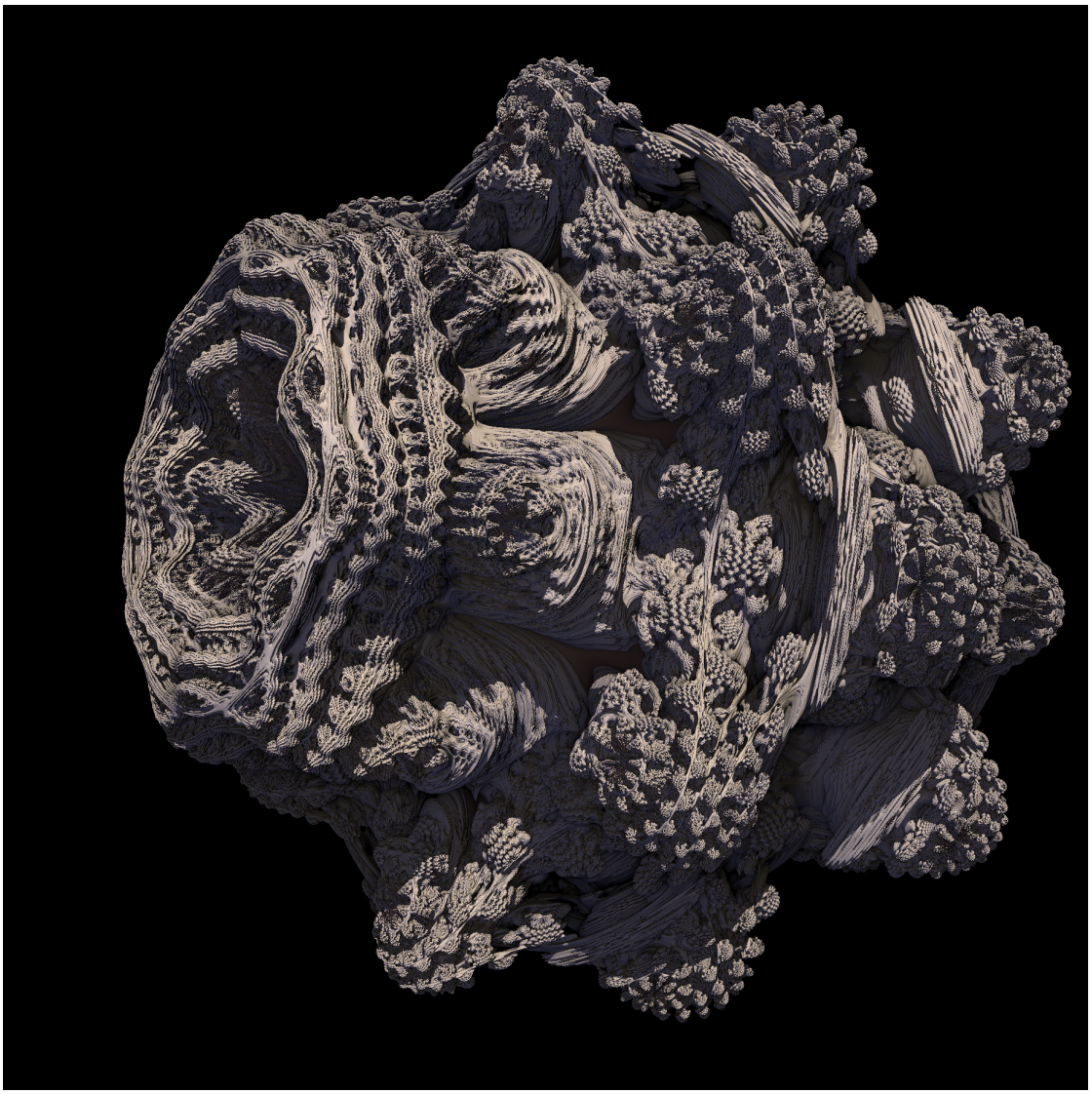
Rendering of a Mandelbulb Fractal in Real Time



Contents

1 Introduction **1**
1.1 Problem statement 1

2 Related Work **3**
2.1 Fractals, Mandelbrot, Mandelbulb 3
2.2 Ray Marching 5
2.3 Ray Tracing and shading 6

3 CUDA and General Purpose GPUs **9**
3.1 Introduction 9
3.2 CUDA 10

4 Method **11**

5 Implementation **15**
5.1 Acceleration structure 16
5.2 Calculation of the normal 16
5.3 Shadows and shading 17
5.4 Adaptive refinement and navigation 17
5.5 Rendering parameters 17

6 Results **19**
6.1 Test Setup 19
6.2 Benchmarks 20
6.3 Limitations 20

7 Conclusion **25**
7.1 Future Work 25

A Images **27**
A.1 Front view 28
A.2 Broccolis 32
A.3 Cave 36
A.4 Back view 40

Bibliography **45**

CHAPTER 1

Introduction

Rendering fractals has fascinated people already for 3 decades [20]. Since then the computational power of computers has increased exponentially [24] and therefore also the quality and speed of rendering fractals has improved. What started with printed asterisks on a sheet [2] developed into beautiful fractal movies [11]. After the Mandelbulb was found in 2009 the community around fractals exploded [9] and many other DE fractals where found, see [8] and [7].

Crane was probably one of the first who started fractal rendering on the graphics card in 2004 [10] when programmable devices appeared. One of the first GPU renderers for the Mandelbulb was programmed by Beddard in 2009 [1]. Today several software packages exist [8].

1.1 Problem statement

The goal of this work was to create a 3d Mandelbulb on the graphics card and render it in real time. The user should be able to control the camera by mouse and keyboard. If the user comes closer to the fractal, more details should be computed on the fly. The object should be illuminated by dynamic lights, including shadows.

CHAPTER 2

Related Work

2.1 Fractals, Mandelbrot, Mandelbulb

A fractal is a mathematical set that can have a non discrete (Hausdorff) dimension, hence the name fractal (from Latin fractus, fractional) [20, 21]. Usually structures have a discrete dimension, for instance a line is 1d, a triangle is 2d and a cube is 3d. On the contrary fractals can have a dimension of for instance 1.654... They are computed by evaluating mathematical formulas in a recursive way. Because of that they have an infinite precision in details, they can be invariant to scaling and they are self similar [15].

Figure 2.1: 7 iterations of the Cantor set, taken from [17].

One of the simplest fractals is the Cantor Set. It is created by starting with a line and recursively removing the middle third part of it. The first 7 iterations of such a Cantor Set are shown in Figure 2.1.

Another well known fractal is the Mandelbrot Set, shown in Figure 2.2. It operates in the domain of complex numbers, represented as the x and y axis of the image. Every point in the domain is fed into a recursive formula 2.1 starting with $z_0 = 0$ and c being the point of interest. If $\lim_{n\to\infty}(z_n)$ stays within certain bounds, the point belongs to the set and otherwise, if it tends towards infinity, it does not [12]. In the figure, black points belong to the set. The shades of grey indicate the speed of divergence of z_n.

$$z_{n+1} = z_n^2 + c \tag{2.1}$$

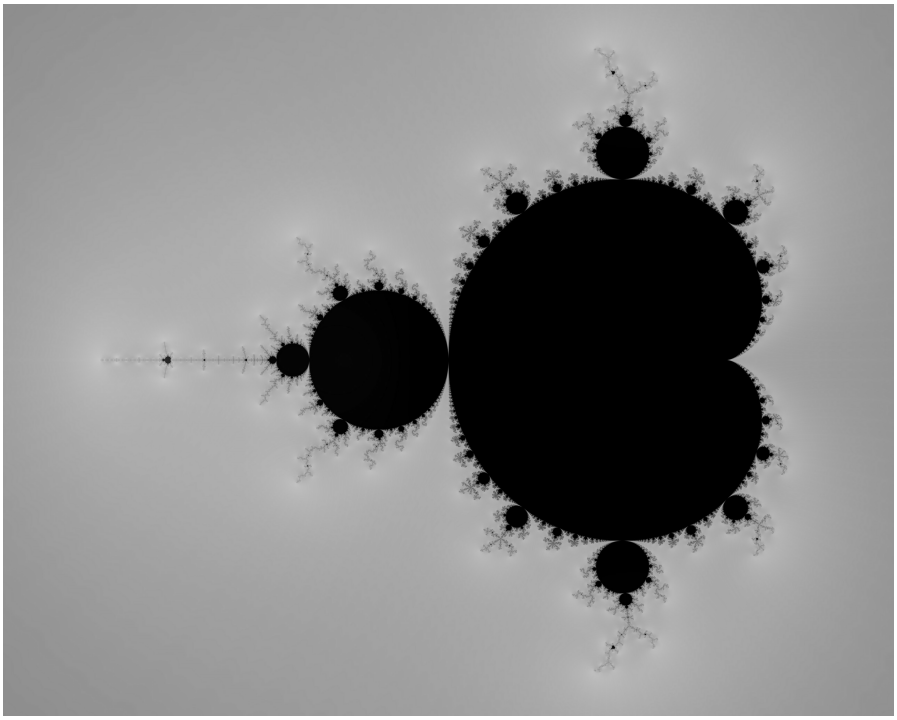


Figure 2.2: Black and white rendering of the Mandelbrot set, taken from [13].

In this project we want to render a Mundelbulb, it appeared as a new type of fractal in 2009 [28]. It is constructed in a similar way as the Mandelbrot Set, but using 3 dimensional vectors instead of complex numbers for z and c (Equation 2.2).

The exponential function is defined as shown in Equation 2.3 [29], using spherical coordinates (2.4 - 2.6). The addition works just as normal vector additions.

A rendered version of such a fractal is shown in Figure 2.3, using a power of 8 (z_n^8) as proposed by Nylander [23].

$$z_{n+1} = z_n^p + c \tag{2.2}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}^n = r^n \begin{pmatrix} sin(\theta * n) * cos(\phi * n) \\ sin(\theta * n) * sin(\phi * n) \\ cos(\theta * n) \end{pmatrix} \tag{2.3}$$

$$r = \sqrt{x^2 + y^2 + z^2} \tag{2.4}$$

$$\theta = atan2(\sqrt{x^2 + y^2}, z) \tag{2.5}$$

$$\phi = atan2(y, x) \tag{2.6}$$

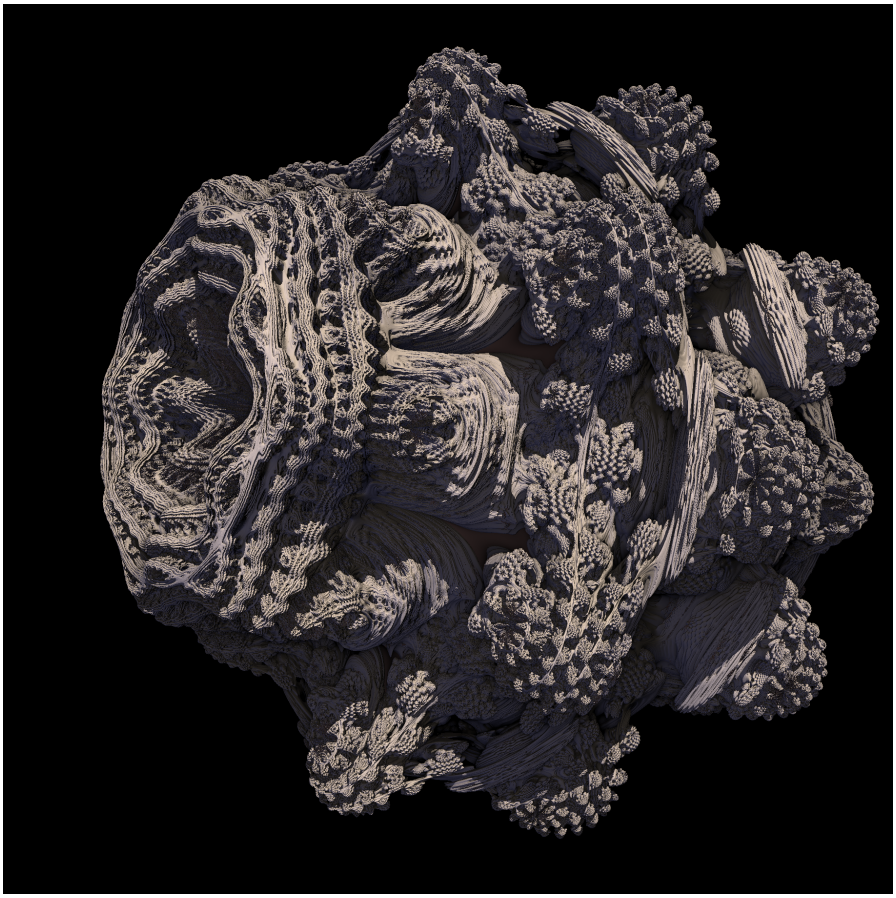


Figure 2.3: Rendering of a Mandelbulb, created with the program developed in this project.

2.2 Ray Marching

Hart et al. [16] introduced ray marching using distance estimation for 3d fractals. The distance estimator (DE) is a function that returns a lower bound for the distance from a given point to the fractal. The DE is evaluated at the starting point of the ray, it is then safe to walk along the ray for this distance. This is repeated, until the distance estimation reaches a threshold. The point reached by the last step is then considered the surface point. Figure 2.4 shows this principle.

Naturally for this to work there must be such a DE. This is the case for many geometric objects including many fractals. There are different versions of the DE for the Mandelbulb in the web, we used the one from Christensen [6], see Listing 2.1.

Listing 2.1: Code for distance estimator taken from [6].

```
float DE(vec3 pos) {
    vec3 z = pos;
    float dr = 1.0;
    float r = 0.0;
```

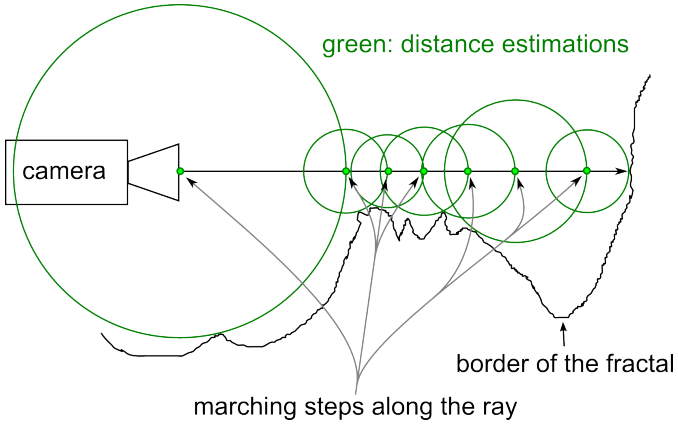


Figure 2.4: This figure shows the basic principle of ray marching.

```
    for (int i = 0; i < Iterations ; i++) {
        r = length(z);
        if (r>Bailout)
            break;

        // convert to polar coordinates
        float theta = acos(z.z/r);
        float phi = atan(z.y, z.x);
        dr = pow(r, Power-1.0) * Power * dr + 1.0;

        // scale and rotate the point
        float zr = pow(r, Power);
        theta = theta * Power;
        phi = phi * Power;

        // convert back to cartesian coordinates
        z = zr * vec3(sin(theta)*cos(phi)
                      sin(phi)*sin(theta),
                      cos(theta));
        z += pos;
    }
    return 0.5*log(r)*r/dr;
}
```

2.3 Ray Tracing and shading

Whitted et al. [30] described how to trace rays from the camera into the scene. Upon hitting an object, secondary rays are sent to compute refractive, reflective and shadow term of the shading. This recursion is visualised in Figure 2.5.

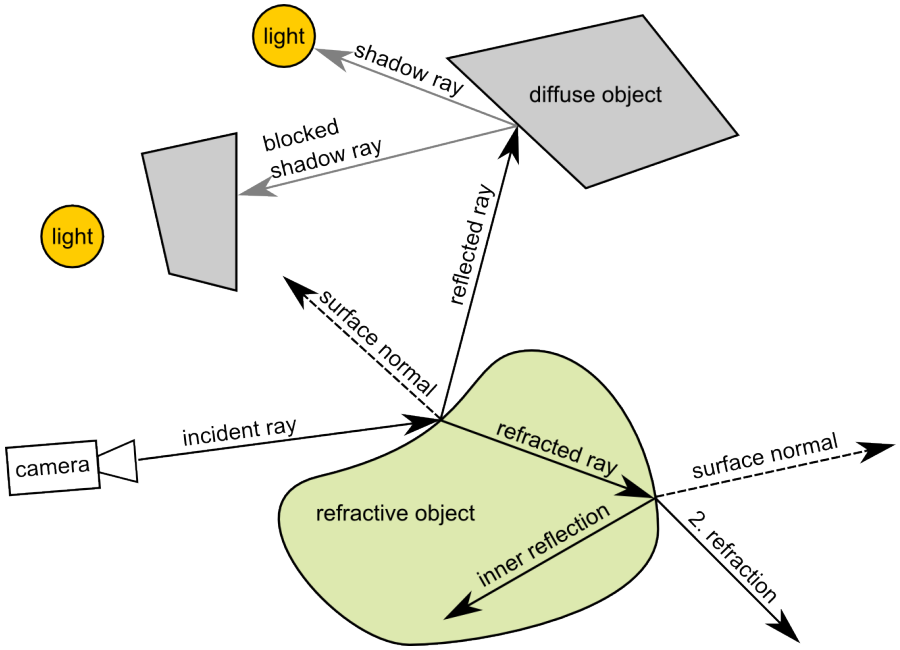


Figure 2.5: Visualisation of ray tracing

Since we decided that the fractal should be rendered solid and non reflecting, the refractive and reflective term is set to 0. We consider only diffuse part including the shadow ray.

Evans described a fast way to approximate ambient occlusion using distance fields [14]. He samples a volumetric distance field along the normal of the shading point in order to get an estimation of how 'hidden' the point is. The values in the distance field will correlate to the distance on the normal only if the point of interest is exposed. On the other hand points inside holes will have smaller values in the distance field.

It is possible to create interesting renders of fractals using only the step count of the ray marching algorithm [5], an example is shown in Figure 2.6. This is an even further simplified version of Evan's method.

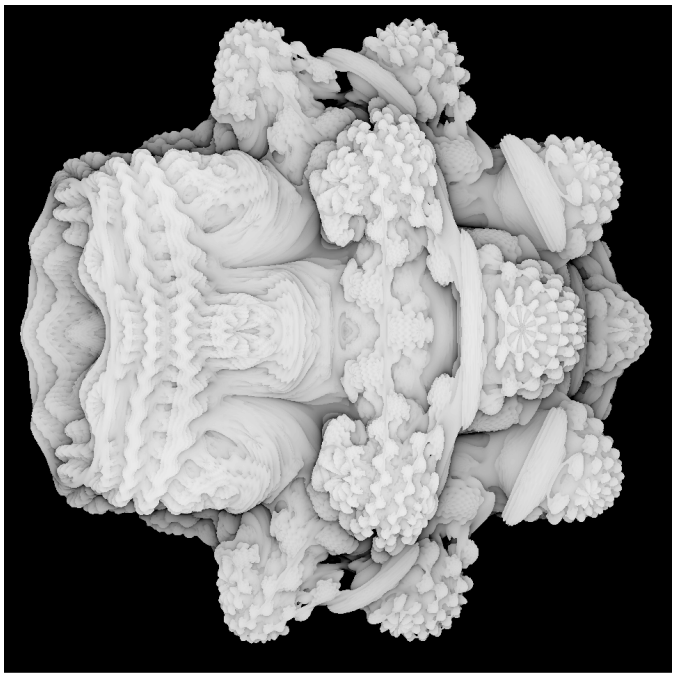


Figure 2.6: Mandelbulb rendered using only the step count: $fakeAmbientOcclusion$ of Equation 5.1.

CHAPTER 3

CUDA and General Purpose GPUs

Fractal rendering has a high computational costs and can be parallelised easily. This is ideal for the GPU [10]. A first Quaternion Julia set was rendered already in 2004 on a graphics card using the fragment shader, which was very limited back then [10]. Modern graphics cards have greater possibilities and can be programmed more easily, which is also used for scientific research [25].

3.1 Introduction

As already said in [4], there were early attempts with the start of programmable GPUs to use them for general computations. However this was still very restricted as the graphics pipeline had to be used [19] [25]. The basic work flow of these systems was as follows:

1. upload input data to the graphics hardware as a texture
2. upload a fragment shader program doing the computation
3. render a screen quad into a buffer using this fragment shader
4. retrieve the result by reading back the buffer into host memory

More general frameworks were invented, they abstracted the graphics pipeline to provide a more convenient programming model [3]. Graphics card manufacturers saw the trend towards general purpose GPUs (GPGPU) and supported the development. Several GPGPU programming systems were released [25]. One of them is CUDA, a C like high level language developed by NVIDIA. There is also a non-proprietary alternative called OpenCL which is very similar to CUDA. However it was shown, that CUDA performs better than OpenCL on appropriate hardware [18].

3.2 CUDA

As already stated in [4], parallelism in CUDA is implemented by running many parallel threads. The code run by one thread is called a kernel. It has a thread id which can have up to 3 dimensions. This thread id is used to select the domain in the data that the current thread is processing and in the output to prevent write conflicts between threads. Threads are organised into blocks which can also have up to 3 dimensions. All threads in one block will be executed on one GPU core. Due to hardware restrictions of such a core the number of threads per block is limited. The exact limit depends on the hardware generation, but a number of 256 threads is common.

Blocks can be executed in any order or parallel on several cores, so the data must be independent. There are mechanisms to sync threads and share memory inside blocks. The memory is organised in a hierarchy with per-thread local memory, per-block shared memory and global memory [22].

CUDA defines only a few extensions to the C language for defining the kernels. The host code is interleaved with the device code, Listing 3.1 shows an example. Two vectors with N elements are added up. Inside the device code the current thread id can be accessed by using threadIdx.

Listing 3.1: CUDA code sample taken from [22].

```
//Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    ...
    //Kernel invocation with N threads per block, 1 block used
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

CHAPTER 4

Method

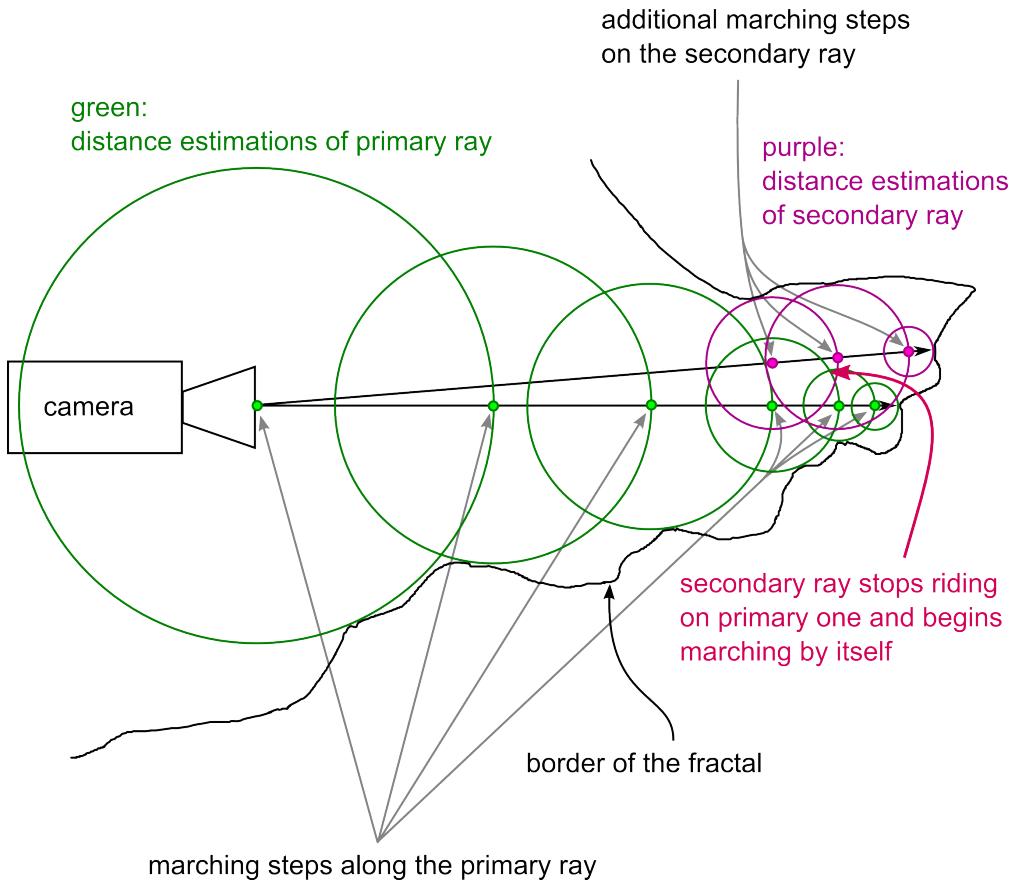


Figure 4.1: Sketch explaining the faster algorithm

The aim of this work was to make it possible to render the Mandelbulb in real time. Even when using the parallel architecture of the GPU as described in Chapter 3, and the DE ray marching approach described in Chapter 2 the rendering was too slow, especially when zooming in closer to the object.

Our newly developed method accelerates ray marching. In every step of the marching algorithm an computationally expensive DE has to be evaluated. The idea is, that the evaluation of this DE is not necessary as long as a ray stays inside of already computed DE spheres.

First a ray is shot using normal ray marching (primary ray). A second ray with the same origin and only slightly different direction (secondary ray, not to be confused with shadow or reflection rays) will stay close to the primary one and therefore can reuse the already computed distance estimations. It can 'ride' on the first ray, until it leaves the DE spheres. Figure 4.1 shows this principle.

The orange colour in Figure 4.2 marks the shortest 'safe' circles around the primary ray. They are formed by the intersection of two neighbouring DE spheres. Note that this is a circle in 3d space, with the primary ray as the axis.

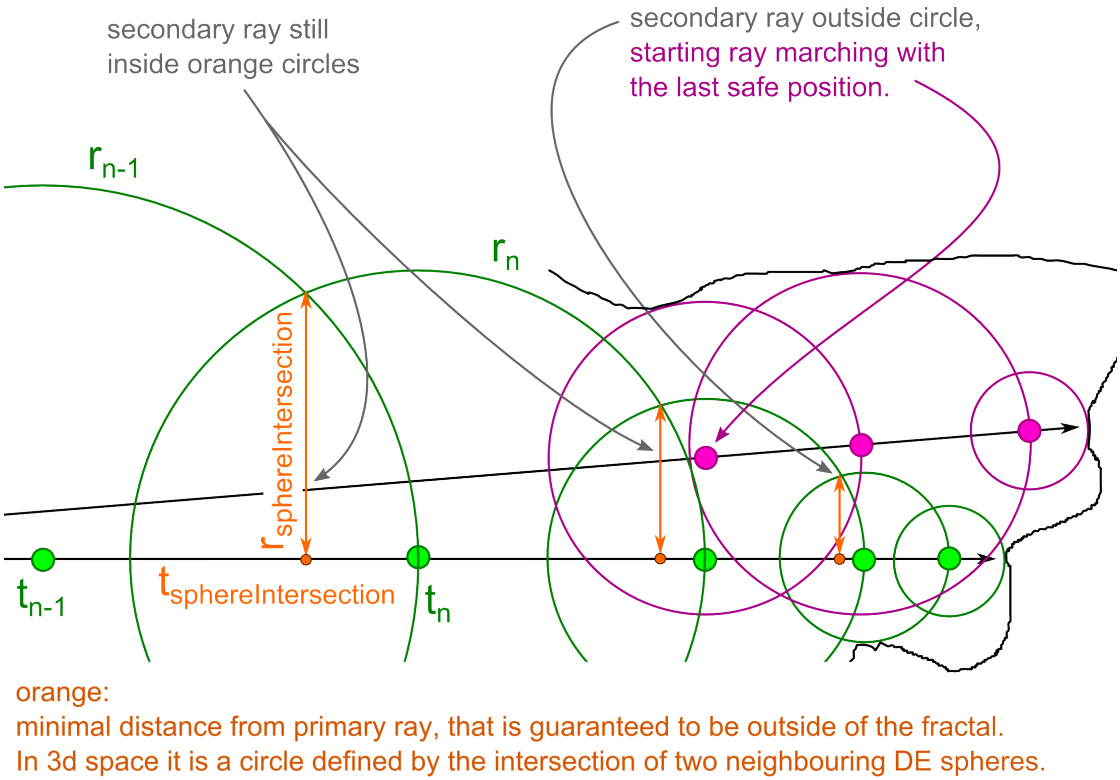


Figure 4.2: It is only necessary to check at the intersections of the DE spheres (marked in orange).

It is enough to do the calculations in 2d space though. Let the primary ray lie on the x-axis of a 2d Cartesian coordinate system. The secondary ray will be a line, starting at $(0/0)$ and having

a slope k from Equation 4.1. The DE spheres are repeatedly calculated after marching t_n on the ray using Equation 4.2. They are represented simply by its radius r_n and t_n.

$t_{sphereIntersection}$ and $r_{sphereIntersection}$ are depicted in Figure 4.2 and can be calculated as shown in Equation 4.3 and 4.4. These equations can be easily derived from plain intersection calculations.

If Inequation 4.5 is true then the secondary ray is guaranteed to be inside the DE spheres until t_n and can therefore start ray marching at position $origin + t_n * direction_{secondary}$.

$$k = |direction_{secondary} - direction_{primary}| \tag{4.1}$$

$$r_n = DE(origin + t_n * direction_{primary}) \tag{4.2}$$

$$t_{sphereIntersection} = t_n - 0.5 * \frac{r_n^2}{r_{n-1}} \tag{4.3}$$

$$r_{sphereIntersection} = \sqrt{r_n^2 - 0.5 * \frac{r_n^2}{r_{n-1}}} \tag{4.4}$$

$$k * t_{sphereIntersection} < r_{sphereIntersection} \tag{4.5}$$

CHAPTER 5

Implementation

The architecture of graphics cards is highly parallel and it would be slow to do the computation in a tree like structure. Therefore we chose to do the computation of the fractal in two passes. In the first pass the primary rays are shot and the acceleration structure is built up. In the second pass all rays are shot again, using the acceleration structure for speed up. Normal computation, shadows and shading is done in a third step.

The primary rays are shot in the middle of either a 3x3 or 5x5 pixel block, see Figure 5.1, depending on the configuration. The secondary rays are shot around the primary ones and use only the primary ray inside their block. In the second pass also the primary rays are shot a second time, but since the direction of the primary rays is exactly the same in the first and second pass, they can reach the fractal very quickly by using the acceleration structure. Therefore not much time is lost.

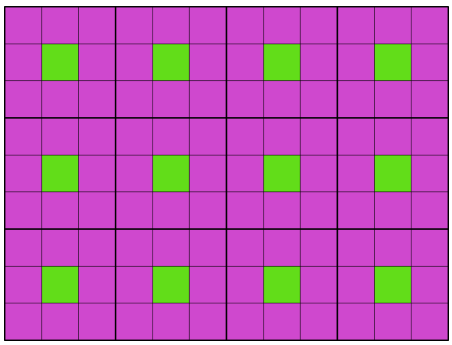


Figure 5.1: Ray marching is done in two passes, first only the primary rays are shot and then all of them. green: primary rays, purple: secondary rays

5.1 Acceleration structure

It would be very slow to save all DE spheres of the primary rays and then iterate over them in the second pass due to memory throughput on the graphics card. Therefore we chose to calculate the 'riding distance' and step count of the secondary rays directly in the first pass and save this value. In the second pass these values are retrieved and marching begins closely to the fractal surface.

These values are saved in the structure shown in Listing 5.1.

Listing 5.1: Ray acceleration structure.

```
struct __align__(8) RayAccelerationData {
    unsigned char rayStepsForRayPosition[ACCELERATION_RAY_COUNT];
    float rayDistanceForRayPosition[ACCELERATION_RAY_COUNT];
};

__device__ RayAccelerationData
rayAccelerators[RENDER_WIDTH/ACCELERATION_BLOCK_SIZE]
               [RENDER_HEIGHT/ACCELERATION_BLOCK_SIZE];
```

The array $rayAccelerators$ holds one $RayAccelerationData$ object per primary ray. $rayStepsForRayPosition$ corresponds to n in Formula 4.2, $rayDistanceForRayPosition$ to t_n. To save memory we don't save data for all secondary rays. k from Formula 4.1 will be very similar within the mirrored values in 5.2, the perspective distortion can be neglected since there are no visible differences in the resulting render.

5	4	3	4	5
4	2	1	2	4
3	1	0	1	3
4	2	1	2	4
5	4	3	4	5

Figure 5.2: Layout of RayAccelerationData's indices in the acceleration block

5.2 Calculation of the normal

The normal is estimated using a simple numerical approximation of the gradient shown in Listing 5.2. The gradient is calculated from the potential field formed by the DE. Sampling the potential field only 4 times instead of 6 is a bit less accurate but faster.

Listing 5.2: Estimation of the normal.

```
vec3 calculateNormal(vec3 p) {
        float e = 2e-6f;
        float n = DE(p);
        float dx = DE(p + vec3(e, 0, 0)) - n;
```

```
        float dy = DE(p + vec3(0, e, 0)) - n;
        float dz = DE(p + vec3(0, 0, e)) - n;

        vec3 grad = vec3(dx, dy, dz);
        return normalize(grad);
}
```

5.3 Shadows and shading

Computation of hard shadows are straight forward in ray tracing. A ray directed towards the light is started at the point of shading, using the ray marching algorithm. There is light if it doesn't hit the fractal and shadow otherwise. Although the fractal computation in shadow rays doesn't need to be as accurate as in view rays, they are computationally expensive.

In our program the fractal is illuminated by 3 lights. Only one of them casts shadows and the others are only very week to provide a more interesting atmosphere. Another factor of our shading is the step count of the ray marching algorithm which provides a very rough but fast approximation of ambient occlusion, see Equation 5.1. The resulting colour of the light shading is simply multiplied with this value as can be seen in Equation 5.2.

$$fakeAmbientOcclusion = 1 - \frac{stepCount}{maxStepCounten} \tag{5.1}$$

$$colour = fakeAmbientOcclusion * (light1 * shadowCalc + light2 + light3) \tag{5.2}$$

5.4 Adaptive refinement and navigation

Basic navigation was implemented. It is possible to move forward and backward and rotate the camera with the mouse. The translation speed is adjusted using the DE of the camera position in order to move slower when the camera gets closer to the fractal.

As proposed by Hart et al. [16] the minimal marching step depends on the distance to the eye. It is calculated as shown in Equation 5.4. As soon as a DE returns a distance smaller than this, the ray stops marching and the found position is considered the border of the fractal.

$$translationDistance = translationSpeed * lastFrameTime * DE(currentPos) \tag{5.3}$$

$$stepThreshold = \alpha * distFromEye \tag{5.4}$$

5.5 Rendering parameters

There is a number of rendering parameters that influence greatly the result and performance of the rendering. The most important are described in this section:

FRACTAL_MAX_RAY_STEPS maximum count of ray marching steps. If set too low, only details close to the eye will be shown. Details further away won't be reached by the

rays, because the maximum steps are already reached. The impact on the performance is noticeable. The default is 50.

FRACTAL_MAX_ITERATIONS How many iterations should be used for the fractal inside the DE function. With low values the fractal will be very 'shallow', with more iterations also deep caves can be explored. The impact on the performance is also reasonable. The default is 10.

FRACTAL_EPSILON_ALPHA α in Equation 5.4. This is a measure how precise the fractal will look like. If set too low, the frequency of the surface will be too high and the sampling frequency (monitor pixels) will be below the Nyquist frequency [26], showing only a maze of differently coloured pixels. If set too high, the frequency will be very low, no detail would be visible, only a bloby surface. Equation 5.4 is also a system for level of detail (LoD), high values make the border between LoD steps visible. The new algorithm gains a bigger advantage over the traditional one, if this value is set higher. This constant has the biggest impact on the performance, the default is 0.001.

SHADOW_EPSILON_ALPHA α for the shadow rays. It can be much higher because it is visible only indirectly, but if it gets too high everything is occluded. Default is 0.05.

RENDER_WIDTH and RENDER_HEIGHT Resolution of the rendered image. These values have to be evenly divisible by THREADS_PER_BLOCK_WIDTH, THREADS_PER_BLOCK_HEIGHT and ACCELERATION_BLOCK_SIZE in order to meet the assumptions made in the code for CUDA and the new algorithm.

ACCELERATION_BLOCK_SIZE The size of the acceleration block, eg 3 for 3x3 and 5 for 5x5. 3x3 is slightly faster and therefore the default.

THREADS_PER_BLOCK_WIDTH and THREADS_PER_BLOCK_HEIGHT The size of the CUDA array block. The fastest setting probably depends on the model of the graphics card, on a GeForce GTX 550 Ti the order from slowest to fastest is $4 \times 4 < 32 \times 32 < 8 \times 8 = 16 \times 16 < 8 \times 16$. The fastest value 8×16 is the default.

CHAPTER 6

Results

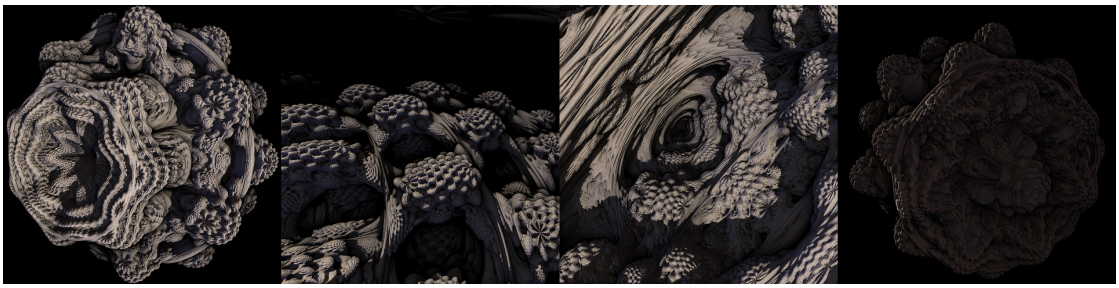


Figure 6.1: Overview of the scenes used in the benchmark section.

When rendering without shadows, the algorithm described in this work can give up to double the rendering speed. The speed up and the performance in general depend on the scene and chosen parameters.

Unfortunately the new algorithm produces artefacts on the fractal. On smooth surfaces boxes with the size of the acceleration blocks (ACCELERATION_BLOCK_SIZE) are visible. The reasons and examples can be found in Section 6.3.

6.1 Test Setup

The performance tests where made on a PC with the following components:

1. AMD Phenom II X4 CPU
2. 4 GiB RAM
3. NVIDIA GeForce GTX 550 Ti with 1 GiB of video memory.
4. Microsoft Windows 7

Scene	Shadows	FPS our method	FPS traditional	Speed up	Images
Front view	Yes	27.7	25.5	8.6%	A.1 A.3
Front view	No	58.9	46.1	27.8%	A.2 A.4
Broccolis	Yes	13.8	9.7	42.3%	A.5 A.7
Broccolis	No	32.3	16.1	100.6%	A.6 A.8
Cave	Yes	11.0	9.4	17.0%	A.9 A.11
Cave	No	32.4	20.6	57.3%	A.10 A.12
Back view	Yes	26.2	24.1	8.7%	A.13 A.15
Back view	No	60.1	48.0	25.2%	A.14 A.16

Table 6.1: Benchmarks of the two different rendering systems.

5. CUDA 5, NVIDIA Driver Version 320.18

All tests are performed with a resolution of 960 * 960 and no anti aliasing. The program was compiled in 32bit mode, single float precision everywhere, Fast Math and O2 optimization enabled, GPU architecture compute_10 and GPU version sm_13 were set. Since in the benchmarks we focus on the new algorithm and not rendering parameters as described in Section 5.5, they were just kept at the default values.

The scenes shown in Figure 6.1 were used for the benchmark. The resulting images are shown in detail in Appendix A.

6.2 Benchmarks

The measurements in Table 6.1 show that the method proposed in this work is faster in all situations. The speed up ranges from 8.6% to 100 % in special situations. It is never slower. Shadows have a great impact, the advantage of the proposed algorithm is less when rendering shadows because shadow rays are not accelerated right now.

The benchmarks also show, that it is possible to render the Mandelbulb in real time, but currently only without shadows and with artefacts.

6.3 Limitations

The biggest limitation of the new algorithm are artefacts shown in Figures 6.2, 6.3 and 6.4. The artefacts are a bit stronger on the 5×5 version but exist in both.

Figure 6.5 tries to explain two of the reasons for the artefacts. The step count, which is also used for the shading, can be different if the secondary ray is further away from the fractal because it can make bigger steps. But the proposed algorithm takes the step count from the primary ray, which is lower or higher and later visible on the edges of the block. The second reason, marked in purple colour in the figure, is the different distance that can be achieved by the algorithms. Usually this shouldn't be visible because it is below the minimal step distance.

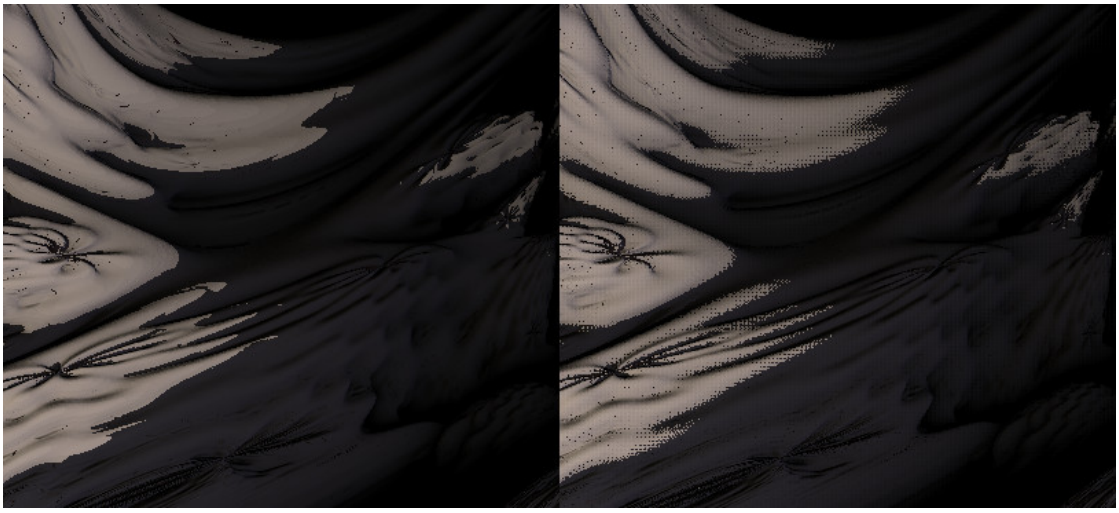


Figure 6.2: Artefacts at the border of the shadow and on smooth surfaces with a block size of 3×3.

Figure 6.3: Artefacts on smooth surfaces with a block size of 5×5.

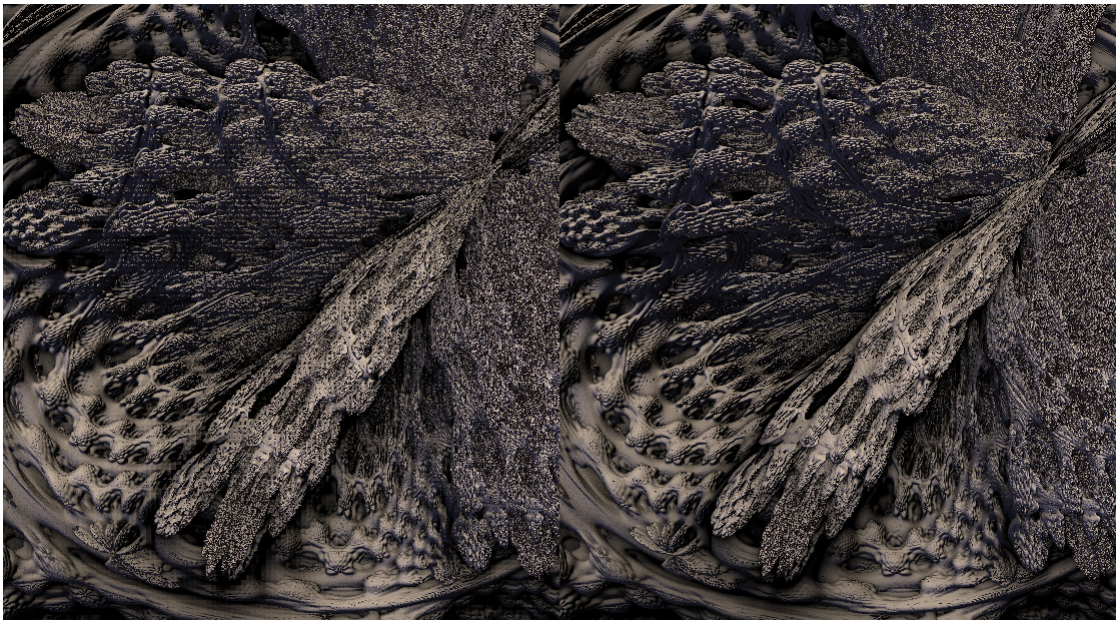


Figure 6.4: Artefacts on dynamic surfaces with a block size of 5×5.

But it matters for instance if the potential field formed by the DE is very dynamic (Figure 6.4) or a shadow is visible (Figure 6.2).

Dynamic surfaces like shown in 6.4 are a limitation for many fractal renderers. A higher sampling would be need and for this probably double precision instead of single precision as used in this work. Today both things would not work in real time on the graphics card due to performance reasons.

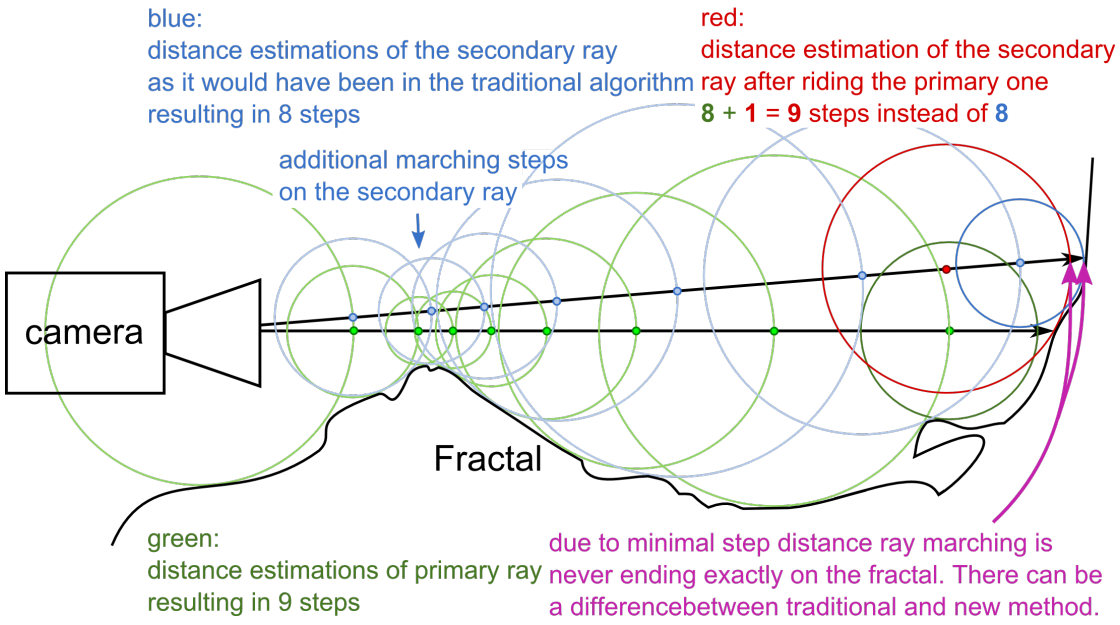


Figure 6.5: The step count and position can be different with the new algorithm

CHAPTER 7

Conclusion

We have presented a new method for ray marching distance estimated fractals and compared it to the traditional one. The presented method is faster in all cases but unfortunately produces visible artefacts. Therefore right now it can be used only for exploring the fractal but not for creating artistic renders.

Exploring is now possible with constantly real time frame rates. This is not the case with the traditional algorithm where the frame rate drops in certain situations dramatically. Today it is not possible to achieve real time frame rates when doing deep zooms into the fractal because double precision calculations are too slow.

7.1 Future Work

The artefacts shown in Section 6.3 can be diminished by using filters. For instance Gaussian blur filter or bilateral filter as proposed in [27] could reduce the artefacts. Theses filters would work directly on the step count and they could also consider the depth value for finding edges.

Currently the algorithm works only for camera rays, but it would be also possible to do something similar for shadow rays. This could make real time shadows in fractals possible in the next generation of graphics cards.

Areas with a highly dynamic potential field appear like a coloured maze right now. This could be improved by adaptive anti aliasing, although it would slow down the rendering.

Generally the algorithm is also a candidate for a fast anti aliasing algorithm. It is not sure if it would work in dynamic areas of the fractal though.

APPENDIX A

Images

In this appendix we present the full size images from our test scenes.

A.1 Front view

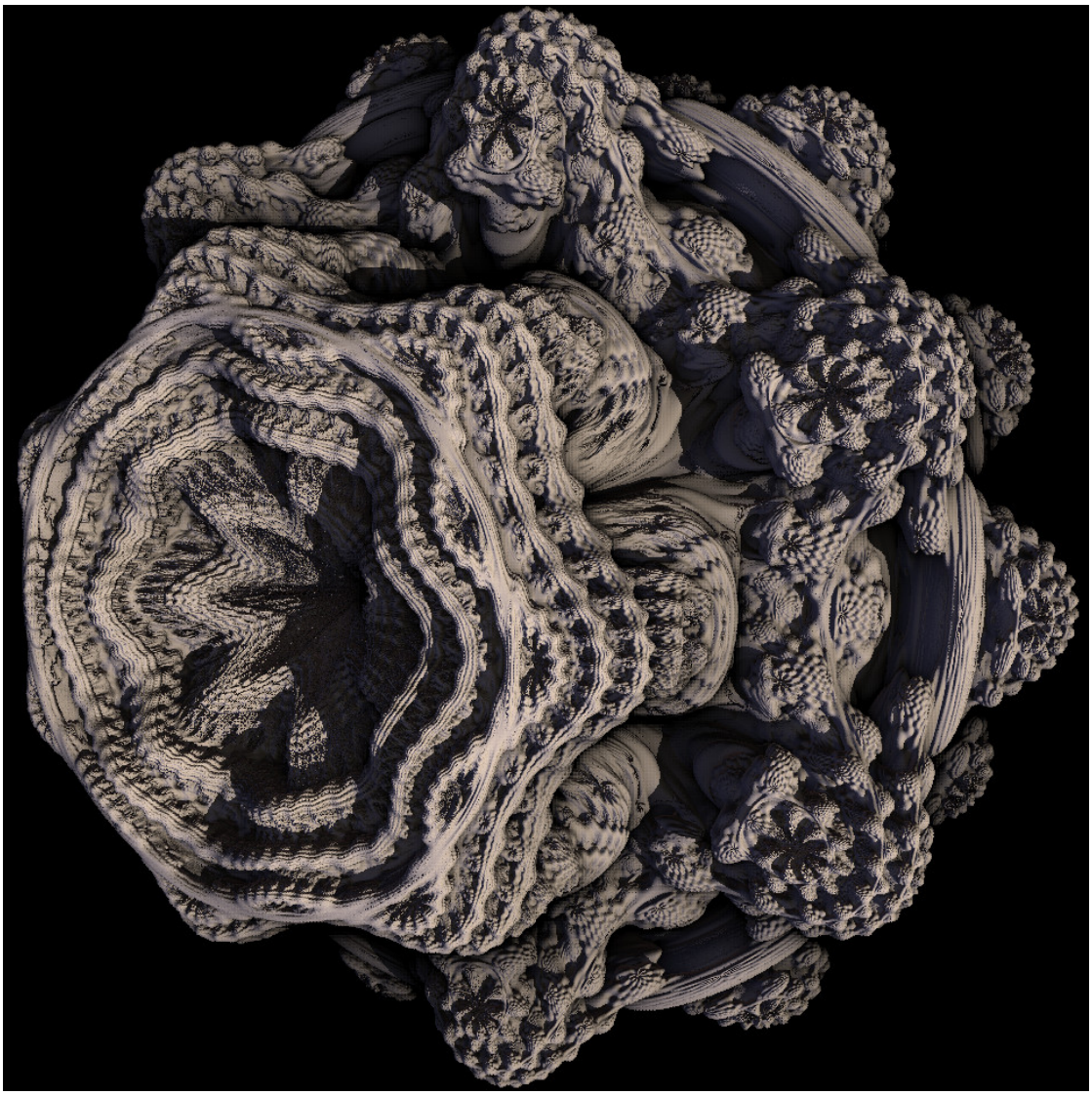


Figure A.1: Front view, our method, shadows, 27.7 fps.

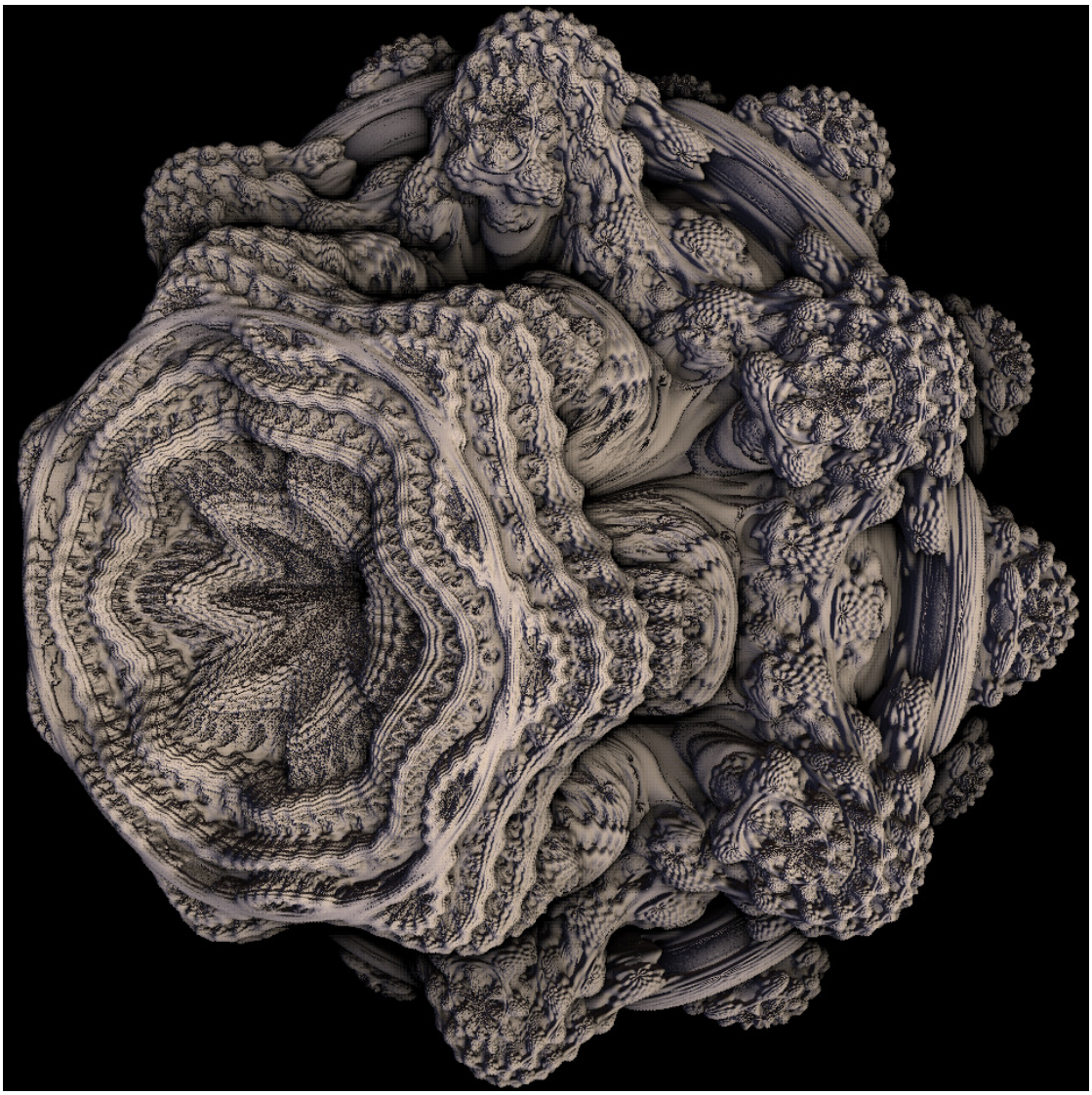


Figure A.2: Front view, our method, no shadows, 58.9 fps.

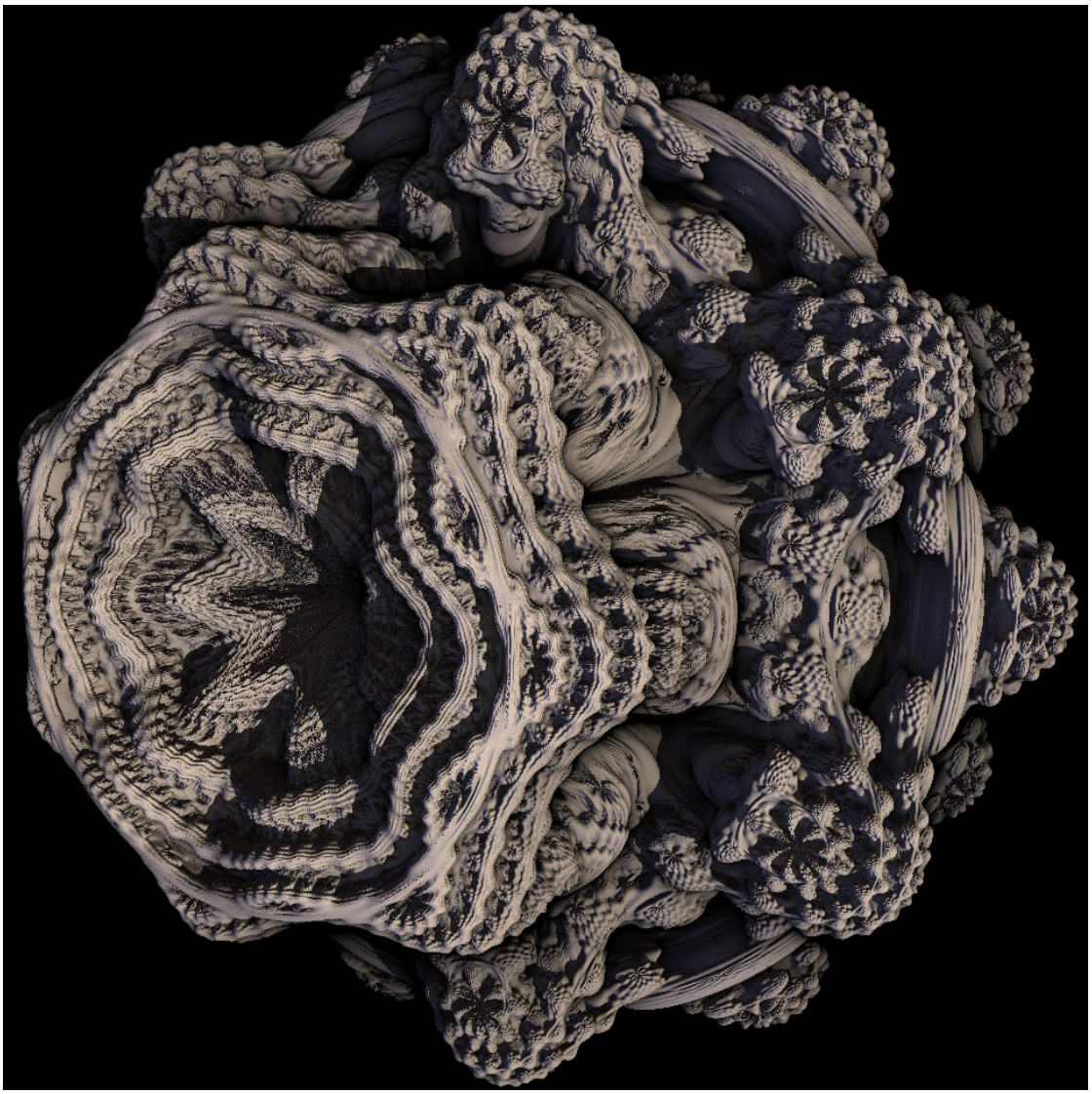


Figure A.3: Front view, traditional method, shadows, 25.5 fps.

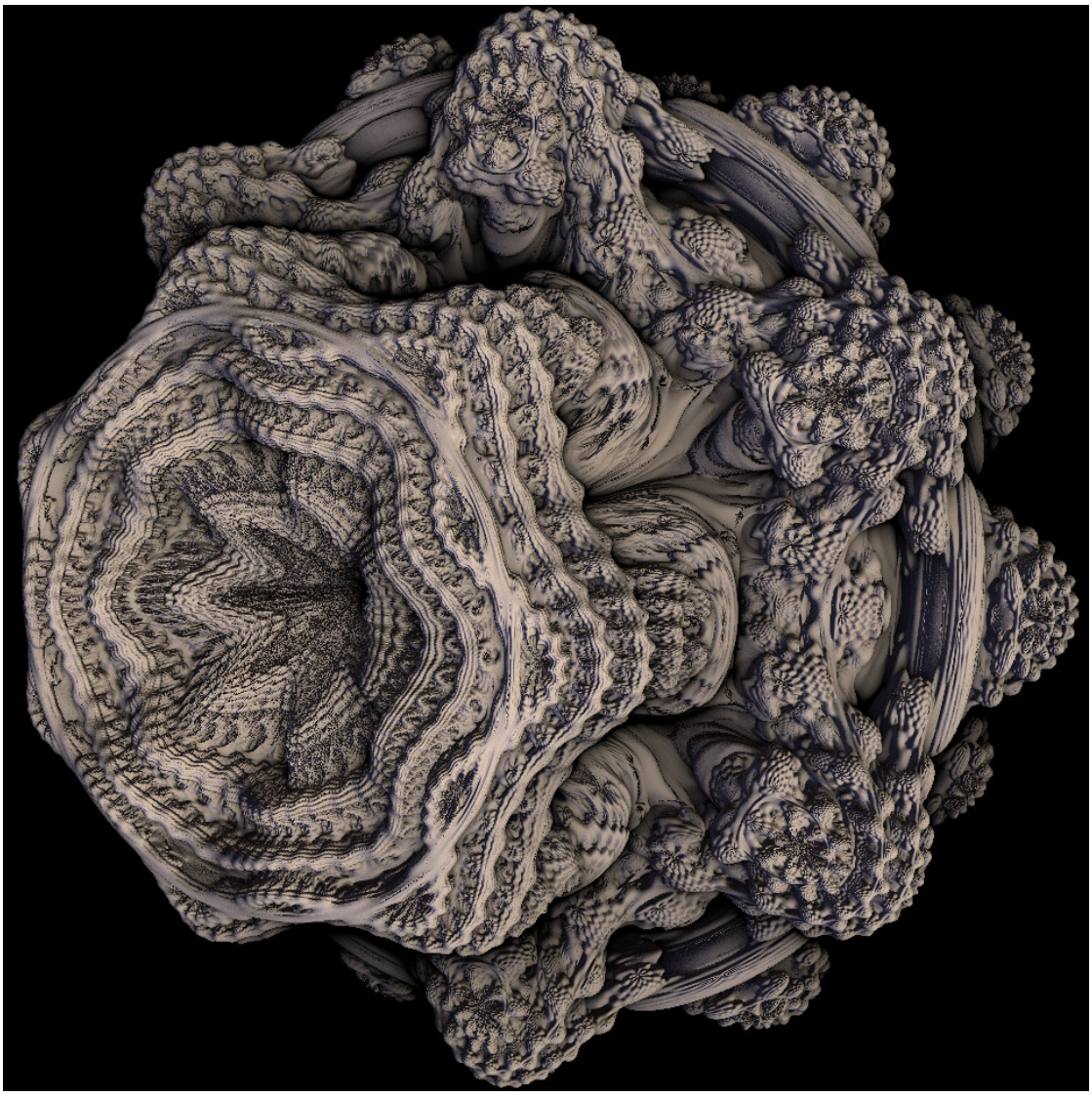


Figure A.4: Front view, traditional method, no shadows, 46.1 fps.

A.2 Broccolis

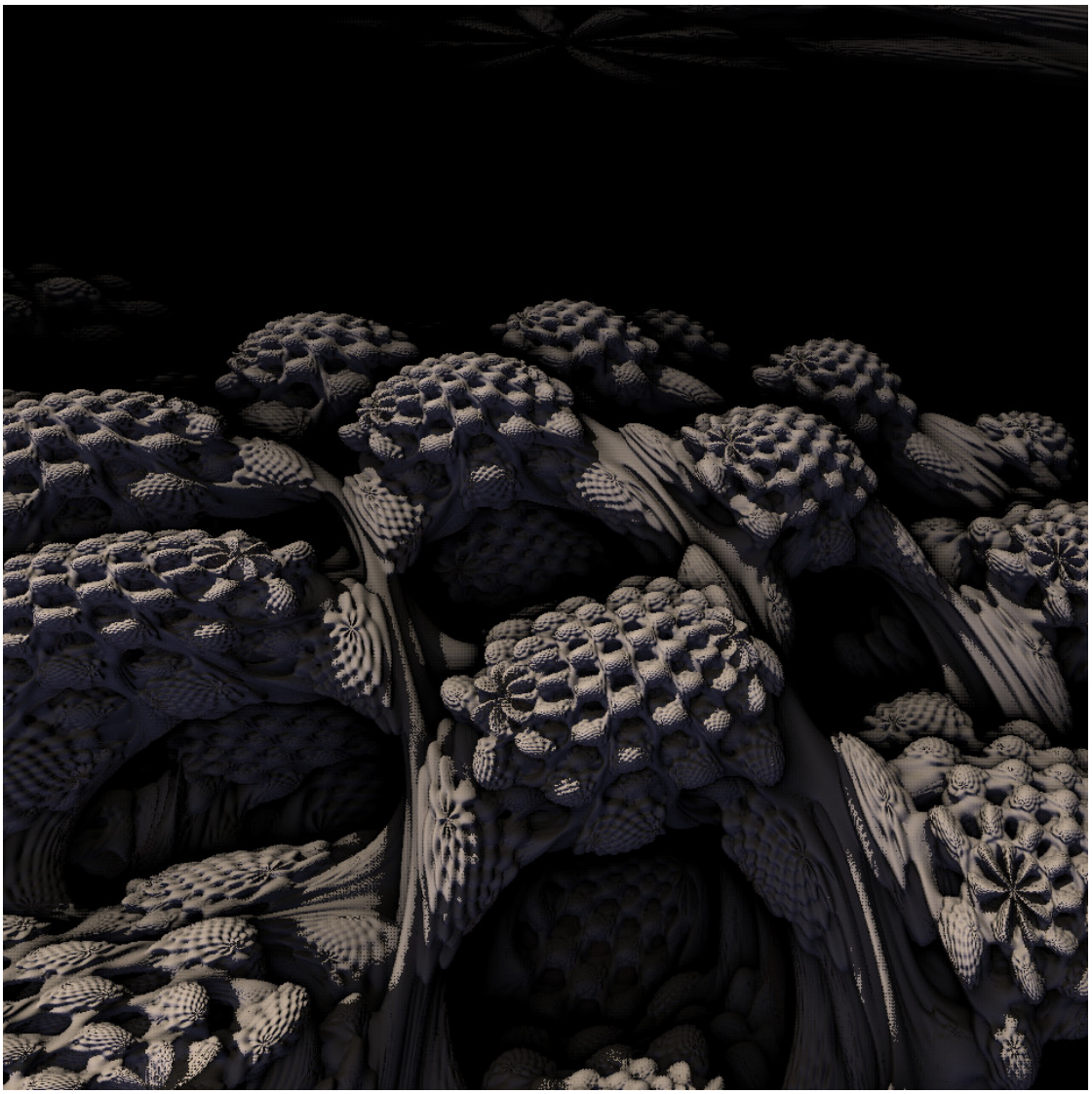


Figure A.5: Front view, our method, shadows, 13.8 fps.

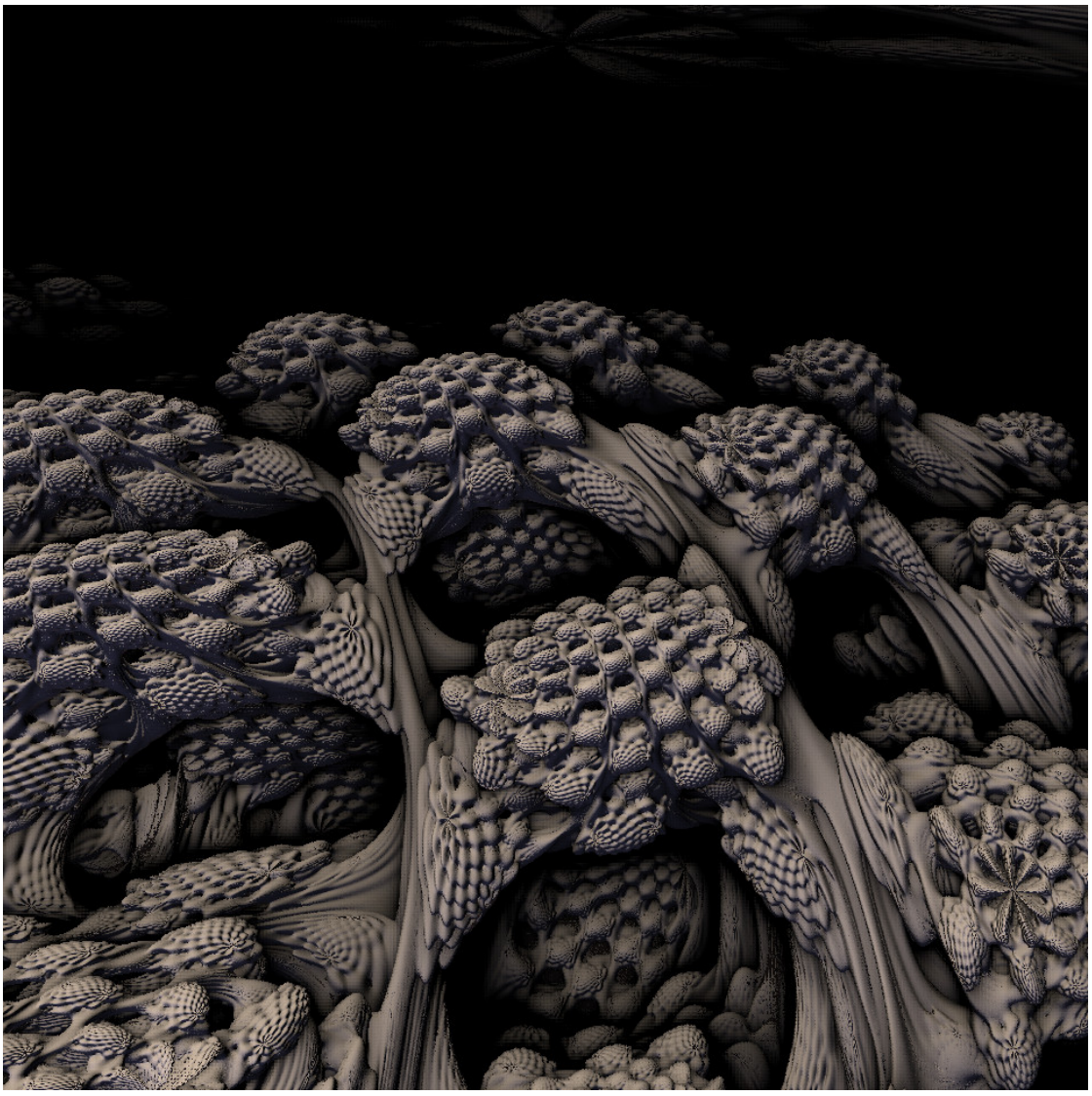


Figure A.6: Front view, our method, no shadows, 32.3 fps.

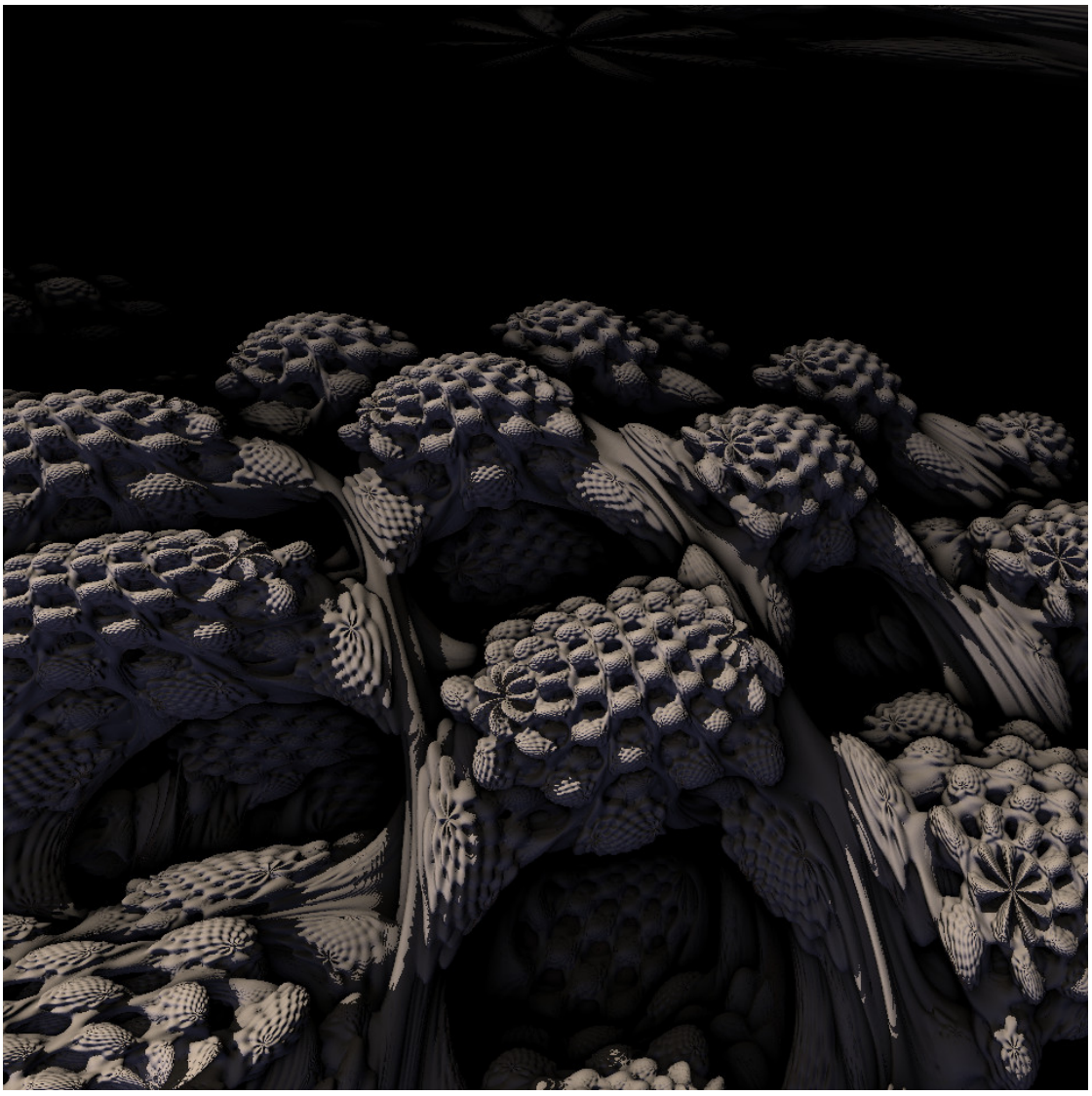


Figure A.7: Front view, traditional method, shadows, 9.7 fps.

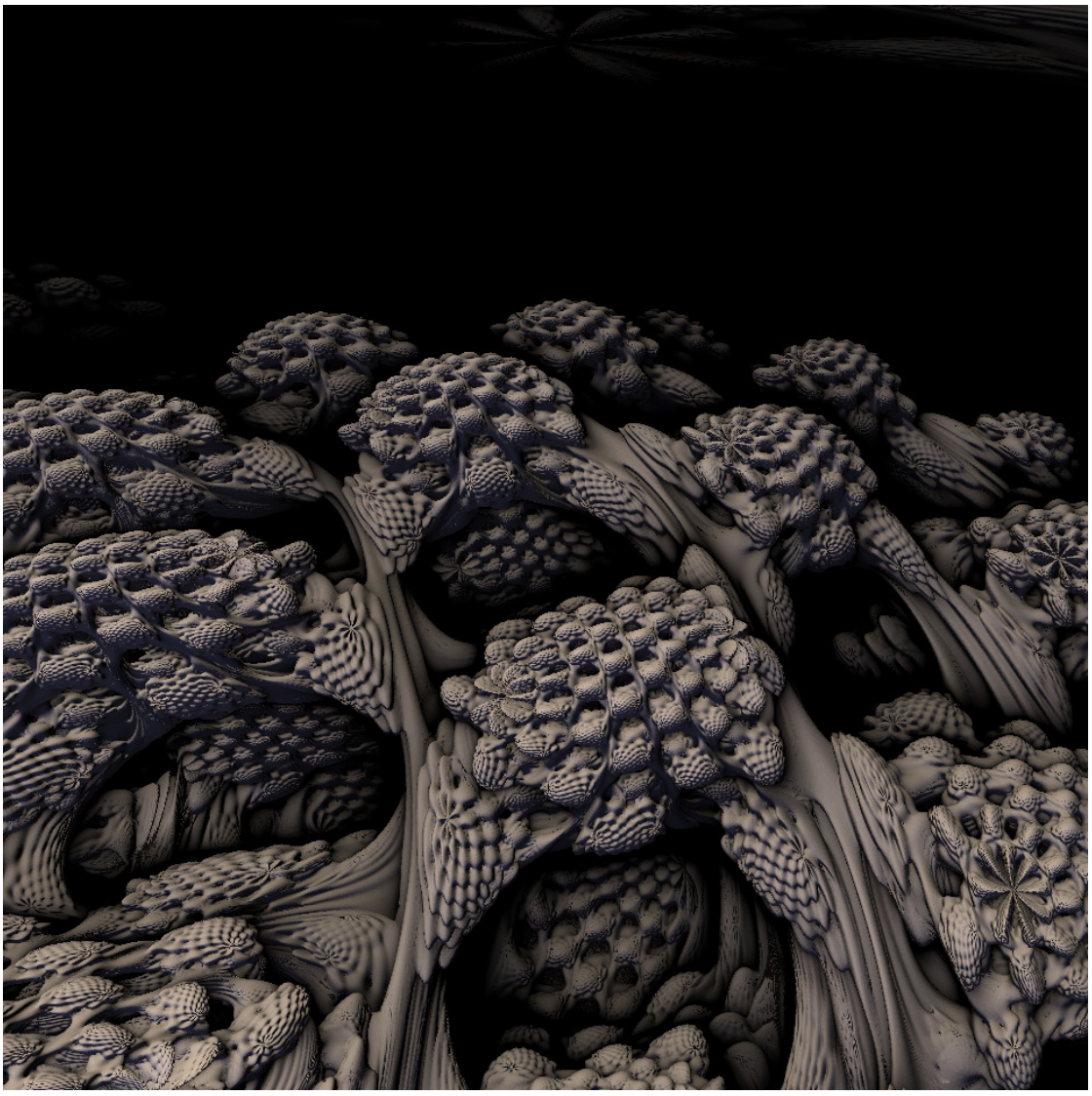


Figure A.8: Front view, traditional method, no shadows, 16.1 fps.

A.3 Cave

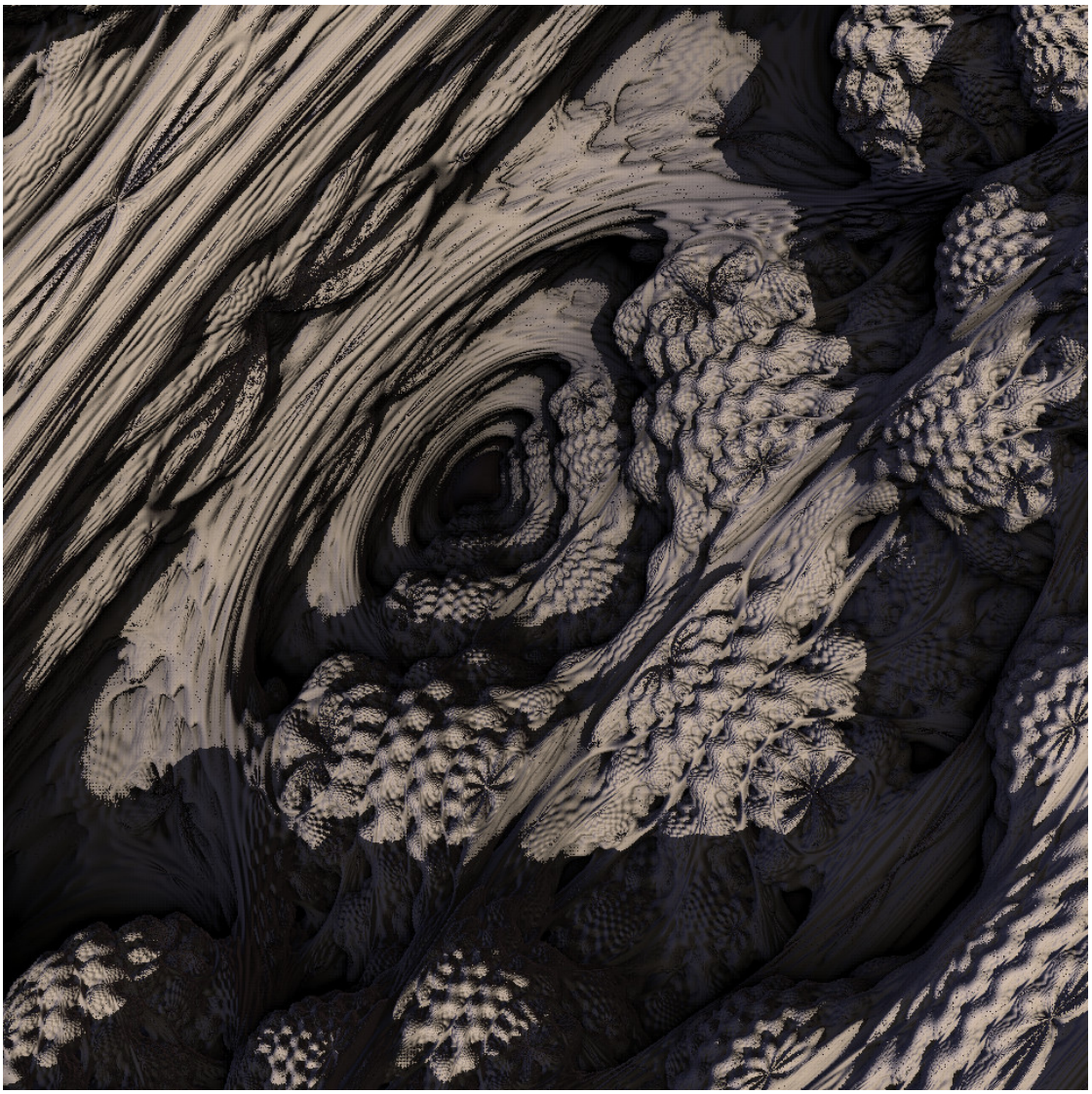


Figure A.9: Front view, our method, shadows, 11.0 fps.

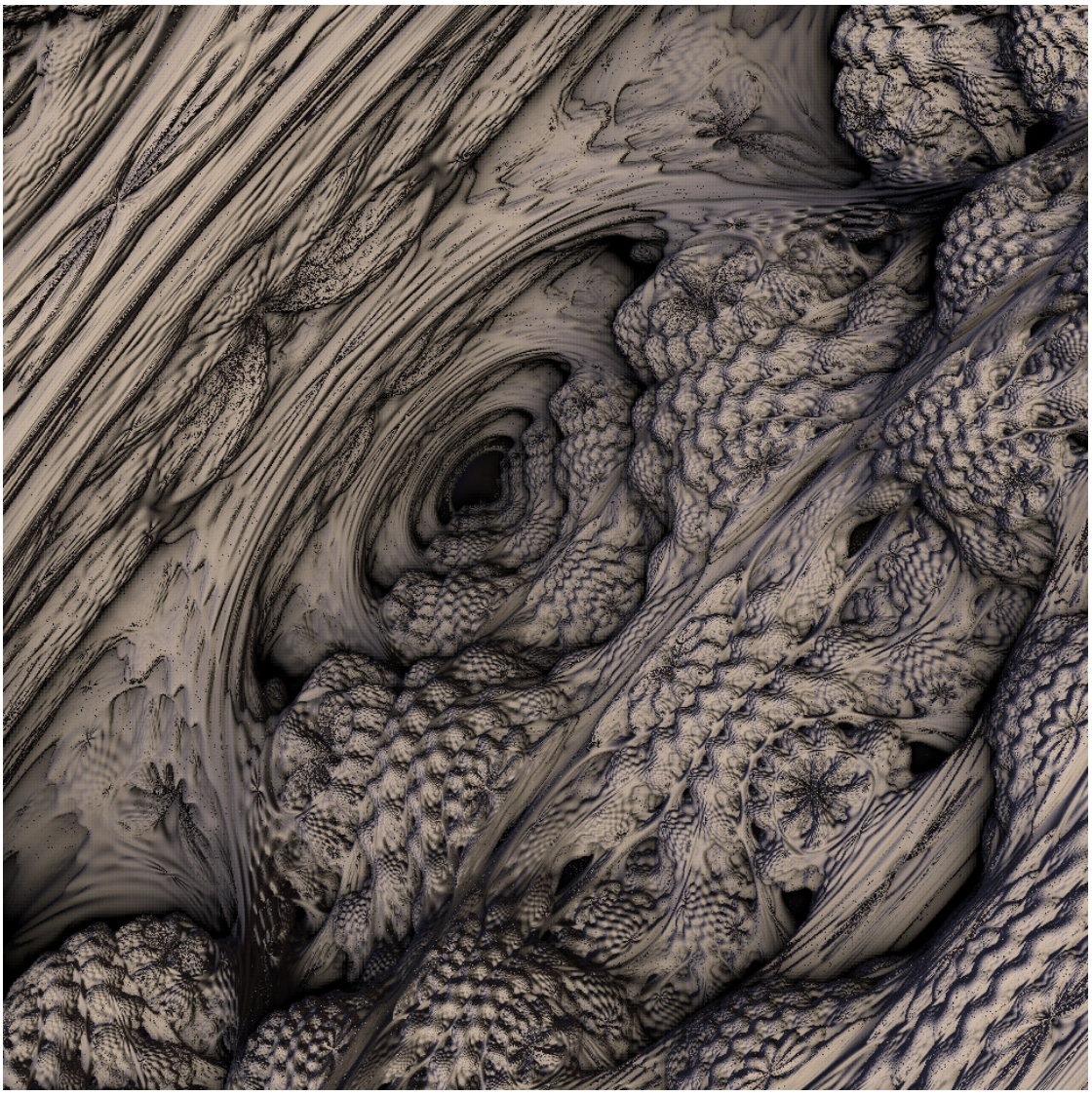


Figure A.10: Front view, our method, no shadows, 32.4 fps.

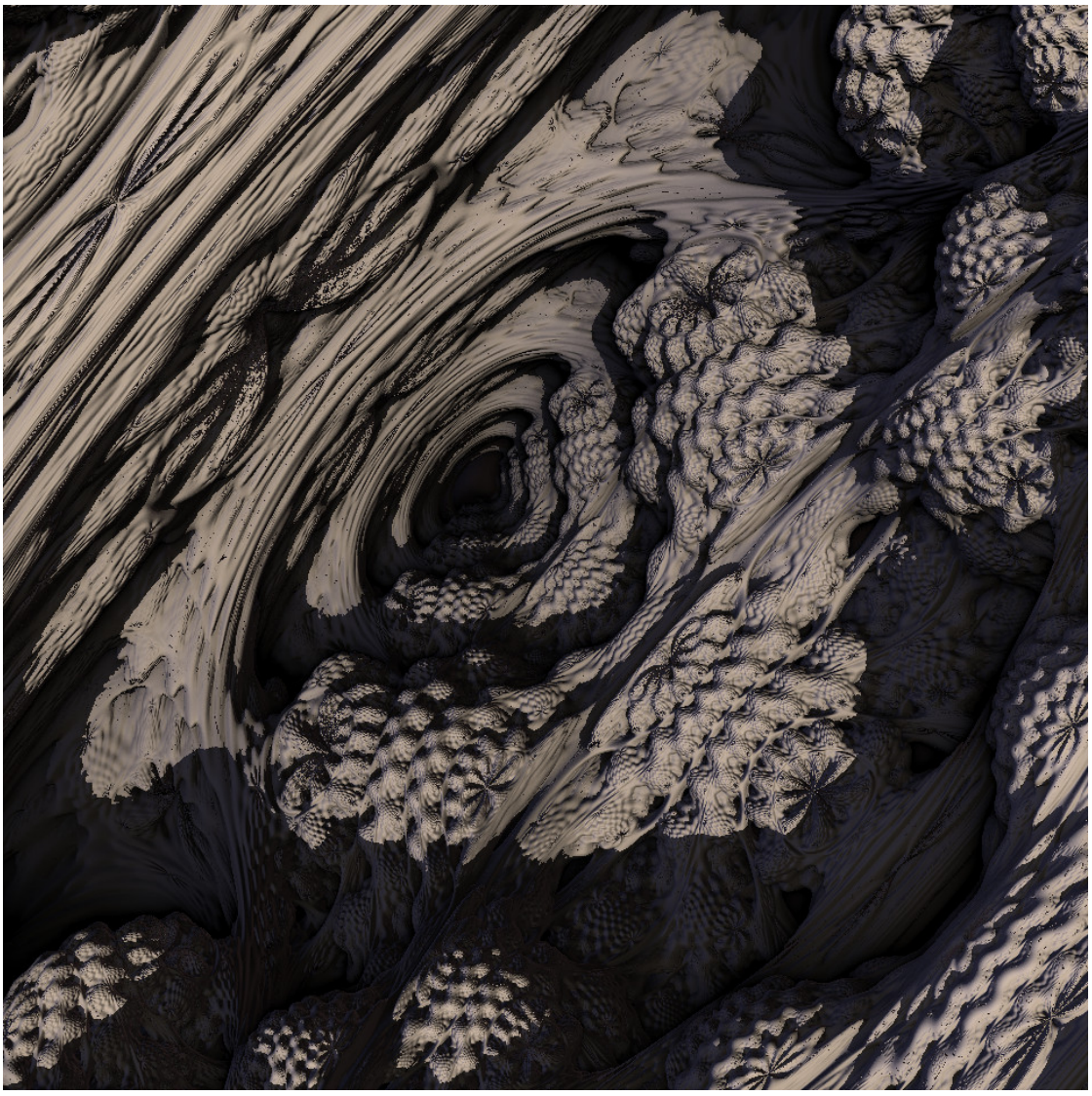


Figure A.11: Front view, traditional method, shadows, 9.4 fps.

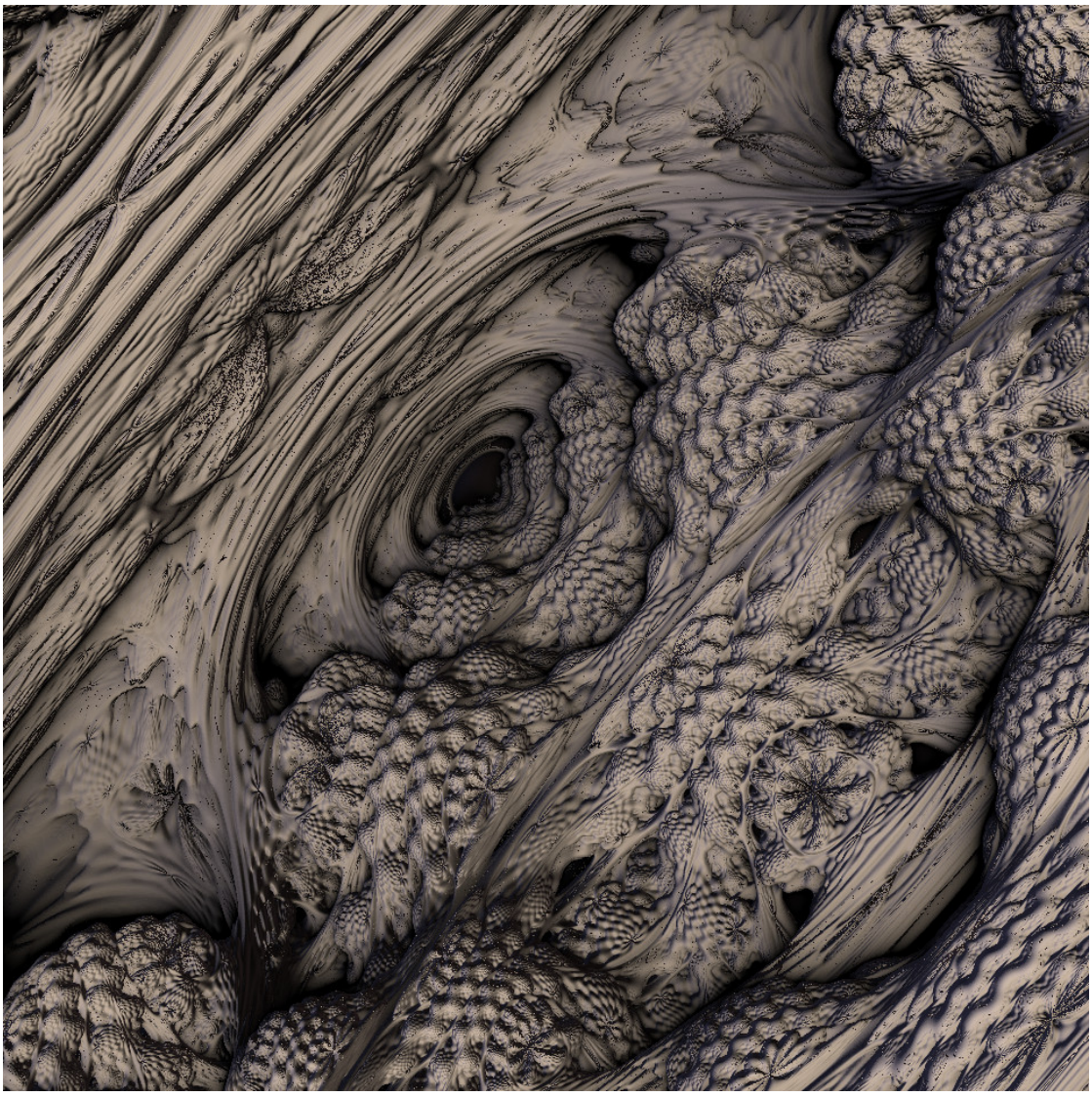


Figure A.12: Front view, traditional method, no shadows, 20.6 fps.

A.4 Back view

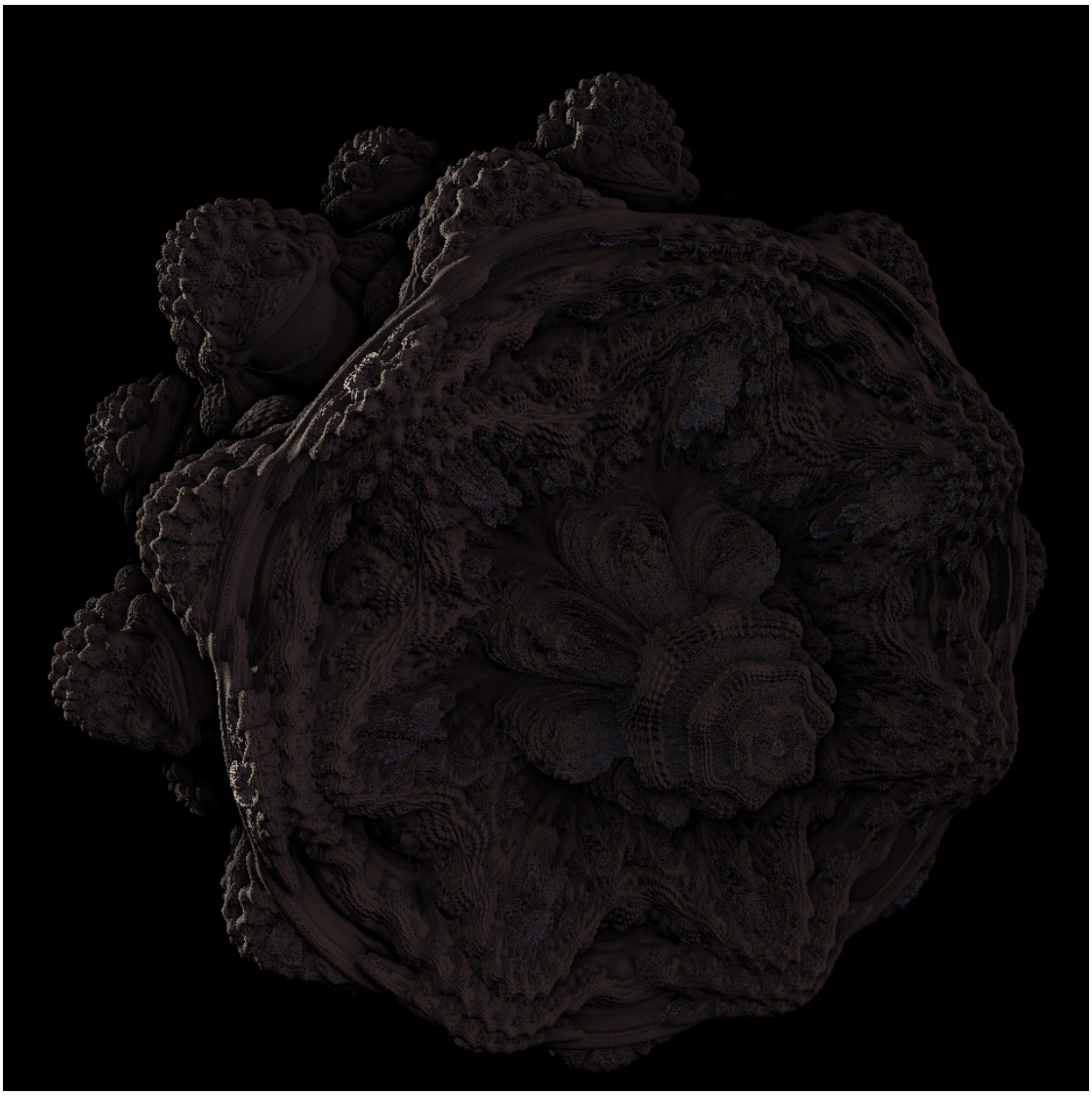


Figure A.13: Front view, our method, shadows, 26.2 fps.

Figure A.14: Front view, our method, no shadows, 60.1 fps.

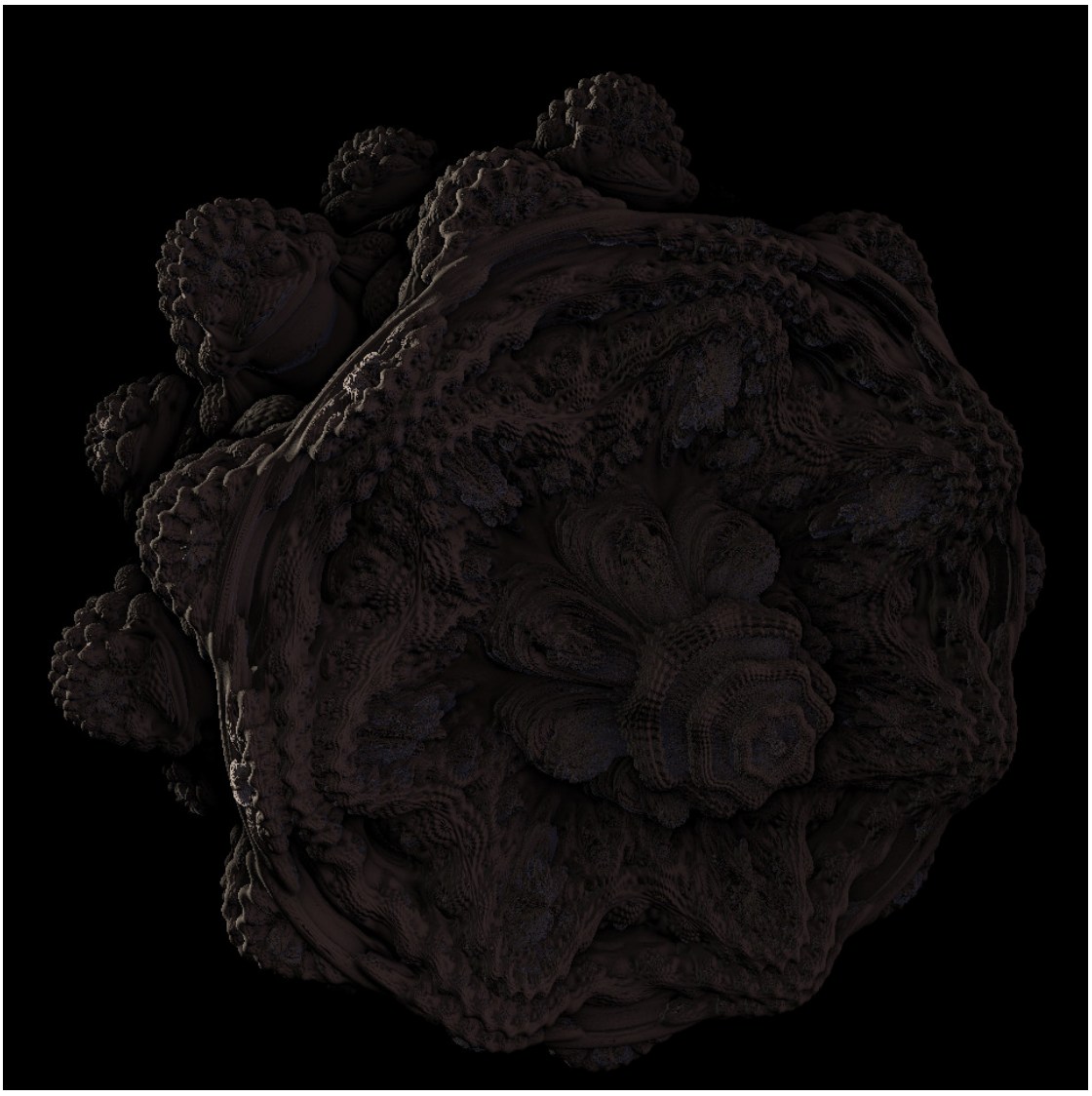


Figure A.15: Front view, traditional method, shadows, 24.1 fps.

Figure A.16: Front view, traditional method, no shadows, 48.0 fps.

Bibliography

[1] Tom Beddard. 3D Mandelbulb Ray Tracer. *http://www.subblue.com/projects/mandelbulb (retrieved 2013-06-22)*.

[2] R Brooks and JP Matelski. The dynamics of 2-generator subgroups of PSL (2, C). *Riemann Surfaces and Related Topics, Proceedings of the 1978 Stony Brook Conference*, 1981.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23(3):777–786, 2004.

[4] Adam Celarek. Merging Ray Tracing and Rasterization in Mixed Reality. *Vienna University of Technology*, 2012.

[5] Mikael Hvidtfeldt Christensen. Distance Estimated 3D Fractals (Part I). *http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/ (retrieved 2013-06-20)*, 2011.

[6] Mikael Hvidtfeldt Christensen. Distance Estimated 3D Fractals (V): The Mandelbulb & Different DE Approximations. *http://blog.hvidtfeldts.net/index.php/2011/09/distance-estimated-3d-fractals-v-the-mandelbulb-different-de-approximations/ (retrieved 2013-06-19)*, 2011.

[7] Mikael Hvidtfeldt Christensen. Distance Estimated 3D Fractals (VI): The Mandelbox. *http://blog.hvidtfeldts.net/index.php/2011/11/distance-estimated-3d-fractals-vi-the-mandelbox/ (retrieved 2013-06-22)*, 2011.

[8] Mikael Hvidtfeldt Christensen. Distance Estimated 3D Fractals (Part VIII): Epilogue. *http://blog.hvidtfeldts.net/index.php/2012/05/distance-estimated-3d-fractals-part-viii-epilogue/ (retrieved 2013-06-22)*, 2012.

[9] Mikael Hvidtfeldt Christensen. Rendering 3D fractals without a distance estimator. *http://blog.hvidtfeldts.net/index.php/2012/09/rendering-3d-fractals-without-a-distance-estimator/ (retrieved 2013-06-22)*, 2012.

[10] Keenan Crane. Ray Tracing Quaternion Julia Sets on the GPU. *http://users.cms.caltech.edu/~keenan/project_qjulia.html (retrieved 2013-06-22)*.

[11] Christopher D'Andrea. PARADIGM - 3D Fractal Animated Short Film. *http://www.youtube.com/watch?v=sanWdJ7rqbQ (retrieved 2013-06-22)*, 2013.

[12] A K Dewdney. Computer Recreations, A computer microscope zooms in for a look at the most complex object in mathematics. *SCIENTIFIC AMERICAN*, pages 16–24, 1985.

[13] User Efecretion. WIKIMEDIA COMMONS: BW Mandelbrot-Set. *http://commons.wikimedia.org/wiki/File:BW_Mandelbrot-Set-Whole_de-called-Apfelmaennchen.jpg (retrieved 2013-06-17)*, 2012.

[14] Alex Evans. Fast approximations for global illumination on dynamic scenes. *Advanced Real-Time Rendering in 3D Graphics and Games SIGGRAPH 2006*, pages 153 – 171, 2006.

[15] K J Falconer. *Fractal geometry: mathematical foundations and applications*. Wiley, 2003.

[16] J. C. Hart, D. J. Sandin, and L. H. Kauffman. Ray tracing deterministic 3-D fractals. *ACM SIGGRAPH Computer Graphics*, 23(3):289–296, July 1989.

[17] User Hellisp. WIKIMEDIA COMMONS: Cantor set in seven iterations. *http://commons.wikimedia.org/wiki/File:Cantor_set_in_seven_iterations.svg*, 2007.

[18] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, (1), 2010.

[19] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

[20] Benoit Mandelbrot. *The Fractal Geometry of Nature*. Henry Holt and Company, 1983.

[21] Benoit Mandelbrot. *Fractals and Chaos: The Mandelbrot Set and Beyond*. Selecta (Springer). Springer, 2004.

[22] C Nvidia. NVIDIA CUDA programming guide. 2011.

[23] Paul Nylander. Hypercomplex Fractals. *http://www.bugman123.com/Hypercomplex/index.html (retrieved 2013-06-19)*, 2009.

[24] K Olukotun and L Hammond. The future of microprocessors. *Queue*, pages 67–77, 2005.

[25] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879 – 899, 2008.

[26] H L Stiltz. *Aerospace telemetry*. Number v. 1 in Prentice-Hall space technology series. Prentice-Hall, 1961.

[27] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pages 839–846, 1998.

[28] Daniel White. The Unravelling of the Real 3D Mandelbulb. *http://www.skytopia.com/project/fractal/mandelbulb.htm (retrieved 2013-06-18)*, 2009.

[29] Daniel White. The Unravelling of the Real 3D Mandelbulb (page 2). *http://www.skytopia.com/project/fractal/2mandelbulb.html (retrieved 2013-06-18)*, 2009.

[30] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.