code of the program 101
D.2 Compiled executable 101
D.3 Experiments 101
D.3.1 Perception study videos 101
D.3.2 Perception study results 101
D.3.3 Real-time toolset evaluation 101
D.3.4 Examples of program's output 101

List of Figures **111**

1. Introduction

1.1 Motivation

> "Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line." — B. Mandelbrot [46]

There is a fundamental natural structure, observable in many fields of both daily and scientific life. Perhaps the most distinctive occurrence of this structure can be observed on plants. Tree branching, wheat spikes, brocoli: all of those are evident examples of objects with self-similar structure composed of repeated patterns. Such structures are denoted as fractals.

> *Big fleas have little fleas upon their backs to bite 'em,*
> *And little fleas have lesser fleas, and so, ad infinitum.*
> *And the great fleas, themselves, in turn, have greater fleas to go on;*
> *While these again have greater still, and greater still, and so on.*
> — Augustus De Morgan [80]

Other occurrences of fractal structure in nature include, for example: The network of veins in leafs, circulatory system of mammals, snowflakes, the Lichtenberg figure, Romanesco cauliflower, river deltas [42] , Chambered nautilus shells, pinecone seeds [31] , mountain surfaces, cloudy heaven, surface of tree bark, lighting trajectory, neurons, galaxies.

(a) Romanesco broccoli, example of self similarity in nature.

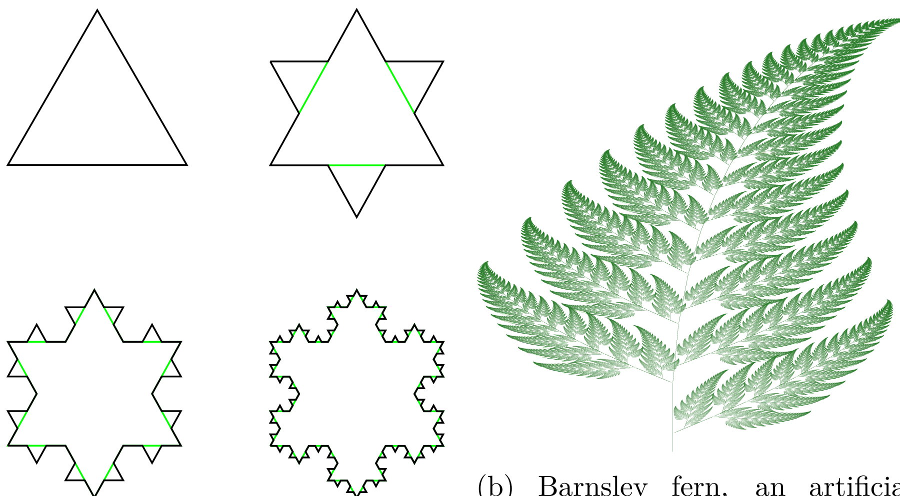
(b) Fern leaf in detail, example of self similarity in nature.

Fractals in mathematics

It seems that the fundamental natural forces themselves induce fractal structures. This leads to the idea that by applying simple formulas in the right manner, very complex patterns can be created. Indeed, we find this very behavior in mathematics.

Mathematically, there are many classes of fractals arising from simple rules. Examples of the most known fractals and their class are: Koch snowflake (Iterated

Function Systems), Barnsley fern and similar branching patterns (L-Systems), and the Mandelbrot set (Escape-time fractals). We define fractals and define and discuss these classes in more detail in Chapter 2.



(a) The first four iterations of the Koch snowflake. Source: [13].

(b) Barnsley fern, an artificial fractal resembling a natural fern. Source: [11].

Perhaps the most fascinating and least understood and examined group is the latter one, **Escape-time fractals**. These fractals are generated by iterating a formula on each point in a space. The fractal is constructed from whether the point has escaped some bound (hence the name) or, more generally, from the convergence speed. Because this group is the most complex one, we want to focus on it.

Fractals utilization

The study of fractals is a broad field with many application. Practical use-cases include following:

All tree-based data structures (Search trees, decision trees, quad/oct trees, heaps) have a self-similar structure and are, in their nature, fractals. In AVL trees, the structure of self-similarity is similar to the branching-patterns observed on plants (and their depth-analysis comes to similar conclusions). Dynamic programming methods, e.g. divide and conquer, are fractals by their very definition.

Space-filling curves, for example The *Hilbert curve* or *Z-order curve* (a.k.a. *Morton code*) that are used by many advanced computer-graphics algorithms and data structures, are examples of fractals; also having the interesting property of having integral Hausdorff dimension, larger precisely by 1 than their topological dimension (more about this in Chapter 2).[1].

In computer graphics, fractals are used for landscape generation [64] or object generation in general [37] and for signal and image compression [15, 27] .

Fractal analysis is widely used to asses fractal characteristics of data in many areas of science, especially in signal processing, ecology and medicine [36] .

Moreover, [78, 79] suggest that some fractal patterns reduce physiological stress when viewed by humans.

[1]Interestingly, not many fractals have this ability, but for example the edge of the Mandelbrot set does.

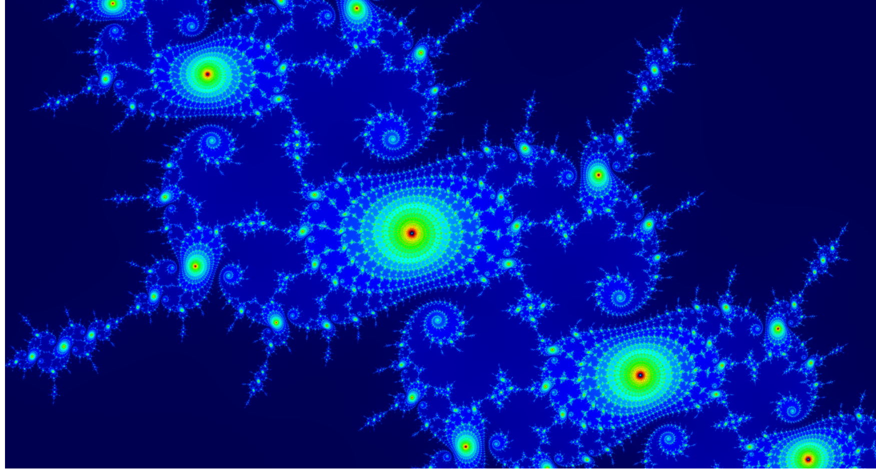


Figure 1.3: Inner part of the Mandelbrot set, example of an Escape-time fractal.

Other usages include e.g. generation of camouflage patterns or measuring coastline complexity [45].

Fractal visualization

As we have showed, fractals are useful and can also be highly fascinating. This is especially true for the Escape-time fractals. That is why people want to see them and explore them in more detail. For this, visualization tools are needed. Visualization always helps when exploring new ideas. B. Mandelbrot himself said "The notion that these conjectures might have been reached by pure thought – with no picture – is simply inconceivable." [43]

There exist tools for visualization of Escape-time fractals, however, we think that these are not appropriate enough, considering the progress in computer graphics in the last years. We are going to analyze this in more detail.

All in all, we think that there is a need for a real-time high-quality visualizer of Escape-time fractals. Who would need such tool? We think that geeks, computer scientists and artists — for example (1) for finding an ideal parameter-constellation when examining a fractal in some research, (2) when referencing a image of a fractal in a paper, (3) for exploring ideas for a wallpaper or poster, (4) for creating a smooth animation.

This can be done with current software, but either the quality is low or it takes a lot of time. We do not want to wait minutes for a single frame and hours for a video, when just exploring ideas. Neither do we want blurry inaccurate images with low level of detail. We want real time response, with sufficient-enough quality for purposes described above.

For some fractal classes, this is easy. For Escape-time fractals, visualization in general is complicated and real-time visualization is still a big issue. We are aiming at solving it.

Chaotic functions

To be able to solve the problem of Escape-time fractal visualization, we need to know them bit better.

The most famous Escape-time fractal is the Mandelbrot set. However, wo do not want to visualize only this instance; we want to analyze and visualize the Escape-time fractal class in general.

The Mandelbrot set is defined by a very simple function, $z^2 + c$ (we introduce this in Chapter 2 in more detail). From such an easy mathematical formula, there emerges an extraordinary complex object — this behavior is not at all clear at the first glance. We want to help exploring such behavior for all other possible formulas.

During the anlysis of Escape-time fractals and their rendering properties, it comes forward that their most immersive property is chaos — the fractals are chaotic. In our later research, it proves most useful to generalize our problem and to analyze visualization of chaotic functions in general, instead of focusing only on Escape-time fractals.

We did not expect this generalization at first. Yet after it emerged, it seemed almost natural. Suddenly, there is a huge overlap to other fields: Weather and climate forecasting [40] , road traffic models [72] or study of economic bubbles [71] are all examples of chaotic systems that, in some sense, behave similarly to Escape-time fractals.

It would be very interesting to explore this overlap in a more detail and to find the connection between visualizing fractals and, for example, weather forecasting or economy. At the same time, we need to maintain the scope, length and difficulty of the thesis.

1.2 Aims

Based on the previous discussion, we summarize the aims of the thesis. This helps us in being ready for defining the specific goals.

We aim at implementing a real time visualizer of chaotic functions, with focus on Escape-time fractals. While doing that, we also aim at discovering chaotic functions and the problems that come forward when rendering them.

1.2.1 Research aims

For visualizing a chaotic function, we will need to know the function's values, at least approximately. Moreover, for achieving real time performance, the calculations need to be very efficient. Some guessing will be needed, for example heuristics. For high visual quality, multiple samples per pixel will be needed, leading to partial integration of chaotic functions.

We are touching an open research problem: prediction of chaotic functions. We do not have the ambition to solve the problem; not even to propose general solutions. We rather touch the problem from the very specific point of view of the rendering of Escape-time fractals.

1.2.2 Implementation aims

To be able to precisely specify the implementation goals, we must know what we are aiming at. We base our aims on the expected target group: geeks, computer

scientists and artists.

Code basis

We want to allow the users to modify the program to suit their needs. Thus the program needs an ability to add new fractals, a good software architecture, maintainable code and an open licence. It should also be cross-platform.

Some of the users will have programming experience. We expect that maybe an open-source community will work together in customizing the program.

On the other hand, the rest of the users will prefer a tool that is easy to use, with minimal learning curve and with pre-made fractals and views.

Device type

Possible target devices are personal computer (desktop or laptop), mobile devices (smartphones or tablets) and cluster in a datacenter.

Mobile devices are highly interactive, but lack computational power needed for real time visualization.

Clusters have great computational power but are not interactive — for fractal examination, we need constant visual feedback in HD, without compression. Utilizing clusters would introduce an unbearable load for the network, or compression artifacts.

Personal computers are an affordable, standard equipment of geeks, computer scientists and artists. They have sufficient computational power, are extensible and interactive enough. That makes them our target platform.

We can utilize high tech gaming PCs — they have been designed for the very purpose of visualization. We can expect an average computer graphics scientist or geek to have a middle to high end gaming PC with a decent CPU and GPU.

Processing unit type

The rendering can be computed either on a CPU or by utilizing GPGPU[2] technology. There exist CPU-based renderers, yet we do not know about GPU-based ones. We want to fill this gap. Based on current market situation, when graphics cards are affordable to anyone, we aim for a GPU-based renderer.

1.3 Related work

We have stated the aims of the thesis. Before defining our goals, we look on relevant existing solutions. Thanks to that, we will have inspiration, will know what gap we want to fill and will be able to define our goals precisely.

We list some existing software for fractal rendering, and then briefly go through relevant papers on fractal rendering and chaos prediction.

[2] General-purpose computing on graphics processing units

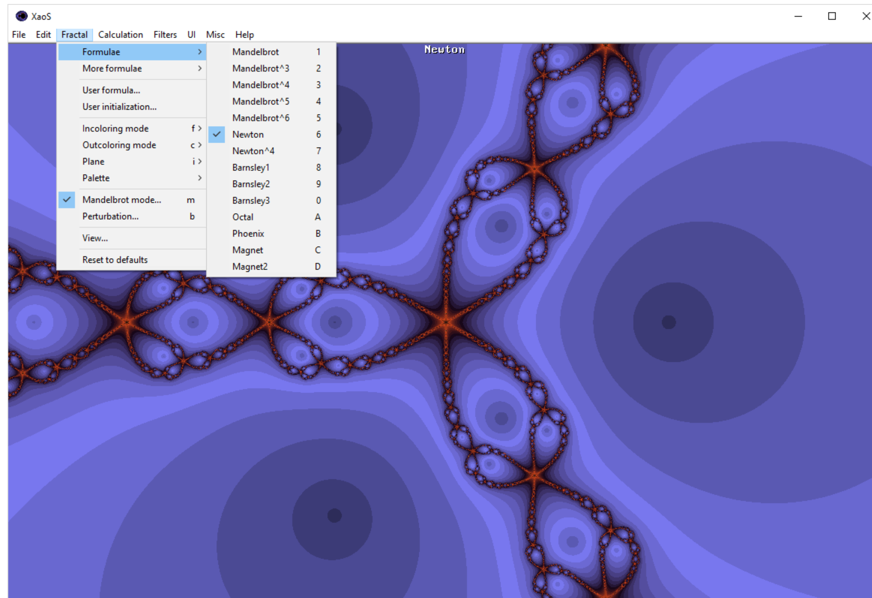


Figure 1.4: *XaoS*, real-time CPU renderer with GUI.

1.3.1 Software for fractal rendering

We want to check whether there already is a program that successes in high quality real time rendering of Escape-time fractals. We list and discuss known fractal-rendering programs.

Quality oriented

- *Fractint* — A quality-based CPU renderer, originated in 1990s.
- *Ultrafractal* — A versatile and very generic renderer, suitable for creating fractal art. Provides fast zoom functionality, but still not real time.

Programs listed above are capable of rendering marvelous animations of e.g. moving around the edge of the Mandelbrot set. However as they are strictly quality-oriented, they cannot effectively be used for interactive real-time rendering. It takes seconds to minutes to hours to produce just one image.

Real Time

- Mobile Apps: *Mandelbrot Explorer* by Defiant Technologies, *MandelBrowser* by Tomasz Śmigielski, *GPU Mandelbrot* by Infinite Worlds, *Fractview* by Searles, *Fractal Universe* by Hochschule Düsseldorf (HSD) — All of them support only Mandelbrot/Julia, and have low visual quality. Most of them start lagging when zoomed in too much.
- *XaoS* by Jan Hubička [33] — A CPU-based pioneer of real time fractal rendering, an industry standard. A well optimized software. It also has support for custom fractal formulas, is cross-platform, versatile, and has many more features. But compared to today's expectations on video and graphic effects, its visual quality seems rather low, with many fragments, lags and blind-spots. We expect that with and GPU-base approach, image quality could be much higher than its.

- Electric sheep [76] — A distributed computing project, with fast rendering, but specializing on fractal flames, not supporting Escape-time fractals.
- *Fractal* by Pavel Fatin [26] — A simple GPU based fractal renderer, supporting only Mandelbrot and Julia. It seems nice but has low FPS (Frames Per Second) and does not allow greater zoom levels. Lags with greater resolutions.
- *Nuclear* by John Tsiombikas [83] — A very simple shader-based implementation of Mandelbrot and Julia fractals. Fast, but with low details and no support for other fractals.
- WebGL mandelbrot by *curran* [23] — WebGL implementation of the Mandelbrot set that runs in the browser. Very low FPS, but a great proof of concept.
- *MPMAndelbrot* by Eric Bainville [2] — An OpenCL multi-precision implementation of the Mandelbrot set, running in real time with about 1 FPS. It is slow, but a great proof of concept.

All listed time renderers have low quality: either (very) low level of detail of fractal's structure or low FPS. Even the programs that use shaders or GPGPU do not use it effectively or do not introduce heuristics that would increase rendering speed. Also, most of the programs support only Mandelbrot/Julia fractals, not fractals in general.

Conclusion

There is no program that meets our requirements. HQ renderers are not real time, and real time zoomers produce low quality images. We have not found a program that would combine both approaches. The aim of our thesis makes sense and is innovative.

1.3.2 Papers on fractal rendering and chaos prediction

Let us have a look on papers that are relevant to the topic of fractal rendering or chaos prediction and at what we can learn from them.

[48] provide algorithms for real time rendering of Escape-time fractals on CPU. The algorithms are not usable in our case, because the threading- and memory-model of CPU differs vastly from that of GPU. Still, the algorithms prove to be inspirational.

[56] is an algorithm for effective rendering of the Mandelbrot set. However, it relies on contectedness of the set and thus cannot be generally applied. Moreover, it is not suitable for the GPU's threading model.

[50] successfully proved that rendering the Mandelbrot set and Julia sets using GPGPU is possible and is fast (yet not real time). This proves that our parallel idea makes sense and it can be our steppingstone to possible rendering methods.

[5] pioneers the rendering of fractals on GPUs. However, it does not speak about their real time visualization nor specifically about Escape-time fractals.

[47] analyzes chaotic systems in general and studies detection of chaotic regions in the parameter space, even providing an implementation. The implementation, however, is CPU-based.

[66] comes with an innovative approach to chaotic precess forecasting, introducing combining of statistical and machine-learning techniques. The paper focuses mostly on weather forecasting and implementing its ideas would be out of the scope of this thesis. Still, it provides a great inspiration in the field of chaos prediction.

[38] describes an adaptive process for Monte Carlo rendering, which could be an inspiration for our adaptive processes, regarding ,for example, supersampling.

[30] and [67] describe how to utilize Foveated rendering, a method where the image is rendered in progressively less detail in ares where the user does not currently look. We could use this method for reducing amount of samples.

1.4 Goals

Having discussed our motivation and aims and having introduced relevant work, let us now formally and precisely define the goals of the thesis.

The main goal of the thesis is to implement a generic real-time renderer of chaotic functions that is parameterisable by a specific chaotic function. It has to have following abilities:

1. **Genericity**: To fully allow examinig the realm of chaotic functions, the renderer should be able to plot arbitrary chaotic function $\mathbb{C} \to \mathbb{R}$. The main focus can be on Escape-time fractals however. We ought to provide implementations of the three most common fractals: the Mandelbrot set, Julia sets and Newton fractal. Other chaotic functions can be added programmatically.
2. **PC & GPGPU-based**: The renderer should use **GPU-accelerated computation** for rendering. We are targeting middle-ranged PCs, with GPU similar to nvidia GeForce GTX 1060.
3. **High quality in real time**: We aim at rendering concerned fractals in **high resolution** and in **high level of detail**, with immediate feedback and **smooth** user experience when zooming in or out or moving the canvas.
   - 3.1. **Resolution**: The resolution has to be scalable and adjustable to window size. The program should support **up to** Full HD, i.e. 1920×1080 px.
   - 3.2. **Detail level**: As the concerned fractals are infinitely complex yet still interesting with even small level of complexity, we cannot formally state what is "good enough" visual quality. We can only rely on user's subjective impressions and comparing our results with existing renderers.
   - 3.3. **Rendering performance**: Smoothness of user interaction can be measured in frames per second (FPS). Usually, application is considered smooth when reaching 24 or 30 FPS, yet some real time animations target up to 60 or 120 FPS. Thus, the target FPS has to be programmatically modifiable and the algorithms should adjust when it changes. Increase of FPS may lead to decrease of level of detail.

   Our formal requirement is, for hardware defined above, to consistently achieve following goals at the same time: HD resolution, i.e. 1280×720 px,

30 FPS and level of detail higher than when using *XaoS* renderer.[3] For higher resolutions, some slowdown is allowed and expected. Also, the problem of fractal rendering inherently gets more complex when zooming in; for higher zoom levels, a slowdown is allowed and expected.

4. **User experience**: To allow the users for straightforward interaction, the program has to have a simple graphical user interface (GUI). The main part of the program is to be a **canvas** that visualizes the fractal and allows following user interaction in a common way: zoom in, zoom out, move around. The resolution of the canvas has to fit the resolution of the window. The GUI also has to allow fractal's **parameter tweaking**, using text fields and buttons.
5. **Cross-platform**: To approach wide audience, the program has to be cross-platform, supporting both Windows and Linux.
6. **Extensibility**: To allow the community to further develop the program to suit their needs, the code base has to be readable and extensible. The program has to have a good **software architecture**, properly utilizing the OOP principles. It should be implemented in a **standard OOP language**, with code following given language's style and best practices.
7. **Color palettes**: To go towards users' individuality, the program has to allow changing the style of coloring the fractal, for example by loading user defined color palettes.

In addition to software-oriented goals, we hope to better understand the perks of sampling a chaotic function. We aim at providing a relevant discussion of the problem and at proposing a few possible solution methods. Our goal is NOT to try or implement all discussed and proposed methods in their full potential; we allow proof-of-concepts and simplified versions.

Our research should focus mainly on following areas:

1. **Progressive visualization**: Interactive navigation with a minimal sample budget, which is then refined whenever the user interaction stops.
2. **Adaptive sampling**: Developing meaningful heuristics for detecting interesting regions of the visualized function, so that more samples can be evaluated for those regions.
3. **Foveated rendering**: Allocating fewer samples to regions away from user's focus (for instance assuming that the user focuses at the cursor). Doing so in a perceptually plausible way.
4. **Sample reuse**: Reprojection of the already evaluated samples between successive frames to avoid duplicated computation.

1.5 Expectations on the reader

We expect the reader to have a standard mathematical education in fields of linear algebra, calculus and statistics and to understand complex numbers.

We expect the reader to understand basic concepts of object oriented programming (OOP) and software development. We also expect the reader to understand

[3] We compare with *XaoS* mostly because it is a pioneer of real time fractal rendering and has been inspiration for this thesis.

the basics of *OpenGL* API and of GPGPU, as sufficient introduction would be beyond the scope of this thesis. Such introduction can be found for example at [22].

1.6 Thesis structure

It is apparent from the goals that this thesis has two aspects: An implementation aspect and a research aspect. The implementation aspect has been mostly fulfilled by the creation of the software; its development documentation is included in Attachment A. The main text of the thesis focuses more on the research aspect.

In Chapter 1, we start the thesis with defining fractals and chaotic functions, followed by introducing the Mandelbrot set and other Escape-time fractals in Chapter 2.

Then in Chapter 3 we analyze the problem that lays before us, focusing on fractals and how to render them. We come to the need for chaotic functions. In the second part of the chapter, we analyze the software-specific goals and discuss what technologies, languages and libraries should be used for the implementation.

We follow with a theoretical Chapter 4, in which we propose what methods could be used for real-time rendering of chaotic functions.

After the methods have been proposed, we implement them, with details described in Appendix A.

Having an implementation, we compare its efficiency and quality with other programs in Chapter 5. We also measure the performance of the proposed real-time methods.

At the end, we summarize what we have done, and show that we have fulfilled the goals stated previously.

2. Mathematical background on fractals

As stated in the first chapter, we aim to visualizing chaotic fractal functions. Many chaotic functions are fractals and vice versa, yet formally, fractals and chaotic functions are independent classes. Let us first have a look at each of them.

First, we define generally used terms. Afterwards we define fractals and chaotic functions. Then we introduce functions that are generally chaotic and yield fractals, complex maps. Finally, we describe most common fractals, such as the Mandelbrot set, Julia sets and the Newton fractal.

2.1 General definitions

- When we say that a function is **visually attractive**, we refer to attractiveness of its graphical representation. We mean that it can be **plotted** in a visually attractive way, usually colorfully.
- **Hausdorff dimension** of an geometric object is a measure of roughness. Informal and vague definition is this: When scaling an object, it is scaled by factor S and its mass (i.e. area, volume etc.) changes by factor M. Then its Hausdorff dimension is such D, that $S^D = M$. For non-fractal geometric objects, it is an integer equal to the topological dimension (e.g. when scaling a 3D cube, its volume changes by power of 3). For fractals, it is a fractional number. Explaining Hausdorff dimension formally is beyond the scope of this thesis; to learn more about it, see [19]. For detailed informal explanation and animation, see (ref).

2.2 Fractals

The origin of the term *fractal* goes back to the 17th century when Gottfried Wilhelm Leibniz debated on recursive self-similarity and used the term "fractional exponents" [68]. The term itself was introduced by Benoit Mandelbrot together with a definition in 1975 [59][44].

According to Mandelbrot, fractal is "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole" [46]. In a more formal manner, Mandelbrot used the term *fractal* to denote a mathematical structure with Hausdorff dimension strictly greater than its topological dimension [44]. There are yet other possible formal definitions and there is no consensus across the scientific community in formal definition of fractal or whether it ought to be defined at all [25]. Mandelbrot's first definition will be sufficient for our purpose.

There are many types of fractals. One common characterization of fractals is by generation techniques. Common techniques for generating fractals are for example:

- Iterated Function Systems — use fixed geometric replacement rules (e.g. Koch Snowflake, Cantor Set, Menger sponge)
- L-systems — use string rewriting (using formal grammars) and turtle graphics, resulting e.g. in branching patterns such as plants
- Random fractals — use stochastic rules, resulting e.g. in Brownian tree or fractal landscapes
- Escape-time fractals — use a map at each point in a space (e.g. the complex plane), resulting e.g. in the Mandelbrot set
- Continuous multidimensional fractals — generalization of previous, use an arbitrary formula at each point in a space

See [16] for more detailed characterization and examples.

As stated in Chapter 1, we examine the last group, Continuous fractals, with restriction on two dimensions, i.e. Continuous 2D fractals.

2.3 Chaotic functions

In Chaos theory, we study functions with sensitive dependence on initial conditions. The intuitive metaphor behind Chaos theory is the butterfly effect.

The butterfly effect describes a phenomenon when a flap of butterfly wings in, for example, Prague, leads to a tornado in, for example, Santa Cruz, California, US. The butterfly itself does not have power to cause the tornado, but its wing flap changes the initial conditions of the chaotic system (weather system in this example), which could lead to different air flow at the area, leading to an air turbulence, leading to a tornado. Had the butterfly not flapped its wings, the system could have behaved vastly differently, possibly not leading to a tornado at all (hence the causality). The effect was described by Edward Lorenz in 1972 [41].

We will examine deterministic chaos, which was summarized by Lorenz as:

> "Chaos: When the present determines the future, but the approximate present does not approximately determine the future." [6]

Definition 1. A function f is considered chaotic when an infinitesimally tiny change of x leads to unpredictable change of $f(x)$.

Examples of chaotic functions are:

- $\sin(1/x)$ for $x \in (0, \varepsilon)$ for some small ε.
- position of a double rod pendulum at time, given initial position [39]
- the Lorenz Attractor
- iterating over evolution function $x \to 4x(1-x)$, $y \to (x+y)$
- iterating over complex evolution function $z \to z^2 + c$, inducing the Mandelbrot set and Julia sets

Although logically not negation nor complement, in some informal sense, chaos can be seen as an opposite of continuity.

2.4 Continuous escape-time fractals

Our goal is to examine continuous 2D fractals, especially those from the escape-time class. Let us have have a look at a class of functions that yield them. We expect those functions to be chaotic.

2.4.1 Maps

Maps, also known as iterated functions or evolution functions [18] are a subclass of chaos functions. Informally, value of an iterated function is obtained by repeatedly applying the function to its result.

Definition 2 (Map)**.** Let X be a set, $f : X \to X$ be a function and $n \in \mathbb{N}, n \geq 0$. Define f^n as the n-th iterate of f by

$$f^0 = \mathrm{id}_X$$
$$f^{n+1} = f \circ f^n,$$

where id_X is the identity function and $f \circ g$ denotes function composition.

We consider only maps on a normed vector spaces V (over a field $\mathbb{F}$), allowing us to measure distances and size. We will use standard notation, where $|x|$ is the norm of vector x.

When examining a map inducing a fractal from the Escape-time class, we will most commonly examine its *escape-time*. Informally, we are examining how fast (regarding number of iterations) the map surpasses a defined boundary, possibly leading to divergence.

Definition 3 (Escape-time)**.** For given map f, $x \in V$ and boundary $G \in \mathbb{F}$, the escape-time is the smallest n_0, such that $\forall n < n_0 : |f^n(x)| < G$.

This can be seen as a function $e_{f,G} : X \to \mathbb{N}^*$; such approach will prove useful in following sections. It should be noted that such n_0 may not even exist; then we set $e_{f,G}(x) = \infty$. The boundary G is usually called the *escape radius*.

The *escape-time* computation can be further generalized to measuring the convergence resp. divergence rate. Such approach is used, for example, by the Newton fractal (there, the map may converge to n different roots, so the single-boundary *escape-time* does not apply).

2.4.2 Complex maps

Our goal is to examine 2D fractals. This is why we restrict to complex maps, i.e. maps where $X = \mathbb{C}$. The complex numbers have proven to yield chaotic results after being mapped with even simple functions.

The set of complex numbers $\mathbb{C}$ is isomorphic with $\mathbb{R}^2$, meaning that we can interpret algebraic operations on $\mathbb{C}$ as operations on two-dimensional vector space over field $\mathbb{R}$.

Thanks to this isomorphism, we will often consider our complex maps to map $\mathbb{R}^2 \to \mathbb{R}^2$. Furthermore, especially in the Implementation chapter, we will use (x, y) instead of (Re, Im) when referring to the real and imaginary component

of a complex number and when referring to the complex plane itself. This will prove practical as computer graphics libraries operate in the terms of the real plane with x, y axis. We will also use terms *complex number* and *point* (referring to a point on the real plane) interchangeably, if there is no ambiguity.

2.4.3 Complex quadratic map

One of the algebraically simplest examples of a complex map that behaves chaotically is map defined by the following complex quadratic monic polynomial.

Definition 4. Let $z, c \in \mathbb{C}$. The complex quadratic map Q_c is an iterated function with iteration given by

$$Q_c : z \to z^2 + c.$$

This map gives rise to the Mandelbrot set and to quadratic Julia sets. After having it mentioned many times, let us now finally introduce those sets.

2.4.4 Mandelbrot set

The Mandelbrot set is one of the most famous mathematical structures, known both to the scientific community and to the general public. There is plethora of descriptions of the set and its properties both in literature and online (for example [55], [43]). For this reason, we will introduce it only briefly.

The basis for its discovery were laid by Pierre Fatou and Gaston Julia in 1918 when looking at the theory of iterated rational functions from a global point of view [77]. In 1980, Benoit Mandelbrot discovered the set (originally called "μ map") during his research of the Julia and Fatou sets [46] and immediately created a widespread interest in it. Later the set has been named after Mandelbrot.

The Mandelbrot set is set of complex numbers c for which the complex quadratic map Q_c does not diverge when iterated from 0 infinitely many times. Formal definition follows.

Definition 5 (The Mandelbrot set M)**.**

$$M = \left\{ c \in \mathbb{C} \mid (\exists s \in \mathbb{R}^+)(\forall n \in \mathbb{N}) : |P_c^n(0)| \leq s \right\}.$$

The set is closed and contained in the closed disk of radius 2 around the origin[1]. Thus, the formal definition can be simplified by taking $s = 2$. This leads to $M = \{c \in \mathbb{C} \mid e_{Q_c,2}(0) = \infty\}$, e being the *escape-time* from Section 2.4.1. We will show practical methods of determining whether given c is in M in Chapter 3.

When analyzing the set as a fractal, we are often more concerned with its boundary rather than with the set itself. Informally, c is at **the boundary of the Mandelbrot set** if there exist numbers a and b in the neighborhood of c such that a is in the set and b is not. For formal definition, see topological definition of set boundary, for example [58].

When rendering the boundary as a colorful fractal, we are concerned with the *escape-time* (see Definition 3) with *escape radius* 2.

Unlike other fractals that we examine, the Mandelbrot set is connected [43].

[1]For formal proof, see for example [43]

2.4.5 Julia set and Julia fractals

Named after French mathematician Gaston Julia, the Julia set of an iterated function f consists of values that cause the function to behave chaotically. It is denoted $J(f)$.

The Julia set of a function can often be a simple curve (e.g. $J(z \to z^2)$ is the unit circle) or may even be empty (e.g. for constant functions). Visually attractive Julia sets arise from complex rational functions, whereas the most famous is the family of quadratic Julia sets, $J(Q_c)$.

Definition 6 (quadratic Julia sets)**.** For given $c \in \mathbb{C}$, quadratic Julia set is

$$J(Q_c) = \{z \in \mathbb{C} \mid e_{Q_c,2}(z) = \infty\}$$

There is a close coupling between quadratic Julia sets and the Mandelbrot set, arising from the fact that both are induced by the same equation. Note the difference between them:

- Mandelbrot: For each c in the plane, iterate Q_c, starting at 0.
- Julia: For fixed c, For each z in the plane, iterate Q_c, starting at z.

This is why there is only one Mandelbrot set, but infinitely many quadratic Julia sets. The mostly known property is that for a given $c \in \mathbb{C}, J(Q_c)$ is connected $\Leftrightarrow c \in M$.

In the following text, when referring to a *Julia fractal*, we will mean the quadratic Julia sets.

Same as with Mandelbrot, and for the same reason, when rendering the set as a colorful fractal, we are concerned with its boundary and the *escape-time problem* with *escape radius* 2.

2.4.6 Newton fractal

The Newton fractal is based on the Newton's numerical iterative method for finding roots of a polynomial. This method is generally numerically unstable, and for complex numbers in particular, it behaves chaotically in some regions. This is why it gives rise to a fractal.

It is induced by the following map.

Definition 7 (Map inducing the Newton fractal)**.** Let $n \in N, z \in C, p \in \mathcal{P}_{\mathbb{C},n}$ complex polynomial of n-th order, p' its first derivative. The map for the Newton fractal, N_p, has iteration given by

$$N_p : z \to z - \tfrac{p(z)}{p'(z)} \tag{2.1}$$

The method divides the plane to regions of attraction. In an attraction region (a.k.a. basin), all values converge to the same root of p when iterated by N_p. The boundaries of the regions do not generally converge, and behave chaotically (infinitesimal change of z leads to z' diverging or converging to an arbitrary different root of p). These regions are the desired fractal.

The fractal is parameterized by an integer n and $n+1$ complex coefficients of p, providing a rich variety of concrete instances to choose. There are combinations of

coefficients that will not behave chaotically or won't result in a visually attractive images, but generally N_p will always diverge on some open regions of the complex plane if $n \geq 4$ [52]. A commonly used simple instance of coefficients is (1,0,0,-1), i.e. $p(z) = z^3 - 1$.

When rendering the set as a colorful fractal, we usually pick a different color for each region of convergence. The convergence rate (how many iterations are needed to approach given root) may be used to further modify the color.

Sources: [21, 73].

2.4.7 Other examples

There are other examples of fractals induced by complex maps that are relevant for this thesis and whose implementation could extend the real time renderer. However, finding a visually attractive 2D complex map that is not visually similar to Mandelbrot/Julia set is believed to be hard [74, 34]. Examples of fractals arising from two dimensional complex maps are:

- **The Mandelbar set**, a generalization of the Mandelbrot set with iteration in the form $z \to z^k + c$ or $z \to (\overline{z})^{\,k} + c$ for $k \in N^+$, $\overline{z}$ denoting the complex conjugate. [86, 85]
- **The Exponential map**, with iteration given by $z \to e^z + c$. [10]
- Generalization of the previous, with iteration $z \to f(z) + c$ where f is an arbitrary non-linear transformation. It has been shown to yield visually attractive results for trigonometric and hyperbolic functions.
- **The Burning ship fractal**, a modification of Q_c, where each component is put in absolute value first: $z \to (|\mathrm{Re}\,(z)| + i\,|\mathrm{Im}\,(z)|)^{\,2} + c$.
- **The Gingerbread Man map**, with iteration given by

$$\begin{aligned} x_{n+1} &= 1 - y_n + |x_n| \\ y_{n+1} &= x_n. \end{aligned}$$

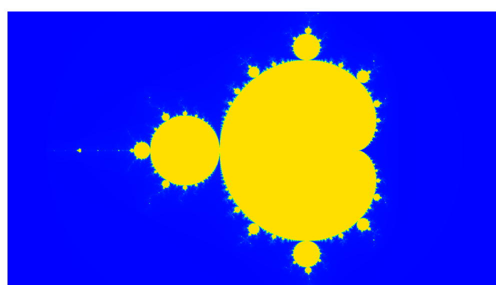
- **The Nova Fractal**, generalization of the Newton fractal with addition of c at each step. Let $c, \alpha \in \mathbb{C}, p \in \mathcal{P}_{\mathbb{C},n}$, the iteration is given by

$$z \to z - a\frac{p(z)}{p'(z)} + c$$

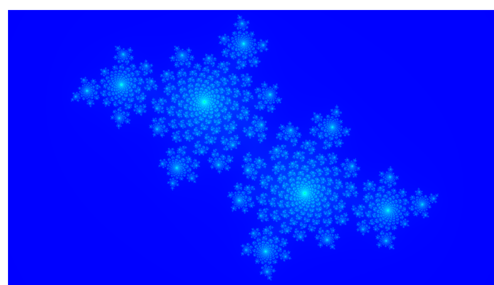
- Generalization of the previous, replacing $p(z)$ with arbitrary differentiable complex function. Again, it has been shown to yield visually attractive results for trigonometric and hyperbolic functions. [21, 84, 7]

For a broader list with detailed explanation and examples to each fractal, see [3].

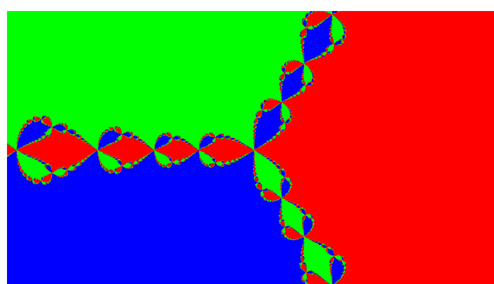
Figure 2.1: Visualization of the three given fractals.



(a) the Mandelbrot set (in yellow)



(b) Julia set of Q_c with $c = -0.4 + 0.6i$ (in light blue)



(c) Newton fractal for $p = x^3 - 1$, each of the root colored in different color.

3. Problem analysis

We have successfully laid the mathematical foundation of fractals and chaotic functions. Now we will analyze the problem specified by the thesis' goals and discuss possible solutions.

First, we describe algorithms for rendering of the three given fractals: Mandelbrot, Julia and Newton. Then we try to generalize the rendering to all Escape-time fractals. We find the need to generalize the problem to chaotic function. Then we take a look at rendering itself and how a general chaotic function can be visualized as image.

Finally, we discuss the ways the program could be implemented, including programming language, GPGPU technology, and graphic libraries.

3.1 Rendering algorithms

The thesis' goals specify three concrete fractals that are to be implemented: Mandelbrot, Julia, Newton. We now look at each of them and describe possible rendering algorithms.

3.1.1 Making a fractal from the Mandelbrot set

The Mandelbrot set is the most common example of Escape-time fractals. There are many books, papers, articles and blogs dedicated to it, some of them discussing modern rendering methods. Rendering it on the GPU is also a very common idea, because it is a great example of a parallel-computable function with a great visual result.

As introduced in Chapter 1, we do not want to render the set itself, we are interested in values c in its boundary and the convergence resp. divergence rate of Q_c when iterated from 0.

Proving divergence is easy: we iterate Q_c, starting at 0. When, for any complex number z', it holds $|z'| \geq 2$, then we know that $Q_c(z')$ diverges. So, if any intermediate value of the iteration satisfies $|z'| \geq 2$, we stop and say that at point c, the Q_c mapping diverges.

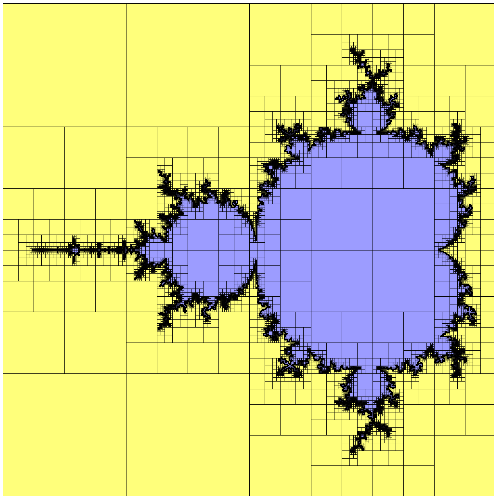
On the other hand, with iterative methods, determining convergence is hard. We can generally never be sure that a value is in the set (for some values, e.g. $z = -1$, it can be proven mathematically).

Bounded computation

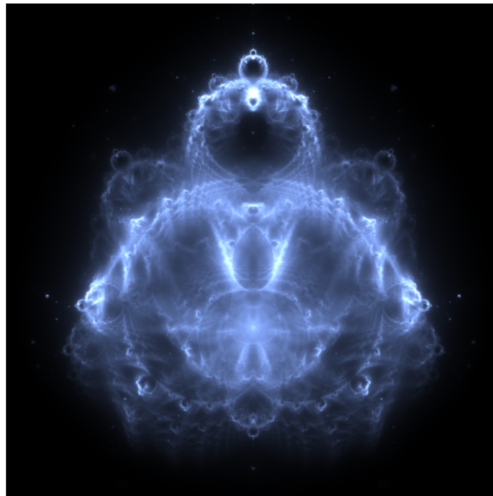
The usual solution to this uncertainty is introducing a parameter, the maximal number of possible iterations, let us name it *maxIter*. If after *maxIter* iteration, it still holds $|z'| < 2$ we abort and say that Q_c mapping probably converges and c is **maybe** inside the Mandelbrot set.

This way, we enclose the set from outside — higher the *maxIter*, the more points are marked as divergent. *maxIter* can be increased infinitely, producing more and more accurate method. of course, the computational complexity rises with higher *maxIter*. Empirically, usual *maxIter* values are about 250–400 on

Figure 3.1: Alternative algorithms for visualizing the Mandelbrot set.



(a) An illustration of the Mariani-Silver algorithm.



(b) A classical depiction of Buddhabort, resembling Gautama Buddha.

simple renderers. (For example, the default value of *maxIter* in *XaoS* is 250). Remark: Later in Section 5, we will show that values 900–3000 are possible in real time, introducing unprecedented level of detail; but let us get back to the analysis.

Escape time algorithm for the Mandelbrot fractal

The analysis above leads to the most common algorithm for making a fractal out of the Mandelbrot set, the Escape time algorithm.

Algorithm 1 Escape time algorithm for the Mandelbrot fractal

```
1: procedure EscapeMandelbrot(c : complex, maxIter : integer)
2:     z : complex, i : integer
3:     z ← 0
4:     i ← 0
5:     while i ≠ maxIter and abs(z) < 2 do
6:         z ← z^2 + c
7:         i ← i + 1
8:     end while
9:     return i
10: end procedure
```

The returned value is then used for determining a color. The simplest way is a linear mapping of i using $(0, maxIter) \rightarrow$ (`BLACK`, `WHITE`). More interesting methods are discussed in Section 3.5

The Escape time algorithm is very easily parallelized, making it an ideal candidate for an example GPU algorithm. There are, however, other possible approaches.

Mariani-Silver algorithm

Mariani-Silver algorithm is an algorithm that speeds up rendering of the Mandelbrot significantly.

[56] describes it as follows: "The Mariani-Silver algorithm starts with a rectangular grid of pixels. The pixels on the boundary of the grid are evaluated, and if they all evaluate to the same result the rectangle is filled in with a solid color; otherwise the rectangle is divided in half (or quarters) and the algorithm is applied again to each half/quarter." See Figure 3.1 for visualization.

The algorithm relies on the connectedness of the Mandelbrot set. It also relies on the CPU computing model. This recursively-refining algorithm with unpredictable branching would be very hard to efficiently implement on the GPU.

Buddhabrot

Buddhabrot is another technique of rendering the Mandelbrot set. Its name reflects the output's resemblance to the classical depiction of Gautama Buddha. [8] See Figure 3.1 for illustration.

Each pixel with value z has an associated integer value, a counter. The counter is incremented by one each time any iteration of Q_c (for any other point c) acquires intermediate value z. Q_c is iterated for all pixels, the image is then colored based on the counters values.

The algorithm yields very beautiful results. At the same time, it introduces an enormous memory pressure: literally chaotically incrementing values at different regions of the memory, almost every memory access is a cache miss. Also, data are concurrently written. The GPU memory model and concurrence model are not suitable for this kind of use and would probably lead to performance decrease in orders of hundreds or thousands.

Other algorithms

For the Mandelbrot set, there have been many other algorithms introduced in literature and online. Variations on the Escape-time algorithm are proposed for example by [75]

Based on the discussion above, we decide to create the fractal using the Escape time algorithm, for its simplicity, parallelizability and the future generalization potential.

3.1.2 Julia fractals

Proceeding form the Sections 2.4.5 and 3.1.1, we can easily modify the Escape time algorithm of the Mandelbrot fractal for Julia:

As can be seen from the algorithm, and in unity with Section 2.4.5, the algorithms for Julia and Mandelbrot fractals are almost the same. This should be no surprise, as the idea behind the fractals is also almost the same. One only has to pay attention to their parameters and note which point is the point being rendered.

Algorithm 2 Escape time algorithm for Julia fractals

```
1: procedure EscapeJulia(c, z : complex, maxIter : integer)
                ▷ c is algorithm's general parameter, z is the desired point
2:     i : integer
3:     i ← 0
4:     while i ≠ maxIter and abs(z) < 2 do
5:         z ← z^2 + c
6:         i ← i + 1
7:     end while
8:     return i
9: end procedure
```

3.1.3 Newton fractal

After having scrutinized Julia and Mandelbrot fractals, let us now pay more attention to the third given fractal, Newton.

Following from the Equation 2.1 and Section 2.4.6, we assume that we are given a complex polynomial p and are asked to make a fractal out of it.

The idea of the fractal is, for each point c, to find to which root of p (if to any) the Newton's method converges. For that, we need to know two more things: the derivative of p, p', and the roots of p, $r_1 \ldots r_n$.

Finding the first derivative and the roots

Computing the derivative of a polynomial is a simple task with a generally known solution, so we assume p' to be known. Regarding the roots, we could (1) use some of the known methods for finding them or (2) require them as algorithm input. We now analyze both approaches.

(1) The *compute-approach*: If no roots have been given, we need to compute them. A suitable method needs to be used: for example, the Newton method. Although we use the numerically unstable Newton method for creating a fractal, core of it lying in the imperfectness of the method, the same method can still be used for finding the roots, if applied many times and on wide-enough universe.

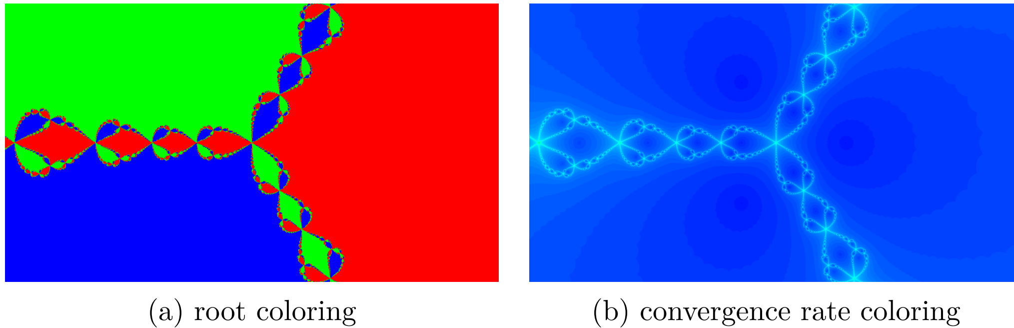
(2) The *provided-approach*: we require the user of our algorithm to provide correct roots of p. If the provided values are not correct, algorithm behavior is undefined.

The *compute-approach* is imperfect, but possible and could yield good-enough solution. It needs applying the method to many points and then analyzing the results, for example by building a histogram and marking the n most common values as roots. Other sub-problems should be solved, for example:

- checking if a root-candidate found by the method really is a root,
- deciding how to build the histogram,
- deciding whether to compute the histogram on the GPU or on the CPU,
- deciding how to pre-compute and store the values with the GPU's restricted threading- and memory-model.

The *provided-approach* puts responsibility on the user of the algorithm. This partially makes sense, as roots are an inherent property of a polynomial and

Figure 3.2: The two coloring methods for Newton fractal.



(a) root coloring (b) convergence rate coloring

should be computed only once, at the beginning. It is much easier to implement than the *compute-approach*.

For the sake of maintaining the scope of this thesis, we choose to implement the easier solution, where the roots are provided. However, as our renderer is going to be generic, its users are free to implement the *compute-approach*.

Coloring method

There is a variety of methods for coloring the Newton fractal. The two most common are following. (1) Root coloring: choose a fixed color for each root and colorize a point based on the root it converges to. (2) Convergence rate coloring: colorize each point based on a rate at which it converges (for example by the number of iterations needed, same as Julia and Mandelbrot). Both methods are illustrated in Figure 3.2.

In our implementation, we implement both approaches, to demonstrate the idea that a fractal can be colored in multiple, independent ways.

The two approaches can also be combined, creating even more visually attractive fractals. We do not implemented the combined version, for the sake of thesis' scope. But again, the users are free to implement it.

The algorithm

Knowing the derivative, the roots and coloring methods, we can now formulate the algorithm. The algorithm is based on iteratively applying the Newton's formula, exactly as prescribed by the Newton's method. It is basically the the canonical of performing the Newton's method. Thus, no further discussion is needed regarding this approach. See Algorithm 3.

The output of the IterateNewton can be colorized by any of the methods described above. If r is `undefined`, we use some default color, for example black.

Algorithm analysis

The computational complexity of this algorithm is $O(n * maxIter)$. In each step of the algorithm, we need to evaluate the polynomial of degree n, which needs $\Theta(n)$ operations (for example using the Horner's scheme), and then we compare z with every of the n roots. The need to compare z with roots does not asymptotically slow down the program, due to the linear nature of the polynomial evaluation.

Algorithm 3 The iterative algorithm for Newton fractals

```
1: procedure IterateNewton(z, r_1 ... r_n : complex, maxIter : integer, p, p' : function)
2:     i : integer
3:     r : complex
4:     i ← 0
5:     i ← undefined
6:     while i ≠ maxIter do
7:         z ← z − p(z)/p'(z)
8:         if for any i ∈ 1...n holds : z ≈ r_i then
9:             r ← r_i
10:            break while
11:        end if
12:        i ← i + 1
13:    end while
14:    return tuple(i, r)
15: end procedure
```

3.1.4 Rendering a fractal as a whole

We have shown how to compute the fractal value of the specific fractals. But how to populate the whole screen with pixel data? Now we propose a very simple approach that we call *naïve*. It is based on [50] and on the approach used in many other projects, especially those listed in Section 1.3.

To be able to render a fractal, we need to be specified the Region of interest, a rectangular segment of the complex plane that should be visualized. We also need to be given a pixel grid (texture) to render on, and, of course, a fractal function. Here we propose the general idea how to do it; the implementation details are discussed in Appendix A.

The segment of the complex plane can be represented, for example, by its bottom-left and right-top corners. Other representation are introduced in Appendix A.

Algorithm 4 The naïve approach to fractal rendering

```
1: procedure RenderFractalNaive(BL, RT : complex, output : texture, f : fractal, maxIter : integer)
2:     for i in 1... width(output) do
3:         for j in 1... height(output) do
4:             p ← Transform(i,j,BL,RT) : complex
5:             v ← f(p, maxIter, otherParams... ) : fractalResult
6:             c ← Colorize(v) : color
7:             output[i, j] ← c
8:         end for
9:     end for
10:    return output
11: end procedure
```

In the Algorithm 4, f is implementation of one of the fractal rendering algorithms described above, the COLORIZE method is one of the coloring methods described above and the TRANSFORM method linearly transforms given indices to corresponding coordinates within the complex plane.

The naïve approach gives the basic idea how to render an fractal image. We are going to generalize this approach in the next sections and vastly improve it in Chapter 4.

3.2 Generalization

We have shown how to visualize the three given fractals. Yet, our goal is more general, to visualize a generic Escape-time fractal. How to achieve that?

Finding some abstract entity that generalizes Newton, Mandelbrot and Julia fractals, and allows for others, is not hard. We use the least common denominator of their parameters, or allow *fractal-specific parameters*. Those can be implemented in many ways, for example as an array of objects or a text string in a defined format. There are other troubles, however.

Both Mandelbrot and Julia fractals base their value on the number of iterations. Those two fractals can be generalized by an iterative process that receives a complex map as input. The map can be supplied for example in form of an algebraic equation. However, for the Newton fractals, this is not enough — it uses the convergence root as its value (beside other approaches). Our generalization should allow the user-defined fractals to choose other ways of finding the value than the convergence rate.

Performance

Beside finding a general abstraction, we still need to fulfil thesis' goal of achieving high quality in real time with the generic fractals. This is surely not going to be possible with the naïve approach; even with the high performance of GPUs.

The naïve approach invests the same amount of resources effort into every pixel, into every iteration. A process that decides to stop iterating or to sample some non-fractal pixels with less resources would help us with the real time performance.

For the Mandelbrot set, there are many known approaches to optimizing the rendering time, see [20] . However, they rely on the specific properties of the Mandelbrot set and cannot be generalized for our cause.

For proposing some general speedup methods, we would need to make more assumptions about the functions that are being rendered. For example, the Mariani-Silver algorithm from Section 3.1.1 relies on connectedness. But connectedness is too restrictive; most of the Escape-time fractals are not connected. Generally, the more assumptions, the less generic the rendering process is. How to find the balance?

In the last few paragraphs, we have come up with many considerations. Getting an inspiration from other fields of computer graphics would help us in finding the solutions.

Nowadays, the most vivid rendering-addressing CG-branch is photo-realistic rendering of 3D scenes. In this field, rendering is often viewed abstractly as per-pixel computation of 2D integral of some abstract image function (for example, the Monte Carlo integration method is used for the Path Tracing technique.) For computing the integral, samples of the function are taken at different points and then combined.

We tried to look at our problem from this abstract point of view too. We found that chaotic functions were suitable for us. Let us discuss them.

3.3 Chaotic functions

The idea of an image function that is being sampled and integrated, and which is usually reasonable, but sometimes very complex and unpredictable, seems very fitting.

This is where chaotic functions came to our mind. Let us remind, what is meant by "chaotic": Informally, a function is chaotic when an infinitesimally small change of the input changes the output unboundedly. Formal definition can be found in Chapter 2.

The Escape-time fractals are examples of chaotic functions: The result of an escape-time iteration often cannot be estimated from samples at near points; Also, the number of iterations and the computation result may differ rapidly for values that are near to each other.

For the rendering process, instead of considering a somehow-defined 2D fractal, we will now consider a general 2D chaotic function.

This puts us before a more general problem, which is very challenging and thus interesting: The sampling of chaotic functions. The core challenge of the task lies in the fact that there is only little information that we know about the function that we render. Yet, we still assume some properties, which we now review.

Restrictions

At each sample that we take, we cannot be sure, how complex its computation is going to be. Neither can we be sure that the function value will be similar to other near values. Yet still, we anticipate that the function shows something "reasonable" and that in many cases, it behaves "nice" in some way, for example is connected. The core challenge would then be to identify the chaotic regions and concentrate the computational power on them, while applying some interpolation or a similar method on the connected regions.

We lay some more restrictions on the general chaotic function f. We need the assumption that f is dependent only on its argument; especially, it is independent on function values from its neighborhood. This assumption would exclude many non-Escape-time fractals, and probably excludes many other chaotic functions. On the other hand, it perfectly fits Escape-time fractals induced by complex maps. For such functions, each sample can be computed independently. This leads to our goal of parallel implementation.

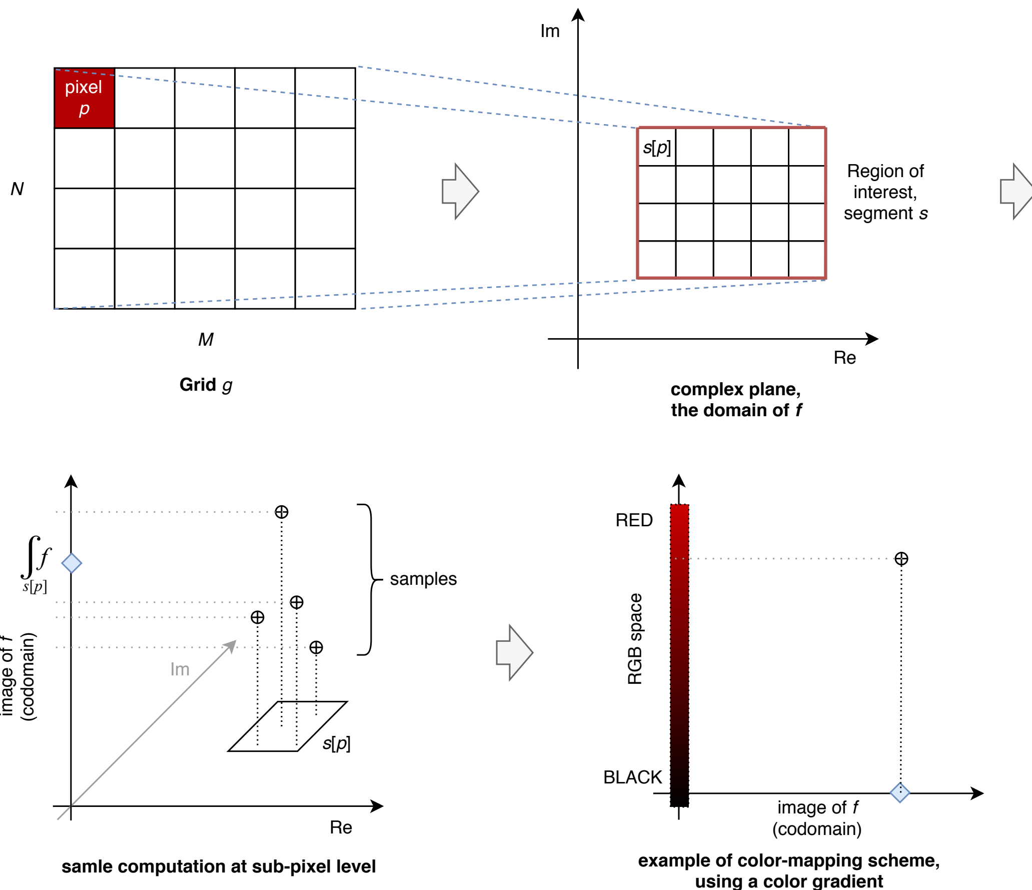


Figure 3.3: Illustration of the definition of the rendering problem: pixel p is mapped to dark-red color.

3.3.1 Formal definition of the rendering problem

Having thought about chaotic functions and having laid our restrictions on them, let us now formally state what is the rendering problem that we want to solve. We hope that this is going to be helpful in our future analysis of performance-increasing methods.

The idea of the following paragraphs is following: We want to project an infinitely-detailed image on a $M \times N$ pixel grid. We want to compute a 2D integral of the area of each pixel. For this, we compute many function values (a.k.a samples) at sub-pixel level, and then combine them using a weighted average. The idea is also illustrated on Figure 3.3.

Now formally and precisely. Let $f : \mathbb{C} \to \mathbb{R}$ be a provided chaotic function[1]. Let g be a grid of pixels, $g = \{1 \dots N\} \times \{1 \dots M\}; M, N \in \mathbb{N}$. Let $s \subset \mathbb{C}$ be a finite rectangular segment of the complex plane, with width-to-height ratio M/N. s is also known as *region of interest*.

We also assume

1. Semi-niceness: At some domain, f is chaotic, yet at some domain, f behaves as if it were continuous.
2. Determinism: For any given floating-point precision, we can analytically compute $f(c)$, and the result is deterministic.

[1]for example our Mandeblbrot implementation, or a user-defined fractal

3. Independence: The computation of $f(c)$ is independent of any other values than c, specially of c' and $f(c')$ where $c' \in U(\delta, c')$.

We map the pixel grid g to the complex plane segment s, so that each pixel $p \in g$ corresponds to $s[p]$, a square-shaped segment of s.

For each $s[p]$, we want to compute the piecewise 2D integral

$$\int_{c \in s[p]} f(c)dc$$

using sample-based numerical integration. This is the core challenge of Chapter 4.

Combining the first mapping and the integral, we have obtained a mapping of pixels $\in g$ to mean values of f at the pixels.

Finally, we colorize the pixel grid, i.e. use some predefined mapping $\mathbb{R} \to RGB$, where RGB is the standard RGB color-space.

3.3.2 Performance-increasing heuristics

Having formally stated our problem and the assumptions we take, we are ready to discuss and explore more performance-speedup methods, aiming at the thesis' goal of real-time performance.

We would need a method that determines regions of the image that contain fractals, so that we can focus the computational capacity on them, rather than on wide single-color areas. This means that we are looking for a method that determines whether a region is chaotic or non-chaotic.

From the realm of photo-realistic rendering, that inspired us, we have gotten us some idea of other heuristics-based methods that can be used for a performance speedup.

We feel that the discussion about those methods is going to be very wide. Also we feel that it could be more dreaming or hoping rather than an analysis. Finally,there are no granted results. For all these reasons, we introduce a separate chapter for this topic, Chapter 4.

Let us now go back to the analysis of the problem, and let us see how the other thesis' goals can be achieved.

3.4 Supersampling

Having introduced the precise formal model, let us dig into another goal of the thesis: high quality.

Similarly to many other visualization methods that render a continuous objects on a discrete pixel grid, our integral-based method is prone to aliasing, as illustrated by Figures **??** and 3.4. The intuitive way to high quality is suppressing the aliasing. The most common anti-aliasing method is supersampling.

3.4.1 Sample distribution

One of the main challenges of supersampling is sample distribution. A distribution that does not lead to other aliasing effects is desired.

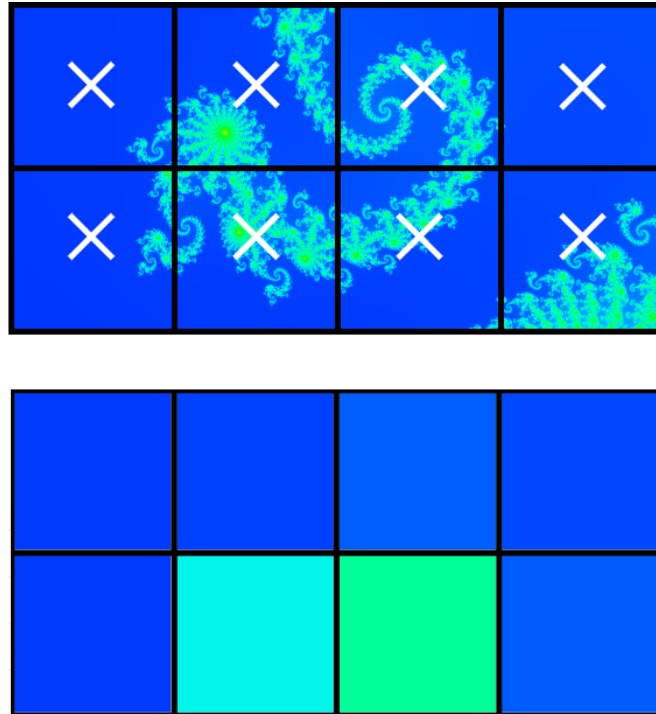


Figure 3.4: Rasterisation: what should be the color of each of the 8 pixels? Below are pixel colors based on only one sample, taken in the pixel's center.

There are many possible sample distribution methods, including uniform distribution, pseudo-random distribution, low discrepancy quasi-random or Poisson-Disc. See Figure 3.5 for illustration.

Uniform distribution is generally not used with smooth objects because it can interfere with objects' edges. For the random-based distribution methods, the *discrepancy* of the random sequence is crucial. Sequences with low discrepancy are needed, such that their values are distributed more evenly and do not make clusters; those sequences are denoted as quasi-random.

In the field of photo-realistic rendering of daily-life objects, mostly pseudo- or quasi-random sequences are used, to prevent interference of the sub-pixel layout with the smooth shape of the daily-life objects.

We argue that the Escape-time fractals, our main rendering focus, are rarely smooth objects and thus the main drawback of using the uniform distribution does not apply for them. At the same time, the uniform distribution is very easy to implement and further develop.

The scope of our thesis is wide, so in this case, for the sake of simplicity, we decide to use uniform distribution. However, another distribution method is a definite candidate for a future work.

3.4.2 Sample composition

After the samples have been taken, arithmetic mean should be used to compose them together to one value. We have two options as what to compute the mean from: the chaotic value $f(c)$, or the color that has been derived out of it. Both approaches have their pros and cons.

Chaotic value composition is algebraically straightforward and does not introduce new colors. However, for points at the chaotic edge, it would produce non-existent "mean" values that the chaotic function may actually never gain.

Color composition is more complex, needing a different color space than RGB (for example LUV or LAB). It is also prone to introducing new colors, that weren't contained in the original color palette. It could lead to smoother results, which is neither positive or negative with fractals (many fractals are inherently sharp).

As there seem to be no right answer, we choose the approach that is simpler: composing the chaotic values.

3.5 Fractal coloring

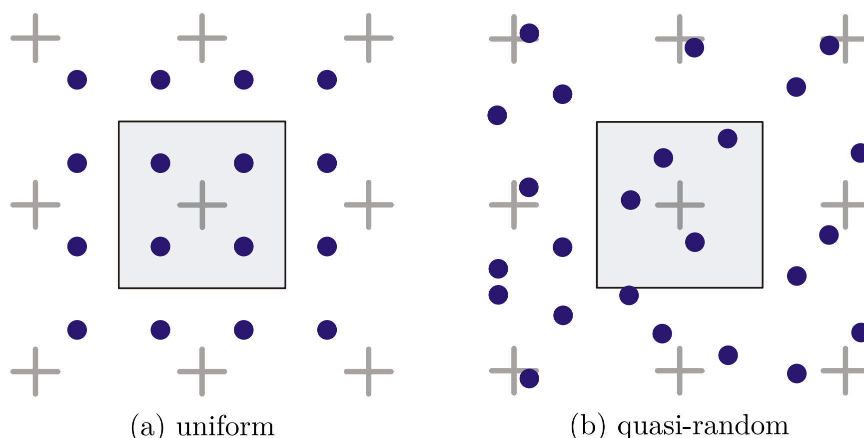
After the fractal has been sampled and its chaotic value has been computed by methods described above, we need to assign it a color. Many techniques for fractal coloring are known. [28]

Since we work with generic chaotic functions, we need a mapping $\mathbb{R} \rightarrow RGB$ as described in Section 3.3.1. There is a canonical way of mapping a real value to a color: an infinite 1D texture. In reality, the texture is not infinite, but the value is mapped modulo texture length. This approach is absolutely generic, as the texture can contain arbitrary data, and is easy to implement on graphic cards.

A usual implementation would be a symmetrical texture with a color-gradient. We will provide few of such textures (hence fractal color schemes) in our implementation.

There is also one thesis' goal regarding color: user defined coloring schemes. Because our program already uses color palettes, it seems logical to implement user-specific color schemes via palettes. A standard way of providing a palette is via an image file. We follow this standard way and simply let the user to load a color palette from a file that they provided.

Figure 3.5: Examples of sample distribution methods within a pixel.



(a) uniform (b) quasi-random

3.6 GPGPU technology

After the analysis of the rendering process has been done, let us now analyze the design of the program that is to be implemented. Proceeding from the thesis' goals, we need to decide what GPGPU technology to use.

Currently, there are two main GPGPU implementations to choose from: CUDA and OpenCL.

Our main demands are high computational power and an easy to use API, proceeding from the thesis aiming at an extensible and readable implementation. Learning tutorials for other programers who want to extend our program would be a plus.

GPGPU technology	High computational power	Easy to use API	Learning tutorials	Vendor support
CUDA	✓	✓	✓	nvidia
OpenCL	✓	×	×	nvidia, AMD, Intel

Based on the analysis, we decide for the CUDA technology. It limits us only to nvidia's GPU's, but this limitation is not in a contradiction with any of the thesis' goals.

3.7 Backend and Frontend

Continuing with the analysis of the program, the next challenge is program's architecture. When facing the problem of designing a program with a user interface, the usual common approach is to split the application into at least two parts: backend and frontend. This is true both for web applications and for desktop applications. As this is de-facto industry standard, we follow this approach.

The **backend** is going to be responsible for efficiently computing the fractal. We expect it to have an API that is non-dependent on specific technologies, fractals and frontend technology. We want the backend to be as fast as possible, and to exploit human perception flaws for this. Thus, information about the user need to be passed to the backend as well, for example mouse position or what interaction is the user performing, or what the user did and probably plans to do next. Specific details depend on the methods used for achieving real time performance.

This leads to following expectations on the backend's API. It should provide rendering methods that need to be given following information:

- what fractal is being rendered
- plane segment
- maximum iterations, maximum number of samples
- user related information

The **frontend** is going to be responsible for displaying the computed fractal and for translating user interaction to commands that the backend understands. Its architecture heavily depends on the language and GUI technology that we use.

The resources spent by computing the fractal are expected to be larger by magnitude(s) compared to the resources needed by maintaining the GUI. For this reason, there is no pressure on effectiveness of the frontend, as opposed to the backend. For sure, it is important to keep the overhead of running a GUI minimal. Still, when implementing the frontend, we do not need to cling to implementation details and may focus more on a clear design.

To create a maintainable program with readable code, as demanded by the thesis' goals, both frontend and backend should be further split into modules if they grow too big on responsibilities. Middleware layers may be added, such as service layer or quality-control layer. We are going to explore that after we have chosen programming language(s), libraries and other technologies and after we dig deeper into their specific details.

3.8 Programming languages

Now let us analyze what is the best programming language for achieving our goals. Our goals specify that an OOP-based language should be used. We also aim at a low learning curve for future developers, and thus we stick to a standard widely-spread language.

Standard OOP languages to choose from are: C++, C#, Java, JavaScript, Python. Let us compare them.

Proceeding from our goals, we define what we need from the language. We need the language to be well-suited for GUI creation, GUI being on of our main goals. We need the language to be cross-platform, this being one of our other goals. Aiming at real time, we need a language with maximum speed regarding the support for large bitmaps, or large arrays of primitive types (containing rendering data). And, of course, we need the language to have some support for programming in CUDA.

Need for two languages

Thinking about our needs on the language, CUDA-support and GUI-support seem to be contradictory. Usually, CUDA code is written in low level languages or scripting languages and GUI in more advanced languages. We now consider using actually two languages: one that targets CUDA and one that targets GUI.

Pros of choosing two different languages: For each of the tasks, we use the language that suits the task best, leading to a readable and maintainable design and code, which is one of our goals.

Cons of choosing two different languages: There is a need for integration. This introduces implementation overhead and may introduce more bugs and make debugging harder.

Choosing two different languages leads to our goal of maintainability, and this is why we decide for this approach.

CUDA-targeting language

CUDA is a complex platform, where bugs and bad-design happen easily. Our main requirement on the CUDA-targeting language will thus be industry and

community support. We also need the best efficiency possible and possibility to integrate with the GUI-targeting language.

The language recommended by nvidia company for writing effective CUDA code is its own CUDA C/C++ language. Being based on C++ and compiled mostly by the C++ compiler, it is for sure the most effective solution. The language allows compilation to modules that can then be launched by library calls from arbitrary other programming language, making it our ideal candidate. **We choose to use CUDA C/C++** as our CUDA-targeting language.

GUI-targeting language

Having chosen the low-level language, let us finally choose the GUI-targeting language.

language comparison	suited for GUI	cross-platform	CUDA binding	minimal performance overhead
C++	×	✓	✓	✓
C#	✓	×	✓	✓
Java	✓	✓	✓	✓
JavaScript	✓	✓	×	×
Python	×	✓	✓	✓

Remarks: C# is cross-platform in core features. However, its GUI libraries, WPF and WinForms, are both Windows-specific [87]. Integration of JavaScript with CUDA can be achieved, but in a very complicated way and has no library-based support.

Concluding from the analysis, **we choose Java** as our GUI-targeting language. Thanks to Java's versatility, speed and CUDA bindings, we will also be able to write most of the backend in Java.

Integrating Java with CUDA

For calling CUDA-related methods from Java, there is JCuda library. It is an open-source project with positive references on maven site. Jcuda is a straightforward Java wrapper of CUDA native API with 1:1 mapping. Thanks to this, there is no need to learn JCuda — one just uses the standard cuda api, only from the Java code. There seem to be no other maintained open Java↔CUDA binding library. For this reason, we choose to use the **JCuda** library.

3.9 Genericity

Coming back to our goals, the first goal of the implementation is for the program to be generic with respect to fractals. This means the program has to be extendable and be able to visualize arbitrary 2D chaotic function (possibly expressed by a formula or other means).

There are two possible approaches to allowing generic functions: (1) Dynamic: The user inputs function's definition to a text field, the program is able to dynamically load it, evaluate it and render it. (2) Compiled: The user writes function's

definition in a programming language and compiles it into a module that extends the backend. Let us have a closer look at them.

The **dynamic approach** is used, for example, by *XaoS*. It requires the implementation to parse the user-input. In our case, it means parsing the formula to a cuda-compatible language and then compiling it to CUDA code. Its advantages are user-friendliness and quick feedback. The main disadvantages are implementation complexity and the requirement on the end user to have a CUDAA-compiler installed, including a full C++ compiler — this is a big limitation. Also, the approach is restricted only on chaotic functions that can be expressed by an algebraic formula.

The **compiled** approach needs the user to write its custom formula/function as a code in CUDA C/C++ and to to compile it with a cuda compiler. The main disadvantage is the technical requirements on the user: the user needs to have programming skills. The main advantages are: (1) Robustness; this approach allows more complex formulas, using full expressivity of the Turing-complete language. (2) Easier implementation and (3) portability: after compilation, the user's custom function can be distributed with rest of the application and viewed by users who have no programming knowledge nor compiler installed.

Translation of user input to a cuda-compatible language is way beyond the scope of this thesis. We do not want to force our user to install a C++ compiler. And proceeding from Chapter 1, we expect the user who wants to add a new fractal to have basic programming skills. This leads us to deciding for the **compiled approach**.

3.10 Designing a graphical user interface

Having chosen an appropriate language, let us now choose how to build the GUI, that is another of the thesis' goals.

Fractal texture visualization

First, we need to have a look at how to visualize the rendered fractal. We want the visualization process to be *hardware accelerated*, i.e without the need to copy the image data to CPU memory to be processed by the front end and then copy the data back to GPU memory to be displayed — such behavior would mean a significant overhead.

CUDA has its own memory space, and data stored in it are not accessible by standard means. CUDA provides means to copy the data to texture memory, where it can be handled by standard API, for example **OpenGL**. Thus, we only need to display the image as an OpenGL texture.

OpenGL

For accessing OpenGL from Java, we use **JOGL** library by JogAmp.org, because it provides a straightforward way use OpenGL from Java with minimal overhead and it is maintained and supported.

Building user interface

For building user interface, we need to choose a GUI-library. Java provides 3 standard libraries that we can choose from: Java AWT, Java Swing and JavaFX. Each one is the historical successor of the previous one, JavaFX being the most modern one.

JavaFX is officially supported by the Java community as "the GUI library to choose nowadays", while the other ones are being discouraged from using. Other pros of JavaFX are: (1) It is a standard part of Java SE. (2) It has a convenient XML designer and (3) is supported by modern IDEs. However, it has no direct support for OpenGL. [61]

Swing is nowadays discouraged from using by the Java community. The visual components produced by it are old-looking. The design of the API is hard-to-grasp. On the other hand, it has a direct support for OpenGL.

AWT is recommended to use only in special circumstances. Our case does not fit them.

Support for OpenGL is a must for us. However, since Swing is not generally recommended, we want to minimize its use if possible. We strongly prefer using JavaFX, for reasons stated above. There is an option, though: JavaFX and Swing both provide an reliable easy-to-use integration for the other platform.

We can use **Swing** for displaying the OpenGL texture with hardware acceleration. And at the same time, we can use **JavaFX** for building the rest of the GUI.

3.10.1 Class architecture

The recommended way of designing a class architecture of a GUI-desktop application is using one of the following patterns: MVC, MVP or MVVM.

Each of the GUI frameworks, as well as OpenGL, introduces its own way of doing this.

We can expect our program to consists of several views, models and controllers/presenters. To follow a good OOP design, one of the thesis' goal, we use an appropriate design pattern for interconnecting them. For this use case, there is the PAC design pattern.

PAC

Presentation–abstraction–control (PAC) is an architectural design pattern used for interconnecting multiple MVC triplets. It is a hierarchical structure of agents, each consisting of a triad of presentation, abstraction and control parts. [70]

In the program, each of the controller/presenter would communicate with the others, while the views and models stay independent.

We do not aim at rigidly implementing the literal wording of the pattern; we adjust it to the specific needs of the implementations, as always recommended with design patterns. For example, in our case, the UI and the rendering-backend is expected to contain some same data. They can be both backed by a single implementation of a model implementing two interfaces.

3.11 Number precision

One the thesis' goals is support for zooming in or out of a certain part of the plane. When zooming, the order of magnitude needed to represent the points in the plane changes very quickly. This makes number precision an issue.

Quality-based fractal renderers use floating-point number representations with arbitrary precision. This allows for infinite zooming, but increases rendering time significantly.

CUDA is designed to work mainly with single precision, and supports double precision too (IEE 754). However, switching to double precision affects performance significantly. (Usually, the double arithmetic is 16x slower [22].)

We can choose to support only single-precision floating point arithmetic, both single- and double-, or arbitrary precision arithmetic.

1. Supporting **only single** floats is most easy to implement. The negatives are that only small zoom is supported, only down to E-7. It is a rather big value that is achieved quickly if zooming into a fractal. This significantly restricts the amount of fractal that can be visualized.
2. Supporting **both single- and double-precision** needs C++ templates to be used, which is harder to implement. Also, corresponding Java-CUDA binding gets more complicated. On the other hand, the pros are twice the zoom of the fractals, down to E-15.
3. Supporting **arbitrary precision** needs a special library to be used and is expected to be tremendously slow on the GPU. The implementation complexity would be much bigger than with two previous approaches. Its biggest advantage is the support for the infinite zooming in and out.

Implementing arbitrary precision would be out of the scope of the thesis. However, we to demonstrate the versatility and ability of generic rendering, and thus we decide for the combined approach, **supporting both single- and double-precision**.

We are going to implement the floating points in CUDA kernels using C++ templates. In Java, we use double precision everywhere, because the CPU is not a bottleneck and it does not introduce any slow-down. When passing the arguments to CUDA, they will be casted if needed.

4. Methods of achieving real time performance

During the problem analysis, we have found out that the process of fractal rendering needs performance improvement. We hope to find methods that will allow it. We will now look at how to achieve real time performance of the sampling process of chaotic functions, using heuristics-based methods.

Assumptions

The core challenge lies in the lack of assumptions that can be adopted in this task — chaotic functions are nowhere differentiable, cannot be integrated analytically, and even resolving a single sample can require orders-of-magnitude more computation than its immediate neighbors. Furthermore, function-specific assumptions that many visualization methods use (for instance the connectedness of the Mandelbrot set) need to be omitted as well.

Building on Section 3.3.1, we assume **semi-niceness**: At some domain, given function is chaotic, yet at some domain, it behaves as if it were continuous. We try to define magnitude of chaos and examine whether we can use this to determine which parts of the fractal are the most chaotic ones and thus need to be sampled more.

Then, instead of making more assumptions about the functions, we will make **assumptions about the user** and the way our program is used. Building on the third research goal of the thesis, Foveated rendering, we assume that the user focuses at the cursor.

Techniques overview

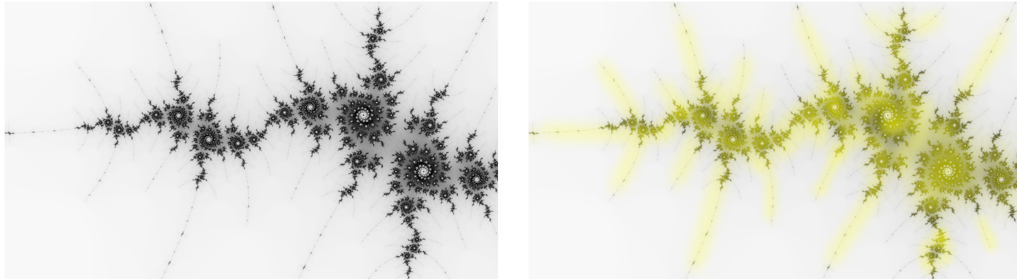
Building on Chapter 1 and on Section **??**, we introduce several heuristic-based methods, that are commonly used for sampling optimization. These include:

- Adaptive super-sampling: using higher density of samples within chaotic regions.
- Progressive visualization, technique where the user is shown a computationally cheap snapshot of the result first, and the result is made more accurate when the user interaction stops.
- Fixed frame resource budget, method that promises always-smooth user experience, with variable image quality.
- Foveated rendering: preferably sampling regions where the user probably looks.
- Technique of reusing samples from previous calculations.

We describe and analyze each heuristic and then propose a specific implementation that suits our cause. With the exception of adaptive super-sampling, we do not see the heuristics as independent methods. Quite opposite, we aim at combining all the other heuristics together to cooperate in the allowing of real-time rendering. When introducing a new heuristic, we rely on the previously introduced ones.

We do not go into implementation details. We leave those for next Chapter.

Figure 4.1: Illustration of chaotic regions of a function.



(a) Image of a chaotic function. (b) In yellow, regions that look chaotic.

Expectations

As heuristics are often non-exact, we do not formally state expected results. We state that we expect a decrease of the rendering time (compared to the naïve approach) with 'nonsignificant' loss of quality, which is vague. In Chapter 5, we compare rendering time and image quality when using the program with and without some of the heuristics.

4.1 Adaptive super-sampling

Supersampling yields very good visual result, however at a very high cost. Almost all the computational resources of our application are spent on computing the samples.

We may not increase speed using for example some interpolation method — due to the chaotic nature of our problem, each sample has to be taken independently.

However, the aliasing effects do not occur in the whole picture, but only in its chaotic parts. Very often only some (small) part of the picture is chaotic, and the rest looks rather continuous (e.g. the big areas of the $x^3 - 1$ Newton fractal or inner areas of the Mandelbrot set). It would be convenient to perform supersampling only on the aliased regions, where the fractal is chaotic.

For this, we need to develop a meaningful heuristic for detecting chaotic regions. This is one of the research challenges of the thesis. We try to propose an adequate method in the following paragraphs.

Adaptive process

General idea of an adaptive sampling process is following, according to [82]:

1. Take k initial samples, $x_1 \dots x_k$. Set $i \leftarrow k$.
2. Compute a credibility of the samples, $p(x_1 \dots x_i)$. This can be realized for example by probability (mapping to $(0, 1)$) or an indicator (mapping to $\{0, 1\}$).
3. Based on the credibility, either stop or set $i \leftarrow i + r$, take another r samples and continue with step 2.

Solution idea

Our first idea was statistical analysis of the samples. It is a simple idea, yet turned out to work surprisingly well with the family of continuous fractals.

Samples in non-chaotic regions would have similar values, samples in chaotic regions are expected to be distributed in some complex, non-uniform manner, possibly resembling some random distribution. Such behavior can be statistically analyzed using *variance*. We try to tell if the variance of the samples is high or low. High variance would correlate to chaotic region, low variance to non-chaotic region.

The value of data variance is data-dependent. We seek for a characteristics that is data-independent, normalized. The simplest such characteristics is the Index of dispersion.

Index of dispersion

The index of dispersion (a.k.a. relative variance) is a statistical characteristics of data. It is a normalized measure of the dispersion of a probability distribution. Being normalized, it is independent of the scale of the data. In our context it means that it is not dependent on the user-provided chaotic function, hence it is generic.

Formula for its computation is

$$d = \frac{\sigma^2}{\mu},$$

where σ^2 is the data variance and μ is the arithmetic mean.

Theoretically, the index of dispersion is not meant for the analysis that we are planning to using it for. We hope to get some results nonetheless, even if not exact or statistically proven.

The values of index of dispersion categorizes the data into three categories. We interpret those three categories in our own, chaos-related way.

1. d is zero or almost-zero — Data are uniformly distributed. This means the data are **non-chaotic**. ⇒ No more sampling needed.

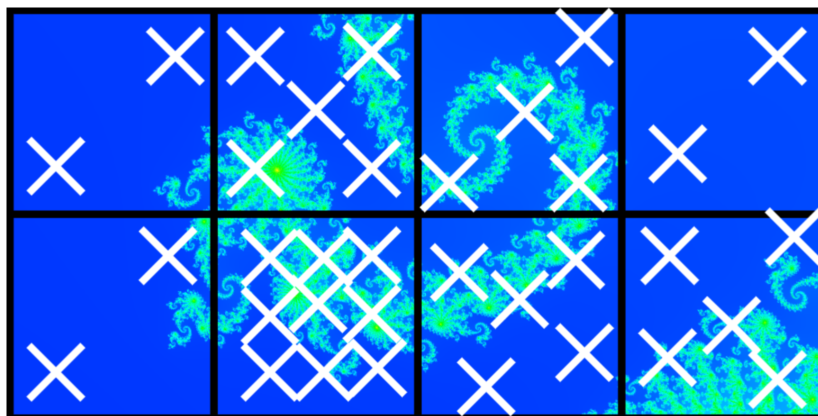


Figure 4.2: Adaptive supersampling: illustration of possible number of samples per pixel and sample distribution.

2. $d < 1$ — Data are under-dispersed, i.e. non-uniform. Our interpretation is that there is some **fractal**, but not the most chaotic region. ⇒ Few more samples are needed.
3. d = 1 — Not relevant for fractals; won't happen often. We join this with the previous case.
4. $d > 1$ — Data re over-dispersed. We interpret this as chaos. ⇒ As many samples as possible are needed.

4.1.1 Algorithm for adaptive supersampling

Based on the previous discussion, we propose the following algorithm. See Algorithm 5.

Algorithm 5 First attempt to adaptive supersampling

```
 1: procedure AdaptiveSuperSampling(maxSuperSampling, k : integer)
 2:     i : integer, samples : float[], sum, mean : float
 3:     i ← k
 4:     sum ← 0
 5:     samples[1...k] ← TakeSamples(k)  ▷ take initial samples to start with
 6:     while i ≠ maxIter do
 7:         samples[i] ← TakeOneSample
 8:         sum ← sum + samples[i]
 9:         if decision time then
10:             mean ← sum / i
11:             d ← IndexOfDispersion(samples, i, mean)
12:             if d < ϵ then
13:                 return samples                 ▷ No chaos. No more sampling.
14:             else if d < 1 then
15:                 decide to take few more samples
16:             else                               ▷ d ≥ 1, we are at a chaotic region
17:                 decide to take as many samples as possible
18:             end if
19:         end if
20:         i ← i + 1
21:     end while
22:     return samples
23: end procedure
```

On statistical significance

The described method may produce false-positives (a.k.a. Type I error), marking chaotic regions as non-chaotic. False-negatives (a.k.a. Type II error), marking regions with uniform samples as chaotic, are improbable. The smaller the initial data set, the higher the chances of a false-positive.

We allow some level of error of both Type I and Type II, as the method is only a heuristic and if few of the pixels are wrongly under- or over-sampled, it is no problem, as long as there is only a few of them.

We determine the significance level experimentally: In Chapter 5, we compare rendered fractals and visualized sample count. If we conclude that the parts of the fractal that are most interesting are usually marked as most sampled, we conclude that the significance level of our method is good enough. We start with the lowest possible (the initial samples per pixel $k = 2$) and eventually progressively increase k until satisfied.

The first idea to adaptive super-sampling that we had, statistical analysis of the samples, has proven to give some results. This is satisfying. There is still a big room for improvements, but we leave this for future work. Having introduced the basic method that we can build on, let us look at another ways of speeding up the rendering process.

4.2 Interactive mode and progressive visualization

We review other methods that speed up the rendering process and thus can lead to real-time performance. The methods that we are going to examine usually lay in reducing the visual quality of the image in some smart way that does not distract the user much, saving o lot of computational resources.

For those smart approaches, we need to take some assumptions about the user. For doing this, we introduce two different phases of user interaction. We discuss how the program should behave in each of them. For each phase, we then introduce techniques that can be used during the phase.

Introducing two phases

When a user interacts with our program (zooming in/out, moving the canvas, parameter change), fastest possible response is desired. Quick response time is in tens of milliseconds, bearable response time is about hundred milliseconds.

Let us call this kind of interaction *the interactive phase*. In some contexts, we need to also discern *the zooming phase* and *the moving phase* within the interactive phase. Those phases may interleave (if the user moves the canvas and zooms in/out at the same time).

Then, when user interaction has stopped, the user typically wants to explore the image in full detail before interacting again. Let us call this kind of interaction *the progressive phase*. During the static phase lower response time is needed and thus higher quality image can be rendered. At the same time, the user is expected to examine the whole image, thus the higher quality is actually desired.

Phases properties

During the static phase, higher quality images should be shown, and more computational resources are available. Adaptive super-sampling can be used during this phase, as it is expected to positively affect performance with a minimal effect on image quality. On the other hand, heuristics that significantly affect image quality should not used.

For the interactive phase, we assume that the user is not interested in image details. We have already assumed in the previous section that the user focuses on the cursor. Thus, lower quality and rendering fragments are allowed in outer regions of the image. This allows us to introduce a variety of other heuristics to be used during the interactive phase. We will do this in the rest of this Chapter.

Phase detection and lifecycle

One of the challenges of this approach is determining and switching the phase. We propose following, straightforward solution:

Every time a user interacts with the program, i.e. canvas move, zoom in or out, parameter change, interactive mode is started. After k milliseconds of user inactivity, progressive mode is activated, starting the progressive rendering process described above. After the progressive rendering has finished, the program is idle.

We have empirically stated that values of k about 50 milliseconds yield satisfactory results.

4.2.1 Fixed resource budget per frame

What is meant by "fast response time" during the interactive phase? Following up with the goals of the thesis, we aim at constant frame per second time (FPS) during the phase. The target FPS should be programmatically modified. This is a non-trivial demand, because the computational complexity of a chaotic function cannot be generally predicted.

This problem can also be rephrased as aiming at fixed GPU occupancy time per frame, and, more generally, fixed resource budget per frame.

In our search for a solution, we assume that two frames successively following during the interactive phase have very similar structure and hence computational complexity.

The idea of the method that we propose is following: Let T be the desired rendering time of one frame. We render frame F and measure its rendering time, $t(F)$. Then, we modify parameters of the successive rendering by the $T/t(F)$ ratio. I.e. as if we were rendering F again but were aiming exactly at rendering time T.

For this method to work, we need to be able to scale the rendering parameters in a way that directly affects rendering time, and know the parameter-to-rendering-time relationship.

For sake of simplicity, we assume that the program spends most of its resources on sampling the chaotic regions. Thus, changing the number of samples taken at chaotic regions directly affects the rendering time. This can be done via the `maxSuperSampling` parameter. We also assume that the relationship is linear, i.e. multiplying `maxSuperSampling` by k changes rendering time to $t * k$. These assumptions are very strong and are definitely a candidate for future work. On the other hand, the assumptions quickly lead to the following method: see Algorithm 6.

Algorithm 6 Simple method for achieving constant FPS

```
1: procedure AutomaticQuality(desiredFPS : integer)
2:     desiredFrameRenderTime ← 1 / desiredFPS : float
3:     relativeQuality ← 1      ▷ Realized by maxSuperSampling in our proposal
4:     lastFrameRenderTime ← desiredFrameRenderTime;
5:     while program not closed do
6:         relativeQuality ← desiredFrameRenderTime / lastFrameRenderTime
7:         lastFrameRenderTime ← RenderFractal(relativeQuality)
8:     end while
9: end procedure
```

4.2.2 Progressive visualization

What about the progressive phase? Common approach to rendering images during the progressive phase is progressive visualization.

Progressive rendering is based on the idea that when the user interaction stops, a quick low-quality preview of the fractal should be shown, and then higher-quality images should progressively be computed and shown. At the end, full quality image is shown. [51]

There a straightforward idea of achieving this: For improving the image quality, the maxSuperSampling parameter seems ideal. We start with the default maxSuperSampling value and then double the value for each iteration of the progressive rendering, until we have reached maximal desired quality (maximal supersampling level) or maximal frame rendering time.

We propose maximal frame rendering time to be 1 second. This would mean that progressive rendering takes up to 2 seconds ($1 + 1/2 + 1/4 + \ldots$ seconds) and the stabile full quality image is shown after two seconds. This seems like a reasonable time that the user would be willing to wait before losing interest.

We have shown two techniques that can be used during the progressive phase: progressive rendering and adaptive super-sampling. We will now look more at the methods for speeding up the interactive phase. Thus, in the rest of this Chapter, we introduce heuristics that improve performance while affecting image quality and are thus suitable for the interactive phase only.

4.3 Sample reuse

How to speedup the interactive phase more? Let us think if there is a process that seems like a waste of computing resources. When a frame is rendered, it usually does not differ much from the previous frame. Recomputing every sample again from scratch seems like a waste that we are looking for and that could be improved.

Instead of recomputing each sample from scratch, information from the last rendering could be reused. This leads us to another goal of the thesis, sample reuse. When changing the viewpoint, values from previous frames should be used and projected into the current frame to save resources.

This is one of the classical problems of computer graphics with many known

approaches. [32, 1] .

This topic is very complex and introduces many challenges. To not run out of the scope of the thesis, our discussion is rather brief, focusing on the most tangible challenges of sample reuse. Generally, this method could be further advanced and extended by even other research results. It has a potential of yielding very great results.

What data is reused?

When reusing data, we stand at the decision of what data to reuse. We have two possibilities: reusing the value of the chaotic function (the result of the integral) or reusing the color that has been derived from it.

None of the approaches is correct or wrong. The approach with color reuse would allow us to use GPU's built-in texture filtering and other tools. The approach with chaotic value reuse seems to be more accurate, with less information lost. Ideally, we want the reused sample to approximate the computed integral, not to approximate a color.

We decide for chaotic value reuse. However, color reuse may be examined as a future work.

Sample reprojection

When we want to reuse previous rendering data, the next challenge is finding the right data. For a given pixel in the texture being rendered, we need to find a corresponding value in the reused texture. Figure 4.3 illustrates this problem.

Generally, we are looking for an image warping method, that maps the previous texture to the current one. A linear warping is sufficient, as the user transforms the rendered complex-plane plane-segment in linear way only.

We need a simple linear transformation with following property: For given texture coordinate, find texture coordinates to the reused texture such that both represent the same point in the complex plane (chaotic function's domain). Let us call it GETWARPINGORIGIN.

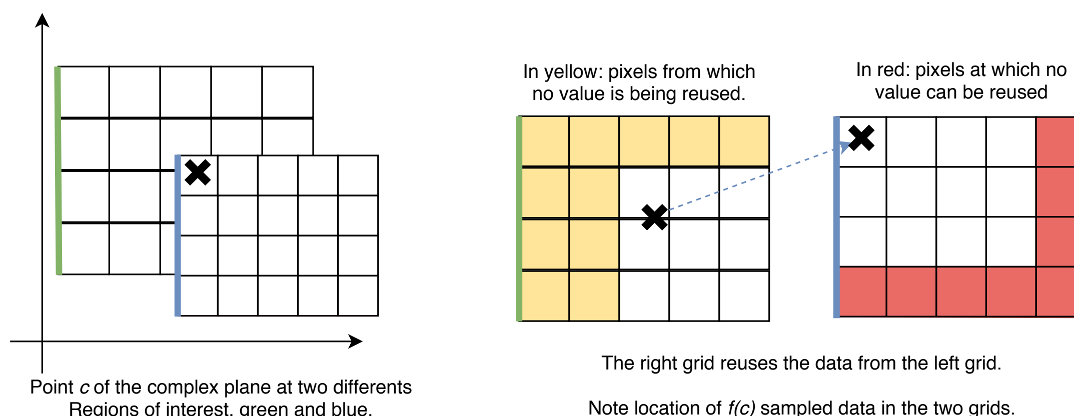


Figure 4.3: Illustration of sample reprojection when reusing samples.

Filtering

When reusing a pixel after the texture has been only moved around, GetWarpingOrigin returns integer indexes. Indexing the reused texture is straightforward.

When reusing a pixel after zooming in or out, GetWarpingOrigin returns float indexes, pointing at a mid-pixel. For reading a value out of the reused texture in this case, a filtering method is needed.

Texture filtering is a standard operation provided by the GPU's. However, we have decided not to reuse color data from a texture, but to reuse the chaotic samples. In this case, we need to implement the filtering ourselves.

There are many filtering methods available. As a proof of concept, we implement simple bicubic filtering. We leave a room for future improvements, for example for finding filtering methods that are less blurry and maintain the sharp edges that fractals usually have.

Memory access

There are two memory access models to choose from: scatter or gather.

Scatter lies is one pixel's value being written to its neighbors. Gather lies in pixel reading values from its neighbors. [54]

In this case, the discussion is quick and the conclusion is evident: on CUDA, only gather comes in question. Scatter approach needs concurrent writes, while gather only needs concurrent reads. The performance difference of both approaches is tremendous.

Fragments and sample degradating

By computing the per-pixel integral of a chaotic function, we perform basically rasterization. Rasterization often introduces artifacts. We have tried to to repress them by supersampling. Our raster-based sample reuse method is prone to even more artifacts, even with a texture-filtering method applied. The more a sample is reused, the more artifacts occur.

To reduce this effect, we introduce sample confidence: Each pixel's chaotic value is assigned a confidence, a real value telling us "how good" the value is (how confident we are about its quality). The initial confidence can be, for example, the number of samples taken to produce the value. Then, within time, its confidence decreases by a given coefficient.

As discussed before, during fractal moving, the artifacts of sample reuse do not occur. Thus, we decrease the confidence of a pixel's value only when zooming.

When combining new and reused samples, we compute their weighted average, the confidence being the weight.

Zooming center

When zooming in into the fractal, we assume that the user focuses at the zooming center. The quality of sample reuse is lowest at the zooming center — there is little information to reuse in there. We do not want that the one pixel at the cursor is reused in all directions. This would produce a blurry single-colored blots.

So, when sample reuse is used when zooming in, it is important to take more samples in the zooming center. A technique that focuses on this is Foveated rendering.

4.4 Foveated rendering

Foveated rendering is a technique that reduces workload by reducing image quality in the peripheral vision. [30] Originally, the method was meant mainly for head mounted displays. [67]. We utilize it for convenient wide displays.

The idea of the method is straightforward: concentrate the computational resources to areas at the focus' center and reduce them increasingly for more distant areas.

For controlling the amount of resources spent by the different focus/distant areas, we propose tweaking the `maxSuperSampling` rendering parameter, same as in Section 4.1.

Foveated rendering relies on determining where the user looks. Eye trackers are often proposed. We rely on a much simpler method, and in compliance with the introduction of this chapter, we simply assume that the user is looking at the mouse cursor.

This assumption seems very accurate during the zooming phase — The user points with the mouse cursor to the area that should be zoomed, thus very probably focusing at the area. On the other hand, the assumption seems inaccurate during moving or during the static phase, when the user is expected to rather look at the picture as a whole. For this reason, we utilize foveated rendering only during the zooming phase.

One of the perks of Foveated rendering is in quick responses to situations when the user quickly changes its focus, typically when using a VR headset. We elegantly solve this problem thanks to the "cursor points at focus" assumption, because tracking the cursor is trivial. The rest of the method seems to be rather simple to implement and may increase the performance significantly.

Human vision

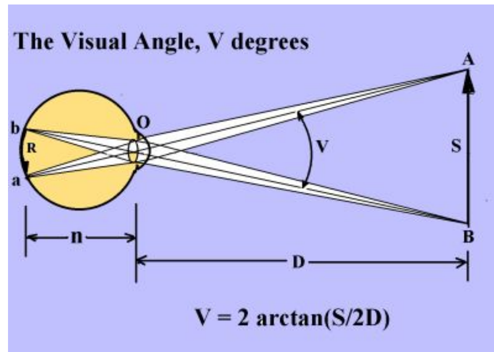
To determine which part of the sample grid is *distant* and which is at the *focus*, we rely on object's *visual angle*.

Visual angle is the angle the pixel subtends at the users' eye. It is usually defined in degrees. See Figure 4.4 for visual explanation and see e.g. [63] to learn more about it.

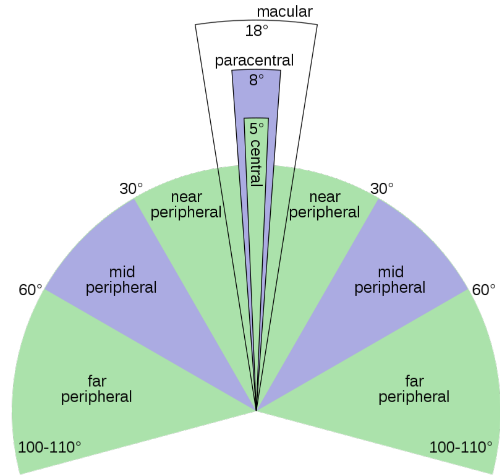
[53] divides human vision into various segments, in which we have different sensitivity to details. The most precise part of our vision is the *foveal vision*. Then there is *macular*, *near-* and *mid-peripheral* vision. Finally, in the *far-peripheral vision*, the visual acuity is very low. Figure 4.4 illustrates the segments.

[53] defines the segments as follows: The edge of the foveal vision is 5° in horizontal direction, the edge of the far-peripheral vision is 60° in horizontal direction.

Thus, only in 5° of the horizontal visual field we see precisely, whereas behind 60°, we are unable to spot any detail.



(a) Visual angle illustration. Source: [12].



(b) Segments of human vision. Source: [14].

Figure 4.4: Human vision

For simplicity, we now assume same values in the vertical direction as in horizontal. This should introduce minimal error, as we are working with wide-screen displays, that are small in the vertical direction. This is equal to modelling the human's field of view by fovea-centric circles, thus allowing to measure only one angle.

A model for calculation of number of samples based on human vision

Previous section examines the human vision using the visual field, and it precisely defines areas at which we see accurately and areas at which our vision has flaws. Based on that, we can propose a model for calculation of number of samples based on human vision. We build on the work of [67] Patney) and by [30] Guenter), simplifying their ideas by introducing previous assumptions to our model.

We let the user set the *screen distance* and the *real world pixel width*. Then at each pixel, we compute pixel's visual angle v. Based on the visual angle, we change `maxSuperSampling` of the pixel in following way:

1. $v \in [0°, 5°]$ — maximal value (max)
2. $v \in (5°, 60°)$ — $f(v)$
3. $v \geq (60°)$ — 1

Where $f(v)$ maps 60° to max, 5° to 1 and intermediate values interpolates in linear manner. See Figure 4.5

Foveated rendering works great together with adaptive supersampling, massively reducing number of samples needed to be taken in peripheral areas. The speedup should especially noticeable on wide screens, where the peripheral areas are big.

Could the foveation technique also be to any use to the other methods? We think that it could, and that we could combine all the introduced methods together to work as one compact, cooperating, real-time achieving toolset.

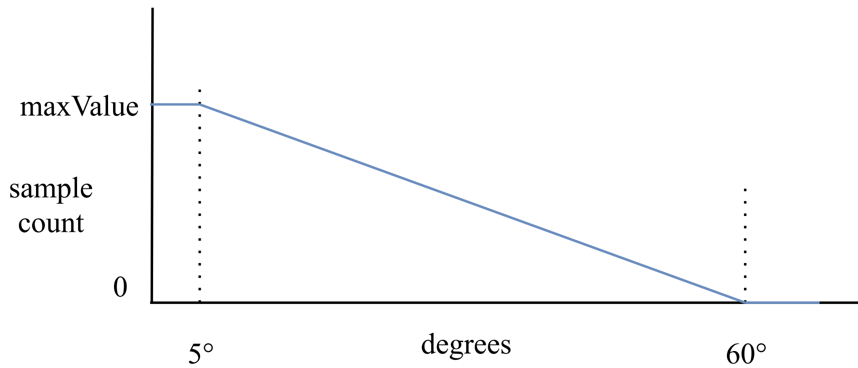


Figure 4.5: Foveated rendering: Mapping of the visual angle to sample count.

4.5 Combining the methods into a toolset

In the four previous sections, we have introduced two phases and five different techniques, four of which can be used during the interactive phase. At the beginning of this chapter, we have claimed that we do not see the methods as independent, but quite the opposite, want them to cooperate in achieving real time performance. Now we want to entitle us to the claim.

We want to put all the four methods available during the interactive phase together to one compact, cooperating, real-time achieving toolset.

Relative quality analysis

For the automatic quality to work, we need to be able to adjust the relative quality of the fractal being rendered. maxSuperSampling, parameter of the adaptive super-sampling methods, works good for this. At least for values bigger than one.

Should maxSuperSampling be lower than one, adaptive super-sampling is technically disabled and cannot help more. How to render images by taking less than one sample per pixel? With sample reuse! At all pixels, we reuse samples from the previous rendering. And then, at some pixels, we also compute some samples.

What pixels to sample if maxSuperSampling < 1? We could decide at each pixel whether to make a sample, with the probability *maxSuperSampling*. This would perfectly match the idea of maxSuperSampling symbolizing the relative image quality. But there should be a smarter way to investing our resources than randomly. What about investing them to the pixels that need them?

Resource investment when maxSuperSampling < 1

During the analysis of sample reuse, we have concluded that when zooming in, then more samples should be taken at the zooming center, to prevent the method from creating artifacts. Zooming center is the user focus. We have a method that allows preferring samples at the user focus: Foveated rendering.

For zoom in, foveated rendering should be plugged together with sample reuse and make sure that at the user focus, some samples are taken. In case that maxSuperSampling is 1, we take one sample at each pixel within the field of

view. If the value is lower, foveated rendering should be responsible for reducing the sample count at the field of view, and even reducing the foveal radius, if needed.

If zooming out or moving, we need to compute samples at the pixels at the image border, because there are no value to reuse. At those pixels, at least one sample must be taken, and then more samples can be taken adaptively if there is enough resources.

Conclusion

We have prescribed how the four techniques available during the interactive phase should rely on each other and how they should behave with respect to relative image quality, symbolized by the maxSuperSampling parameter.

We hope that this cooperation would lead to real time performance, even in cases when less that one sample per pixel is available.

4.6 Conclusion

Throughout this chapter, we have introduced two rendering phases: progressive phase and interactive phase. Progressive phase starts when the user interaction stops and is responsible for progressive rendering. The interactive phase starts when the user zooms in/out or moves the canvas. We introduced a method for adaptive super-sampling, which is used in both phases. For the interactive phase, we introduced more methods: fixed frame budget, foveated rendering and sample reuse. We bundled the methods used during the interactive phase into one toolset.

4.7 Other possible methods

We have introduced, described and analysed several methods that help us achieve real time performance and we have put them together into a toolset. There are for sure many other methods that could have been discussed and maybe implemented. Such a broad discussion would however be out of the scope of this thesis. At least, we list a few other ideas of real-time heuristics and leave them for a future work.

Under-sampling: Pixels in outer regions (determined e.g. by foveated rendering) would be sampled by less that one sample per pixel, i.e. one sample would be taken and then used for example for 8 or 16 neighboring pixels. We believe that this would introduce a significant speed up. However, efficient CUDA implementation is not straightforward. Different threadId-to-pixel transformation would be needed than for the approach when each pixel is sampled. Moreover, for this reason, the level of under-sampling cannot be trivially made dynamic the same way that super-sampling can. Also, for memory access efficiency, the sampled value should be stored only once, not for every pixel, but then rendered for every pixel.

Edge detection: For adaptive super-sampling, instead of the statistical analysis, an edge detection method could be used, for example Sobel. The pixels marked as *edge* would then be marked *chaotic* and supersampled. The main challenge of this approach regarding CUDA is that it depends on values of pixels

in the neighborhood. This leads to the need of introducing a threading model, which, beside others, can dramatically slow the program if implemented wrongly. Another approach would be a simple sampling of the whole image, followed by an edge detection of the whole image, followed by potential supersampling of some regions. Straightforward implementation of such a method would not use resources in CUDA-compliant. It would introduce a great memory pressure, memory being the bottleneck of most CUDA's performance. Also, it would not to easy to make such a process adaptive, i.e. deciding after i iterations whether to continue with $i+1$.

Successive interations: Method introduced primarily for computing the Mandelbrot set [57] , where a sample is computed with a lower number of iterations first, and then the number is successively increased. Usually, the method introduces many fragments during the progressive phase. On the other hand, it produces snapshots very efficiently.

Let us now review how the proposed methods actually work when implemented.

5. Experiments

We have implemented the ideas described in previous sections and created a program that we named *chaos-ultra*[1]. The implementation of *chaos-ultra* is documented in detail in Appendix A.

Having a working implementation, let us now use *chaos-ultra* and measure how fast it can render and how much have the proposed real-time methods helped.

To show that we have fulfilled our goal, we are going to perform following comparisons: (1) Comparing *chaos-ultra* and *XaoS*. (2) Comparing the progressive phase of *chaos-ultra* with and without adaptive supersampling. (3) Comparing the interactive phase of *chaos-ultra* with and without the real-time toolset.

On image quality

All images from this Chapter are included as Attachment D.3.4 in their full resolution. Author of all images in this Chapter is the author of the thesis.

5.1 Measurements methodology

Now that we know what is to be measured, let us discuss the methodology.

5.1.1 Hardware

All results are hardware-dependent. We use hardware compliant with the thesis' goals. The machine that was used for measurements in this chapter has following equipment: Intel Core i7-4771, with 4 cores, 3.5 GHz; 8 GB RAM; nvidia GeForce GTX 1060 6GB; running on Windows 10.

5.1.2 Comparing specific chaotic functions

We need to evaluate general performance of a generic renderer of chaotic functions. This is hard, as every chaotic function may behave differently, moreover differently at different locations and zoom levels. To approximate the comparison, we select several specific chaotic functions and compare them, selecting several locations and zoom levels for each one of them.

For comparison, we choose the three standard fractals that we have implemented: the Mandelbrot set, Julia fractal and Newton fractal. The parameters that change the fractal have to be chosen independently, by someone else than the author of the experiment.

For the Mandelbrot set, we choose values proposed by Alan Dewar in [24] . For the Julia sets, the performance-affecting parameter is the c value; we choose its values based on [17] . For the Newton fractal, the performance-affecting parameter is the inducing polynomial, which we choose based on [21] .

[1] The name is a word play, referencing to real time fractal renderer *XaoS* and high quality fractal renderer *ultrafractal*, which both have been inspiration for the thesis.

5.1.3 Choosing comparison method

Our task is to compare *chaos-ultra* and *XaoS* resp. two versions of *chaos-ultra*. Comparing real-time applications that react on user input is not possible with single images. Video comparison will be needed.

There are basically two approaches for determining video quality: (1) Objective methods, introducing algorithms and statistical models [49] . And (2) subjective methods, usually involving perception studies and user's impressions, with many guidelines and methodologies available [69] .

The objective approach is impossible for comparing output of two different fractal renderers and thus we use a subjective method for that.

Regarding comparison of chaos-ultra with itself, we objectively compare achieved FPS and level of supersampling and then introduce a very simple subjective comparison method.

On video quality

We have decided to use video-based comparison. The video is going to have the target resolution of the thesis, 1280 × 720, and 30 FPS. For video encoding, either lossy or lossless codec can be used. Videos encoded using lossless codecs are extremely large. Lossy codecs introduce quality loss.

As we do not want to introduce attachments with sizes in many gigabytes, we use a lossy codec. We decide for the lossy codec h264, and use highest quality possible. We do this in a hope that even with the encoding artifacts, the quality of our rendering will still be sufficiently visible.

The videos that were used for comparison are included as Attachment D.3.1.

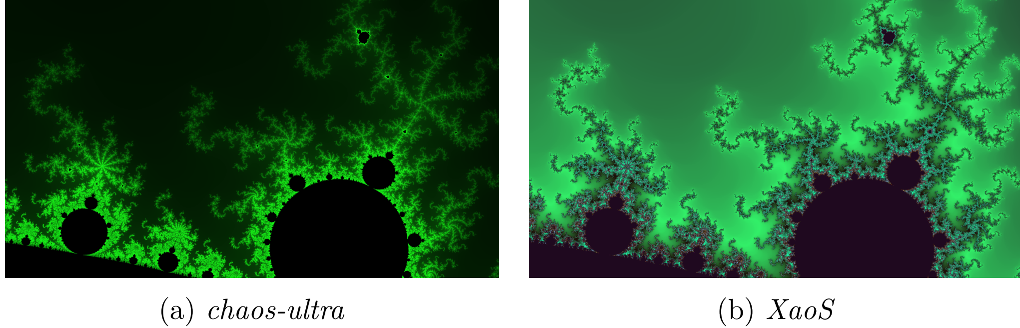
5.2 Comparing *XaoS* and *chaos-ultra*

Making *chaos-ultra* producing more detailed images than *XaoS* is one of the goals of the thesis. Although we have concluded that we need to use a subjective method for comparing the two, we want to stay as independent as possible. Hence, we have organized a simple perception study with independent participants and use its results as our metric.

Study details

The study has taken place on April 26, 2019 in a high school in Prague (Gymnázium nad Štolou) during a student conference organized by the school's students (called *Štolní noc*). 9 participants have been shown videos of 5 different videos produced by *XaoS* and *chaos-ultra* and have been asked to rate them. During the projections, the participants have been asked to focus at the cursor.

The participants have been chosen randomly from conference participants. The study was blind — the participant didn't know which program is which. We made comparison on the Mandelbrot set only; its points have been chosen by external source, see Section 5.1. Participants gave marks 1–5 (lowest–top) in two categories: smoothness and level of detail. We have used similar color schemes



(a) *chaos-ultra* (b) *XaoS*

Figure 5.1: Illustrative screenshots from the perception study, study example 3.

in both renderers.[2]

Study results

Data have been collected on response sheets and evaluated. For *XaoS*, the average smoothness is 2.82 with variance over examples 0.6. The average mark of level of detail is 3.24 with variance over examples 0.05. For *chaos-ultra*, the average smoothness is 3.58, variance being 0.72 and the average level of detail mark us 4.03, with variance being 0.11. All values have been round to two decimal points. We remind that 1 means worst, 5 means best.

Table with response data is included as Attachment D.3.2. Used videos are included as an Attachment D.3.1. For illustration, we include screenshots from the study, see Figure 5.1.

chaos-ultra has surpassed *XaoS* in both categories, by a statistically significant value. We are aware that the performed study was very small and simple and that the number of respondents is low and the Law of large numbers cannot be applied. To prevent any disputes, videos from the study are included and the kind reader may watch study's videos themselves or compare the programs by using them. We conclude that *chaos-ultra* has better rendering results than *XaoS*.

5.3 Adaptive supersampling

Sample count visualization

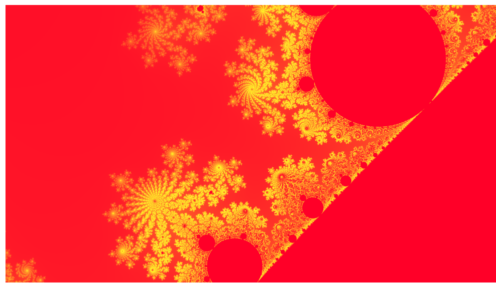
The results achieved by the the method are greatly visualized by Figures 5.2, 5.3, 5.4, and 5.5. The visualization shows relative number of samples taken at each pixel, black representing zero samples, white representing sample count equal to maxSuperSampling.

Technical details: The pictures were taken by *chaos-ultra*, during the progressive phase, by choosing the *visualize sample count* option in the *research options* toolbox, with *automatic quality* option disabled, and maxSuperSampling manually set.

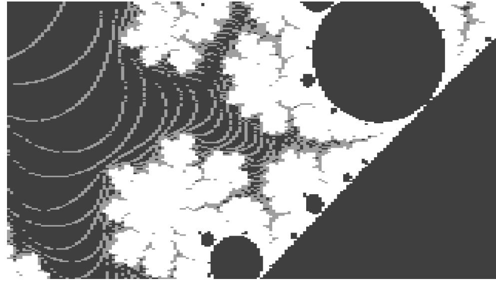
The granularity of the method may seem *too coarse* at the first look — the visualization seems to consist of big black-white rectangles, not granular up to

[2]In *XaoS*, we have used following settings: algorithm 1, seed 30230, shift 0. For *chaos-ultra*, we used the green palette.

Figure 5.2: Adaptive supersampling on Mandelbrot example 4

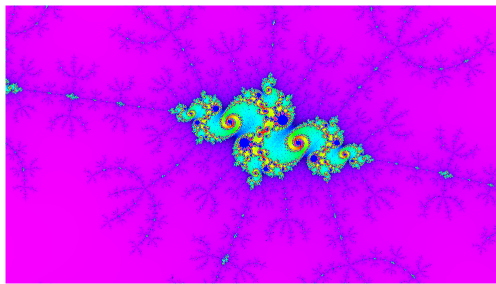


(a) Red color palette.

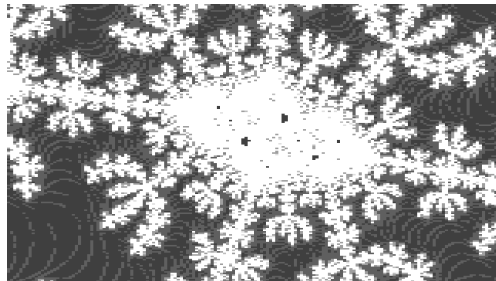


(b) Relative sample count with maxSuperSampling 8.

Figure 5.3: Adaptive supersampling on Mandelbrot example 5



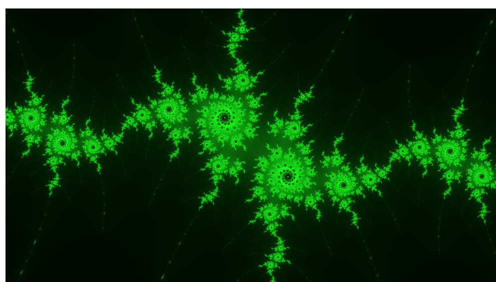
(a) Default color palette.



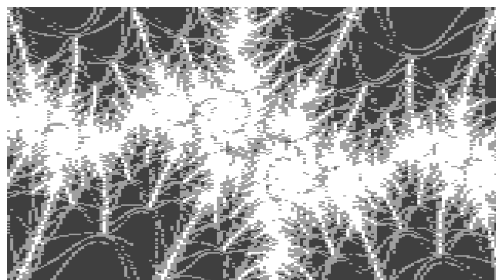
(b) Relative sample count with maxSuperSampling 8.

pixel-level. However, this is not a property of our method but property of CUDA. At any moment, 32 pixels are sampled together, within the so called *warp*. If in any of the pixels, the method decides that the pixel is chaotic, all other 31 pixels can be sampled too, in parallel, without introducing any more overhead (This behavior is discussed in detail in Section A.2.5.). For illustration, see Figure 5.6, where we have turned this functionality on and off. The rendering time of both images is exactly the same.

Figure 5.4: Adaptive supersampling on Julia example 4

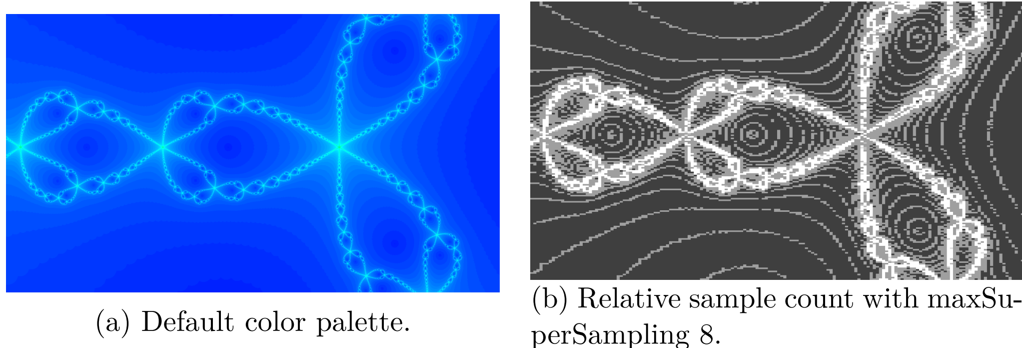


(a) Green color palette.



(b) Relative sample count with maxSuperSampling 8.

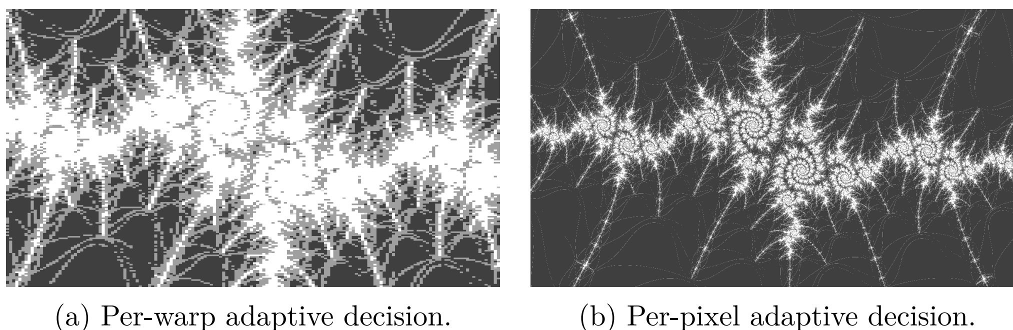
Figure 5.5: Adaptive supersampling on Newton iterations



(a) Default color palette.

(b) Relative sample count with maxSuperSampling 8.

Figure 5.6: Granularity of adaptive supersampling, illustrated on Julia example 4. Both images have the same render time. maxSuperSampling is 8.



(a) Per-warp adaptive decision.

(b) Per-pixel adaptive decision.

Rendering time comparison

Another way of evaluating the method is comparing the rendering time with and without the method. We have performed this comparison with 12 examples. See Table 5.1 and Figure 5.7 for results. The average rendering time of each example has been taken as average of 10 independent renderings.

Technical details: The measurements were taken by *chaos-ultra*, during the progressive phase, by switching the *adaptive supersampling* option in the *research options* toolbox, with *automatic quality* option disabled, and maxSuperSampling manually set. The rendering time has been read from log. Logs were enabled by setting program's property `renderingLogging` to `true`.

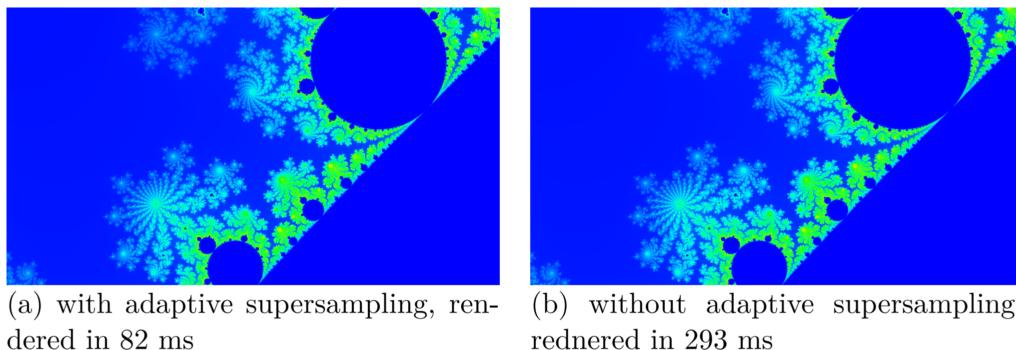
example:	M1	M2	M3	M4	M5
with	187,8	457,8	41,2	85,2	1381,6
without	192,6	487	63,8	191,4	1562,2
speedup in %	2	6	35	55	12

example:	J1	J2	J3	J4	N1	N2	N3
with	171,6	25	18,4	39,6	72,4	44	110,4
without	187	31,2	39,8	44,8	229,2	126,8	308
speedup in %	8	20	54	12	68	65	64

Table 5.1: Average rendering time (in ms) with and without adaptive supersampling. The speedup has been rounded.

The most significant speedup was achieved on the Newton fractal. This is no surprise, Newton has big single-color areas that need no supersampling.

Figure 5.7: Comparison of Mandelbrot example 4 with and without adaptive supersampling, with maxSuperSampling set to 32.



(a) with adaptive supersampling, rendered in 82 ms

(b) without adaptive supersampling, rednered in 293 ms

To demonstrate that the image quality was affected minimally, we compare Mandelbrot example 4, the most sped-up image of Mandelbrots, rendered with and without adaptive supersampling. See the comparison in Figure 5.7. From the comparison we conclude that quality was not affected by the method.

Conclusion

In Table 5.1, we have computed for each image the speedup of adaptive supersampling. We do not introduce the total speedup of the method, because such value does not make sense. The speedup of adaptive supersampling strongly depends on the structure of the chaotic function.

The visualized sample count proves that the method is able to find chaotic regions. The rendering time measurements prove that the method introduces a speedup without a noticeable loss of quality. We conclude that the method works.

In Section 4.1, in the subsection on statistical significance, we have claimed to experimentally determine the initial number of samples per pixel, k. We conclude that the implemented value $k = 2$ suffices.

5.4 The real-time heuristic toolset

Regarding our real-time heuristic toolset, there is no specific objective to defend, nor any specific metric to compute. Our goal was to decrease rendering time with 'nonsignificant' loss of quality, which is vague.

To demonstrate that our toolset allows this, we have created recordings of zooming at the Mandelbrot set with and without the toolset active. The recordings are included in Attachment D.3.3. Also, the kind reader can try this comparison themselves by modifying *research options* in the GUI of *chaos-ultra*, disabling any of those functionalities: automatic quality, sample reuse, foveated rendering, and adaptive supersampling.

Concluding from the recordings, the benefit of our method is undisputed. The improvements are most significant when after about 10 seconds of zooming in, the

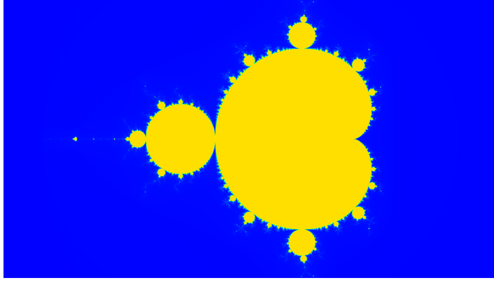
program switches to computation in double-precision. When rendering in double-precision, the naïve approach hardly computes one frame per second, while our methods maintains 30 FPS.

5.5 Rendering review

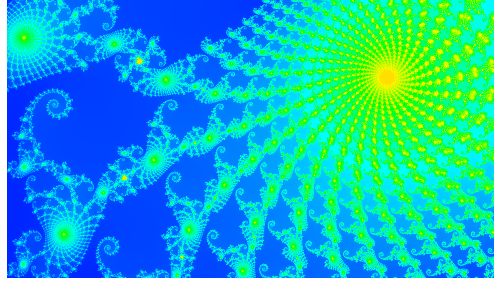
In the last two sections, we have used videos for demonstrations and asked the reader to try the experiments themselves in *chaos-ultra*.

To illustrate what *chaos-ultra* can do within the text of the thesis, without any video recordings and program use, we hereby list examples of images rendered by it: see Figure 5.8.

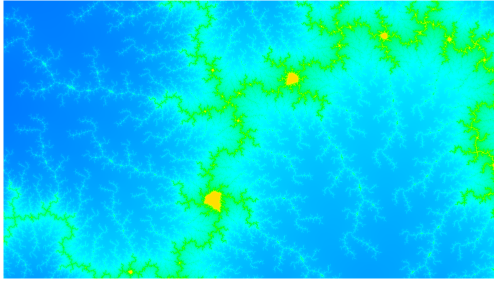
The original resolution is HD (1280×720), color palette is the default one for all images. All images have been saved after progressive rendering has finished, meaning it took approximately 500 ms to render each image. Other rendering parameters are listed in Attachment D.3.4. These examples correspond to the ones available from the GUI of *chaos-ultra* and to the ones from Section 5.1.



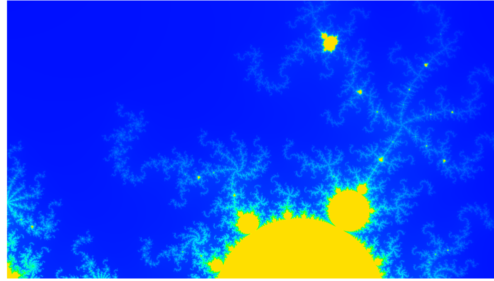
(a) Example M1, a view at the full Mandelbrot set.



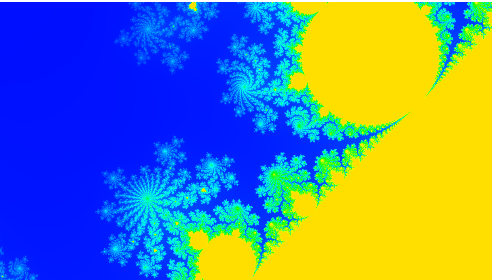
(b) Example M2, Seahorse valley, Mandelbrot set.



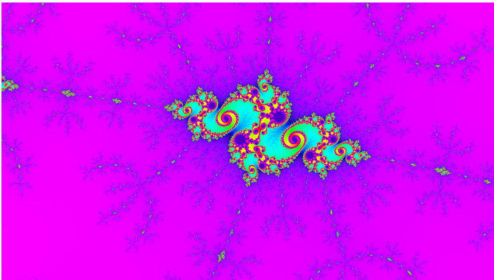
(c) Example M3, Lightnings, Mandelbrot set.



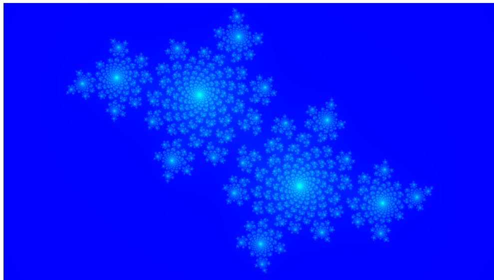
(d) Example M4, Multimandel, Mandelbrot set.



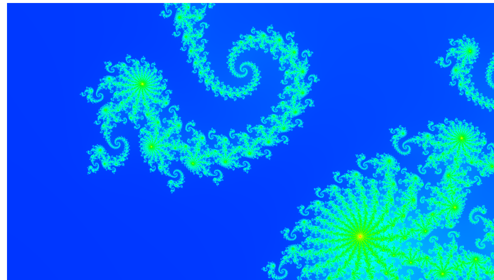
(e) Example M5, Flowers, Mandelbrot set.



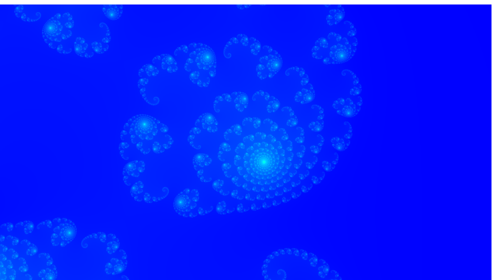
(f) Example M6, Inner Julia, Mandelbrot set.



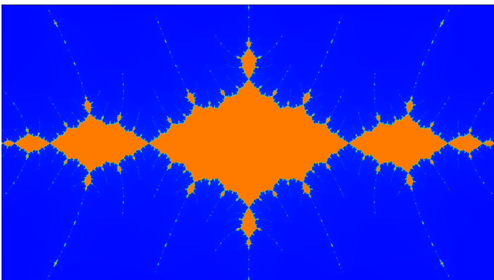
(g) Example J1, Crystals, Julia set $0.4 + 0.6i$.



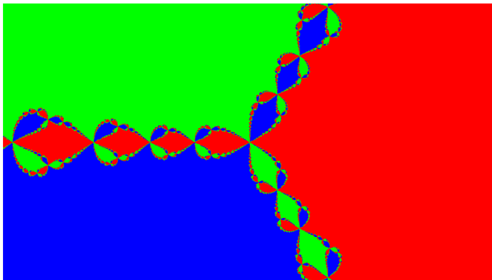
(h) Example J2, Snakes, Julia set $-0.8 + 0.156i$.



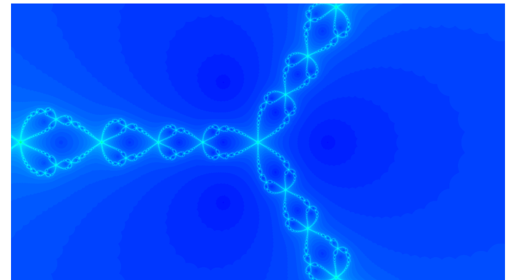
(i) Example J3, Jellyfish, Julia set $0.285 + 0.01i$.



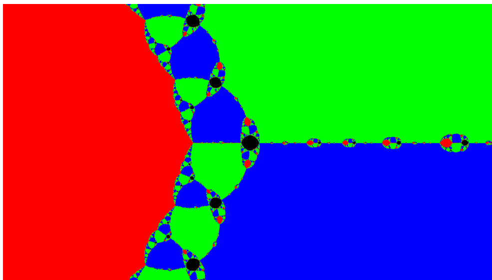
(j) Example J4, Anntenas, Julia set -1.77578.



(k) Example N1, Newton fractal for $x^3 - 1$.



(l) Example N2, same fractal as N1, colored by convergence speed.



(m) Example N3, Newton fractal for $x^3 - 2x + 2$.

Figure 5.8: Examples of *chaos-ultra* output.

Conclusion

We have implemented the real time renderer of chaotic functions, *chaos-ultra*. We have also proposed heuristic-based methods for real-time rendering, implemented them and measured the outcome. Now we are turning back to the the goals of the thesis and evaluate if they have been fulfilled.

Evaluation of the goals

The main goal of the thesis was to implement a generic real-time renderer of chaotic functions. We have implemented a renderer that is interactive, real-time, and supports chaotic functions in general. Hence **we have fulfilled the main goal**.

Now we review the specific goals in detail:

1. **Genericity**: The render supports chaotic functions $\mathbb{C} \to \mathbb{R}$ in general. We have provided implementations of the Mandelbrot, Julia, and Newton fractals, as required. The user can add an arbitrary new chaotic function. This can be done without modifying current code, by adding new files to appropriate modules. We have demonstrated this by adding two more fractals: *goc* and *test*. By all this, we have fulfilled the goal.
2. **PC- & GPU-based**: *chaos-ultra* runs on PCs and the rendering backend runs on CUDA technology, fulfilling this goal. We have performed our measurements on a hardware that matches the hardware required by the goals.
3. **High quality in real time**: Concluding from the section-conclusions in Chapter 5, we have achieved high resolution, high level of detail and smooth user experience at the same time. To be more specific, we now take a more detailed look at this goal:
   - 3.1. **Resolution**: The resolution of the program is scalable, and is adjustable to window size. *chaos-ultra* runs even in Full HD, i.e. 1920×1080 resolution, exactly as required by the goals.
   - 3.2. **Detail level**: The results of the perception study, and our subjective comparison, lead to the conclusion that the level of detail of fractals renderer by *chaos-ultra* is "good enough", fulfilling this vague goal.
   - 3.3. **Rendering performance**: We have introduced a method for achieving constant FPS. The target FPS can programmatically be set. On all tested fractal, the method works as expected when zooming in and moving around, for parameter values as discussed below.

   Our formal requirement was, for the defined hardware, to consistently achieve following goals at the same time: HD resolution, i.e. 1280×720 px, 30 FPS and level of detail higher than when using *XaoS* renderer.
   In HD resolution, we achieve the target FPS most of the time on all tested fractals, *maxIter* values being between 400 and 1500 and zoom levels order of magnitude being above -7. The perception study described in Section 5.2 concluded that the level of detail is higher than that of *XaoS*.

However, for very complex fractals at deep zooms (with order of magnitude lower than -7), with a high *maxIter* value (around 3000) and when computing in double precision, the method is not more able to adjust the quality during the interactive phase, and the target FPS is not achieved and decreases to values as 15 or 10 FPS, leading to a laggy user experience. Example of such behavior is the Example number 5 in the Mandelbrot set.[3]
As stated in the goal, the problem of fractal rendering inherently gets more complex when zooming in, and a slowdown is expected and allowed. We believe that the limit of the method, lying in *maxIter* above 3000 and zoom levels with order of magnitude under -8, is adequate.
Based on this discussion, we conclude that the formal requirement has been fulfilled and thus the goal **High quality in real time** achieved.

4. **User experience**: We have introduced a standard graphical user interface that allows parameter tweaking using text fields and buttons. The fractal is visualized on a canvas that allows user interaction in a common way, including zoom in/out and move around. We have thus fulfilled the goal.
5. **Cross-platform**: *chaos-ultra* is written in Java, running on both Windows and Linux, as required. The CUDA backend runs on both Windows and Linux too.
6. **Extensibility**: We have implemented *chaos-ultra* in Java, a standard OOP language. We have followed Java's style and best practices. The CUDA backend is implemented in CUDA C/C++ and the implementation follows its guidelines.
   We have introduced a module-based software architecture and several useful design patterns, and have discussed and documented our architecture decisions. We utilize OOP principles, relying on interfaces and inheritance. Our classes follow the single responsibility principle. The addition of a new fractal follows the Open/Close principle.
   The code is documented, regarding both method headers and algorithmical parts of the code. All the code uses a consistent and expressive naming convention.
   For these reasons, we conclude that our code is readable and extensible, fulfilling this goal.
7. **Color palettes**: As required, the program allows changing the style of coloring the fractal, by loading user defined color palettes from an image file.

In addition to software-oriented goals, we have also hoped to better understand the perks of sampling a chaotic function. We were aiming at providing a relevant discussion of the problem and at proposing a few possible solution methods. Our goal was NOT to try all discussed and proposed methods in their full potential.

Let us have a look at each of the area of focus:

1. **Progressive visualization**: We have inctroduced the topic, proposed an algorithm, implemented it and concluded that it works as expected.
2. **Adaptive sampling**: We have developed a statistical-based heuristics for detecting chaotic regions of a chaotic function. We have also implemented a

[3]See example 5 in Attachment D.3.1 for details, or Mandelbrot Example 5 in the GUI.

method that adaptively evaluates more samples at those regions. We have shown that it works for all the tested fractals.

3. **Foveated rendering**: We have briefly introduced relevant work in the field of foveated rendering and tried to fit the known methods to our toolchain. Assuming that the user focuses at the cursor, we have proposed and implemented a perceptually plausible heuristic for sample-count reduction in peripheral parts of the image.
4. **Sample reuse**: We have described sample reuse techniques, proposed possible solutions and implemented a proof-of-concept that significantly reduces number of samples needed to compute, although it introduces visual artifacts.

During the thesis, we have come to a better understanding of chaotic functions and the challenges of sampling them. We have discussed the problem, proposed relevant solution methods and partially implemented them.

As expected by the goals, the discussion did not go very deep into the topics. However, if one day, we were to rigorously research the general problems that come with chaotic functions integration, this discussion could be our stepping stone.

From this, we conclude that we have fulfilled the research-focused goal.

We have fulfilled all of the thesis' goals. We now open the realm of chaotic functions to the public.

Known limitations

After having concluded that our goals have been fulfilled and that the program works, let us review what are known limitations of our methods or of *chaos-ultra*.

Limited foveated rendering

Although the method works, which has been proven by various tests, in the final real-time-rendering toolset, foveated rendering is used only partially. We use it for determining additional number of samples to take in the foveal area when zooming in while reusing samples. It is not used for determining number of samples to take in the non-foveal area; this number is always 0 (thus only the reused value is displayed).

The reason for this is that combination of sample reuse and foveated rendering introduced many circular-shaped visual noise during zooming in, hence we decided to disable the combination. Finding a way how to apply foveated rendering in its full potential together with sample reuse is a candidate for future work.

Julia zoom bug

There is a bug in the Julia fractal. Rendering at zoom levels with order of magnitude between -4 and -6 produces highly aliased outputs, as if the floating-point precision were at its limits. After zoom magnitude -7 is surpassed and the program switches to rendering in double-precision, the output is OK again.

This effect has not been observed with other fractals. This effect has appeared during development of *chaos-ultra*, and was for sure not present in earlier phases of the project. However, we were not able to find the cause yet, and we leave it for future work.

Rendering in the UI thread

The UI thread, which launches CUDA kernels, waits for the CUDA-computation to finish in a blocking way, technically causing rendering to be performed in the UI thread. This is a well-known anti-pattern that we know about.

This causes a bug experienced by the users during the progressive phase. When user interaction stops and progressive rendering has started, the program may be unresponsive for hundreds of milliseconds — because the UI event loop thread is blocked by waiting hundreds of milliseconds for CUDA to finish the high-quality rendering.

The bug should be fixed in a next release, by introducing a non-blocking waiting scheme.

Limitations of real-time performance

Even with our real-time toolset, the real-time performance has its limits, due to the infinitely complex nature of the Escape-time fractals.

If desired *maxIter* parameter is too high and the zoom too detailed, the complexity of computing even very few samples at the users' focus is so big, that our method does not manage to achieve real-time and the FPS drops down.

Future work

Apart from fixing the bugs introduced in the previous section, what are other possible future assignments?

Machine learning

A logical improvement of our statistical-based adaptive supersampling method is introducing a machine learning method instead of the statistical analysis of the data, for example a deep learning method.

More advanced sample distribution

A more advanced sample distribution scheme than uniform distribution could improve the results of adaptive supersampling. A quasi-random distribution is a good candidate.

More advanced filtering method

Our implementation uses bilinear texture filtering for texture scaling. We have also implemented bicubic filtering. A more advanced filtering method that works better with chaotic function could be introduced and tested, for example Lanczos.

Software improvements

There are many possible improvements of *chaos-ultra*, many inspired by *XaoS* and its large variety of fractal-rendering-related features. The examples include, but are not limited to: more advanced coloring schemes (based on other inputs than the chaotic value), more rendering-plane options (for example the Riemann sphere), automatic zooming at interesting points, video capture or more advanced image-exporting methods.

As *chaos-ultra* is licensed as open source, the scientific community is free to add any functionality in case it is desired, and to try variety of other possible rendering-improvement methods.

APPENDICES

A. Development documentation

After having discussed the problem theoretically, let us have a look at how the ideas from Chapters 3 and 4 are implemented.

We introduce the program architecture from a broad perspective. We introduce the main five modules: CUDA fractal interface, CUDA backend, Java-Cuda mapper, Java renderer and GUI. Then, we describe each module in more detail, with focus on module's structure, program flow, and, for Java-based modules, OOP design.

Finally, we conclude that we have implemented a functional full-sized Java GUI application backed by CUDA-enhanced backend. We call this application *chaos-ultra*[1].

At the very end of this chapter, we introduce documentation for further development, including technical requirements, compilation and troubleshooting.

But first, let us leave a remark about notation, and define terms that are related for the whole chapter.

Notation remark

This thesis is about chaotic functions in general. Within this chapter, we use the term *fractal* where a more general term *chaotic function* could be used. This is for two reasons: (1) First, the focus of the thesis is especially on fractals. (2) Second, more importantly, during implementation, it is much more convenient to name modules, classes and objects *fractal-Something*[2]. It is broadly understood, as opposed to longer and more vague *chaoticFunction-Something*, which also could be confused with language's *method* in some contexts.

A.0.1 CUDA terminology

For the purpose of explaining all the implemented concepts, we need to use some CUDA-specific terminology throughout this chapter. If you are not familiar with it, we recommend a quick look at the Programming Model section at nvidia's CUDA Programming guide. We briefly summarize the most common terms from the guide in the following paragraph.

- **device code/memory** — code/memory that runs/resides on the GPU
- **host code/memory** — code that runs on the CPU / standard RAM
- **device function** — A function that runs on the device, marked `__device__` in CUDA C/C++.
- **kernel** — Device function that may be called only from host. When called, it is executed N times in parallel by N different CUDA threads.
- **cuda thread** — One instance of the computation process, with own instruction pointer and register state.
- **thread block** — A group of threads that can be executed serially OR in parallel.

[1] The name is a word play, referencing to real time fractal renderer *XaoS* and high quality fractal renderer *ultrafractal*, which both have been inspiration for the thesis.

[2] e.g. fractalRenderer, fractalModule, fractalProvider etc.

- **block dimension and thread index** — For convenience, threads can be arranged into one- two- or three-dimensional blocks and indexed accordingly.
- **grid** — For convenience, blocks are arranged into one- two- or three-dimensional grids. The number of thread blocks in a grid is usually dictated by the size of the data being processed.
- **warp** — 32 parallel threads that are physically executed simultaneously (on the same GPU core).
- **module** — Set of device classes and functions, that can be loaded and later called.

CUDA introduces two APIs: *CUDA Driver API* and *CUDA Runtime API*. As officially stated by a nvidia developer "The two APIs exist largely for historical reasons. ... The runtime API is, for the most part, a very thin wrapper around the driver API" [22]. The two APIs may be used interchangeably. The Runtime API is easier to use, but the Driver API allows, beside others, module loading. As we need the module loading functionality, we use both APIs in our code.

A.0.2 Fractal representation

The rendering process is parametrized by a rectangular segment of the complex plane to be visualized. We need to represent this segment. There are two possible approaches and both are used in the application.

(1) The straightforward approach is to store two corners of the rectangle (their x, y coordinates). We choose to store the left-bottom and right-top corners. Advantage of this approach is simplicity for the programmer. This approach is used by the CUDA backend and by most of the Java modules.

(2) The more common approach among the fractals-concerned scientific community is to store the center of the segment (x, y coordinates) and one-dimensional *zoom*. Zoom specifies the distance from the center to the side of the rectangle (either in x or in y direction). One information is missing in this representation: distance to the other side (in y resp. x direction). It can be computed from width-to-height ratio of the rectangle, which makes this representation window-size independent. This is probably why it is popular within the community. We use this representation to communicate with the user.

The class `Model` from Java package `rendering` is responsible for conversion between the two representations.

A.1 Program architecture

chaos-ultra consists of five main modules: GUI, rendering, Java-Cuda binding, main backend module, cuda fractal. The first three are implemented in Java, the last two are implemented in CUDA C/C++ and run on the GPU. The modules are connected in a chain, each of the module relying on its child module and providing API for its parent module. See Figure A.1 for illustration.

The Java part is build using maven, and has been developed in IntelliJ Idea. The Cuda part is build using a custom script and compiled with nvidia's compiler `nvcc`. See Section A.2 for more details about cuda-module build process.

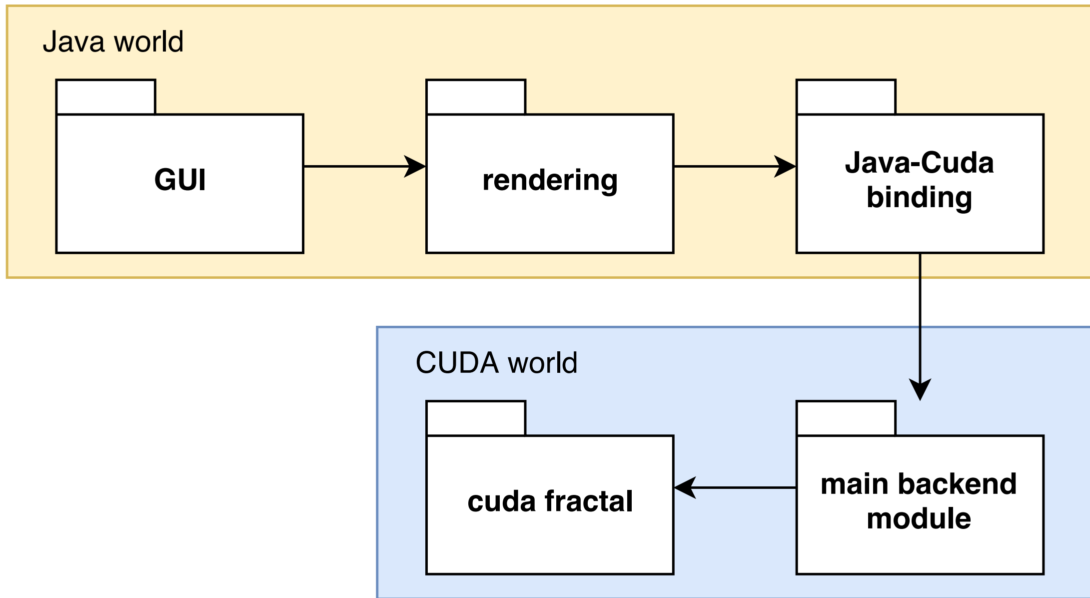


Figure A.1: Program architecture

Let us now see each of the modules separately. Since the *cuda fractal* module is just one interface, we include its documentation together with the rest of the Section A.2.

A.2 CUDA backend

The application is backed by a generic fractal sampler written in CUDA C/C++. It is generic in the sense of concrete fractal and floating point precision. It contains implementation of the logic that has to be performed in parallel. Most of the real-time methods from the previous chapter are implemented here. The module depends on *fractal* interface, representing specific fractal to be rendered. Module's public API are the kernels in the CUDA code. They are expected to be called from the Java application.

CUDA version

To allow for further reuse and maintainability, we support the latest CUDA version, 9. For biggest possible backwards-compatibility, we are targeting the lowest possible compute capability for this CUDA version, which is compute capability 3.0, as explained in [22] and [9]. As a consequence, a GPU with CUDA version 6 or higher is needed, because lower CUDA versions do not support compute capability 3.0.

A.2.1 Data structures

Throughout the module, floating point numbers are represented with the `Real` C++ template. The template is then instantiated to `float` and `double`.

Points and complex numbers

Most of the time, we will need to work with pixels, points in the 2D real plane and complex numbers, and cast between them. For this, we introduce the Point<T> generic type. The arithmetic operations on the it are performed component-wise, as in classical linear algebra.

To represent complex numbers, we use Point<Real>. This has proven to simplify the implementation. Throughout the generic renderer, no complex algebra is needed, and thus this approach suffices. However, for fractal implementation, complex algebra may be useful. In those cases, CUDA's thrust::complex can be used.

For representing a complex plane segment (the region of interest), generic Rectangle<T> class is introduced, containing two points that define the segment.

Pitched arrays

In accordance with Chapter 3, chaotic values are taken and then stored in memory for further coloring or sample reuse. For this, we use CUDA 2D pitched array.

The *pitch* is introduced by CUDA for effective memory alignment. Allocation of the pitched array is done by the CUDA runtime, not by the compiler. Thus, to access data in a pitched array, the standard C++ accessors, [], may not be used — we introduce special methods for wrapping this functionality, for example getPtrToPixelInfo.

Sampling information and textures

For storing the sampling information, we use the pixel_info_t data structure.

For storing the output image and color palettes, OpenGL textures are used. For accessing textures from the __device__ code, CUDA provides *surface* adapter, which itself needs a *cuda resource* adapter.

Although color palettes are 1D, introducing another *surfaces* and *cuda resources* would increase code complexity, and thus we represent palettes as 2D textures with height 1.

A.2.2 Fractal interface

For computing a sample of a fractal (the chaotic value), the module relies on the *fractal* interface. It is represented by header file fractal.cuh, which is well-documented and telling.

It provides three methods: computeFractal, colorize, and debug. The names of the methods are expressive.

To use the cuda-backend module, an implementation of the *fractal* interface needs to be provided. Compilation and linking is described at the end of this section.

A.2.3 Fractal specific parameters

Many fractals need a specific parameters, for example Julia fractal's c value or the polynomial coefficients of the Newton fractal.

We implement this need using CUDA's `__constant__` type. Values declared as `__constant__` can be set from the host code and stay immutable during a kernel launch. They are cached, so there should be no performance overhead to passing the values as kernels' parameters.

It is the responsibility of the caller to initialize the values. We discuss in Section A.3 how this is done.

A.2.4 Specific fractals implementation

Following the thesis' goals, we have implemented the three example fractals: Mandelbrot, Julia and Newton.

The implementation of Mandelbrot and Julia fractals is straightforward. We only note that to gain maximal performance of Mandelbrot and Julia, instead of using the `thrust:complex` type, we have written complex multiplication ourselves, using primitive types. We have also used the standard trick of comparing $z^2 < 4$ rather than root-containing $\mathrm{abs}(z) < 2$.

Newton fractal

Regarding the Newton fractal, we provide several implementations.

`newtonWired` supports only the $x^3 - 1$ polynomial, the formula is "wired" in the source code. `newtonGeneric` allows for any polynomial of 3rd order to be set using fractal-specific parameters. We have implemented both the wired and the generic version to be able to compare the overhead of evaluating the polynomial as generic.[3]

The coloring scheme of those two fractals does not rely on the given palette, for aesthetic reasons. However, the implementation exists, and can be found in `newton_generic.cu` file. The users are free to experiment with it.

`newtonIterations` has same implementation of the `computeFractal` method as the generic version, but it provides different coloring scheme.

We do not provide the most generic version that would support polynomial of nth order, n being a parameter. This is because CUDA does not allow for dynamic allocation of the constant memory, and thus the number of fractal's parameters may not change after compilation.

There is a possible work around for this, in setting the value to, say, 255 and then supporting polynomials up to that value. We leave this for future work.

Other fractals

We have implemented two other fractals that are included as part of *chaos-ultra*. The `test` "fractal" is a function with highly-regular and human-predictable structure and as such, it is meant for testing and debugging purposes.

The `goc` fractal is dedicated to my best friend who has discovered it. It is part of *chaos-ultra* as an acknowledgement of his tremendous help with *chaos-ultra* debugging.

[3] To maintain the thesis' scope, we do not discuss this comparison in Chapter 5. During the implementation, we have estimated it to be about 10%.

A.2.5 Implementing the real-time heuristics

Most of the implementation of the methods described in Section 4 is done in this module. We briefly go through relevant implementation details.

There are two main rendering methods: `fractalRenderMain` and `fractalRenderAdvanced`. The `main` method implements rendering for the static phase, the `advanced` method implements rendering for the progressive phase, and is thus capable of sample reuse, foveated rendering, etc.

Threading model

Following CUDA best practices described in [22], we use one thread for each pixel that is to be rendered.

We don't use one thread per sample, because thread creation is not dynamic in CUDA and it would not allow us to implement adaptive supersampling.

Adaptive super-sampling

The adaptive process is implemented by method `sampleTheFractal`. The implementation is compliant with Algorithm 5.

There is one significant implementation detail that positively affects rendering quality: Per-warp decision. The adaptive decision is made per-warp, so that if any of the pixels in the warp is marked chaotic, all other pixels in the warp are sampled too. This introduces no performance overhead, because all the other threads in the warp must perform same instructions (or wait) anyway.

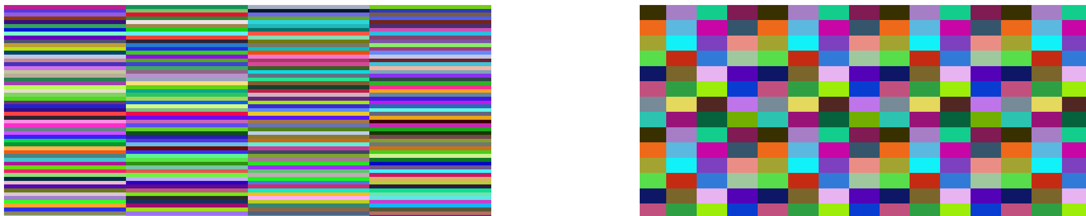


Figure A.2: A texture rendered on CUDA, with all pixels within a warp colorized with the same color. Scaled up for clarity. Left: canonical cuda warp, right: our rectangular warp.

Rectangular warp

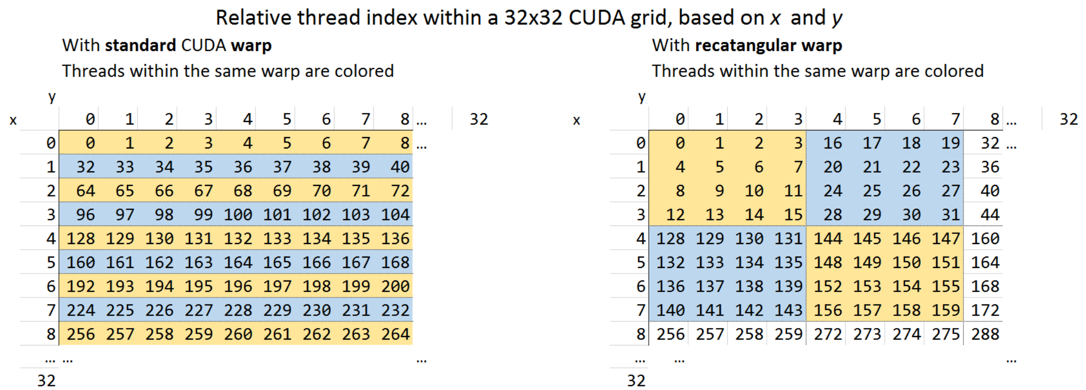
Regarding the warp, we use one more optimizations that positively affects rendering time: rectangular warp.

We do not organize pixels to warps in the intuitive way proposed by CUDA. If we did, then each warp would be responsible for sampling pixels that are organized within the pixel grid as 32 successive pixels in a row. See the left picture in Figure A.2 for visualization.

Instead, we change the mapping of CUDA threads to image-pixels in a way described by Figure A.3. This causes one warp to compute pixels that are organized as 4×8 rectangle within the pixel grid. See Figure A.2 for visualization.

Chaotic areas are expected to cover a rectangular area more often than to cover a single 32 pixels long line. Using this technique, we hope to produce more

warps in which none of the pixels is chaotic and more warps in which many pixels are chaotic.


Relative thread index within a 32x32 CUDA grid, based on *x* and *y*

With **standard** CUDA **warp**
Threads within the same warp are colored

x \ y	0	1	2	3	4	5	6	7	8	...	32
0	0	1	2	3	4	5	6	7	8	...	
1	32	33	34	35	36	37	38	39	40		
2	64	65	66	67	68	69	70	71	72		
3	96	97	98	99	100	101	102	103	104		
4	128	129	130	131	132	133	134	135	136		
5	160	161	162	163	164	165	166	167	168		
6	192	193	194	195	196	197	198	199	200		
7	224	225	226	227	228	229	230	231	232		
8	256	257	258	259	260	261	262	263	264		
...	...									...	
32											

With **recatangular warp**
Threads within the same warp are colored

x \ y	0	1	2	3	4	5	6	7	8	...	32
0	0	1	2	3	16	17	18	19	32	...	
1	4	5	6	7	20	21	22	23	36		
2	8	9	10	11	24	25	26	27	40		
3	12	13	14	15	28	29	30	31	44		
4	128	129	130	131	144	145	146	147	160		
5	132	133	134	135	148	149	150	151	164		
6	136	137	138	139	152	153	154	155	168		
7	140	141	142	143	156	157	158	159	172		
8	256	257	258	259	272	273	274	275	288		
...	...									...	
32											

Figure A.3: Mapping of image indexes to warp index and vice-versa.

Sample reuse

The implementation follows the ideas from Chapter 4.

Sample reprojection follows the idea of Figure 4.3 and is implemented by the `getWarpingOriginOfSampleReuse` method.

For reading samples from the warping origin, we provide implementations of bilinear and bicubic filtering, in the `readFromArrayUsingFiltering` method.

Sample composition is performed by `fractalRenderAdvanced`, with regard to data weight. The initial data weight is the number of samples taken. Weight degradation is not applied when moving the canvas.

Foveated rendering

Foveated rendering is implemented by the `getFoveationAdvisedSampleCount` method. We have implemented all the ideas from Section 4.4. The code is expressive and does not need additional comments.

A.2.6 API of the module: Available kernels

We have gone through the rendering methods, but how to call them from the Java code?

Each CUDA device-code has its API defined by `void` methods marked `__global__`, usually referred to as *kernels*. The kernels run on device and can later call device functions. The C++ templates allow us to make the kernels generic and to call generic device functions from them.

The compiler needs all generic methods to be instantiated during compile time but we do not call the kernels from any CUDA C/C++ code. At the same time, we need to be able to export the generic kernels in a format understood by the JCuda library.

For this reason, why we define floating-point-specific kernels that later call the generic cuda methods and we export them using `extern "C"`.

Following from this, te API of our CUDA modules contains, apart from other, following functions: `fractalRenderMainFloat`, `fractalRenderMainDouble`, `fractalRenderAdvancedFloat` and `fractalRenderAdvancedDouble`. We call those function from JCuda just by their name.

Other available kernels are `compose`, `fractalRenderUnderSampled` and `debug`. The `debug` kernel may be used for debugging, including fractal implementation debugging. The `fractalRenderUnderSampled` kernel is meant as a proof of concept of the under-sampling method described in the previous chapter. However, it is not currently used.

The `compose` kernel is meant to compose data from multiple kernels. It originated as a proof-of-concept of under-sampling too. Currently, it is simply being used for colorizing chaotic values and saving them into a texture.

All the function's parameters are based on the ideas from Chapters 3 and 4 and are well-documented in the source code.

A.2.7 Compilation and build

We need to compile our fractals to cuda binary files that are loadable at runtime of *chaos-ultra*. CUDA introduces the `ptx` file format for this. A `ptx` file contains a compiled ready-to-run CUDA module.

To be able to work with each fractal separately, our goal of the compilation is to produce a `<name>.ptx` file for each available fractal.

This is not easily achieved with standard build tools. At the same time, our CUDA code base is not large. Thus, we have decided to implement our own simple building script, which is sufficient for code base of this size. We provide version for Linux and version for Windows. The usage of the script is described in Attachment D.1, in `README.md`.

A.3 Java-Cuda mapping

Let us have a look at the third-lowest module of *chaos-ultra*, Java-Cuda mapping.

The module is contained within the java package
`cz.cuni.mff.cgg.teichmaa.chaosultra.cudarenderer`.

The role of this module is to abstract away details about using CUDA and to wrap the JCuda c-style calls to an OOP wrapper. The API of the module is expected to used by module `rendering`.

The public API of the module consists of implementations of the `FractalRendererProvider` and `FractalRenderer` interfaces from package `rendering` by `CudaFractalRendererProvider` and `CudaFractalRenderer` classes.

Code base

The code base of this module is rather technical and does not introduce a lot of logic. It works in compliance with [35] and [22]. It contains mostly Java wrappers of JCuda calls. These calls are greatly documented by [35] and [22] . The code is divided into well-named classes with well-named methods, that are mostly commented. Thus there is only little to to document about this particular module.

Still, this module is very interesting regarding the mapping of c-style code and data structures to Java-style OOP code and data structures.

Let us now briefly go through the class structure of the module.

A.3.1 Module object hierarchy

We introduce a one-to-one mapping of cuda modules to Java classes. To represent a CUDA module, type `FractalRenderingModule` is used.

Each specific fractal should provide own sub-class of this abstract class.

The class contains convenient functions for module management, fractal-specific parameters management and, primarily, for accessing the rendering kernels that are to be invoked.

Adding a new fractal

When a developer adds a new fractal, they don't change any existing code but rather add a new class that inherits from `FractalRenderingModule`. This follows the Open/Close principle, one of OOP's best practices, required by the thesis' goals.

Fractal custom parameters

To set fractal custom parameters, the module needs to cooperate with both backend and frontend. The callback `setFractalCustomParameters` needs to be overridden to get parameters set by the user. For parsing the params, convenient methods are provided, named `parseParamsAs<Something>`.

Then, adequate `writeToConstantMemory` method should be called. The method has a non-trivial overhead. The caller is responsible for calling the method only on actual parameters update, instead of before each rendering. Caller is also responsible for calling a proper type-variant, corresponding to the type of the `__constant__` in cuda source.

A.3.2 Cuda Kernel

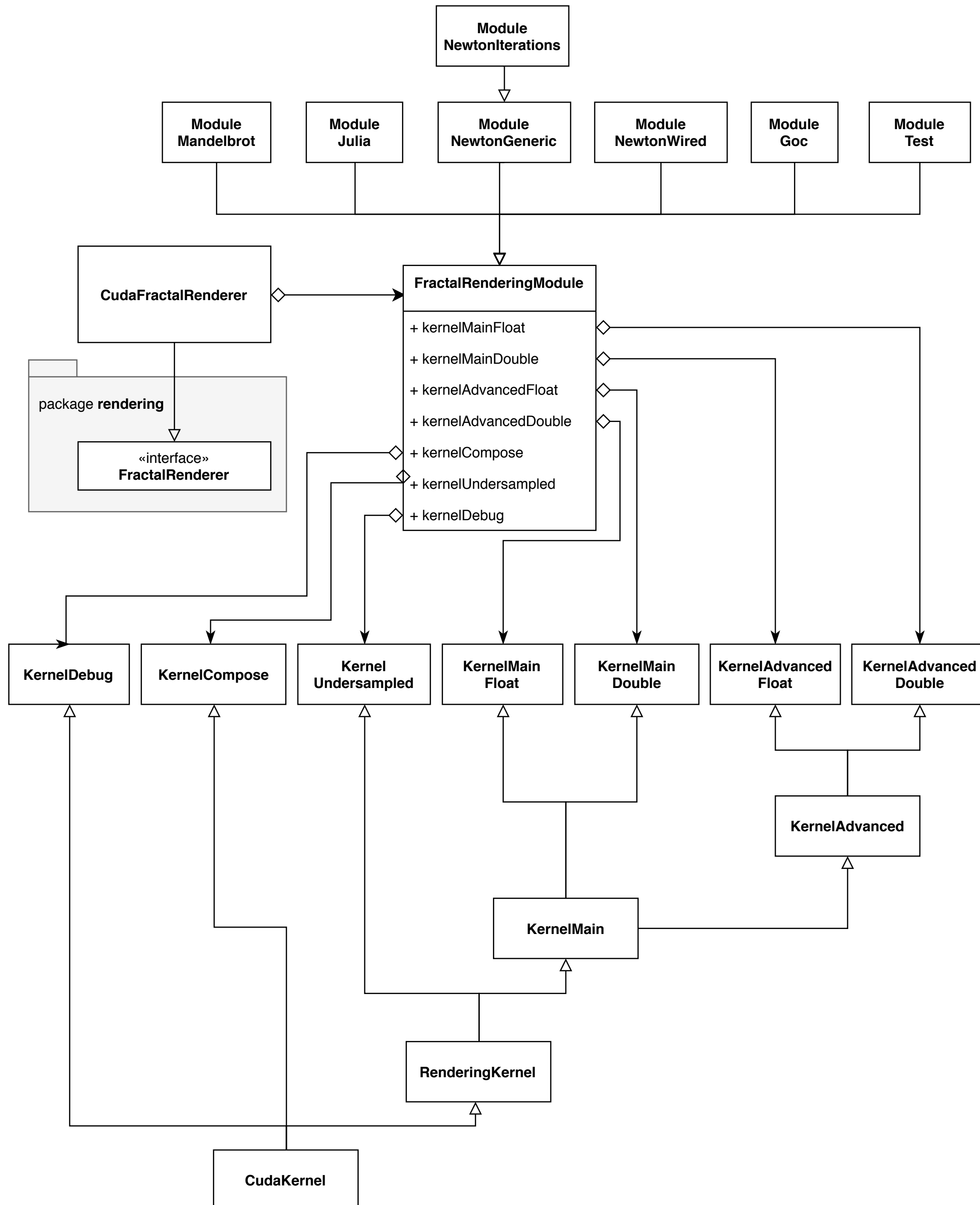
We have stated that `FractalRenderingModule` allows access to CUDA kernels that are to be invoked. Let us now summarize how those kernels are represented in our object-oriented code.

We introduce the `CudaKernel` abstract class as a type-safe object-oriented wrapper of JCuda's `CUfunction`. It is responsible for storing kernel's parameters when set prior to kernel's invocation. It allows Java-style parameter manipulation.

Rendering Kernel object hierarchy

We introduce a complex hierarchy of classes that inherit from `CudaKernel`. Why do we use this complex approach? The whole point of `cudarenderer` package is about launching different kernels at different times, with different parameters. To help us manage this task, some data structure is needed. We consider the introduced hierarchical structure to be appropriate.

Figure A.4: Class structure of the `cudarenderer` package.



The kernels in CUDA C/C++ source code have many common parameters. In our kernel hierarchy, the common parameters are managed by common ancestors. Kernel parameters setting is very cumbersome when using JCuda, and thus this abstraction that conceals the cumbersome details comes very handy.

The whole hierarchy is expressively documented by Figure A.4.

Floating point precision

Similarly as the CUDA C/C++ source code introduced 4 different methods for calling 2 kernels in 2 floating-point-precision variants, we introduce 4 classes for that.

However, this way the kernel hierarchy grows in complexity very fast. If more kernel types or floating-point precisions were to be added, the Bridge design pattern should be used for decoupling. See [65] for details about the pattern.

A.3.3 Class CudaFractalRenderer

`CudaFractalRenderer` implements the `FractalRenderer` interface. It is responsible for calling appropriate kernels with appropriate parameters.

Most of its code is c-style manipulation of JCuda data structures, needed for calling the CUDA API through JCuda.

As we have stated, there are not many implementation details in the `cudarenderer` package. However, quite the opposite is true for the Java renderer module. Lets us discover it now.

A.4 Java renderer

The package `renderer` encapsulates the logic needed to utilize the cuda backend and the graphical front end, and composes them to a single high performing application.

The package is CUDA-independent and, in theory, could use arbitrary underlying sampler based on arbitrary GPU or CPU technology.

The package's structure relies on the MVC pattern. The core idea of the pattern has been adapted to suit the program's specific needs, thus the structure does not strictly follow the traditional description from [4]. The model is represented by `Model` class, the view by `GLRenderer` class and the controller by `RenderingController` class. The whole class structure of the package is visually documented by Figure A.5.

The classes in this package use exclusively `double` Java primitive type to represent floating points, even when working with single-precision based cuda-kernels. This is for two reasons: (1) Because of lack of Java support for generic primitive types, as opposed to CUDA C/C++ language from the previous section. (2) Because there is almost-zero overhead of this, CPU-time and RAM-access not nearly being a hot path of our application.

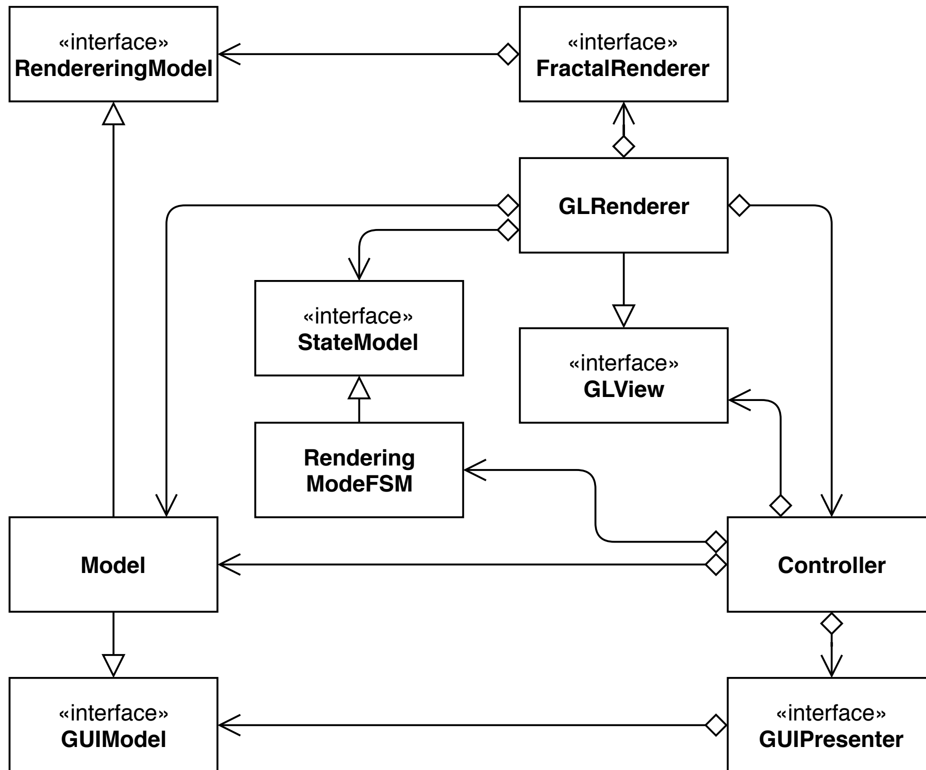


Figure A.5: class structure of the `rendering` package

A.4.1 Supporting multiple fractals

One of the main goals of the thesis is to support multiple, and arbitrary fractals, that can later be added by other users. We achieve it by programming against interfaces and using catalog-based management of available fractals.

The interface `FractalRenderer` specifies functionality needed by the `rendering` package to render a fractal. It is an abstract functionality with no dependence on a specific fractal, with even no dependence on the CUDA technology. The only technology it depends on is OpenGL, assuming that it is a generic and universally supported graphic library [29]. Each FractalRenderer instance can represent a different fractal. More details are described in Section A.4.2.

Catalog

Fractals are cataloged by a `FractalRendererProvider`. We provide one implementation, `CudaFractalRendererProvider` from package `cudarenderer`, which is aware of CUDA and its cuda-modules. In the future, the program is extensible by other potential providers based on other technologies. A catalog needs to know about available fractal rendering classes. The CUDA-aware catalog knows about the generic `CudaFractalRenderer` that needs to be supplied concrete `RenderingModule`. To be able to supply the modules, the catalog needs to know them. Java reflection API does not provide means to scan a package for implementations of `RenderingModule`, hence those have to be registered manually. This is done in the catalog's static initializer.

Call sequence

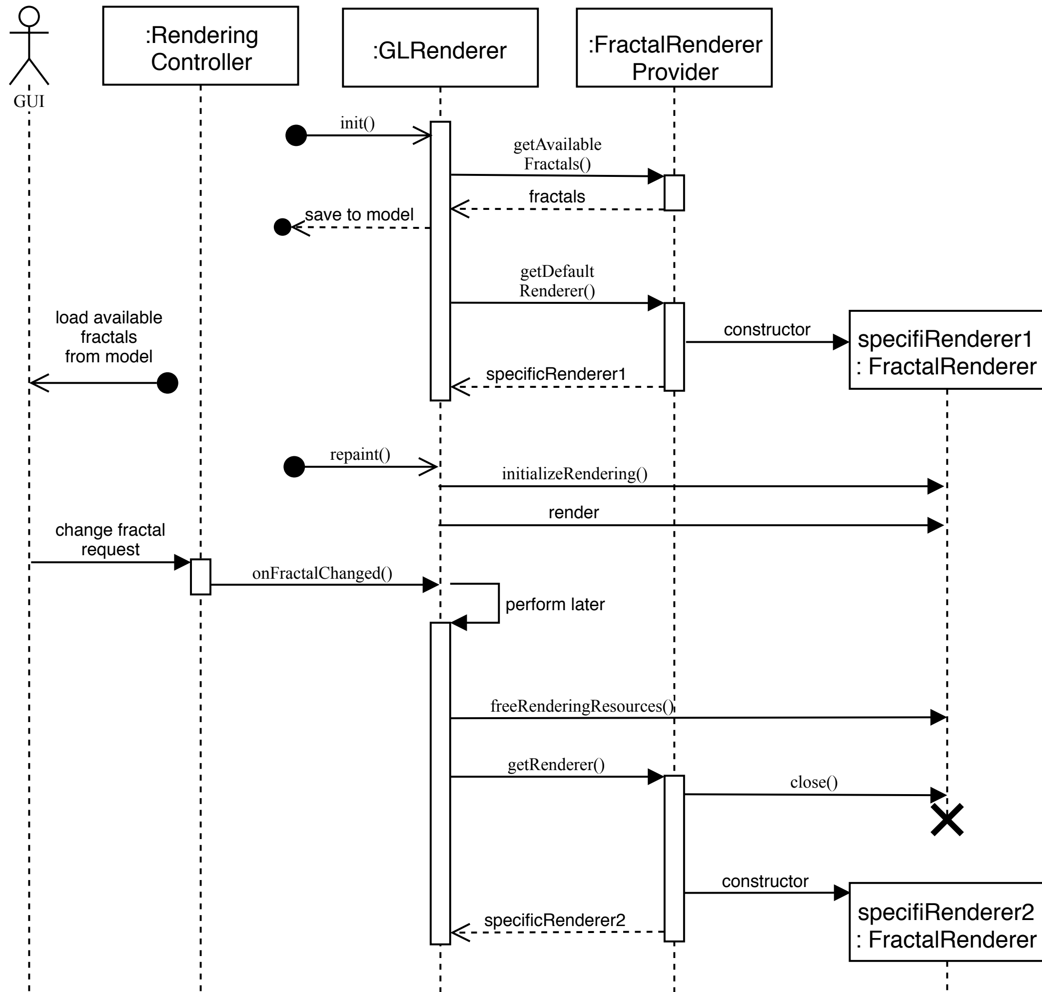


Figure A.6: Sequence of multiple fractals management

The sequence of calls is illustrated by Figure A.6 and is following: After program initialization a list of available fractals is to be shown to the user. It stored to the model by `GLRenderer`, which asks `FractalRendererProvider`. After user action, a specific fractal needs to be shown. The GUI asks the `RenderingController`, which asks `GLRenderer`, which asks `FractalRendererProvider`. The provider is responsible for creating appropriate instances of `FractalRenderer` and for disposing the old ones.

To the rest of the package `rendering`, the multi-fractal behavior is transparent.

A.4.2 Interface FractalRenderer

Interface FractalRenderer provides abstraction of the CUDA-dependent `CudaFractalRenderer` and defines the functionality required for the rest of the application to visualize the fractals. It represents a specific renderer for rendering a specific fractal. It is instantiated using `FractalRendererProvider`. Since it is expected to hold system resources not managed by JVM, it extends the `Closeable` interface.

It provides two core methods: `renderFast` and `renderQuality`. Both take `RenderingModel` as a parameter, explicitly defining, beside others, which heuristics should be used and which not.

The `renderFast` method is meant to provide the real time rendering. It should use any available means to reduce the computational complexity of fractal sampling. It is expected be scalable; in the sense that the lower the values of `maxIterations` and `superSamplingLevel`, the faster computation time. It is responsibility of the caller to set adequate values of those parameters.

The `renderQuality` method is used by the progressive rendering technique to show progressively nicer representations of the fractal when user interaction has stopped and we have relatively enough time for rendering. It should provide high quality images with little artifacts, at the cost of taking longer time to compute. It is expected be scalable; in the sense that the higher the values of `maxIterations` and `superSamplingLevel`, the higher visual quality. It is responsibility of the caller to set adequate values of those parameters.

Lifecycle

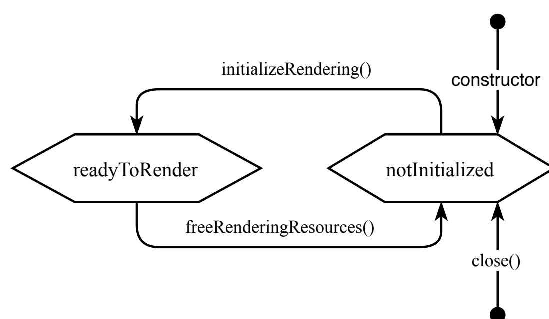


Figure A.7: `FractalRenderer` lifecycle

Before the first use, the renderer has to be initialized by calling `initializeRendering` with appropriate OpenGL parameters supplied. The initialization function expects following parameters (bundled in `GLParams` data class): (1) output texture (in the form of OpenGL integer handle) to draw the fractal on and (2) palette texture to read colors from. Both textures are expected to be valid OpenGL textures, initialized, in 32-bit `RGBA` format and 2D; the height of the palette is expected to be 1.

The implementation is allowed (and expected) to map the textures to its internal resources (e.g. `cudaGraphicsResource` in case of `CudaFractalRenderer`), potentially locking the textures for modification and for exclusive writing.

If the user of the FractalRenderer wants to modify the textures (e.g. after window resize or to pass it to some other renderer), it has to free it first by calling `freeRenderingResources`.

The two methods described above form the lifecycle of the class, switching between `notInitialized` and `readyToRender` states. See Figure A.7.

The approach we use is stateful and thus prone to errors (during development). There exists a simpler, stateless approach: `GLParams` could be supplied directly every time a rendering method is called. However, the stateless version would be

noticeably slower, as mapping a texture to a cuda resource (or similar resource) is a time consuming task [81]. Performing such operation often (30 times per second expected) would introduce a needless overhead.

A.4.3 Class GLRenderer and OpenGL loop

Class `GLRenderer` is implementation of `GLEventListener` provided by the library. Architecturally, it is the view of the MVC-triplet that is responsible for canvas rendering. Its controller is `RenderingController`, and it has two models: `Model` and `RenderingStateModel`.

It implements four lifecycle methods that are called by the GUI framework: `init`, `reshape`, `display`, `dispose`. It also exposes API for RenderingController, to be called after user interactions.

Lifecycle and flow

The program flow of `GLRenderer` is illustrated on Figure A.8 and described in the following text. As can be seen from the illustration, most of the operations performed are stateful and with no return value. This is due to the stateful nature of both OpenGL API and `FractalRenderer`.

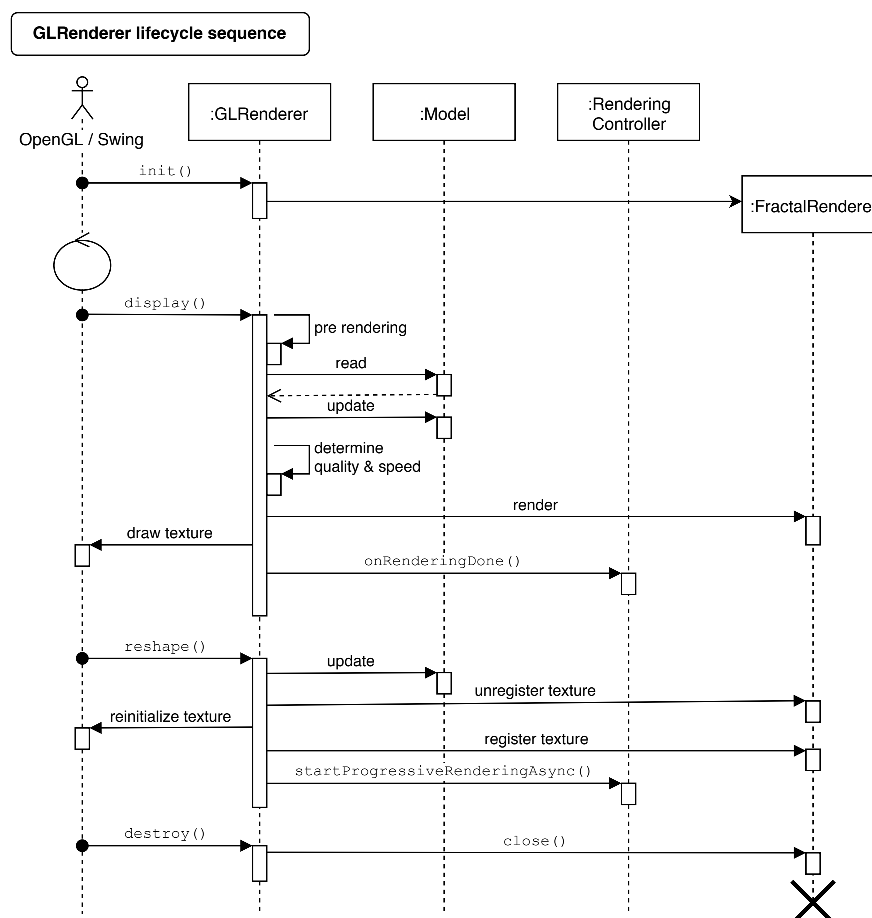


Figure A.8: `GLRenderer` lifecycle sequence

Each time `repaint` method is called, it is forwarded to Swing, which causes

display to be invoked later, by OpenGL. The frequency of repaint calls is managed by the controller.

When display is invoked, following operations are performed:

1. Pre-rendering operations are performed (see in Subsection A.4.3).
2. Data from both the models are read and evaluated.
3. Rendering quality/speed is computed, based on previous rendering experiences, and model is updated accordingly (see in Subsection A.4.3).
4. Adequate FractalRenderer's render is called with adequate parameters, in blocking way.
5. The resulting texture is drawn on the screen using OpenGL.
6. The controller is informed that rendering is done.

When reshape is invoked, following operations are performed: (1) Model is updated, (2) OpenGL texture is unregistered from the FractalRenderer, re-initialized to new size and registered again, and (3) progressive rendering is started, to quickly fill the new resized texture with image data.

The init method is called at the program beginning, the dispose method before program termination. Their implementation is straightforward and expressive.

Automatic quality

When automatic quality is enabled by the user, the GLRenderer is responsible for its implementation. The implementation follows the idea described in Section 4.2. The functionality is implemented by the determineRenderingModeQuality and setParamsToBeRenderedIn methods. The implementation heavily uses the StateModel interface, which should be managed by the controller.

Pre-rendering operations

Due to the asynchronous nature of our program and due to OpenGL context lifecycle, we introduce pre-rendering operations.

Calls to the OpenGL API, as well as CUDA calls that rely on OpenGL[4] have to be performed when an OpenGL context is active in the program. Its lifecycle is managed by the library and the context is available to us only during GLEventListener's lifecycle methods described above.

In response to controller's requests, GLRenderer often needs to access the OpenGL API. However, the callbacks from the controller are not part of the lifecycle[5] and the OpenGL context is not available during those calls.

For this, we introduce the doBeforeDisplay data structure. It is a list of tasks that need to be performed when OpenGL context is active. The list is populated during controller's callbacks (e.g. the onFractalChanged method) and the tasks it contains are invoked later, when display is called by OpenGL.

When display invocation is requested (via repaint) with the intention to perform the doBeforeDisplay tasks without rendering, the doNotRenderRequested flag is set.

[4] For example cudaGraphicsGLRegisterImage.

[5] They run in the same thread, and share the same Swing event loop, but OpenGL manages its context independently

A.4.4 Model

The data model used by the `rendering` package is represented by the `Model` class. Its public API is specified in the `model` sub-package, which contains various interfaces and model-related data structures that `Model` implements/uses.

`Model`'s public API is following. It is also documented in the code.

- **`RenderingModel`** — the data needed by a `FractalRenderer` to render and methods to give feedback
- **`DefaultFractalModel`** — extension of `RenderingModel`, allowing fractals to supply default values (e.g. max iterations)
- **`GUIModel`** — data that ought to be visualized to the user;the interface is used to communicate with the GUI and is located in the **`gui`** package for better code-coherence
- **`PublicErrorLogger`** — allows for logging public error that are to be displayed to the user

Following the MVC architectural pattern, the model is both queried and updated by both the controller and the view. The program's data flow is visually described by Figure A.9.

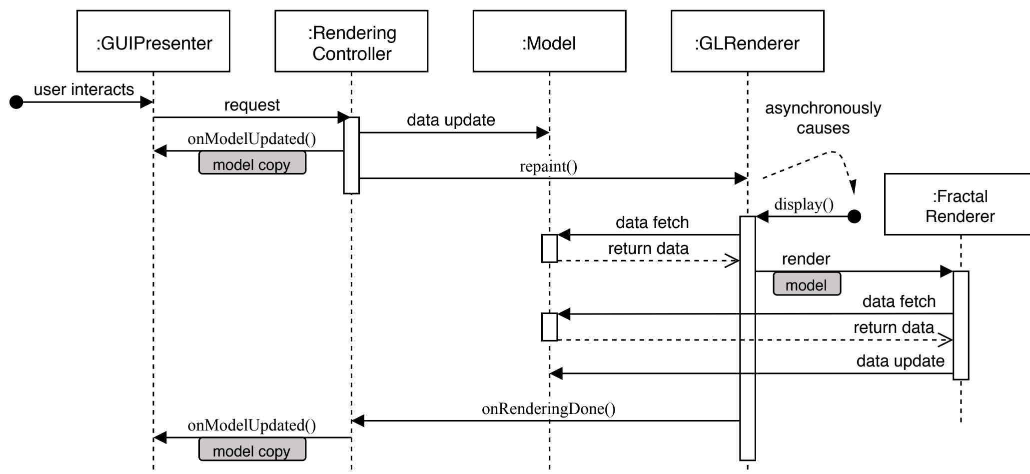


Figure A.9: `rendering` package data flow

A.4.5 Class RenderingController

RenderingController is the logical core of the rendering process. It is responsible for acting upon user requests, evaluating user commands and translating them to model and/or view updates.

It reacts both on `GLRenderer`'s (a component of OpenGL/Swing) and `GUIPresenter`'s (a component of JavaFX) requests. This follows the architectural pattern Presentation–abstraction–control introduced in Section 3.10.1.

Mouse events

`RenderingController` extends Swing's `MouseAdapter` and overrides some of its mouse event handler methods to react upon user's mouse interaction. It is re-

sponsible for recomputing boundaries of the plane segment that is being rendered upon mouse move or zoom in/out.

Automatic quality and progressive rendering

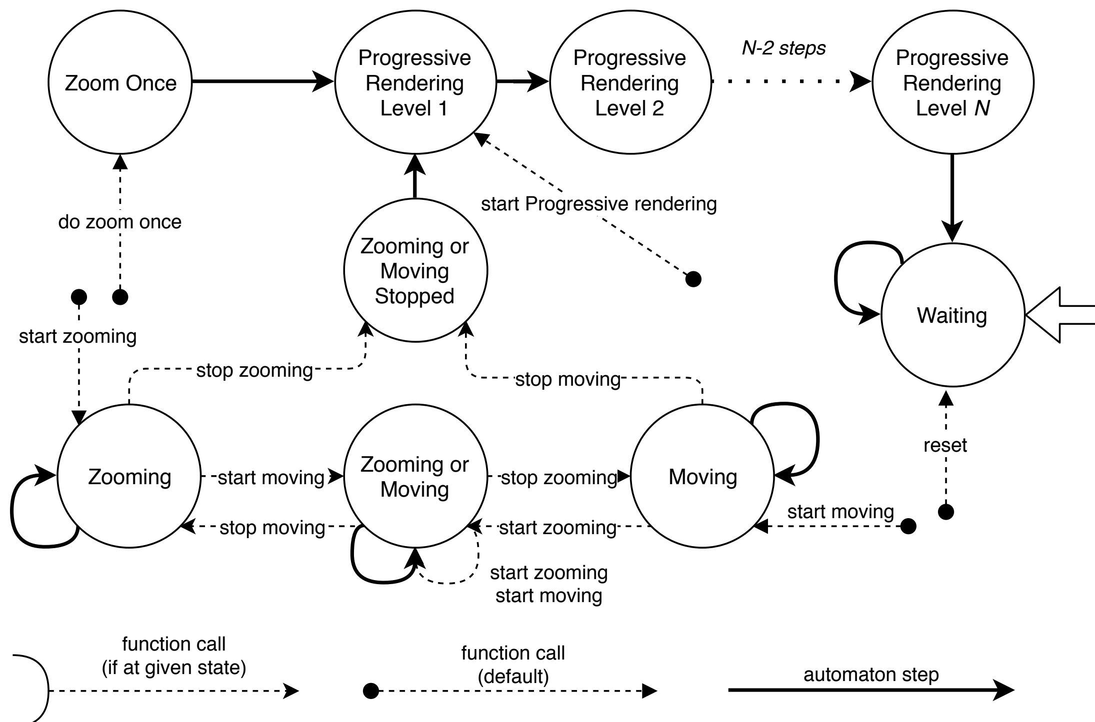


Figure A.10: States and transitions of the rendering mode state machine

The controller is responsible for controlling which of the various rendering approaches is used, especially in deciding whether the user interaction has stopped and progressive rendering should be activated. Internally, this is achieved through a finite state machine implemented by `RenderingModeFSM` extending `StateModel`. The machine is controlled by the controller and queried through the `StateModel` interface by `GLRenderer`. The states and transitions of the machine are visually described by Figure A.10.

A.5 Graphical user interface

To allow for simple user interaction with the fractal, we implement a basic graphical user interface (GUI). As discussed in Chapter 3, we use JavaFX and Swing libraries. The class structure, including gui-component hierarchy, is visualized by Figure A.11.

A.5.1 Swing GLCanvas

The renderer draws the fractal on an OpenGL texture, residing in the GPU memory. This texture has to be displayed by our GUI. Based on the discussion form Chapter 3, we use Swing library for this task.

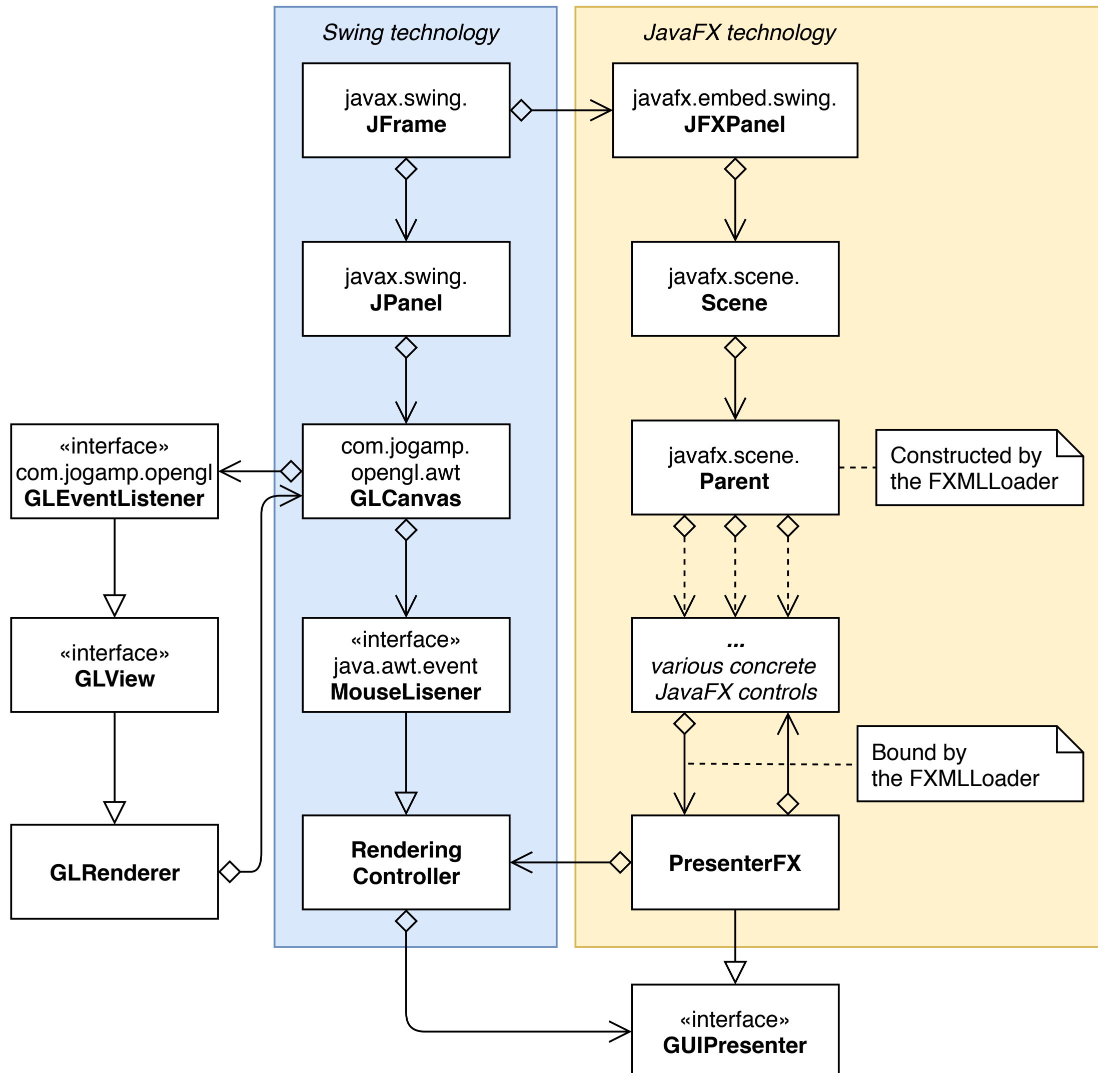


Figure A.11: Class structure of the GUI, including gui-component hierarchy.

Swing provides `GLCanvas` class, a Swing `Component` that allows given OpenGL context to render on it. `GLCanvas` relies on injected `GLEventListener` and, as every Swing `Component`, allows for registering of variable mouse event listeners.

For the mouse listening, we use `RenderingController` class, described in Section A.4.5. As an implementation of the required `GLEventListener`, we provide `GLRenderer` class.

A.5.2 JavaFX and Swing integration

Both JavaFX and Swing provide easy-to-use menas to integrate with the other technology. In both cases, the integration is done via injecting a `JavaFX/Swing` panel into `Swing/JavaFX` window component tree, respectively. JavaFX provides `SwingNode`, Swing provides `JFXPanel`. Each technology maintains its own event loop in its own thread. Event bubbling (e.g. mouse interaction) is translated between the loops transparently to the programmer [60].

Swing divides its components into two categories: *lightweight* and *hard-weight*. Most of the common components are lightweight, but GLCanvas is hard-weight.

When Swing is embedded in JavaFX using `SwingNode`, only lightweight components are enabled. Thus we cannot use JavaFX Panel as the root in our implementation.

We use Swing's `JFrame` as root of the application. The frame contains the `GLCanvas` and the `JFXPanel`, the latter one being responsible for the user-interaction part of the GUI. See Figure A.11 for illustration.

Thread safety

Each library runs in its own thread, in which it operates internal event loop. Our code relies on callbacks from the library's event loop. We have to ensure proper communication between the two threads / loops. Again, both technologies provide easy-to-use means for thread cooperation.

Swing provides `SwingUtilities.invokeLater(Runnable task)`, JavaFX provides `Platform.runLater(Runnable task)` methods. Both methods implement the same idea: they are safe to call from any thread and context, and arrange that the given `task` is run from the platform's event loop.

The OpenGL loop runs in the *Swing event message thread*, and thus so do both the rendering logic of the program and the cuda calls.

The interface between the two threads is the `PresenterFX` class. It is responsible for determining in what thread a method should be invoked when calling methods of other classes or when being called. It is responsible for using adequate `invokeLater/runLater` call. Calling any of its methods from Swing event loop thread is safe. For all other classes in the application, threading is transparent.

Callers of `PresenterFX` (typically `RenderingController` via `GUIController`) must be aware that the called function will be performed asynchronously. For this reason, a deep copy of model, not the original object, is passed to `GUIController.onModelUpdated` when called from `PresenterFX`.

A.5.3 FXML

FXML is user interface markup language for defining JavaFX user interfaces. It is based on XML and CSS technologies. It is a standard JavaFX-related technology and we use it in a standard way[62].

We use this markup language to define the tree structure of the graphical user interface. It is located in `src/main/resources.window.fxml` and loaded by a `FXMLLoader` at the start of the application.

A.5.4 FX Presenter

The `PresenterFX` class is a presenter of the JavaFX part of the GUI. Its purpose is to propagate user events to the rest of the application and to present changes in the application data to the user.

It interacts with the `RenderingController` (following the PAC pattern introduced in Section 3.10.1) and when supplied a `GUIModel`, updates the underlying view.

It is implemented by standard JavaFX paradigm: the underlying view is constructed and supplied by the `FXMLLoader`, the Presenter can modify it programmatically and can simply implements user-event handlers.

A.6 Developer README

All details needed for future-development are included in Attachment D.1, in `README.md`. You can find there information about:

- Technical requirements
- Software prerequisites
- Build and compilation process
- Running the program
- Documentation of developer part of the GUI
- Troubleshooting

B. User documentation

B.1 Technical requirements

To run *chaos-ultra*, you will need a PC with Windows or Linux operation system, equipped with CUDA-capable graphic cards by nvidia corporation. A GPU with CUDA version 6 or higher is needed. To see if your graphic card is CUDA-capable and has sufficient version, check `https://www.geforce.com/hardware/technology/cuda/supported-gpus` or `https://en.wikipedia.org/wiki/CUDA`.

B.2 Installation guide

If you meet the technical requirements, following steps apply for installation.

1. If you do not have it installed already, download and install CUDA toolkit from `https://developer.nvidia.com/cuda-downloads`.
2. If you do not have it installed already, download and install Java (version 8 or higher), for example from `https://java.com/en/download/`. Note: You need JavaFX library to be installed too. If you have downloaded Oracle Java, you should be fine. If you are using `openjdk`, you will need to install `openjfx`, see `https://openjfx.io/`.
3. Download and extract the compiled version of *chaos-ultra* from `https://gimli.ms.mff.cuni.cz/~tonik/chaos-ultra.zip`. It is also included as Attachment D.2.
   - If you prefer to compile the application yourself, see Chapter A.6 for instructions.
4. Run `chaos-ultra-1.0-jar-with-dependencies.jar`, either by double clicking it or with command
   `java -jar chaos-ultra-1.0-jar-with-dependencies.jar`.
   - Note: for the program to start correctly, the directory `cudaKernels` must be located in the same directory as the `chaos-ultra.jar` file and it must be populated with `.ptx` files.

B.3 Navigating the user interface

Interaction with the fractal

- use `left` mouse button to `drag` (move around)
- use `right` mouse button to continuous `zoom in`
- use `middle` mouse button to continuous `zoom out`
- use mouse `wheel` to `one-step zoom` in/out

When you stop the interaction, the program may be unresponsive for about one second, before you can interact again. This is a known bug. Please, be patient.

Right menu

The right menu allows changing of default parameters. Its labels should be explanatory.

Increasing *maximum iterations* usually leads to higher quality images but significantly decreases real-time performance and vice versa.

Changing *max supersampling* does not take effect if *automatic quality* is enabled in *research options*.

Note that the *research options* are meant for experimenting, and tweaking them may lead to bad user experience, for example laggy behavior.

B.4 Fractal change

The program allow for changing the displayed fractal. This is done via the choice box in the right menu. Only correctly added fractals are available; consult Chapter C if you want to add one.

Parameters

If the fractal needs some parameters, supply them to the text field.

The format of the data is not universal and is defined by each fractal's provider. Usual format, however, is comma/semicolon separated values, comma/semicolon separated key value pairs in format `key=value` or JSON.

Parameter format of the default fractals is following:

- Mandelbrot — no parameters
- Julia — the c parameter in format `re;im`, `re` and `im` being floating point numbers.
- Newton wired — no parameters
- Newton generic — JSON containing the polynomial coefficients and roots. The provided example should be self-explanatory.
- Newton colored by iterations — Same as Newton generic, also adds *color magnifier* that is used by the its coloring method to map chaotic values to large palettes.

B.5 Changing the color palette

To change the color palette used for fractal coloring, edit the file palette.png to contain desired colors (if the file does not exist, create it). The program needs to be restarted for this to take effect.

Alternatively, you can change location and file-format of the palette through program's arguments, details are described in Attachment D.1, in `README.md`.

The format of the palette file is following: the first row of the image is interpreted as the palette, all other rows are ignored. Common image formats are supported, including `jpg` and `png`.

The palette is recommended to consist of smooth gradients rather than discrete color areas; however, this depends on the coloring method of each fractal. (For example for colorization of the Newton fractal, discrete palette would be adequate.)

B.6 Image export

To export currently viewed image to a file, use the *save as image* functionality from *additional functionality* pane.

The image is then stored in `saved_images` directory with current timestamp used as its name. The name and output location cannot be changed from the program.

B.7 Troubleshooting

If an error occurs, the corresponding error message is shown in the *system messages* area.

If it is a CUDA-related error, the program usually needs to be restarted before working properly again.

C. Adding a custom fractal

If you want to add a new fractal, following steps apply.

See implementations of the provided fractals for examples.

Prerequisites

We expect that you have gone through Chapter A.6 and know how to compile and build the program.

We expect that you know the formula of the fractal you are adding.

We expect that you have basic programming skills in C or C++ and in Java.

In the cuda backend module

First, choose a unique name and filename for your fractal. In the following text, we will use macros <name> and <filename> to refer to them.

1. Create file <filename>.cu in src/main/cuda/fractals.
2. Include #include "fractal.cuh"
3. Implement the computeFractal method defined in fractal.cuh.
   - If you need complex numbers arithmetics, include thrust/complex.h and convert z to thrust::complex type. See newton_generic for examples.
   - If your implementation needs fractal-specific parameters, declare them as __constant__. See more details in Section A.2.1.
4. Implement the colorize method defined in fractal.cuh. You can use and copy-paste proposed default implementation.
5. Implement the debug method defined in fractal.cuh. Feel free to leave the body empty. Common convention is to add printf("<name>\n").
6. Compile with
   - .\compile.bat <filename> on Windows.
   - ./compile.sh <filename> on Linux.

In Java

1. Create a new class in package cz.cuni.mff.cgg.teichmaa.chaos_ultra.cuda_renderer.modules. The name of the class can be arbitrary, common convention is Module<name>.
2. Inherit from cz.cuni.mff.cgg.teichmaa.chaos_ultra.cuda_renderer.FractalRenderingModule.
3. Implement a **parameterless** constructor by adding the following call: super(<filename>, <name>);
4. Implement the setFractalCustomParameters method.
   - Leave the method blank if your fractal takes no parameters.
   - Parsing the parameters from a user string is up to you. This allows for great flexibility, as the format and number of parameters is unlimited.
   - You can use helper functions parseParamsAsDoubles, parseParamsAsIntegers and parseParamsAsKeyValPairs.

- To set the fractal-specific parameters in the CUDA kernel, use adequate `writeToConstantMemory` method. Details are described in Section A.3.1.

5. Optional: you may override `supplyDefaultValues` callback if you want to set specific initial rendering parameters for your fractal, e.g. `maxIterations` or specific `planeSegment` or fractal-specific parameters.
6. Register your newly created class by adding it to `modules` collection in the static initializer of `cz.cuni.mff.cgg.teichmaa.chaos_ultra.cuda_renderer.CudaFractalRendererProvider`.
7. Compile the program with maven, see more details in Attachment D.1, in `README.md`.

Hello world

To see that your fractal code is called, you can use the *Invoke debug method* button in the GUI, in *additional functionality* pane. Clicking the button will call your `debugFractal` method on the device in a 1×1 grid.

D. Electronic attachments

Hereby we list electronic attachments of the thesis. Electronic attachments have been uploaded to the repository as a `zip`-file file with following inner directory structure.

D.1 Source code of the program

The source code of the program is located in directory `chaos-ultra-source` and has a standard maven-project structure. The directory contains `README.md` file that is referenced from the thesis.

D.2 Compiled executable

A compiled version of the source code, that can be executed at once, is located in directory `chaos-ultra-executable`.

D.3 Experiments

D.3.1 Perception study videos

The videos that have been used for perception study are located in directory `perception-study-videos`.

D.3.2 Perception study results

The full results of the perception study are located in directory `perception-study-results`.

D.3.3 Real-time toolset evaluation

The videos that have been used for evaluation of the real-time toolset are are located in two directories:

1. Recordings of *chaos-ultra* with the real-time toolset **activated** are same as the recordings used for the perception study and hence are located in `perception-study-results`.
2. Recordings of *chaos-ultra* with the real-time toolset **deactivated** are located in directory `recordings-with-real-time-toolset-deactivated`.

D.3.4 Examples of program's output

Examples of program's output are located in directory `program-output-examples`.

Bibliography

[5] Munesh Chauhan. “FRACTALS IMAGE RENDERING AND COMPRESSION USING GPUS”. In: *International Journal of Digital Information and Wireless Communications (IJDIWC)* 1 (Jan. 2012), pp. 813–818.

[25] K. J. Falconer. *Fractal geometry : mathematical foundations and applications*. 2nd edition. Wiley, 2006. ISBN: 978-0-470-84862-3.

[27] Yuval Fisher. “Fractal Image Compression”. In: *SIGGRAPH ‘92 Course Notes* (1992).

[30] Brian Guenter et al. “Foveated 3D graphics”. In: *ACM Transactions on Graphics* 31.6 (Nov. 2012), p. 1. DOI: 10.1145/2366145.2366183.

[36] et al. Jing Z. Liu. “Fractal Dimension in Human Cerebellum Measured by Magnetic Resonance Imaging”. In: *Biophysical Journal* 85.6 (Dec. 2003), pp. 4041–4046. DOI: 10.1016/S0006-3495(03)74817-6.

[37] Martin Kahoun. “Procedural generation and realtime rendering of planetary bodies”. MA thesis. Charles University in Prague, Faculty of Mathematics and Physics, 2010.

[39] S. M. Levien R. B.; Tan. “Double Pendulum: An experiment in chaos”. In: *American Journal of Physics* (1993). URL: https://aapt.scitation.org/doi/10.1119/1.17335.

[40] Edward N. Lorenz. “Deterministic Nonperiodic Flow”. In: *Journal of the Atmospheric Sciences* 20.2 (Mar. 1963), pp. 130–141. DOI: 10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2.

[41] Edward N. Lorenz. “Predictability; Does the flap of a butterfly’s wings in Brazil set off a tornado in Texas?” In: *AAAS 139th meeting* (1972). URL: http://eaps4.mit.edu/research/Lorenz/Butterfly_1972.pdf.

[43] Benoit Mandelbrot. *Fractals and chaos : the Mandelbrot set and beyond*. Vol. C. Springer, 2004. ISBN: 978-1-4757-4017-2.

[44] Benoit Mandelbrot. *Fractals: Form, Chance and Dimension*. W.H. Freeman, 1977. ISBN: 0-7167-0473-0.

[45] Benoit Mandelbrot. “How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension”. In: *Science* 156.3775 (1967), pp. 636–638. ISSN: 0036-8075. DOI: 10.1126/science.156.3775.636. eprint: https://science.sciencemag.org/content/156/3775/636.full.pdf. URL: https://science.sciencemag.org/content/156/3775/636.

[46] Benoit Mandelbrot. *The Fractals Geometry Of Nature*. San Francisco, Calif. : Freeman, 1982. ISBN: 978-0-7167-1186-5.

[47] Athanasios I. Margaris. “Simulation and Visualization of Chaotic Systems”. In: *Computer and Information Science* 5.4 (June 2012). DOI: 10.5539/cis.v5n4p25.

[49] José Martínez et al. “Objective video quality metrics: A performance analysis”. In: (June 2019).

[50] Will. D. Mayfield et al. “Fractal Art Generation using GPUs”. In: (Nov. 8, 2016). arXiv: http://arxiv.org/abs/1611.03079v1 [cs.GR].

[52] Curtis T McMullen. “Families of rational maps and iterative root-finding algorithms”. In: *Annals of Mathematics 125(3): 467–493* (1987). URL: https://dash.harvard.edu/handle/1/9876064.

[53] Michel Millodot. *Dictionary of Optometry and Visual Science E-Book*. Elsevier Health Sciences, July 30, 2014. 450 pp. URL: https://www.ebook.de/de/product/23252845/michel_millodot_dictionary_of_optometry_and_visual_science_e_book.html.

[58] James Munkres. *Topology (2nd Edition)*. Pearson, 2000. ISBN: 0131816292.

[63] Robert P OShea. “Thumbs Rule Tested: Visual Angle of Thumbs Width is about 2 Deg”. In: *Perception* 20.3 (June 1991), pp. 415–418. DOI: 10.1068/p200415.

[64] J P. Lewis. “Is the Fractal Model Appropriate for Terrain?” In: (Jan. 1990).

[66] Jaideep Pathak et al. “Hybrid Forecasting of Chaotic Processes: Using Machine Learning in Conjunction with a Knowledge-Based Model”. In: (Mar. 9, 2018). DOI: 10.1063/1.5028373. arXiv: http://arxiv.org/abs/1803.04779v1 [cs.LG].

[67] Anjul Patney et al. “Towards foveated rendering for gaze-tracked virtual reality”. In: *ACM Transactions on Graphics* 35.6 (Nov. 2016), pp. 1–12. DOI: 10.1145/2980179.2980246.

[68] Clifford A Pickover. *The math book from Pythagoras to the 57th dimension, 250 milestones in the history of mathematics*. Sterling : New York, NY, 2009. ISBN: 978-1-4027-5796-9.

[70] Kai Qian et al. *Software Architecture And Design Illuminated*. Jones and Bartlett Publishers, Inc, Feb. 19, 2009. 388 pp. ISBN: 076375420X. URL: https://www.ebook.de/de/product/8104374/kai_qian_xiang_fu_lixin_tao_chong_wei_xu_software_architecture_and_design_illuminated.html.

[71] Ping Chen Richard Hollis Day. *Nonlinear dynamics and evolutionary economics*. 1993. ISBN: 9780195078596.

[72] Leonid A. Safonov et al. “Multifractal chaotic attractors in a system of delay-differential equations modeling road traffic”. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 12.4 (Dec. 2002), pp. 1006–1014. DOI: 10.1063/1.1507903.

[74] Mitsuhiro Shishikura. “The Hausdorff Dimension of the Boundary of the Mandelbrot Set and Julia Sets”. In: *The Annals of Mathematics* 147.2 (Mar. 1998), p. 225. DOI: 10.2307/121009.

[75] P. D. Sisson. “Fractal art using variations on escape time algorithms in the complex plane”. In: *Journal of Mathematics and the Arts* 1.1 (Mar. 2007), pp. 41–45. DOI: 10.1080/17513470701210485.

[77] Scott Sutherland. “An Introduction to Julia and Fatou Sets”. In: *Springer Proceedings in Mathematics and Statistics* 92 (Jan. 2014), pp. 37–60. DOI: 10.1007/978-3-319-08105-2__3.

[78] R.P. Taylor. “Reduction of Physiological Stress Using Fractal Art and Architecture”. In: *Leonardo* 39.3 (June 2006), pp. 245–251. DOI: 10.1162/leon.2006.39.3.245.

[79] Richard P. Taylor et al. “Perceptual and Physiological Responses to Jackson Pollock’s Fractals”. In: *Frontiers in Human Neuroscience* 5 (2011). DOI: 10.3389/fnhum.2011.00060.

[80] Gordon Teskey. *The Poetry of John Milton*. Cambridge, Mss. : Harvard Univ. Press, 2015. ISBN: 978-0674416642.

Electronic sources

[1] Oskar Elek Ahmed Hassan Yousef. *Can Scatter algorithm become faster than the Gather Algorithm in GPU?* Sept. 9, 2015. URL: https://www.researchgate.net/post/Can_Scatter_algorithm_become_faster_than_the_Gather_Algorithm_in_GPU (visited on 06/26/2019).

[2] Eric Bainville. *CPU/GPU Multiprecision Mandelbrot Set.* 2009. URL: http://www.bealto.com/mp-mandelbrot_intro.html.

[3] Paul Bourke. *Fractals, Chaos, Self-Similarity.* 2019. URL: http://paulbourke.net/fractals/ (visited on 04/06/2019).

[4] Steve Burbeck. *Webarchive: Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC).* URL: https://web.archive.org/web/20120729161926/http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html (visited on 06/26/2019).

[6] Ph.D. Christopher M. Danforth. *Chaos in an Atmosphere Hanging on a Wall.* Mar. 17, 2013. URL: http://mpe.dimacs.rutgers.edu/2013/03/17/chaos-in-an-atmosphere-hanging-on-a-wall/ (visited on 04/02/2019).

[7] hpdz.net collective. *Technical Info - Convergent Fractals.* URL: http://hpdz.net/TechInfo/Convergent.htm#Nova (visited on 06/26/2019).

[8] Wikipedia contributors. *Buddhabrot — Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/wiki/Buddhabrot (visited on 06/26/2019).

[9] Wikipedia contributors. *CUDA — Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/wiki/CUDA (visited on 06/26/2019).

[10] Wikipedia contributors. *Exponential map (discrete dynamical systems) — Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/wiki/Exponential_map_(discrete_dynamical_systems) (visited on 06/26/2019).

[11] Wikipedia contributors. *File:Barnsley Fern — Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/wiki/Barnsley_fern (visited on 06/26/2019).

[12] Wikipedia contributors. *File:EyeOpticsV400y.jpg — Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/wiki/Visual_angle#/media/File:EyeOpticsV400y.jpg (visited on 06/26/2019).

[13] Wikipedia contributors. *File:KochFlake — Wikipedia, The Free Encyclopedia.* URL: https://commons.wikimedia.org/wiki/File:KochFlake.svg (visited on 06/26/2019).

[14] Wikipedia contributors. *File:Peripheral$_v$ision.svg———Wikipedia,TheFreeEncyclopedia* URL: https://en.wikipedia.org/wiki/File:Peripheral_vision.svg.

[15] Wikipedia contributors. *Fractal compression — Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/wiki/Fractal_compression (visited on 06/26/2019).

[16] Wikipedia contributors. *Fractal, Common techniques for generating fractals — Wikipedia, The Free Encyclopedia*. Mar. 16, 2019. URL: https://en.wikipedia.org/wiki/Fractal#Common_techniques_for_generating_fractals (visited on 04/02/2019).

[17] Wikipedia contributors. *Julia Set — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Julia_set#Quadratic_polynomials (visited on 06/26/2019).

[18] Wikipedia contributors. *List of chaotic maps — Wikipedia, The Free Encyclopedia*. Apr. 1, 2019. URL: https://en.wikipedia.org/wiki/List_of_chaotic_maps (visited on 04/02/2019).

[19] Wikipedia contributors. *List of fractals by Hausdorff dimension — Wikipedia, The Free Encyclopedia*. Mar. 16, 2019. URL: https://en.wikipedia.org/wiki/List_of_fractals_by_Hausdorff_dimension (visited on 04/02/2019).

[20] Wikipedia contributors. *Mandelbrot Set — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Mandelbrot_set (visited on 06/26/2019).

[21] Wikipedia contributors. *Newton's method — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Newton's_method (visited on 06/26/2019).

[22] nvidia corporation. *CUDA C Programming Guide*. May 30, 2019. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 06/26/2019).

[23] curran. *WebGL Mandelbrot*. URL: https://github.com/curran/mandelbrot.

[24] Alan Dewar. *Images from the Mandelbrot Set*. URL: http://www.cuug.ab.ca/dewara/mandelbrot/images.html (visited on 06/26/2019).

[26] Pavel Fatin. *gpu-fractal-renderer*. URL: https://github.com/pavelfatin/gpu-fractal-renderer.

[28] *Fractal Tutorial - Color Techniques For Fractals*. URL: http://fractalarts.com/ASF/fractal_color_tutorial_1.html (visited on 06/26/2019).

[29] Anthony Frausto-Robledo. *A Unified Graphics Future—How The Khronos Group Intends To Get Us There (Part 1)*. Aug. 28, 2017. URL: https://architosh.com/2017/08/a-unified-graphics-future-how-the-khronos-group-intends-to-get-us-there-part-1/ (visited on 06/26/2019).

[31] Shea Gunther. *14 amazing fractals found in nature*. Apr. 13, 2013. URL: https://www.mnn.com/earth-matters/wilderness-resources/blogs/14-amazing-fractals-found-in-nature (visited on 06/26/2019).

[33] Jan Hubička. *GNU XaoS*. URL: http://matek.hu/xaos/doku.php (visited on 06/26/2019).

[34] Isaac. *Mandelbrot-like sets for functions other than f(z)=z2+c?* July 29, 2010. URL: https://math.stackexchange.com/questions/1099/mandelbrot-like-sets-for-functions-other-than-fz-z2c (visited on 06/26/2019).

[35] *JCuda Documentation*. URL: http://www.jcuda.org/documentation/Documentation.html.

[42] Michael Lucy. *Fractals in nature*. URL: https://cosmosmagazine.com/mathematics/fractals-in-nature (visited on 06/26/2019).

[48] Thomas Marsh and Jan Hubicka. *XAOS ALGORITHMS*. URL: https://web.mit.edu/kolya/sipb/afs/root.afs/athena/activity/p/peckers/OldFiles/Programs/XaoS-2.2/doc/algorithm.txt (visited on 06/26/2019).

[51] Steven C. McConnell. *Progressive Image Rendering*. Dec. 14, 2005. URL: https://blog.codinghorror.com/progressive-image-rendering/ (visited on 06/26/2019).

[54] Bryon Moyer. *How Does Scatter/Gather Work?* Feb. 9, 2017. URL: https://www.eejournal.com/article/20170209-scatter-gather/ (visited on 06/26/2019).

[55] Robert Munafo. *Mu-Ency - The Encyclopedia of the Mandelbrot Set*. 1996. URL: https://mrob.com/pub/muency.html (visited on 04/06/2019).

[56] Robert P. Munafo. *Mariani/Silver Algorithm*. Sept. 8, 2010. URL: https://www.mrob.com/pub/muency/marianisilveralgorithm.html (visited on 06/26/2019).

[57] Robert P. Munafo. *Successive Tradeoff Methods*. 1993-01-23. URL: https://mrob.com/pub/muency/successivetradeoffmethods.html (visited on 06/26/2019).

[59] Jeremy Norman. *Mandelbrot's "The Fractal Geometry of Nature" : HistoryofInformation.com*. en. 2019. URL: http://www.historyofinformation.com/detail.php?entryid=1151.

[60] Oracle. *Enriching Swing Applications with JavaFX Functionality*. URL: https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/jtable-barchart.htm#CHDBHIJJ.

[61] Oracle. *The JavaFX Advantage for Swing Developers*. URL: https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/overview.htm#CJAHBAHA (visited on 06/26/2019).

[62] Oracle. *Using FXML to Create a User Interface*. URL: https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm (visited on 06/26/2019).

[65] PANKAJ. *Bridge Design Pattern in Java*. URL: https://www.journaldev.com/1491/bridge-design-pattern-java (visited on 06/26/2019).

[73] sgtatham. *Fractals derived from Newton-Raphson iteration*. URL: https://www.chiark.greenend.org.uk/~sgtatham/newton/ (visited on 06/26/2019).

[76] LLC Spotworks. *Electric Sheep*. URL: https://electricsheep.org/ (visited on 06/26/2019).

[81] Thomasdb. *CUDA-OpenGL interop performance*. May 26, 2014. URL: https://devtalk.nvidia.com/default/topic/747242/cuda-opengl-interop-performance/ (visited on 06/26/2019).

[82] Steve Thompson. *Adaptive Sampling*. July 16, 2011. URL: http://www.mathstat.helsinki.fi/msm/banocoss/2011/Presentations/Thompson_web.pdf (visited on 06/26/2019).

[83] John Tsiombikas. *Nuclear / Mindlapse*. URL: http://nuclear.mutantstargoat.com/articles/sdr_fract/.

[84] ultrafractal. *Class* $Standard_N ovaMandel$. URL: http://formulas.ultrafractal.com/reference/Standard/Standard_NovaMandel.html (visited on 06/26/2019).

[85] Wolfram. *Mandelbar Set — Wolfram Math World*. URL: http://mathworld.wolfram.com/MandelbarSet.html.

[86] Wolfram. *Mandelbrot Set — Wolfram Math World*. URL: http://mathworld.wolfram.com/MandelbrotSet.html (visited on 06/26/2019).

[87] *WPF - Mono*. URL: https://www.mono-project.com/docs/gui/wpf/ (visited on 06/26/2019).

List of Figures

1.3 Inner part of the Mandelbrot set, example of an Escape-time fractal. 7
1.4 *XaoS*, real-time CPU renderer with GUI. 10

2.1 Visualization of the three given fractals. 21

3.1 Alternative algorithms for visualizing the Mandelbrot set. 24
3.2 The two coloring methods for Newton fractal. 27
3.3 Illustration of the definition of the rendering problem: pixel p is mapped to dark-red color. 31
3.4 Rasterisation: what should be the color of each of the 8 pixels? Below are pixel colors based on only one sample, taken in the pixel's center. 33
3.5 Examples of sample distribution methods within a pixel. 34

4.1 Illustration of chaotic regions of a function. 42
4.2 Adaptive supersampling: illustration of possible number of samples per pixel and sample distribution. 43
4.3 Illustration of sample reprojection when reusing samples. 48
4.4 Human vision 51
4.5 Foveated rendering: Mapping of the visual angle to sample count. 52

5.1 Illustrative screenshots from the perception study, study example 3. 57
5.2 Adaptive supersampling on Mandelbrot example 4 58
5.3 Adaptive supersampling on Mandelbrot example 5 58
5.4 Adaptive supersampling on Julia example 4 58
5.5 Adaptive supersampling on Newton iterations 59
5.6 Granularity of adaptive supersampling, illustrated on Julia example 4. Both images have the same render time. maxSuperSampling is 8. 59
5.7 Comparison of Mandelbrot example 4 with and without adaptive supersampling, with maxSuperSampling set to 32. 60
5.8 Examples of *chaos-ultra* output. 63

A.1 Program architecture 75
A.2 A texture rendered on CUDA, with all pixels within a warp colorized with the same color. Scaled up for clarity. Left: canonical cuda warp, right: our rectangular warp. 78
A.3 Mapping of image indexes to warp index and vice-versa. 79
A.4 Class structure of the `cudarenderer` package. 82
A.5 class structure of the `rendering` package 84
A.6 Sequence of multiple fractals management 85
A.7 `FractalRenderer` lifecycle 86
A.8 `GLRenderer` lifecycle sequence 87
A.9 `rendering` package data flow 89
A.10 States and transitions of the rendering mode state machine 90
A.11 Class structure of the GUI, including gui-component hierarchy. . . 91