FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

GPU Based Rendering of Complex Fractals in Three Dimensions

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Clemens Rögner
Matrikelnummer 0825045

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn Michael Wimmer
Mitwirkung: M.Sc. Thomas Auzinger

Wien, 12.11.2012

(Unterschrift Clemens Rögner) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

GPU Based Rendering of Complex Fractals in Three Dimensions

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Clemens Rögner
Registration Number 0825045

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn Michael Wimmer
Assistance: M.Sc. Thomas Auzinger

Vienna, 12.11.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Clemens Rögner
Ottakringer Strasse 215/3/7, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Clemens Rögner)

Acknowledgements

I would like to thank Günther Voglsam for his CUDA-support and Ralf Habel, as well as Thomas Auzinger for their professional input and support while writing this paper and for giving me the possibility to do this as my bachelor thesis.

Abstract

Fractals are mathematical sets with interesting features but without clear mathematical definition. They are infinite-detailed and have self-similar patterns, which makes them well-suited for rendering and artistic purposes.

A common problem when rendering fractals is, that choosing an appealing camera position and path is difficult due to the delay generated by computing the fractals. Furthermore choosing the right lighting for a scene is time consuming as well and therefore we propose an interactive application to generate visual interesting images of fractals to solve those issues.

In this paper we describe the theory behind rendering fractals and an approach to the problem of interactive fractal rendering. Our approach consists of a ray caster, which uses the CUDA-API as well as real-time rendering techniques to give interactive previews of fractals. We also provide a high quality renderer to generate detailed images of user-generated camera paths and lighting conditions.

To enhance the visual quality of the images, we implemented effects such as High Dynamic Range rendering (including a tone mapper), a bloom effect and a global illumination approximation. We also present our solution of calculating the normal vectors, which is essential for evaluating the light transport on the fractal.

Contents

CHAPTER 1

Introduction

Fractals are used to create terrain and other nature like objects as well as objects for creating visual appealing images. Because of their infinite complex geometry with fractal structures on every scale, fractals are a worst case scenario for rendering. The infinite details of their structure will always lead to aliasing artifacts at some point. Therefore extra computational power is needed to reduce aliasing and increase the quality of the resulting image.

This results in increased rendering time. When it comes to exploring a fractal, changing the rendering environment (the light setup, reflectance properties, etc. for fractals in three dimensional space or coloring techniques for two dimensional fractals for example) or generating a camera flight/zoom, a delay between the users action and the corresponding image generation makes an exploration of the available parameter space time consuming. We observe that the recent improvements in general-purpose computing on graphic processing units (GPGPU) and the increased computing power on graphics cards are sufficient enough to achieve ray casting of fractals at interactive frame rates.

This paper presents our implementation of an interactive fractal renderer using common real-time rendering techniques and hardware. We provide the users with various lighting modes and progressive refinement to give them the possibility to choose between faster rendering and more realistic results. Furthermore we include visual effects to further increase the quality of the generated images, which are described in chapter 4.

In the following section we describe the theory behind fractals and relate to previous work. In chapter 3 we describe the problems of rendering fractals and our approach to solve them. A detailed description of our implementation is given in chapter 4. In chapter 5 we conclude our work and in chapter 6 we give an outlook on how to extend it.

CHAPTER 2

Fractal Theory

2.1 Introduction to Fractals

In mathematics it is not clearly defined which properties qualify an object to be a fractal, but they share various properties as described below:

One feature that is associated with fractal geometry is that they are usually self similar, which means that a part of an object matches the object itself exactly or approximately. Some fractals even have the same shape under every scale, a property called scale invariant. This feature is also the reason why fractals are often described as “infinite detailed“ [5].

Non-fractal objects (without infinite detail) can always be represented by a linear approximation of them, which converges to the correct solution with increasing quality. On the other hand fractal objects can not be represented with such an approximation that would converge with increasing accuracy to the correct solution [5].

A common misconception when it comes to fractals is, that they do not have a full integer dimensionality, but rather fractional dimensionality (in other words, dimensionality that lies between two integers). For example the Mandelbrot set has the Hausdorff dimension of 2 as proven by [12] (consult the next section for more information on dimension).

One should note, that fractals are not only bound to be geometric object, but can also stretch for example over time or any other space.

2.2 Dimension

As mentioned in the section above, an object’s dimensionality is not an intrinsic property to classify a fractal amongst other objects. When it comes to fractals the dimension of an object is of interest, because objects with fractional (non-integer) dimension are often fractals and the dimension of a fractal is not intuitive. Because of this we give the reader a short introduction on the topic of dimension in this section.

In mathematics a dimension of an object is an intrinsic property of the object defined by the minimum number of independent values, that are needed to describe any point on the object so that it is continuous and continuously invertible. For fractal sets the definition of a dimension is more complex compared to non-fractal geometry like lines, circles, cubes etc., due to the fact, that fractals are not continuous and not differentiable. One can easily show, that a fractal line segment would converge to filling up an area if the calculation of it is repeated infinitely times. This leads to the discovery that fractal sets behave more like a volume than a shape. A way to describe such complex objects dimensionality was introduced by Felix Hausdorff [4].

The general idea of the Hausdorff dimension is to estimate the dimensionality by covering the border of an object with any shape, that has notion of a radius (a circle in two dimensional space, a sphere in three dimensional space etc.). Such a covering is shown in figure 2.1. The Hausdorff dimension is calculated by using the relation between that radius and the number of objects with that radius needed to cover the border. The radius used to cover the border stays the same and no point of the objects border should be uncovered, even if this means, that shapes are overlapping. Furthermore should the number of shapes with a specific radius be minimal, meaning not a single of those shapes should be redundant. This implicates, if the radius goes to zero, the coverage of the border will get more exact and therefore the correctness of the approximation to the Hausdorff dimension increases.

To put this idea into a mathematical formula the covering of the border N objects of the size S is defined as a function $N(S)$. The Hausdorff dimension D_H is than defined as:

$$N(S) \sim \frac{1}{S^{D_H}} \tag{2.1}$$

This leads to:

$$D_H = -\lim_{S \to 0} \frac{\log(N(S))}{\log(S)} \tag{2.2}$$

2.3 History

Benoit B. Mandelbrot first mentioned the term ffractalïn 1975 [5]. Before that they were called mmathematical monstersäs seen in figure 2.2. Such mmathematical monstersäre for example:

1. The Cantor Set, introduced by George Cantor in 1870, by given an abstract description of a set, that shows fractal properties [4].

2. The Sierpinski Triangle is named after Waclaw Sierpinski who mentioned it in 1915 as part on his work in set theory, similar to George Cantor [5].

3. The Koch Snowflake was introduced by Helge von Koch, a Swedish mathematician. Koch described the Snowflake fractal to give an example of a curve without any tangents [14].

Another important development for the field of fractals was the introduction of complex numbers. Gaston Julia and Pierre Fatou both independently showed the applicability of complex

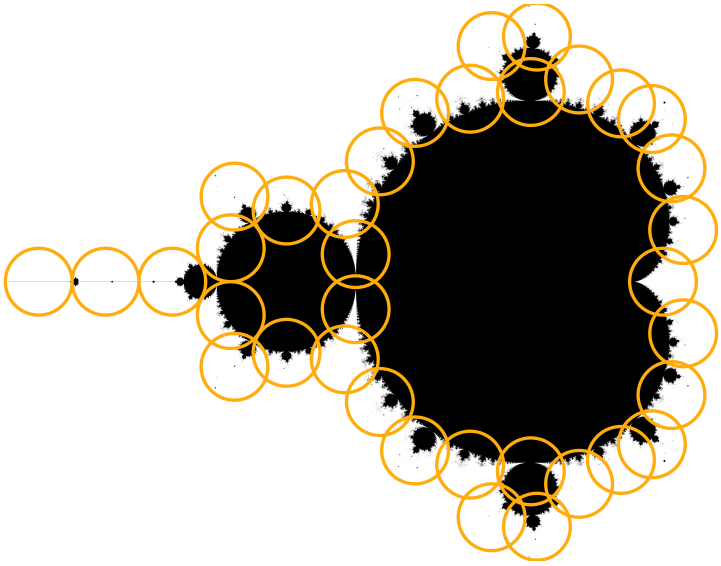


Figure 2.1: A possible covering of the border of the Mandelbrot fractal with circles. This image was created by hand and has 38 circles with a radius of roughly 0.0526.

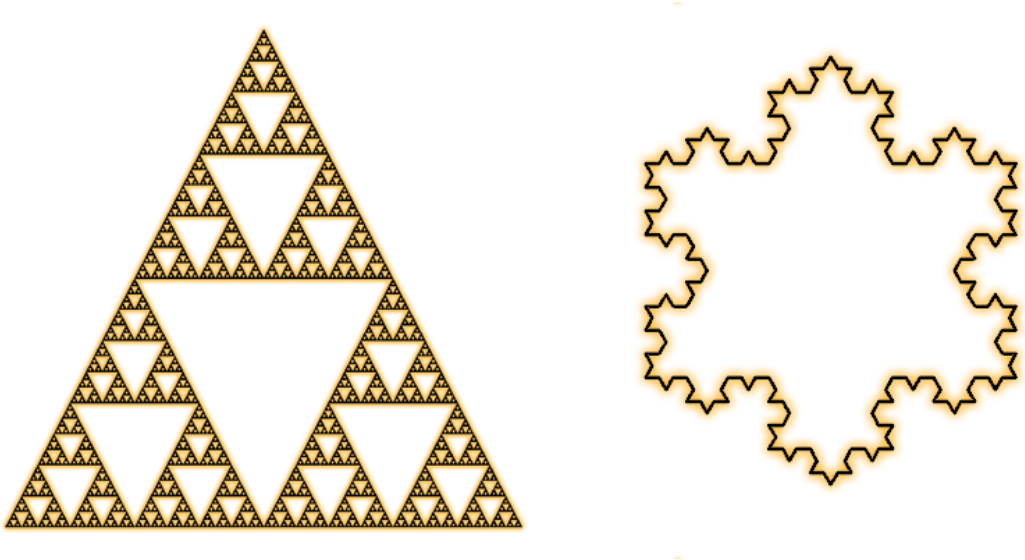


Figure 2.2: "mathematical monsters": The Sierpinski Triangle on the left and the Koch Snowflake on the right.

numbers for fractals. After their work was submitted, Felix Hausdorff introduced the previously mentioned Hausdorff dimension in the same year [4].

2.4 Classification

In this section we provide a subjective classification to give the reader an overview of the various fields in which fractals are used. Other ways of classifying fractals are valid too.

Non Deterministic

Non deterministic fractals always include some random element. They are often used to model natural phenomena such as clouds, terrain, fluids etc as seen in Figure 2.3.

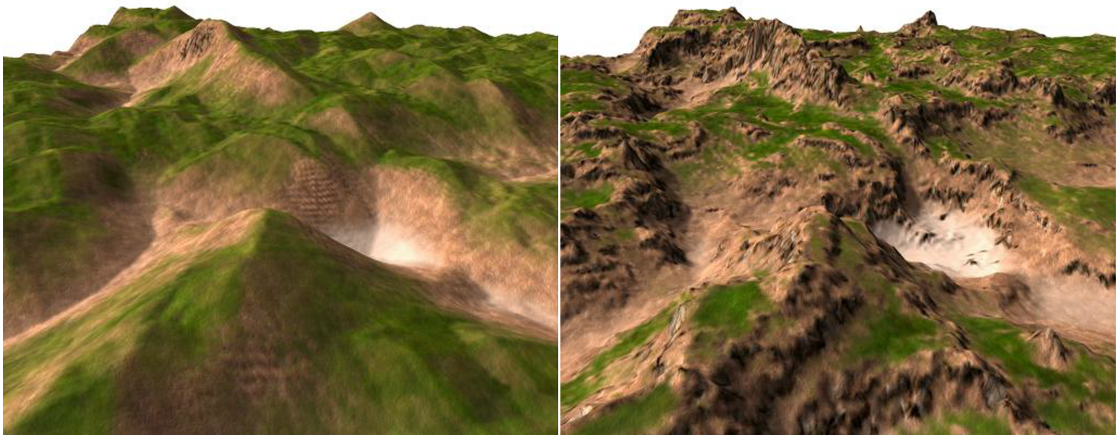


Figure 2.3: Images of terrain created with fractal algorithms. The left terrain was created with less iterations than the right one. Images taken from [8]

Deterministic

Deterministic fractals on the other hand are always the same under the same parameters. There are two main types of deterministic fractals:

Linear deterministic fractals are Iterated Function Systems, such as the Cantor Set, Koch Curve and Sierpinski Gasket. But also the Lindenmayer systems, which are commonly used to simulate the growth process of plants, are linear deterministic fractals. [9]. An example is shown in Figure 2.4.

The non linear fractals, on the other hand, always contain at least one non linear term in their equation. The Mandelbrot set and Julia sets are such non-linear deterministic fractals as seen in Figure 2.5.

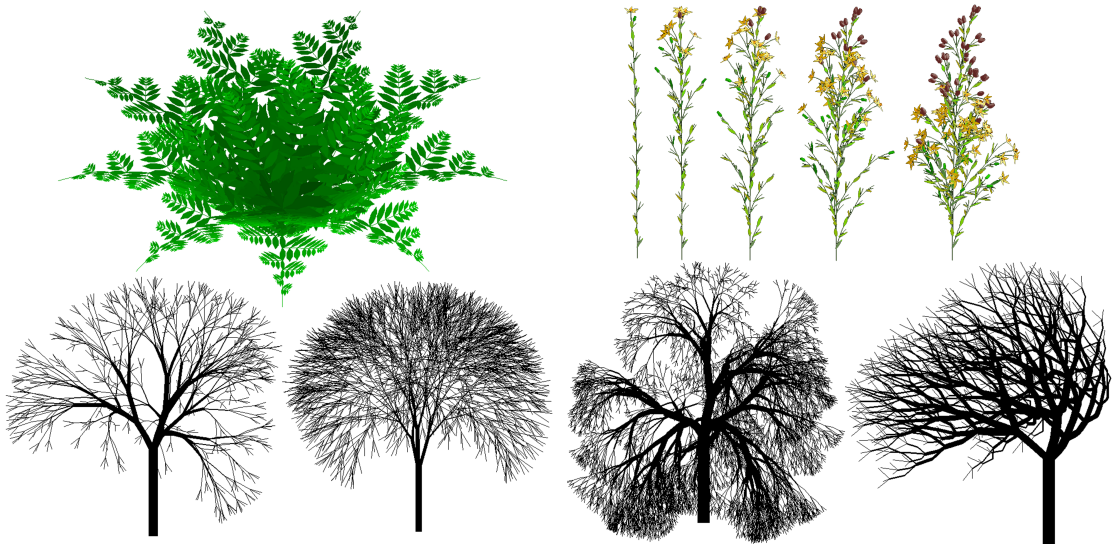


Figure 2.4: Plants and trees generated with Lindenmayer systems. Images taken from [9]

Artistic Fractals

As to this point we mentioned the mathematical background of fractals, as well as an short overview on how fractals can be used.

However, we observe that there is certain artistic interest on fractals due to their intricate shape. In particular fractals which make use of complex numbers or quaternions are used in the artistic purposes. We refer to those artistic complex fractals as Hyper-complex Fractals (which use complex numbers as well). Because of the creative potential of those fractals and the increasingly convenient way to visualize them, creative new formulas for fractals are created by various artists.

In our application we choose to use the formula called MMandelbulbäs our exemplary fractal. The Mandelbulb fractal was introduced by Daniel White in 2007 as an extension to the 2D Mandelbrot set [15].

2.5 Calculation of Fractals

In this section, we give a short overview over the various methods used to create fractals and give an example of four formulas to create fractals including the Mandelbrot and the Mandelbulb Set.

Methods

Fractals are estimated by a recursive feedback system, meaning that the function converges towards the correct fractal by increasing the number of iterations steps.

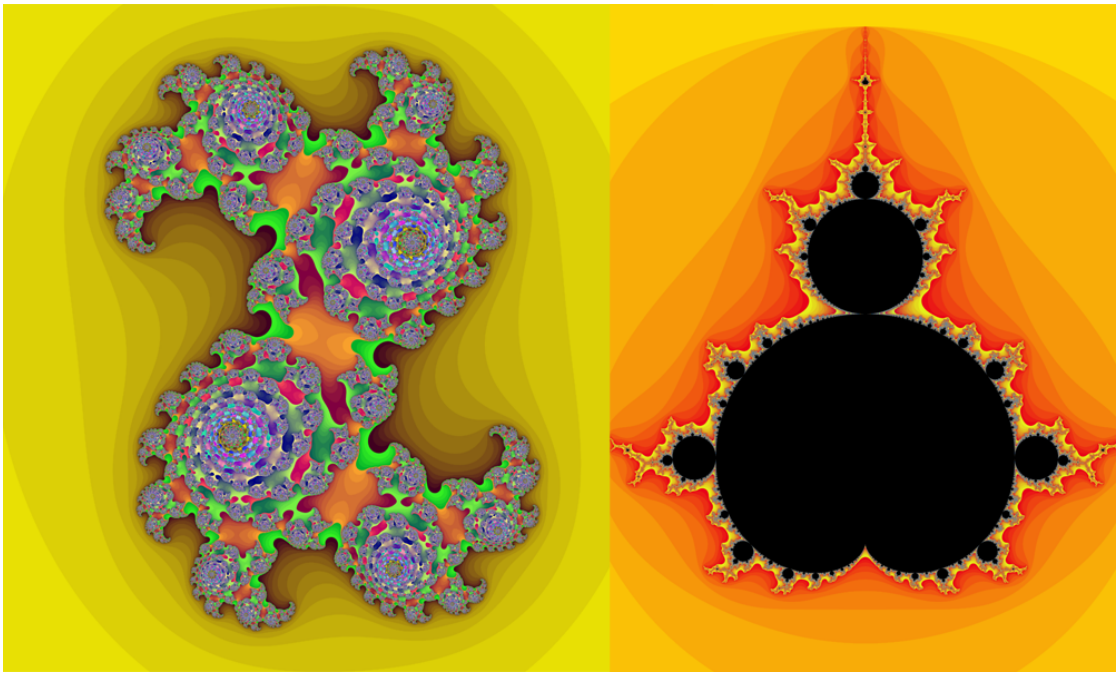


Figure 2.5: A colored Julia set on the left and a colored Mandelbrot set on the right. Images created with the NVIDIA GPU SDK 4.0 Sample Mandelbrot.

The following list contains a selection of the methods of how those iteration steps can be realized:

1. **Iterated Function Systems** are a formal description for constructing linear fractals. An Iterated Function System replaces geometry with other geometry (which allows further replacement). Examples of Iterated Function Systems include the Cantor Set, Koch Curve, Sierpinski Gasket etc.

2. **L-Systems**(named after their introducer Aristid Lindenmayer) are rewriting systems operating on strings. A L-System is defined by finite set of symbols, an initial word and a finite set of productions. L-Systems replace the symbols parallel. Furthermore L-System allow context sensitive production, parametrized production and various other extensions introduced by Prusinkiewicz [9]. To render a graphic, the resulting string of a L-System has to be interpreted by a drawing program. Fractals, that can be generated by an Iterated Function System, can be drawn by L-Systems as well.

3. **Escape-time fractals** use a particular function as a sequence. Wether a start-value is part of the fractal set is determined by checking if the point is within a preset boundary after a certain number of iterations. Such Escape-time fractals are for example the Mandelbrot-Set or the Julia-Set. [5]

4. **Strange Attractors** To generate a fractal, the chaotic behavior of dynamic systems can be used. Chaotic behavior means that the result of a system etc. changes drastically, when altering the starting conditions only minimally. Some chaotic systems have an attractor(the set of system states to which the system evolves after sufficiently long time) that exhibits fractal nature.

5. **Stochastic Rules** The Levy flights, Brownian motion or other stochastic models can be used to generate fractals as well. [13]

Sierpinski Triangle

This section describes one of the various methods to generate the Sierpinski Triangle.

The first step to generate Sierpinski Triangle is to create a triangle. Followed by:

1. Scale down the triangles height and width by 2 and copy it twice.

2. For each vertex of the original triangle, place one copy of the scaled triangle with the corresponding vertex on it.

Those steps have to be repeated for each triangle.

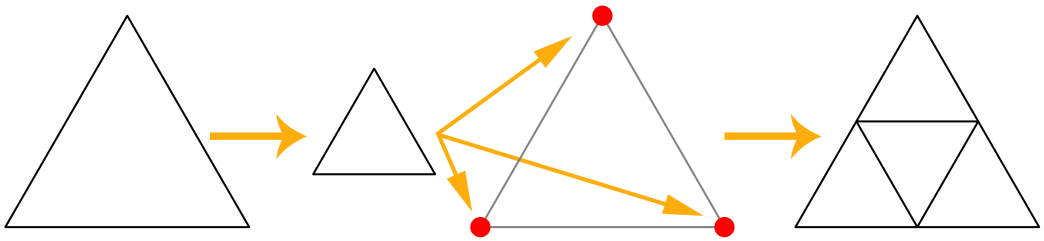


Figure 2.6: The steps to generate a Sierpinski Triangle.

Koch Snowflake

Similar to the Sierpinski Triangle, described above, the following description is just one of many ways to generate the Koch Snowflake.

The starting point for our method is to generate a equilateral triangle. Then the following steps for each line have to be done:

1. Divide the line into three equal pieces.

2. Add two lines on top of the middle piece from step 1, so that an equilateral triangle (with the edge-length of that middle piece) is created

3. Remove the middle piece from step 1.

Those steps have to be repeated for each generated line.

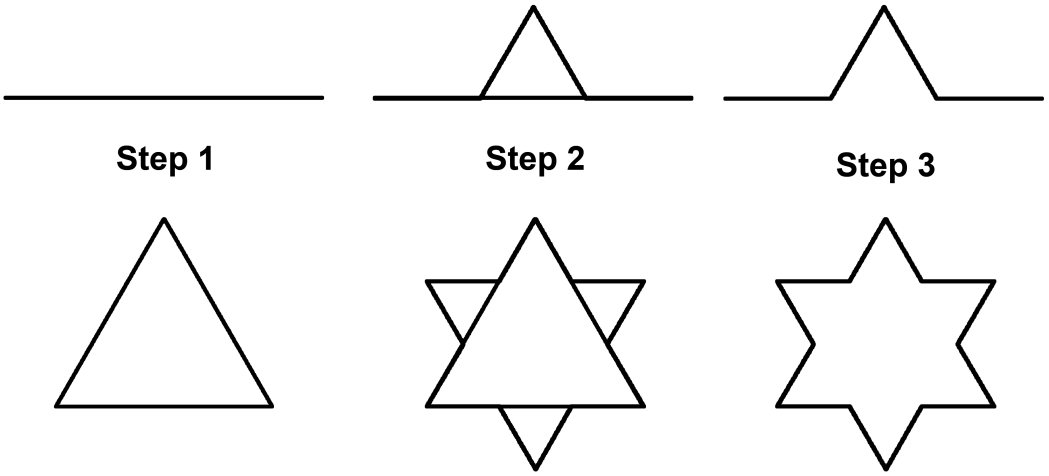


Figure 2.7: The steps to generate a Koch snowflake. On the top this is done for one line segment and on the bottom it is done for three line segments of a triangle.

Julia Set

To understand the Mandelbrot fractal, a first look on Julia Sets of the sequence $f_c = z^2 + c$ has to be taken. In this sequence, the starting point z_0 lies on the complex plane on which the Julia set is mapped and c is an arbitrary constant complex number. As in every Julia set, the sequence has to be tested if it converges to a fixed point for a specific starting value z_0. When this test succeeds, then the point z_0 is part of the corresponding Julia set.

Like every mathematical set, a specific Julia set can be a disconnected or connected set. To see if a Julia set is connected, the sequence $f_c = z^2 + c$ with the starting point $z_0 = 0$ has to be tested if it escapes to infinity (disconnected Julia Set) or not (connected Julia set). This property of connectivity is essential for the Mandelbrot set, as described in the next chapter.

Mandelbrot Iteration

The basic idea of Mandelbrot was to see if the Julia set is connected for a specific c in the complex plane. We use the term complex plane to describe a two dimensional space, where the real part of a complex number is represented via one dimension in this space and the imaginary

part of the same number is represented by the other dimension. This complex plane directly corresponds to the mapping used in visualizations of the Mandelbrot Fractal.

To determine whether a point is in- or outside of the Mandelbrot set, the convergence of the particular point has to be calculated by starting the iteration $z_{n+1} = z_n^2 + c$ with $z_0 = 0$ and the c as the point of interest. If $|z| > 2$ is true, the iteration can be stopped, because the corresponding Julia set is disconnected. In other words, a divergence can be detected in every iteration step. An example of this iteration is given in Figure 2.8. The resulting visualization gets more accurate as iterations increase.

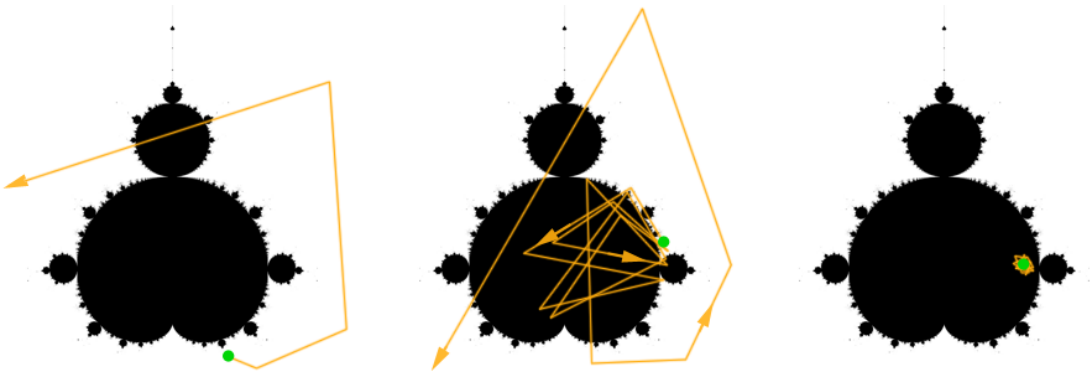


Figure 2.8: The convergence of a point (marked in green) in the Mandelbrot set. The first two images show divergent iteration steps. The last one is a convergent iteration step.

Mandelbulb Iteration

The operations on the complex numbers in the Mandelbrot formula can be seen as operations on two dimensional vectors. Daniel White suggested to port the Mandelbrot fractal into the three dimensional space, to do those same operations with three dimensional vectors as well [15].

The complex multiplication done in the Mandelbrot formula is equivalent to a rotation of this number (seen as a two dimensional vector) in the complex plane. In his formula, Daniel White rotated the three dimensional vector, which represents the former complex number, along its spherical coordinates. The movement of such a three dimensional vector can be seen in Figure 2.9a and will be explained later in this chapter.

After Daniel White experimented with various spherical coordinate systems and multiplication factors for those coordinates, Paul Nylander came up with the idea to use a higher power than 2 when it comes to moving the point by the original distance as seen in Figure 2.9b (d^w represents the scaled distance) [15]. This led to the exploration of the formula using the Power 8 by David Makin, which provided enough self similarity at high zoom levels in all three dimensions according to Daniel White [15].

The mathematical equation for the Mandelbulb (with the iteration depth of n) is:

$$\{x, y, z\}^n = r^n\{\cos\theta\cos\phi, \sin\theta\cos\phi, -\sin\phi\} \tag{2.3}$$

where

$$r = \sqrt{x^2 + y^2 + z^2}, \theta = n * atan2(y, x), \phi = n \sin^{-1}(z/r) \tag{2.4}$$

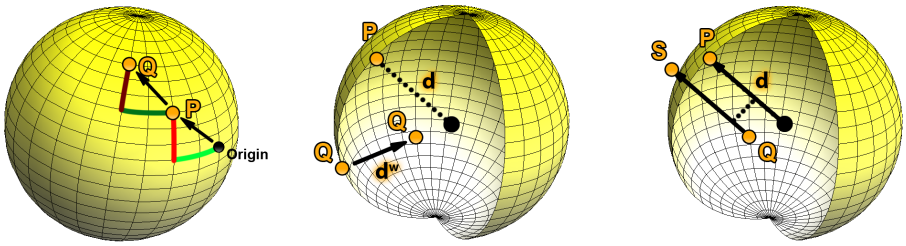
The algorithm for the Mandelbulb Iteration is described below:

1. At first we have to rotate the three dimensional point P around the origin by the same degrees as P is already angled from the origin (in other words: double the angle), which generates a new point $Q(q_x, q_y)$. This happens in spherical coordinates, as shown in Figure 2.9a.

2. Then the distance from the origin to the point P has to be calculated: $d_p = \sqrt{p_x^2 + p_y^2 + p_z^2}$. The point Q, which can be seen as a vector from the origin as well, then has to be scaled towards the origin relatively by the amount of d_p^w as seen in Figure 2.9b. This step is equal to a complex multiplication. The parameter w is 8 for the Mandelbulb, but any other number greater than 2 can be used for experimenting with certain shapes of the bulb.

3. This altered point Q now has to be increased by the value P, creating a new point S (in other words: a complex addition has to be performed). This step is shown in Figure 2.9c.

4. The Steps above now have to be repeated over and over again, but with the point S as the starting point P in Step 1. But the distance calculated in Step 3 has to be the distance from the original point P. If the calculated point S is inside a sphere with radius 2 after a specified amount of iterations, the point P is part of the Mandelbrot set.



(a) Step 1 (b) Step 2 (c) Step 3

Figure 2.9: Graphics describing one Mandelbulb iteration step.

CHAPTER 3

Rendering Fractals

The complexity of fractal geometry demands computer graphics to visualize and investigate its objects, because the amount of calculations can not be done by hand. In this section the basics of rendering those objects are described.

3.1 2D Fractals

As shown in Figure 3.1 the Mandelbrot set spreads from -2 to 1 on the axis of the real component and from -1 to 1 on the imaginary component axis. To determine where certain phenomena appear, the cardioid and bulb like shapes, the hyperbolic components, on the Mandelbrot figure are ordered by their size.

In general, coloring the distance from a pixel to the fractal, is the most common visualization method for this kind of fractals. The distance can be calculated as mentioned in 3.1.

Limitations

The limitations of rendering 2D-fractals are caused by finite rendering time, inaccuracy of data types and arithmetical operations. The computational time depends on resolution, used data types and number of iterations. The last one changes the shape of the Mandelbrot set by making it more accurate and therefore adding more detail to it. Increasing zoom levels require more detail, because at some point fractal shapes do not appear anymore. Accuracy of data types on the other hand, limit the possibilities to zoom into the fractal as shown in Figure 3.2.

Coloring

In most cases, the points, which are not in the Mandelbrot, set are colored by the number of successful iterations. This requires either color maps with the size of the maximum number of iterations or a cyclic mapping of a color-set.

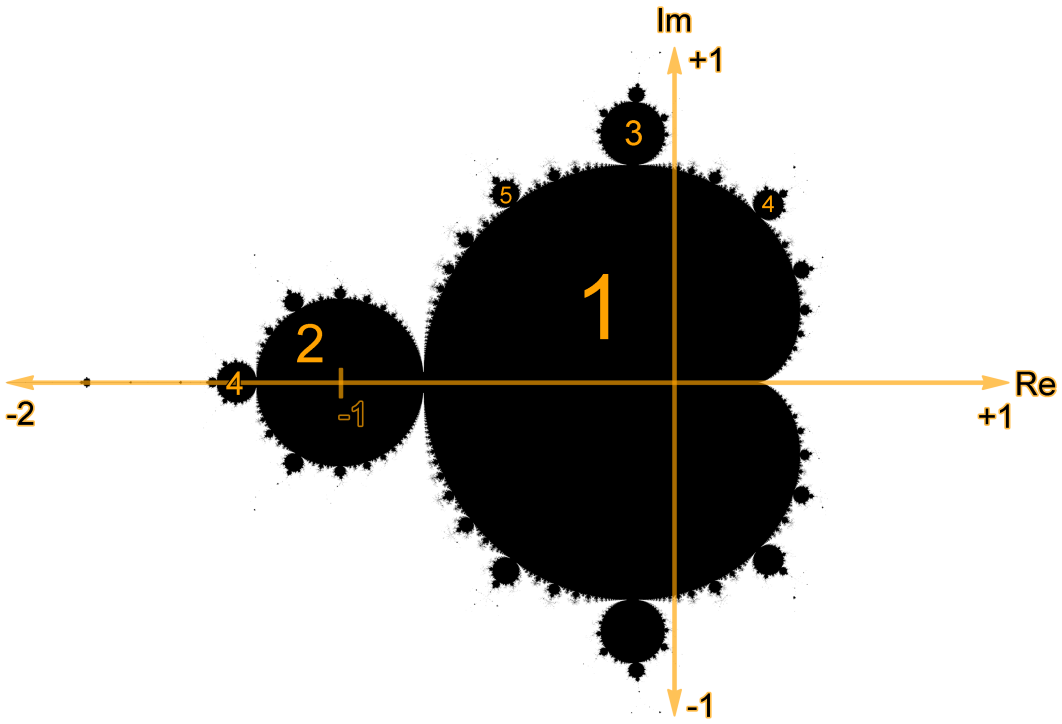


Figure 3.1: The Mandelbrot set. The orange numbers are a subset of the periodic hyperbolic components.

To hide the banding effects, that occur when using one of the mappings above, an artistic approximation for such continuous coloring can be used. Of particular interest is a formula called Normalized Iteration Count [3]:

$$color = Num_{iter} - \frac{log(\frac{log(d_p^2)}{log(4)})}{log(2)} \tag{3.1}$$

Note that this formula is just one possible way to give a continuous color gradient.

3.2 3D Fractals

Rendering 3D-fractals suffers from the same limitations as 2D-fractals. In addition, we have to take problems into account that come along with rendering three dimensional objects. In the following sections we describe those problems and our approach to solve them.

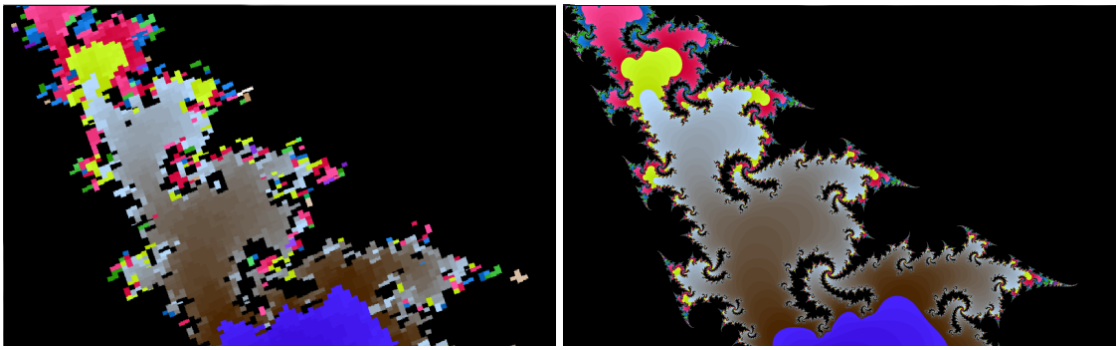


Figure 3.2: A deep zoom on a Julia Set with 32 bit floating point values on the left and 64 bit double precision floating point values on the right. Images created with the NVIDIA GPU SDK 4.1 Sample Mandelbrot [7].

Theoretical Background of Rendering 3D Fractals

To visualize a three dimensional fractal, it has to be mapped onto a two dimensional plane (an image) in order to show it on 2D display. Therefore one dimension of information for the observer (the human eye) is lost during this process, which would be equivalent to rendering a 2D fractal onto a (one dimensional) line.

However, it has not yet been investigated how light travels through objects with surfaces like the Mandelbulb, which are not only of fractal nature but also irregular structured unlike a Sierpinski cube etc.. We suggest that light travels through an non-virtual Mandelbulb (or similar fractal structures), that is created at atom-level detail, like it would through translucent materials, in other words: light scatters through that structure. Our suggestion is based on the following observation: when light from a source outside of the Mandelbulb hits one point at the surface on it, it may or may not exit the fractal object. To be more specific, the light can bounce infinity times on the Mandelbulb until it is fully absorbed. It could also bounce of several times (or one time) and escapes the Mandelbulb with decreased intensity (corresponded to the number of bounces). Furthermore it is also possible that the light travels past the objects structure without hitting the fractal itself. The chance of light getting absorbed or decreased in intensity gets higher as the light travels deeper into Mandelbulb, since the density of the structure increases with increasing depth. This behavior of light travel is similar to those in translucent materials, which leads to our idea of scattered light transport within the Mandelbulb.

Rendering scattering materials in real-time is complicated to achieve. In our application we choose not to render the scattering effect at all, because we find that a diffuse light reflection model is sufficient enough to generate visual appealing images and furthermore is inexpensive to compute. In particular we choose to use the Blinn - Phong shading model in our application.

Discretization Method

When using a voxelization or triangulation or any other pre-calculated discretization, two major disadvantages have to be taken into account:

1. The quality of zooms and in general the detail of the visualized surface are preset by the calculation before the rendering and the camera movement starts. An adaptive refinement of the pre-calculated surface has to be implemented to overcome this disadvantage.

2. The amount of memory transfered on the GPUs data buses is higher compared to ray casting, potentially decreasing performance. See Section 4.4 for more details.

Therefore we choose an ray casting approach [16] over a pre-calculated discretization, due to the better handling of close zooms and easier implementation of adaptive optimizations compared to the methods mentioned in the previous paragraph. The ray casting algorithm is explained in the section 4.1.

Calculating the Normal Vector

The third dimension requires the shading and lighting of the surface so that the human eye can perceive the structure of the fractal. Using a Blinn - Phong shading model, or any other BRDFs (a shading and lighting model) for that purpose, requires a normal vector of the surface, in other words: a statement about the surfaces structure has to be made. Since a fractal surface is infinite detailed an approximation of its real structure is inevitable.

Our approach is to sample the surrounding area of the fractal on a regular grid. We are testing several sampling patterns within this grid and evaluate the best method by weighting the image quality against the performance costs (which increases when using more samples). As a reference image we use figure 3.3a. It was created by leaving out the normal vector from the lightning calculation to see which areas could be potentially lit. The graphic below shows the sampling patterns, followed by our observations on the result by an example scene.

a) the reference image.
b) only 3 sample points distributed on the positive main axis of the grid. As someone can observe large areas in the middle image, that should be lit but are not (they should be lit, because they face the same direction as other lit ones).
c) 6 sampling points are distributed on the main axis. The large areas from the previous sample are lit, however small areas (as seen on the right image) are still unlit.
d) has 18 sample points. The result is similar to c, but the size of the small dark spots increased.
e) 26 sample points bring similar results as d).

We decide to use option c), as it provides not only the best image in terms of aesthetics, but also uses less sampling points as d) and e) and is therefore faster than those two options.

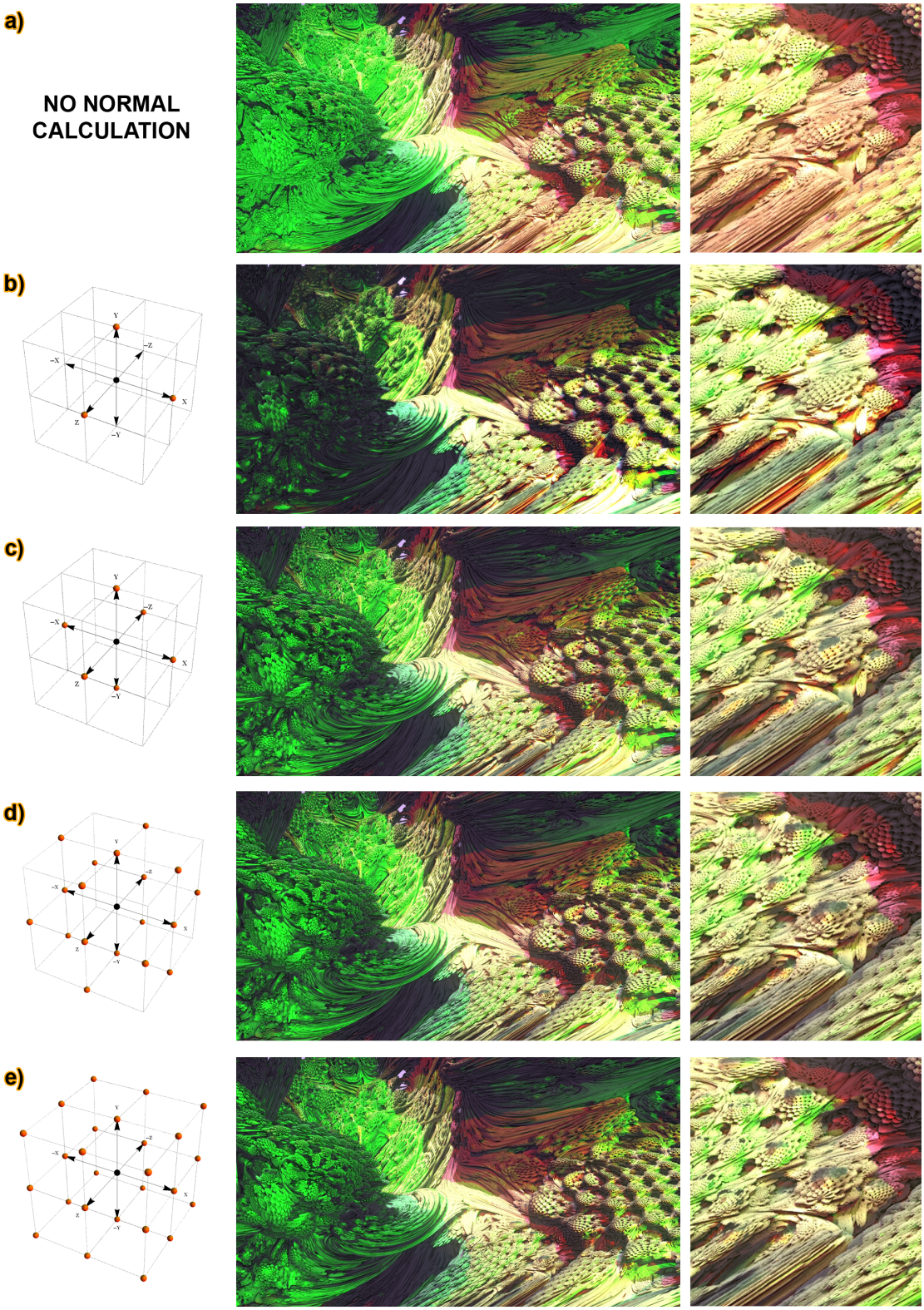


Figure 3.3: The test cases for the normal calculation. Left Side: sampling pattern of the normal calculation. The orange dots are the sample points. Middle: result of test scene. Right Side: closeup of the result in the test scene.

CHAPTER 4

Parallelized Ray Casting of Fractals

To speed up ray casting of fractals we use specialized processing units, the Graphics Processing Units (GPUs). In this chapter we explain the details of our implementation on the GPU.

4.1 GPU Ray Tracing

GPUs are in general used to increase performance when rasterizing and texturing triangles. Their architecture can be described with the key-attributes “Single Instruction, Multiple Data, High Parallelism“.

The common APIs to run custom Code on GPUs, such as OpenGL and DirectX, offer too few options and have too much overhead for the requirements of our real-time application. In particular those APIs don’t provide enough flexibility when it comes to loop and double precision support, and in addition are bound to the rasterization pipeline.

The advantages of general-purpose computing on graphics processing units (GPGPU) APIs, are the reason why we decide to use this approach instead of other programming interfaces and methods. The implementation is realized with Nvidias CUDA APIs on GPUs with the Fermi architecture.

4.2 Basic Approach for Fractals

To render the fractal, its location has to be calculated. This is done by using its distance function as in listening 1. With this formula, stepping along the viewing ray as shown in Figure 4.1 can be done until a point is reached, that is close enough (Epsilon-area) to the fractal.

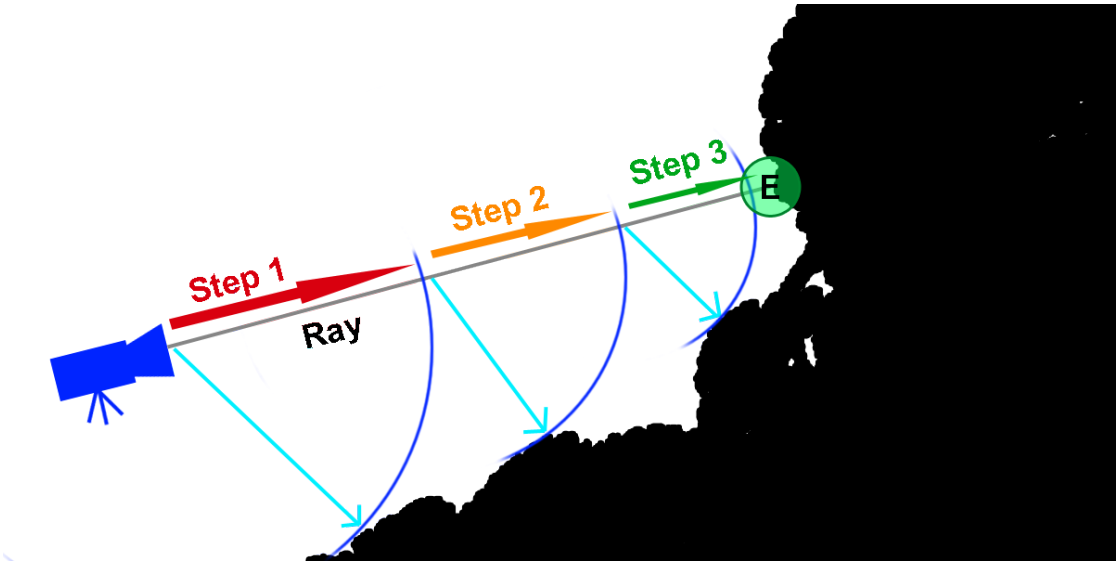


Figure 4.1: The concept of ray-stepping. At each step, the distance to the 3D fractal is estimated and added to the viewing ray until it reaches a distance close enough (E - the Epsilon area).

4.3 Advantages of GPU Architectures

Pixel Parallelism

The main advantage of the GPU-Architecture is, that the parallelized data processing happens with a higher thread-count compared to the common CPU-architecture. But unlike the CPU architecture, the GPU is not as flexible on the instruction set of different threads, because of two reasons:

1. Each thread on the GPU is running with the same source code, but on the CPU each thread, that runs in parallel, could potentially have entirely different source code.

2. The GPU is divided into smaller processing units (similar to the CPU). Those so called Single Instruction Multiple Data (SIMD) units require the same instructions set for each thread that runs on it. Because of this, threads that already finished computing on one SIMD unit, have to wait for the last thread to finish on this SIMD. CPUs, in contrast, feed every thread with a unique instruction set on each kernel and no thread has to wait for another to finish.

An schematic overview of the different workflow between CPUs and GPUs for the ray casting of fractals is given in figure 4.2. The high number of parallel threads is the reason why GPUs are better suited for rendering fractals than CPUs as described in 4.4.

Algorithm 1 Calculate fractal distance for the Mandelbulb

Require: $point, maxiter, power, divergence$

$z \Leftarrow point$
$r \Leftarrow length(z)$
$dr \Leftarrow 1$
$i \Leftarrow maxiter$
while $r \leq divergenceandi > 0$ **do**
  $i \Leftarrow i - 1$
  $ph \Leftarrow asin(z.z/r)$
  $th \Leftarrow atan2(z.y, z.x)$
  $zr \Leftarrow pow(r, power - 1)$
  $dr \Leftarrow zr * dr * power + 1$
  $zr \Leftarrow zr * r$
  $sph \Leftarrow sin(power * ph)$
  $cph \Leftarrow cos(power * ph)$
  $sth \Leftarrow sin(power * th)$
  $cth \Leftarrow cos(power * th)$
  $z.x \Leftarrow zr * cph * cth + point.x$
  $z.z \Leftarrow zr * sph + point.z$
  $r \Leftarrow length(z)$
end while
$return \Leftarrow 0.5 * log(r) * r/dr$

Floating Point Operations

Current GPUs are optimized for executing single floating point arithmetic operations, but not for double precision floating point operations. To improve performance for the former, the CUDA API offers an option to speed up floating point computations at the price of accuracy [6]. This option is used for the real-time approach, because the numerical inaccuracy is covered up by artifacts from the reduced sampling per pixel - another quality cutoff we made to achieve rendering in real-time.

Furthermore double-precision calculations are significant slower than the corresponding single-precision operations. This is not only because of the increased data transfer but also because of the lower amount of operations per clock cycle for one multiprocessor for double precision arithmetics compared to single precision arithmetics (consult chapter 5.4.1 in [6] for detailed information).

4.4 Challenges in GPU Based Fractal Rendering

Since fractals are infinite detailed, calculations on CPUs and GPUs always contain artifacts, caused by the finite accuracy of data types and limited computing time. But there are also some other difficulties and limitations when it comes to rendering fractals, which we describe below.

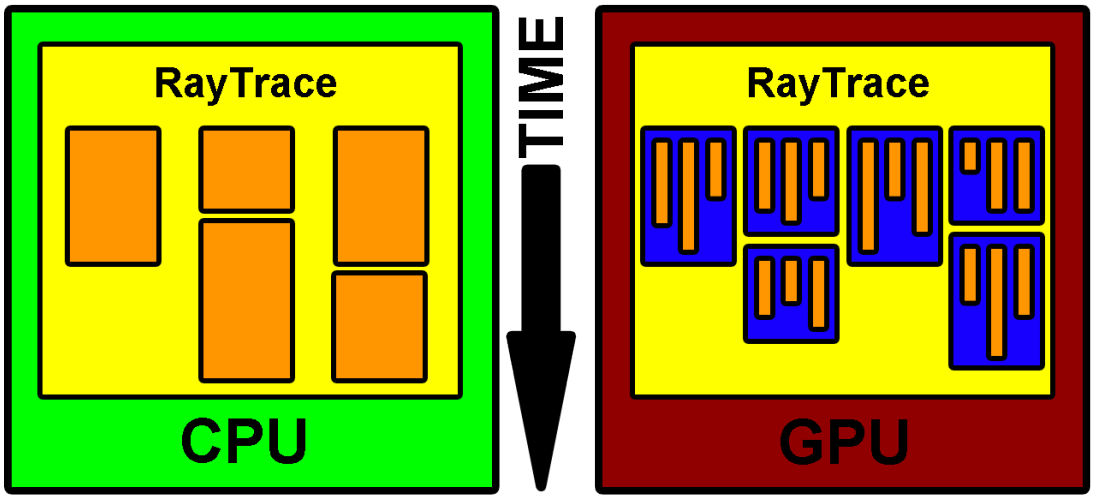


Figure 4.2: The different workflow of GPUs and CPUs. The orange rectangles represent one single thread. The blue rectangles represent the SIMDs of the GPU.

Limitations of the GPU-Architecture

As stated above the GPU's SIMD units execute only the same instruction set. This leads to a disadvantage of the GPU over the CPU when different operations have to be applied on threads on the same SIMD. Threads have to wait for each other in this case. This reduces the effectiveness of the GPU's architecture when loops are executed, that do not have the same number of passes. When it comes to calculating the distance to the fractal or stepping along the the viewing-ray, using loops with breaking conditions that cannot be resolved at compile time is essential. Because of this a great amount of computational time is unused.

Oversampling

To reduce aliasing artifacts, we oversample the fractal geometry. This is done by casting multiple rays for one pixel and averaging those samples to get the final value (see memory consumption for details). To further reduce aliasing artifacts, the casted rays are jittered with pre-calculated random values stored in a 2D-texture. For further informations about the anti-aliasing methods used, consult "Real-Time Rendering 3rd Edition" [1].

Memory Consumption

The memory needs are an important factor of the applications performance. To be more specific, increased data traffic along the GPUs data lines in our application slows down the rendering significantly.

In the implementation a similar approach to deferred shading is used for storing the sampling-points. This is done to accomplish a global illumination approximation called Image-Space Horizon-Based Ambient Occlusion (HBAO) [2], as described in section 4.5.

For each sampled point, the position, normal vector and a counter which increases if the sampled point is in the fractal or outside. We use this counter to tell if a point is inside or outside, which we need for the global illumination approximation described in 4.5. All three vectors of a sampled point are three-dimensional, which leads to a memory consumption of 37 byte per sampled point when using single precision and 73 bytes when using double precision.

The number of sampling points stored in the memory is equivalent to the resolution of the resulting image. For a Full-HD resolution of 1920x1080 and single-precision sampling this sums up to a total of ~73.5 MB and ~144.5 MB for double precision.

For calculating the final color and post processing effects a texture with the resolution of the final image and four channels red, green, blue and alpha, is needed. Each channel is represented by a 32-bit floating point value to enable High Dynamic Range output.

The tone mapper used in the application needs the average color of all pixels, which is done by calculating the mipmapping-levels. To do so, 1/3 of the storage of the texture mentioned before has to be allocated.

For a resolution of 1920x1080, the texture for the final image is ~32 MB and ~11 MB for the mipmap-levels. The texture and the mipmap-levels use single-precision floating point values in both precision modes. This makes up to a total of ~116.5 MB for single- and ~187.5 MB for double-precision to be available on the GPU, as shown in Table 4.1.

	float	double
Sample Points	73.5 MB	144.5 MB
Texture	32 MB	32 MB
Mipmap	11 MB	11 MB
Total	116.5 MB	187.5 MB

Table 4.1: This table shows the memory usage of our application

Real-time-Approach

To accomplish a real-time rendering, several cutbacks on quality in the exchange for speed are done. Those include single-precision computing, using the fast-math-option, doing no oversampling, lowering the resolution and using only a few lights.

Those restrictions allow around 3 frames per second on current hardware. To get more frames per seconds, progressive refinement, a technique used in visualization, was implemented to enable smooth navigation. Progressive refinement decreases the number of samples while moving. See Figure 4.3.

We also provide the user with the option to turn the shadow calculations off.

Performance

As mentioned before in our real-time rendering part of the application we get around 3 frames per second using 3 environmental lights, activated bloom effect (see chapter 4.5 for more) and tone mapping (see chapter 4.5 for more) on a still frame (without progressive refinement and no oversampling, in other words 1 sample per pixel). With progressive refinement activated we get around 16 to 25 frames per second an the NVIDIA Geforce GTX 470, whilst using 1 sample for 5 pixels.

As for the high quality render, a Mandelbulb image with 1920x1080 pixels, 16x oversampled, 3 light sources and per channel tone mapping, our application needs 135 seconds with single precision and 718 seconds with double precision to render on a NVIDIA Geforce GTX 470.

In both the real-time application as well as in the high quality renderer the frame rates are calculated in scenes where the entire screen is covered with the fractal. This is the worst case scenario for our application, because rendering time decreases, when samples for the image do not hit the fractal, due to thread-blocks can finish their work earlier.

4.5 Lighting / Illumination / Shading

When it comes to 3D-Objects, the human perception determines the structure of an object by its shading and lighting. In this section we describe our methods for approximating light transport and generating visual effects.

Environment Lighting

Environment lighting simulates the incoming light from every direction on an object. We store the incoming light in a list containing all lights with their direction and color. Every light can be interpreted as an directional light, which casts shadows.

To calculate the shadows for one particular light, a ray is casted to sample for intersections between the sampled point and the light source. These rays are casted to the sampled point from a point, that lies in the direction of the light, but is outside of the fractal.

The implementation uses deferred shading, which is known to be faster when it comes to multiple lights. In general the shadowing of a pixel is done after averaging the sample points. This would lead to diverging normals and positions as a result of the discontinuity of the fractal or sampling different parts of it. We therefore cast the rays for shadowing for each sample point and merge the results afterwards.

Point Lights

To reach parts of the Mandelbulb with light, that cannot be reached with the environmental light sources, point light sources are also part of the application. Each point light is defined by position and color. Similar to environment lights the shadows are calculated by shooting a ray towards the light source from every sampled point.

Global Illumination

We use the screen space approach to approximate the indirect lightning. This is realized by two different screen space operators in our application.

The first one uses a uniform sampling area and weights the color channels by the cosine of the normal vectors as shown in the following formula:

$$L_{indir}(P) = \frac{\sum_{y=-S}^{S} \sum_{x=-S}^{S} L_{dir}(x,y) dot(N(P), N(x,y))}{(S*2+1)^2} \tag{4.1}$$

This operator results in a continuous color gradient as shown in Figure 4.4, which does not give any clues about the structure of the fractal.

For a more detailed global illumination approximation we implemented Image-Space Horizon-Based Ambient Occlusion [2]. The technique was improved, after several tests, by dividing by the number of the samples, which are closer to the camera. The result is the second operator as seen in the listening 2.

Algorithm 2 Horizon Based Ambient Occlusion

1: $sp \Leftarrow getSamplePoint(pixelCoord)$
2: $curAngle \Leftarrow 0$
3: $a \Leftarrow 0$
4: $ao \Leftarrow 0$
5: **for all** $directions$ **do**
6: **for** $d = 1 \rightarrow numSamplesPerDirection$ **do**
7: $p \Leftarrow getSamplePoint(pixelCoord + d * direction)$
8: **if** $isNotBackground(p)$ **then**
9: $vec \Leftarrow p.pos - sp.pos$
10: $normalize(vecToPos)$
11: $angle \Leftarrow saturate(dot(sp.normal, vec))$
12: **if** $angle \geq curAngle$ **then**
13: $curAngle \Leftarrow angle$
14: $angle \Leftarrow 1 - angle$
15: $ao \Leftarrow ao + angle * Color(sp) * (ambientfactor + Color(p))$
16: $a \Leftarrow a + 1$
17: **end if**
18: **end if**
19: **end for**
20: **end for**
21: $ao \Leftarrow ao/a$

We considered the screen space approach to calculate the direct lighting in this application as well. To accomplish this, we implemented Screen-Space Directional Occlusion (SSDO) [11] for fast results. We rejected that idea, because calculating the directional occlusion with ray tracing

proofed to generate sufficient fast results for the needs of our application. In addition, the ray tracing technique does provide physically correct shadows without the artifacts, that occur in the approximation of SSDO [11].

High Dynamic Range Rendering

Since environment lighting uses a high number of light sources, we use High Dynamic Range (HDR) rendering to provide the user with the full spectrum of light intensity. In our application there is an option to directly return a HDR-image or to use a tone mapper to scale images values to a displayable range.

We decide to implement Reinhard's tone map operator [10] in our application for that purpose. We evaluate the average pixel-color for this operator by using mipmap-levels of the rendered image, which allows us to parallelize this process. After this calculation, each pixel is mapped to its new value (either for each color-channel separately, or by the pixels luminance value and we leave the choice to the user).

Bloom

To further improve the quality of the images produced, a bloom effect [1] can be applied on demand. Our application uses one of the mipmap-levels to realize this effect. To save performance the blurring is done in a horizontal and vertical step separately. The filtering is done by a uniform kernel. The bloom effect is applied after the tone mapping in the application. In figure 4.5 one can see difference between a scene with and without the bloom effect. An unintended effect of bloom is, that it gives back a small amount of saturation to the final image, which is lost after tone mapping.

Stereoscopic 3D

The high-quality rendering part of our application also features stereoscopic rendering. To accomplish stereoscopic vision, the renderer produces two images in two separate steps. The user is able to define the size of the screen, the distance between the eye and the screen and the distance between the two eyes. The size of the screen is defined by only one parameter in the application, the height of it. The width of the screen is calculated by the ratio between the pixel width and height. An example of stereoscopic images is given in figure 4.6.

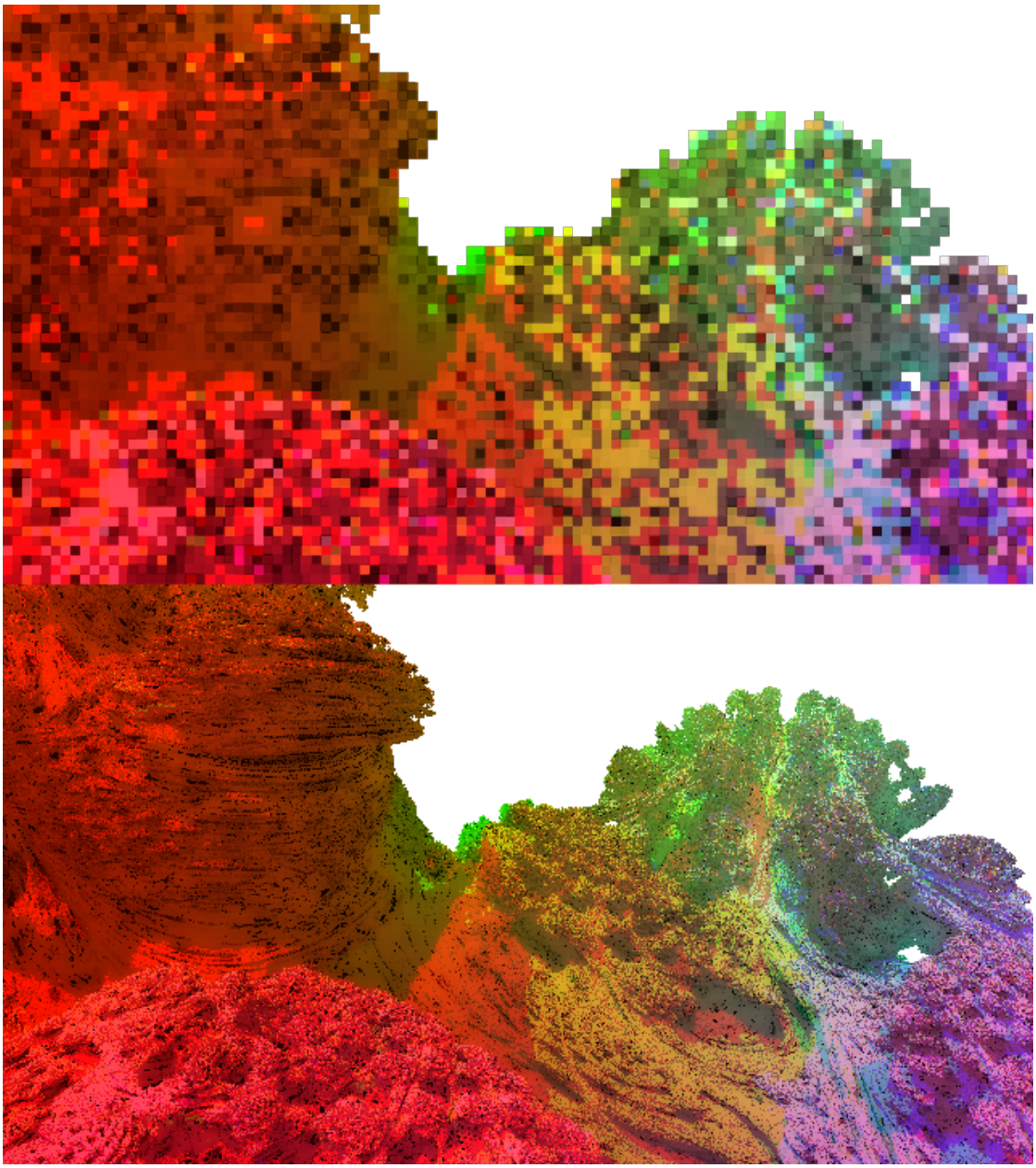
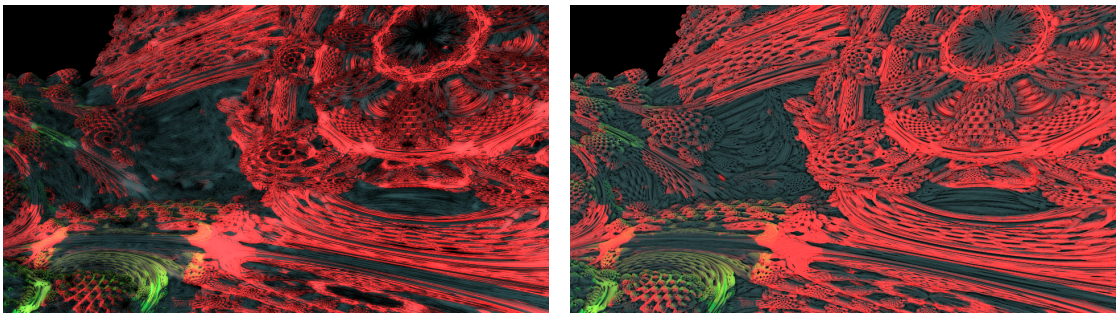
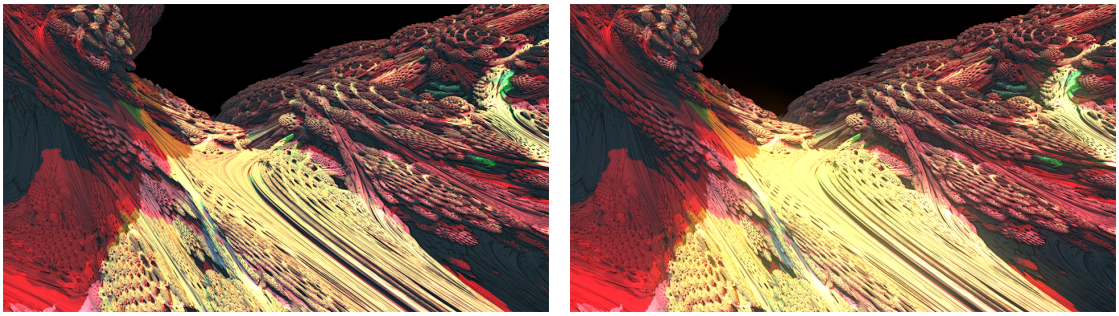


Figure 4.3: A real-time rendering of the Mandelbulb. The image on the top is rendered while moving, the image on the bottom is captured when the camera was fixed.



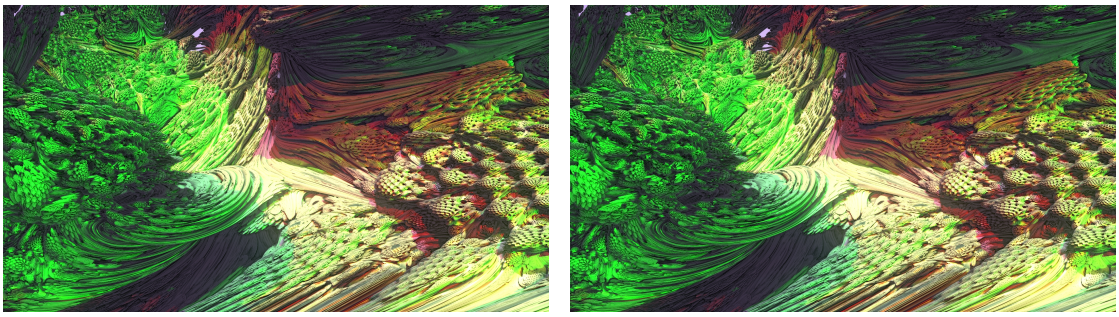
(a) Screenspace Ambient Occlusion (b) Horizon Based Ambient Occlusion

Figure 4.4: Different global illumination approximations.



(a) No bloom effect added (b) Bloom effect added after tone mapping

Figure 4.5: An example of a rendering with and without the bloom effect.



(a) image for the left eye (b) image for the right eye

Figure 4.6: An example of stereoscopic 3D rendering of a test scene.

CHAPTER 5

Conclusion

The rendering of fractals on the GPU is, due to the nature of the fractals, a worst-case-scenario for anti-aliasing and a lot of cutbacks of correct rendering need to be done to provide a real-time visualization. The current GPU-architecture offers enough computational power for single precision calculations, but still needs more power to enable deep zooms into fractals with double precision calculations and good visual quality at interactive frame-rates. Furthermore we succeed to include image enhancing effects and provide a high quality rendering mode as well. To accomplish this, we solve the problem of calculating the normal vector for the fractal and investigate the GPU architecture to achieve maximum performance.

CHAPTER 6

Future Work

For future work we suggest to implement a scheme for adapting the number of iterations on the camera-distance. Such an adaptive iteration number reduces sampling artifacts and speeds up rendering (by potentially using less iterations) in order to increase performance without losing a significant amount of visual quality and detail. To further improve visual quality, we propose to implement a better and more accurate global illumination method. We also emphasis a more sophisticated coloring option for the fractal to increase artistic freedom in our application.

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2] Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. Image-space horizon-based ambient occlusion. In *In SIGGRAPH '08: ACM SIGGRAPH 2008 talks*, 2008.

[3] Francisco Garcia, Angel Fernandez, Javier Barrallo, and Luis Martin. Coloring dynamical systems in the complex plane. 2008.

[4] Felix Hausdorff. Dimension und äußeres Maß. *Mathematische Annalen*, 79:157–179, 1918.

[5] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freedman and Co., New York, 1983.

[6] NVIDIA. Cuda C Programming Guide version 4.0, 2011.

[7] NVIDIA. Nvidia GmbH, 2011.

[8] Jacob Olsen. Realtime procedural terrain generation: Realtime synthesis of eroded fractal terrain for use in computer games, 2004.

[9] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[10] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 267–276, New York, NY, USA, 2002. ACM.

[11] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 75–82, New York, NY, USA, 2009. ACM.

[12] Mitsuhiro Shishikura. The Hausdorff dimension of the boundary of the Mandelbrot set and Julia sets. *The Annals of Mathematics*, 147(2):225–267, 1998.

[13] T. Vicsek. *Fractal Growth Phenomena*. World Scientific, 1992.

[14] Helge von Koch. On a continuous curve without tangents, constructible from elementary geometry. 1904.

[15] Daniel White. Mandelbulb explorer site. `http://www.skytopia.com/project/fractal/mandelbulb.html`(last visited on 18 July 2011), 2009.

[16] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, jun 1980.