



NTNU – Trondheim
Norwegian University of
Science and Technology

3D Visualization of X-ray Diffraction Data

Thomas Løfsgaard Falch

Master of Science in Computer Science

Submission date: June 2012

Supervisor: Anne Cathrine Elster, IDI

Co-supervisor: Dag W. Breiby, IFY

Norwegian University of Science and Technology
Department of Computer and Information Science

3D VISUALIZATION OF X-RAY DIFFRACTION DATA

Thomas Løfsgaard Falch

Problem Description

With the introduction of fast area detectors, X-ray diffraction experiments now routinely yield data sets on the order of several gigabytes when studying nanostructured materials with the method known as "reciprocal space mapping". Due to complicated scattering geometries, the data points do not fall on any grid, nor is the density of points constant in all directions. These facts, along with the large data volumes, makes creating meaningful visualizations challenging. The overall goal of this project is to develop methods to create high-quality visualizations of three-dimensional X-ray diffraction data. This will be done by extending and modifying volume rendering techniques such as ray casting. Due to the computationally intensive nature of the problem, modern multi-core processors and programmable GPUs will be considered.

Assignment given: 15. January 2012

Supervisor: Anne C. Elster, IDI

Co-supervisor: Dag W. Breiby, IFY

Abstract

X-ray diffraction experiments are used extensively in the sciences to study the structure, chemical composition and physical properties of materials. The output of such experiments are samples of the diffraction pattern, which essentially constitutes a 3D unstructured dataset. In this thesis, we develop a method for visualizing such datasets.

Our visualization method is based on volume ray casting, but operates directly on the unstructured samples, rather than resampling them to form voxels. We estimate the intensity of the X-ray diffraction pattern at points along the rays by interpolation using nearby samples, taking advantage of an octree to facilitate efficient range search. The method is implemented on both the CPU and the GPU.

To test our method, actual X-ray diffraction datasets is used, consisting of up to 120M samples. We are able to generate images of good quality. The rendering time varies dramatically, between 5 s and 200 s, depending upon dataset, and settings used. A simple performance model is developed and empirically tested to better understand this variation. Our implementation scales exceptionally well to more CPU cores, with a speedup of 5.9 on a 6-core CPU. Furthermore, the GPU implementation achieves a speedup of around 4.6 compared to the CPU version.

Sammendrag

Røntgendiffraksjonseksperimenter er i utstrakt bruk i naturvitenskapene for å studere stukturen, den kjemiske sammensetningen, og de fysiske egenskapene til materialer. Resultatet av slike eksperimenter er punktprøver av diffraksjonsmønsteret, som i hovedsak er et 3D ustrukturert dataset. I denne oppgaven utvikler vi en metode for å visualisere slike dataset.

Vår visualiseringsmetode er basert på volumstrålekasting, men opererer direkte på de ustrukturerte punktprøvene, heller enn å repunktprøve dem for å lage volumelementer. Vi estimerer intensiteten til røntgendiffraksjonsmønsteret på punkter langs strålene ved å interpolere med nærliggende punktprøver, og tar fordel av et octree for å tilrettelegge for effektive søk. Metoden blir implementert både for CPUer og GPUer.

For å teste metoden vår, blir virkelige røntgendiffraksjonsdataset, bestående av opptil 120M punktprøver, brukt. Vi er i stand til å generere bilder med god kvalitet. Bildedannelsestiden varierer dramatisk, mellom 5 s og 200 s, avhengig av datasetet og instillingene brukt. En enkel ytelsesmodell blir utviklet og empirisk testet for bedre å forstå denne variasjonen. Vår implementasjon skalerer usedvanlig vel til flere CPU-kjerner, hastigheten øker med en faktor på 5,9 på en 6-kjerne CPU. Videre oppnår GPU-versjon en hastighetsøkning på omkring 4,5 sammenlignet med CPU-versjonen.

Acknowledgements

This masters thesis is the result of work done at the HPC-lab at the Department of Computer and Information Science in collaboration with the Department of Physics at the Norwegian University of Science and Technology.

I would like to thank my supervisor, Dr. Anne C. Elster and my co-supervisor Dr. Dag Breiby for invaluable feedback and guidance, and for allowing me to work with a problem I found truly interesting. I would also like to thank PhD candidate Jostein Fløystad for his valuable help and support.

I and Dr. Anne C. Elster would like to thank the Department of Computer and Information Science, NVIDIA and AMD for their hardware donations to the HPC-lab, which made this work possible.

I am indebted to my family for their encouragement and support throughout my studies, and in particular these last hectic months.

Finally, I want to thank the other students at the HPC-lab for creating a fun, inspiring and motivating work environment, and the Computer Science and Nanotechnology classes of 2012 for five great years in Trondheim.

Thomas Løfsgaard Falch
Trondheim, June 2012

Contents

Problem Description	iii
Abstract	v
Sammendrag	vii
Acknowledgements	ix
1 Introduction	1
1.1 Outline	3
2 X-ray Diffraction	5
2.1 X-rays Interaction with Matter	5
2.2 X-ray Diffraction Patterns	6
2.3 Experimental Setup	7
3 Volume Rendering	11
3.1 Volumetric Data	11
3.2 Indirect Volume Rendering	11
3.3 Direct Volume Rendering Techniques	12
3.3.1 Image Order Techniques	12
3.3.2 Object Order Techniques	14
3.4 The Volume Rendering Integral	14
4 Parallel and GPU Computing	17
4.1 Parallel Computing	17
4.1.1 Parallel Computer Architecture	17
4.1.2 Parallel Scaling	18

4.2	GPGPU Computing	19
4.2.1	CPUs Compared to CPUs	20
4.3	CUDA	21
4.3.1	Programing Model	21
4.3.2	Hardware Implementation	23
4.3.3	Performance Considerations	24
4.3.4	Maximize Memory Throughput	24
5	Multivariate Interpolation	27
5.1	Trilinear Interpolation	27
5.2	Inverse Distance Weighting	28
5.3	Kriging	29
5.4	Anisotropic Interpolation	30
6	Related Work	35
7	Implementation	37
7.1	Overview	37
7.1.1	Design Choices	39
7.2	Preprocessing and Filtering	40
7.2.1	Input Data and Preprocessing	40
7.2.2	Filtering	41
7.3	Volume Ray Casting	42
7.3.1	Ray Creation	42
7.3.2	Casting a Single Ray	43
7.3.3	Optimizations	44
7.3.4	Colors	46
7.4	Range Search	47
7.4.1	Octree	47
7.5	Parallelization	51
7.6	Interactivity	51
7.6.1	Incremental Update	52
8	GPU Implementation	55
8.1	Overview	55
8.2	Removing Recursion	56
8.3	Memory Considerations	58
8.3.1	Precision	58
8.3.2	Optimizations	58

9	Results and Discussion	61
9.1	Methodology	61
9.1.1	Datasets	61
9.1.2	Testing Environment	62
9.1.3	Measurements	63
9.2	Overview	63
9.3	Filtering	65
9.3.1	Median Filtering	66
9.4	Interpolation	66
9.4.1	Interpolation Techniques	66
9.4.2	Anisotropic Interpolation	68
9.5	Memory Consumption	70
9.6	Performance	72
9.6.1	Performance Model	73
9.6.2	Performance Results and Discussion	75
9.6.3	Parallel Scaling	80
9.7	GPU	81
9.7.1	GPU Compared to CPU	81
9.7.2	GPU Optimizations	82
10	Conclusion and Future Work	85
10.1	Future Work	86
A	RSV User Manual	95
A.1	Quick Start	95
A.2	Overview	95
A.2.1	Raycasting	96
A.2.2	Files	96
A.3	Compilation	97
A.3.1	Dependencies	97
A.3.2	Compilation	98
A.4	Use	99
A.4.1	Input File	99
A.4.2	Moving the Camera	100
A.4.3	Grid	100
A.4.4	HUD	100
A.4.5	Color	100
A.4.6	Saving Images	100
A.4.7	Trace Single Ray	101

A.4.8	Configuration File	101
A.4.9	Batch Mode	104
A.4.10	Single Image Mode	105
A.4.11	Quitting	105
A.5	Development	105
B	Transfer Functions	109
C	Selected Source Code	111
D	Poster	117

List of Figures

2.1	Scattering of X-rays	6
2.2	X-ray scattering by free electrons	7
2.3	X-ray diffraction experiment setup	8
2.4	Geometry of results of diffraction experiment	9
3.1	Illustration of volume data	12
3.2	Image order direct volume rendering	13
3.3	Illustration of emission and absorption	15
4.1	Comparison of GPU and CPU	20
4.2	Thread hierarchy	22
5.1	Trilinear interpolation	28
5.2	The effect of rapidly changing functions on interpolation radius	31
5.3	Anisotropic distance	32
5.4	Anisotropic functions and interpolation	33
7.1	Overview of implementation	38
7.2	Spatial distribution of input data	41
7.3	Schematic depiction of camera	42
7.4	Illustration of problems with empty space skipping	45
7.5	Illustration of our empty space skipping algorithm	45
7.6	Transfer function	46
7.7	Sample octree	48
7.8	Searching in octree	50
7.9	Screenshot	52
7.10	Multi-resolution/incremental update	53

8.1	Stack optimization	57
9.1	Visualization of the 27uc dataset	63
9.2	Visualization of the 00571 dataset	64
9.3	Results of filtering	65
9.4	Visual results of different interpolation methods.	67
9.5	Effect of varying radius and anisotropic matrix	69
9.6	Memory consumption	71
9.7	Octree depth	71
9.8	Timing results with varying search radius	76
9.9	Timing results with varying step size	76
9.10	Timing results for different interpolation methods	77
9.11	Timing results for filtering	79
9.12	Timing results with varying transfer function	79
9.13	Speedup with more threads	80
9.14	Speedup on GPU	81
9.15	Results of GPU optimizations	83
A.1	Raycasting	96
A.2	Schematic overview of raycasting in 2D	97
A.3	Callgraph	107
B.1	Transfer function f1	109
B.2	Transfer function f2	109
B.3	Transfer function t1	110
B.4	Transfer function t2	110
B.5	Transfer function t3	110
B.6	Transfer function t4	110
B.7	Transfer function t5	110

List of Tables

9.1	Hardware and software used in tests	62
9.2	Settings used in Figure 9.5	68
9.3	Tree depth, and memory consumption	72
9.4	Timing results	73

Chapter 1

Introduction

X-rays are perhaps best known for their application in medicine. However, X-rays are also extensively used in the sciences to study the structure, chemical composition and physical properties of materials. In one kind of experiment, an X-ray beam is directed onto a sample of a material, and measurements of how the material scatters, or spreads, the X-rays are made. The outcome of these experiments is essentially samples of a three-dimensional scalar field, known as the diffraction pattern.

It is difficult to analyze these samples of the diffraction pattern automatically. To extract information about the material being studied, it is necessary to visualize the diffraction pattern and let experts interpret the resulting images. The ability to turn the measured data into high quality visualizations is therefore of great importance.

When the data is two-dimensional, generating such images is trivial. However, recent advances in sensor technology has made it possible to perform measurements much more efficiently. This has led to an increasing number of three-dimensional datasets. Visualizing this kind of data remains a challenge.

One approach is to visualize two-dimensional slices of these datasets. While simple, the utility of the images generated with this method remains limited, as much mental effort is required to reconstruct the full, three-dimensional, diffraction pattern.

A better solution is to visualize the three-dimensional data directly. While techniques and tools exists for visualizing three dimensional datasets of scientific origin, the nature of the X-ray data makes most of them unsuitable. For practical reasons, the diffraction pattern cannot be sampled on a uniform grid.

Hence, the data is not in the form of voxels, but rather in what can essentially be regarded as an unstructured 'cloud' of samples. To complicate matters further, the samples are not uniformly distributed, but organized in layers, where the distance between different layers is significantly larger than the distance between points in the same layer. The sheer size of the datasets also present a challenge, they frequently consists of millions of samples, resulting in gigabytes of raw data.

Since their invention, computers have been an indispensable aid for the sciences, making problems involving massive amounts of numerical computation possible to solve. As the preceding discussion hints at, and the remainder of this thesis will make clear in more detail, visualizing 3D diffraction patterns is one such problem. Previously, the most demanding problems required specialized and costly computers. However, the continued validity of Moore's law has made consumer grade hardware ever more capable. In particular, modern, programable graphical processing units (GPUs) have brought teraFLOPS scale computing to the PC [1].

While originally developed to render 3D graphics for games and similar applications, GPUs have recently emerged as a compelling alternative to the CPU for general purpose computations. The GPU is designed for a special kind of workload, those that combine high degrees of parallelism with high computational intensity, making it ideally suited for scientific computing.

In this thesis, we tackle the problem of creating 3D visualizations of X-ray diffraction data. Our main goal is to generate images of the highest possible visual quality. In order to be useful for scientists analysing the results of X-ray diffraction experiments, the generated images must accurately depict the diffraction pattern, and be free of artefacts. Furthermore, we aim at making our method and implementation as flexible and adjustable as possible, so that it can successfully be used for datasets with varying properties, and to highlight different parts of the same dataset.

A secondary goal is to achieve good performance. This includes both selecting an efficient rendering algorithm using fast data structures, and also developing a high performing, optimized, implementation. In particular, we are interested in how modern, programmable GPUs can be used.

Finally, realizing that the two previous goals are somewhat conflicting, we attempt to find ways to dynamically trade of speed for performance, thus allowing a low quality overview of the diffraction pattern to be rendered at interactive or even real time speed. This can be highly useful to quickly find good camera positions and settings, without having to wait for the slow, high quality images to become ready.

1.1 Outline

This thesis is structured as follows:

Chapter 2 provides background information about X-rays, X-ray scattering and diffraction, and X-ray diffraction patterns.

Chapter 3 provides background information about direct volume rendering, with an emphasis on volume ray casting.

Chapter 4 provides background information about parallel computing, general purpose GPU computing, and NVIDIA's CUDA framework.

Chapter 5 provides background information about multivariate interpolation of unstructured samples, and anisotropic interpolation.

Chapter 6 describes previous and related work in the fields of GPGPU computation, and direct volume rendering, with an emphasis on rendering of unstructured data.

Chapter 7 describes our implementation of a method using volume ray casting to visualize 3D X-ray diffraction patterns.

Chapter 8 describes how we modified our implementation to allow it to run on graphical processing units, using NVIDIA's CUDA framework.

Chapter 9 describes the performance results of our tool, both in terms of speed and image quality, and discusses these results.

Chapter 10 concludes, and discusses possible future work.

Appendix A contains the user manual for the tool we have implemented.

Appendix B lists the transfer functions used in some of our experiments.

Appendix C contains selected source code.

Appendix D contains a poster summarizing this thesis, to be displayed at the International Supercomputing Conference 2012.

Chapter 2

X-ray Diffraction

This chapter provides an introduction to X-rays, X-ray scattering and diffraction, and X-ray diffraction patterns. The remainder of this thesis describes how to visualize results from X-ray diffraction experiments, so an understanding of the origin of the data is vital for the full appreciation of the material. We will not attempt to cover this vast field in any detail, but rather provide a simple introduction aimed at the non-expert. The interested reader is referred to the extensive literature on the subject, see for instance [2] or [3].

2.1 X-rays Interaction with Matter

X-rays are electromagnetic waves with a wavelength in the range of 0.01 - 10 nm. While perhaps best known for their application in medicine, X-rays are also used extensively to study the structure, chemical composition and physical properties of materials. The wavelength of X-rays has the same order of magnitude as the distance between atoms and molecules in most common materials. For this reason, X-rays passing through a sample of a material will be scattered, that is, spread in new and different directions relative to the original beam, as illustrated in Figure 2.1. Measuring how the X-rays are scattered will reveal valuable information about the sample.

When X-rays interact with matter, several phenomena occur, which account for the above mentioned scattering. For simplicity, we will limit our discussion to elastic scattering, which we explain next.

When a sample is hit by an X-ray beam, each of the charged particles, such

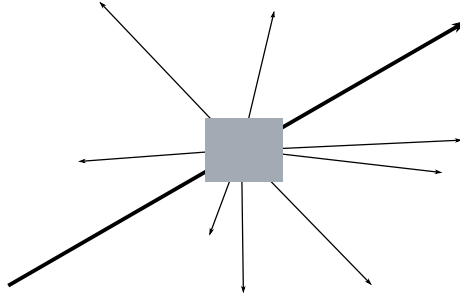


Figure 2.1: The X-rays of the incident beam are scattered as they pass through the sample.

as electrons, in the sample will experience the time varying electrical field of the X-rays. The varying electrical field will cause the particles to oscillate. The oscillation will cause each of the charged particles to give rise to its own electromagnetic field, with the same wavelength as the waves originally causing the oscillation. That is, each particle becomes an X-ray source.

This concept is illustrated in Figure 2.2. The incoming X-rays, the wave crests of which are shown in dark gray, cause the two charged particles at a and b to oscillate. The oscillating electrons give rise to two new spherical X-ray waves, spreading out from the electrons.

The incoming X-ray wave and the new waves will combine to form a composite wave which will be the superimposition of the individual waves. In some directions the waves will cancel each other out, in others combine constructively. Measuring the composite wave will therefore result in a diffraction pattern.

2.2 X-ray Diffraction Patterns

The diffraction pattern is essentially the intensity of the scattered X-rays as a function of the direction of scattering. This direction is specified by the scattering vector \mathbf{Q} which is defined as $\mathbf{Q} = \mathbf{k}_r - \mathbf{k}_i$. Here \mathbf{k}_i is the direction of the incoming beam, while \mathbf{k}_r is the direction of the outgoing, scattered wave. Both of these have the same length, that is: $|\mathbf{k}_i| = |\mathbf{k}_r| = \frac{2\pi}{\lambda}$, where λ is the wavelength of the X-rays. This is illustrated in Figure 9.1.

While the derivation is beyond the scope of this exposition, it can be shown that:

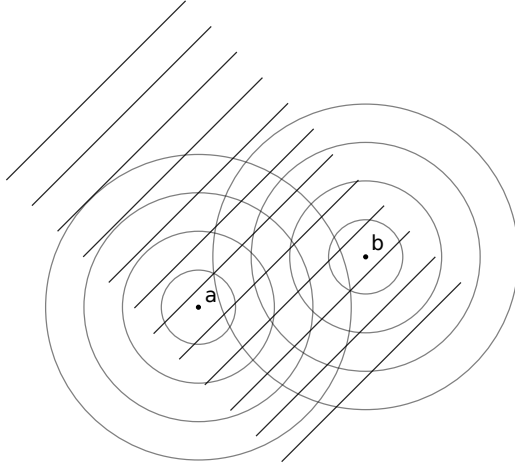


Figure 2.2: Illustration of X-ray scattering by free electrons. The incoming X-rays, indicated by the dark gray, straight lines, cause the electrons at a and b to oscillate, and send out their own X-ray waves, indicated by the concentric, light gray, circles.

$$A(\mathbf{Q}) = \int \rho(\mathbf{r})e^{i\mathbf{Q}\cdot\mathbf{r}}d\mathbf{r}$$

$$I(\mathbf{Q}) = \overline{A(\mathbf{Q})}A(\mathbf{Q})$$

Where $\rho(\mathbf{r})$ is the electron density of the sample, and $A(\mathbf{Q})$ is the amplitude and $I(\mathbf{Q})$ the intensity of the scattered wave. That is, the diffraction pattern is essentially the Fourier transform of the electron density of the sample. This relationship is at the heart of X-ray diffraction analysis, making it possible to relate the diffraction pattern to the structure and properties of the sample.

2.3 Experimental Setup

Figure 9.1 shows the setup of the kinds of experiments performed to generate the data we use as input for the tool described in this thesis [4, 5]. A 2D sensor array is used to measure the scattered X-rays from a beam incident on the sample. For each pixel in the array, the corresponding scattering vector can be computed, to obtain a set of samples of the diffraction pattern. By rotating the

sensor array relative to the sample and beam, or rotating the sample relative to the beam and sensor array, a new set of diffraction pattern samples can be gathered.

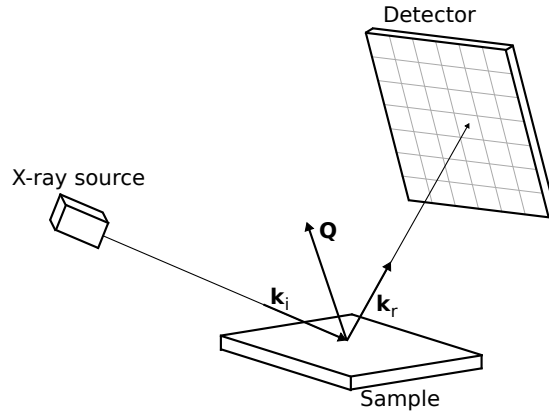


Figure 2.3: X-ray diffraction experiment setup. X-rays from the source hits the sample, and are scattered. The scattered X-rays are measured with an area detector. Each pixel in the sensor has a corresponding scattering vector \mathbf{Q}

The samples will then have the form (Q_x, Q_y, Q_z, I) where Q_x , Q_y and Q_z are the coordinates of the scattering vector \mathbf{Q} , and I is the measured intensity. The samples can be thought of as points in 3D space, with an attached intensity value.

Because of the geometry of the setup, all the samples from one frame will be placed on the same curved surface in 3D space, as shown in Figure 2.4. Measuring multiple frames will result in multiple surfaces, each with a slightly different curvature. For practical reasons, the distance between two such surfaces will be greater than the distance between two neighbouring samples on the same surface.

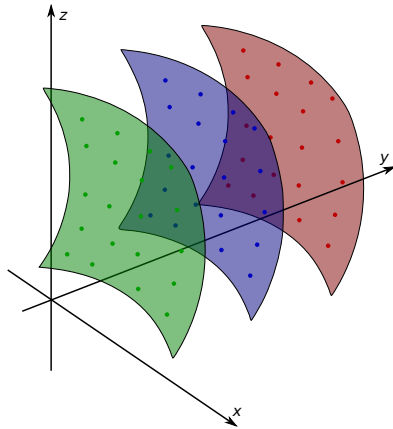


Figure 2.4: All the sets of samples from one frame will fall on the same curved surface. Here, three such curved surfaces are shown, in different colors, with the samples belonging to each surface indicated in the same color. The shape of different surfaces is not identical.

Chapter 3

Volume Rendering

Volume rendering is the process of generating a 2D image of a 3D volumetric dataset. In this chapter, we examine some volume rendering techniques, with a special emphasis on volume ray casting, which is the method we use.

3.1 Volumetric Data

Volumetric data is a set of samples of the form (x, y, z, v) , where v is the value of some property at the location (x, y, z) in 3D space. In general, the samples may be taken at random locations. However, it is frequently the case that the samples are taken on a regular three-dimensional grid. The latter case will result in a set of *voxels*¹. A voxel is a cubic volume element corresponding to a sample at its center. Volumetric data is illustrated in Figure 3.1.

Volumetric datasets arise frequently in medicine and the sciences. Some examples include stacks of 2D magnetic resonance imaging (MRI) and computed tomography (CT) scans, the results of numerical simulations using finite element methods, and geological, seismic and meteorological data.

3.2 Indirect Volume Rendering

Before proceeding to volume rendering proper, it should be noted that it is frequently possible to render volumetric datasets by converting it into, or ex-

¹From volumetric pixel, or volumetric picture element.

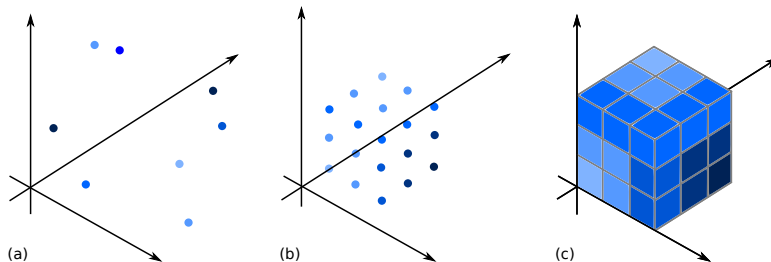


Figure 3.1: Illustration of volume data. (a) shows unstructured data, where the samples are taken at random locations. (b) shows samples taken on a uniform grid. (c) shows a voxel representation of the samples from (b), that is, (b) and (c) show different representations of the same data. Different shades of blue indicate different values of the sampled property.

tracting from it, geometric primitives, and then render those primitives using standard techniques. An example of this is isosurface extraction. The set of samples is partitioned by a binary classifier in an application dependent way, for instance into a high-intensity and low-intensity partition. The interface between the partitions is then rendered as a polygonal mesh, using an algorithm like marching cubes [6]. Such techniques are effectively discarding much of the information in the 3D dataset, resulting in inferior images. In particular, objects without a clearly defined surface, such as clouds or flames, are a poor fit for this kind of rendering.

3.3 Direct Volume Rendering Techniques

Following [7], direct volume rendering techniques can broadly be classified into *image order* and *object order* techniques. Image order techniques starts with the pixels, and determines, for each pixel, which samples will influence its value. Object order techniques starts with the samples, and determines, for each sample, the pixels whose values it will influence.

3.3.1 Image Order Techniques

All image order techniques share the same framework, as illustrated in Figure 3.2. Form an eye/camera, rays of sight are cast, through each pixel, and into the volume. At points along the ray samples of the value of the volume will be

obtained, using some kind of interpolation. The estimates at the points are then mapped to color and opacity values, and finally combined to obtain the color of the pixel. It is in this last step that the techniques vary. Some alternatives are:

X-ray The values of the samples are simply summed, and the sum mapped to a color.

Maximum intensity projection The sample with the highest value along each ray is mapped to a color. All other samples along the ray are discarded [8]. A variation, local maximum intensity projection, uses the first value along the ray above a certain threshold, rather than the global maximum [9].

Full volume ray casting All of the samples along a ray are combined to find the color of the corresponding pixel. Different techniques with varying degrees of sophistication exists for performing the combination.

Full volume ray casting, commonly referred to as just ray casting or ray marching, is clearly the most capable and flexible of these, and have been widely used and studied [10, 11, 7].

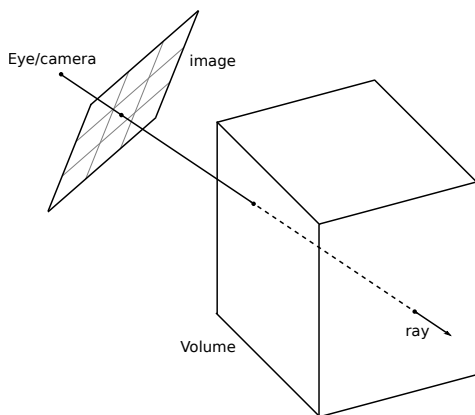


Figure 3.2: The standard image order direct volume rendering framework. From a eye/camera, rays are cast through each pixel and into the volume. The value of the volume along the ray is used to determine the color of the pixel.

3.3.2 Object Order Techniques

In object order techniques, each sample is projected onto the screen, and assembled into a final image. Several different techniques exist, which we will describe in the following.

Splatting

One object-order technique is *splatting* [12]. Here, an image plane footprint is calculated for each sample. The samples are traversed in either front-to-back or back-to-front order, and each new footprint is composited into an incrementally updated buffer.

Shear-Warp Factorization

The basic idea of shear-warp factorization [13] is to transform the volume into an intermediate form where projection is simpler. In more detail, the view transformation matrix, which transforms samples from volume to image space is factorized:

$$M_{view} = S * M_{warp}$$

S shears the volume so that the front face of the volume is parallel with the image plane. The volume is then projected onto the image, and the image is warped with M_{warp} to correct for the distortion caused by the shear.

Texture Mapping

Texture mapping [14] works by cutting slices through the volume, parallel to the image plane. The volume is then sampled to find the values at each of the pixels of the slices. Finally, the slices are composited together to form the final image. The technique resembles shear-warp factorization, but the volume is not sheared prior to cutting the slices, so in general, they are not parallel with any face of the volume. Texture mapping can be done in hardware on GPU's [15, 16].

3.4 The Volume Rendering Integral

As described above, during full volume ray casting, it is necessary to combine all the samples along a ray to determine the color value of the corresponding

pixel. This can be done by evaluating the *volume rendering integral* [17] along the ray.

In the following, the simple emission-absorption optical model is assumed. Each point in the volume emit and absorb certain amounts of light. More advanced models, taking into account effects such as scattering and indirect illumination, exists [18, 17], but will not be presented here.

Furthermore it is initially assumed that the volume is continuous. This assumption will be relaxed later.

Let $\mathbf{x}(t)$ denote a ray, parameterized by t , the distance from the eye. Then $s(\mathbf{x}(t))$ is the scalar value, $c(s(\mathbf{x}(t)))$ the emitted light and $\kappa(s(\mathbf{x}(t)))$ the absorbed light a distance t form the eye along the ray. For notational convenience, we define:

$$c(t) := c(s(\mathbf{x}(t)))$$

$$\kappa(t) := \kappa(s(\mathbf{x}(t)))$$

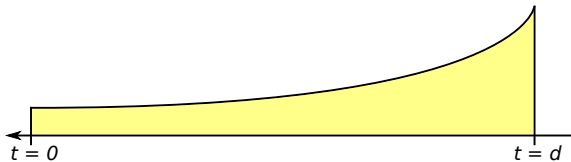


Figure 3.3: Illustration of emission and absorption. Some of the light emitted at $t = d$ will be absorbed along the way to the eye. Figure from [17].

If c light is emitted a distance d from the eye, only a part of it, c' will reach the eye, the remainder will be absorbed along the way, as shown in Figure 3.3. If constant absorption of κ is assumed, then:

$$c' = c \cdot e^{-\kappa d}$$

If, however, the absorption varies, it must be integrated along the ray:

$$c' = c \cdot e^{-\int_0^d \kappa(\hat{t}) d\hat{t}}$$

This is the contribution to the total amount of light reaching the eye along the ray from a single point. The total amount C can be found by integrating over all the points along the ray:

$$C = \int_0^N c(t) \cdot e^{-\int_0^t \kappa(\hat{t}) d\hat{t}} dt$$

Where N is the distance from the eye to the most distant point in the volume. This is the volume rendering integral.

In practice this integral cannot be evaluated analytically, but must be approximated numerically. Letting

$$C_i = c(i \cdot \Delta t) \Delta t$$

and

$$K_i = \kappa(i \cdot \Delta \hat{t}) \Delta \hat{t}$$

the integral can be approximated using the standard rectangle method:

$$C \approx \sum_{i=0}^n C_i \cdot e^{-\sum_{j=0}^i K_j} = \sum_{i=0}^n C_i \cdot \prod_{j=0}^i e^{-K_j}$$

Here, C_i can be thought of as the light emitted, while e^{-K_j} is the light absorbed, by a small ray segment. Rather than specifying absorption, it is more common to use *opacity*, defined as:

$$A_i = 1 - e^{-K_i}$$

which results in:

$$C \approx \sum_{i=0}^n C_i \cdot \prod_{j=0}^i (1 - A_j)$$

If the volume consists of discrete samples, rather than being continuous, as assumed so far, neither $c(t)$ nor $\kappa(t)$ can be found directly. One approach is to find these values based on the known samples, using interpolation. Multivariate interpolation is discussed in detail in Chapter 5.

Chapter 4

Parallel and GPU Computing

A recent trend in high performance computing is using graphics processing units for general purpose computation (GPGPU) [19, 20]. GPUs are massively parallel computational units. Combining high computational power, low cost and energy efficiency, GPUs offer a compelling alternative to CPUs for compute intensive tasks.

In this chapter we first provide a short introduction to parallel computing in general. We then proceed to introduce general purpose GPU computing, and explore NVIDIA's CUDA GPGPU framework in some detail.

4.1 Parallel Computing

Here, we will briefly introduce some concepts of parallel computing. More details can be found in our previous work [21], or in standard textbooks, such as [22].

4.1.1 Parallel Computer Architecture

There are several ways of classifying parallel computers. Firstly, we might look at the relationship between memory and compute units:

Shared memory In this type of computer, all the compute units access the same shared memory, and can use this memory for communication.

Distributed memory Each compute unit has its own private memory. Communication must be performed through explicit message passing.

Another classification scheme, known as Flynn's taxonomy [23], looks at the relationship between instruction and data streams:

Single Instruction, Single Data stream (SISD) These are nonparallel computers, where one processor uses one instruction stream to operate upon one data stream.

Multiple Instruction, Multiple Data streams (MIMD) This is essentially a combination of several SISD computers. Each compute units uses one instruction stream to operate on one data stream.

Single Instruction, Multiple Data streams (MIMD) Here, each compute unit uses the same instruction stream to operate upon different instruction streams, typically in lockstep.

The final combination, Multiple Instruction, Single Data stream (MISD) is of little practical interest.

4.1.2 Parallel Scaling

Ideally, increasing the number of processing units by a factor of n should decrease the computation time by the same factor. Most problems are, however not perfectly parallelizable, so more sophisticated models are required.

First we define parallel speedup as:

$$S = \frac{T_{serial}}{T_{parallel}}$$

Where T_{serial} and $T_{parallel}$ is the serial and parallel execution time, respectively. Given a problem consisting of a serial part which cannot be parallelized, and a perfectly parallelizable part, the speedup as a function of the number of processors is:

$$S = \frac{T_{serial}}{T_{parallel}} = \frac{s + p}{s + \frac{p}{N}} = \frac{1}{(1 - p) + \frac{p}{N}}$$

Where s is the fraction of execution time spent on the serial part, $p = 1 - s$ the fraction spent on the parallel part (on a serial computer) and N is the number of processors. This result is known as Amdahl's law [24].

We might instead start with s' and p' , the fraction of time spent on the serial and parallel parts on a parallel computer, respectively. The speedup will then be:

$$S = \frac{T_{serial}}{T_{parallel}} = \frac{s' + p' \cdot N}{s' + p'} = s' + p' \cdot N$$

This result is known as Gustafson's law [25]. While apparently contradictory, it is important to emphasize that Amdahl's and Gustafson's laws are equivalent, because s is not equal to s' . The two laws are based on different assumptions, Amdahl assumes that as the computer becomes more parallel, it will still be used to run the same problem. Gustafson assumes that the more parallel computer will be used on a larger instance of the problem.

4.2 GPGPU Computing

Graphics processing units were developed to accelerate the rendering of 2D, and later 3D graphics. This is a highly parallel task. The input, a list of geometric primitives, typically triangles, must be shaded and converted into screen space. This can be done independently for each vertex of each triangle. The triangles are then rasterized to fragments. Each fragment must then be shaded, and possibly textured. This can also be done in parallel for each fragment. Finally, all the fragments of one pixel are composited to find the color of that pixel.

Originally, GPUs were fixed function units. However, to satisfy the demand for better and more photorealistic graphics, they gradually became more configurable and programmable. In particular, the fixed per-vertex and per-fragment operations were replaced with shader programs, which would be executed once for each vertex or fragment.

As the programmability as well as the computational power of GPUs grew, it was realized that they could be used for other things than graphics. Initially, this was done rather awkwardly, by mapping the input to graphics primitives, the computation to shader programs, and the output to one or more images.

Soon, programming environments were developed that abstracted away the graphics details. At the same time, the underlying hardware became ever more flexible, and the combination allowed the GPU to be used as a general purpose, highly parallel, processor.

4.2.1 CPUs Compared to GPUs

While both are capable of general purpose computation, the hardware architecture of GPUs and CPUs differ dramatically. This is illustrated in Figure 4.1.

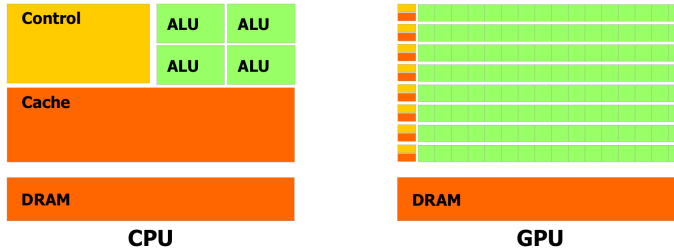


Figure 4.1: Comparison of GPU and CPU architecture. In a CPU, a large fraction of the transistors are used for flow-of-control logic and caches. GPUs, on the other hand, devote most of the transistors to computation. Figure from [1].

CPUs have traditionally been optimized to increase the performance and reduce the latency of a single thread. For this reason, a large number of the available transistors are used to implement advanced flow of control mechanisms such as branch prediction and out-of-order execution. Furthermore, large caches are used to reduce memory access latency. While modern CPUs have multiple cores, each of these are fairly independent, targeted at task parallelism.

As described above, graphics rendering is a highly parallel task. GPUs have therefore been designed to maximize the throughput of a large number of threads, at the expense of the performance of any single thread. As a consequence of this goal, GPUs devote a larger parts of the transistors to actual computation, sacrificing caches and flow-of-control logic. Thread switching is inexpensive, so latency can be hidden by simply executing another thread. While modern CPUs have 2-8 cores, modern GPUs have several hundred [1]. To reduce the amount of control logic even further, these cores are organized into groups, where all the cores in the group execute the same instruction, on different data. GPUs are therefore best suited for data parallel problems.

Since GPUs and CPUs have different memory spaces (this will be described in more detail in Section 4.3), referring to Section 4.1 above, it is clear that a system with a CPU and GPU is essentially a distributed memory computer with two nodes, the CPU and GPU. Each of these nodes can be considered a shared memory computer. Furthermore, such a heterogeneous system will be

beneficial for application performance. Referring to Amdahl’s law, the serially oriented CPU can be used to reduce s , while the GPU increases N . Thus, the two platforms complement each other.

4.3 CUDA

Released in 2006, NVIDIA’s *Compute Unified Device Architecture* (CUDA) is a parallel computing architecture that allows developers to use a modified version on C to do general purpose computation on GPUs [1]. In this section, we will briefly describe the CUDA programing model.

4.3.1 Programing Model

The CUDA programing model is based upon the SPMD paradigm, where the same program is executed in parallel on several data items. While resembling the SIMD paradigm, described in Section 4.1, SPMD is more flexible. Rather than having the same instructions performed on all the data in lock-step, SPMD allows for branching and loops.

Kernels

The program that is executed on the different data items is known as a kernel. Kernels are structured like regular functions, but are executed N times in parallel by N different threads. Each kernel is given a unique thread id that allows it to identify the data to operate on. A simple kernel, which multiplies the elements of two arrays are shown in listing A.1.

Thread Hierarchy

As described above, the kernel is executed by N threads in parallel. The number N typically depends upon the input data size. All the threads are part of a one-, two- or three-dimensional block of threads. The blocks are organised into a one-, two- or three-dimensional grid of thread blocks. The thread hierarchy is illustrated in Figure 4.2. Threads in the same block can perform simple synchronization, threads in different blocks cannot synchronize.

```

1  __global__ multVectors(int* a, int* b, int* c){
2      int i = threadIdx.x;
3      c[i] = a[i] * b[i];
4  }
5
6  int main(){
7      ...
8      //execute 1 block of N threads
9      multVectors<<<1, N>>>(a,b,c);
10     ...
11 }

```

Listing 4.1: Sample kernel multiplies the elements of two arrays. The kernel proper is the function on lines 1-4. The kernel is launched on line 9. The triple chevron syntax is used to indicate the number of blocks and the number of threads per block.

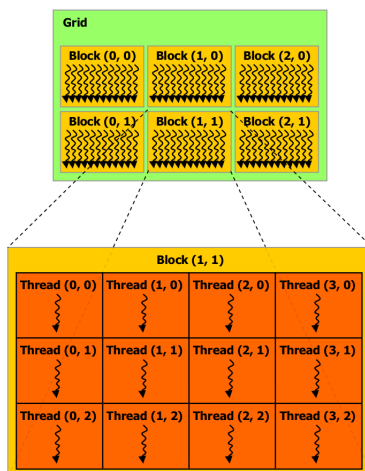


Figure 4.2: Illustration of thread hierarchy. Each grid consists of several blocks, and each blocks consists of several threads. Figure from [1].

Memory Hierarchy

The GPU has access to memory in two different physical locations: fast on chip memory, and slower, off chip (but on the graphics card) memory, known as device memory. In addition, there is the computers main memory, which the GPU cannot access. The CPU has to explicitly copy data from the main memory to the GPU memory before the GPU can start computing.

The two physical memory locations are divided into several logical memory spaces, with different properties. In addition the per thread, on chip, registers, there are:

Local Each threads has private local memory, with the same lifetime as the thread. Local refers to visibility, not location, as it is placed in device memory.

Shared All the threads in a block have access to shared memory, with the same lifetime as the block. Is in fast, on chip memory.

Global Global memory is visible to all the threads in block, and persists across kernel launches. Is in device memory.

Texture Read-only memory optimized for sequential access. The data is in device memory, but is cached.

Constant Read-only memory optimized for simultaneous access by different threads. The data is in device memory, but is cached.

Similar to global memory, texture and constant memory are visible to all threads, and persist across kernel launches.

4.3.2 Hardware Implementation

As described above, GPU cores are organized into units that execute the same instructions in lock-step. In CUDA parlance, these units are known as *streaming multiprocessors* (SM). Each SM consists of several processing cores (known as CUDA cores), some special function units, a register file, shared memory and shared control logic.

At runtime, each SM is assigned a number of blocks, which it executes concurrently. If the number of blocks exceeds the combined capacity of all the SMs, some blocks may be queued. In more detail, the threads of each block is partitioned into groups of 32, known as *warps*, and scheduled for execution.

When executed, each thread in the warp is issued the same instruction. If branches are encountered, each path is executed serially, with those threads taking other paths disabled. In this way, the SIMD nature of the SM are hidden from the programmer.

4.3.3 Performance Considerations

The performance of CUDA kernels is highly sensitive to small changes in the program [26]. Following [1] and [27], we will here discuss two key topics when optimizing for performance on the GPU.

Maximize Utilization

At a high level, maximizing utilization includes dividing work between the GPU and CPU, and overlapping the computation on the CPU with either computation on the GPU or transfer of data between the GPU and CPU.

On a lower level, it is important to maximize the occupancy of the SMs. Rather than using deep pipelines, and caches to hide latency, GPUs simply issue instructions from other threads. Having a high number of threads available and ready for execution is therefore vital. *Occupancy*, defined as the ratio of resident warps to the highest allowable number of resident warps should therefore be maximized. Each SM has a fixed number of registers and amount of shared memory, and it can only host an integer number of blocks at the same time. Hence, the occupancy depends upon the number of registers used by each thread, the amount of shared memory used by each block, and the number of threads in a block. This is best illustrated with an example.

The Tesla C1060 GPU has 16 K registers, and 16 KB of shared memory. If each thread uses 16 registers, there are 512 threads per block, and no shared memory is used, each SM can host $16K/16 = 1024$ threads, that is 2 blocks or $1024/32 = 32$ warps, which is the maximum, resulting in a occupancy of 100%.

If the register usage is increased to 17, each SM can only host $16K/17 = 936$ threads. This is not enough for 2 full blocks, so it can therefore only host 1 thread block of 512 threads, giving an occupancy of 50%.

4.3.4 Maximize Memory Throughput

Maximizing memory throughput involves two key steps. Firstly, low throughput memory accesses should be avoided. Secondly, the most efficient memory access patterns for the given type of memory should be employed.

The first step involves minimizing the data transfer between the host and device, which have low bandwidth. Furthermore, the usage of global and local memory, both of which are physically located in slow, off-chip, and on older devices non-cached, DRAM memory should be minimized, while shared memory, which is on chip and hence faster should be employed to a as large degree as possible. In particular, shared memory can often be used as a user managed cache. Finally, texture memory, while being located in off-chip DRAM memory, is cached, and optimized for certain access patterns. It can therefore be beneficial to use it in certain scenarios.

The second step involves changing the memory access patterns to better fit the type of memory. Global memory accesses are coalesced, so if threads of the same warp access a contiguous area, this results in fewer memory transactions, compared to a situation where the accesses of the threads are spread. Furthermore, the minimum transaction size is 32 bytes. Unless it can be coalesced with other accesses, reading a 4 byte integer will therefore result in reading 32 bytes, effectively reducing throughput by a factor of 8.

Chapter 5

Multivariate Interpolation

Interpolation is a method of finding the value of some unknown function $f(x)$ at a new x given the value of f at several other x -values, x_0, x_1, \dots, x_n [28]. In general, x might be a vector, as in our case, where we need to find the value of the diffraction pattern $I(\mathbf{Q})$ at locations along rays, given only samples of I at other locations. In this chapter, we examine some methods for multivariate interpolation.

5.1 Trilinear Interpolation

If the samples are on a uniform grid, that is, in the form of voxels, trilinear interpolation [29] can be used. In essence, it performs linear interpolation three times in a row, using the eight surrounding samples of the interpolated point.

Referring to Figure 5.1, given the eight samples taken at x_0, x_1, \dots, x_7 , with values $D(x_0), D(x_1), \dots, D(x_7)$, the unknown value $D(x_{14})$ at x_{14} is interpolated by first calculating:

$$\begin{aligned} D(x_8) &= l(x_0, x_1, x_8) & D(x_9) &= l(x_2, x_3, x_9) \\ D(x_{10}) &= l(x_4, x_5, x_{10}) & D(x_{11}) &= l(x_6, x_7, x_{11}) \end{aligned}$$

These values are then used to calculate:

$$D(x_{12}) = l(x_8, x_9, x_{12}) \quad D(x_{13}) = l(x_{10}, x_{11}, x_{13})$$

And finally:

$$w_i(x) = \frac{1}{d(x_i, x)^u}$$

with $d(x_i, x)$ being the euclidean distance between x and x_i .

The value of the exponent u can be varied to change the behaviour of the algorithm. High values of u assigns higher weights to close samples and lower weights to distant samples compared to low values of u .

One variation of the basic IDW method described so far is to only use the samples close to a interpolated point, rather than all the samples. This might be done either because the distant points are considered to not carry any information about the value at the interpolated point, as discussed further in Section 5.4, or simply to reduce the required computational effort.

Close samples can be defined as either all those within some radius r of the interpolated point, as the n nearest neighbours of the interpolated point, or some combination.

5.3 Kriging

Kriging¹ [31] is a family of interpolation techniques where the value at the interpolated point is a weighted average of the samples, and the weights are chosen to minimize the error of the prediction.

In particular, it is assumed that the N samples, taken at x_1, x_2, \dots, x_N with values $z(x_1), z(x_2), \dots, z(x_N)$ are the realizations of a stochastic process $Z(x)$. To find the unknown value at the point x_0 , kriging computes a predictor $\hat{Z}(x_0)$ of $Z(x_0)$ given by:

$$\hat{Z}(x_0) = \sum_{i=1}^N \lambda_i(x_0) z(x_i)$$

where the weights $\lambda_i(x_0)$ (henceforth denoted simply λ_i) are chosen such that the variance of the prediction error:

$$\text{Var}(Z(x_0) - \hat{Z}(x_0))$$

is minimized. The details of how this is done depends upon the assumptions made about Z , the stochastic process. In the following we present *ordinary*

¹Named after Daniel Gerhardus Krige, who pioneered the technique.

kriging, which assumes that the mean μ , of Z is unknown, but constant, and that the covariance function of Z is known.

Ordinary kriging adds the additional constraint of unbiasedness:

$$\sum_{i=1}^N \lambda_i = 0$$

The weights minimizing the variance can then be found using the method of Lagrange multipliers, resulting in the equation system:

$$\begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_N \\ \mu \end{bmatrix} = \begin{bmatrix} \gamma(x_1, x_1) & \cdots & \gamma(x_1, x_N) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \gamma(x_N, x_1) & \cdots & \gamma(x_N, x_N) & 1 \\ 1 & \cdots & 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \gamma(x_1, x_0) \\ \vdots \\ \gamma(x_N, x_0) \\ 1 \end{bmatrix}$$

Here, γ is the *semivariogram* of Z , (2γ is the *variogram*) which, under the assumptions of ordinary kriging, is related to the covariance as follows:

$$2\gamma(x_i, x_j) = \text{Cov}(x_i, x_i) + \text{Cov}(x_j, x_j) - 2\text{Cov}(x_i, x_j)$$

As is evident from the relationship with the covariance function, the variogram is used to describe how related the value at two locations are. In practice it is rarely known, and must be estimated from the samples, or simply guessed.

5.4 Anisotropic Interpolation

Anisotropy refers to the property of being directionally dependent. In anisotropic interpolation [32, 33] the influence of a sample on the interpolated point depends not only on the distance to and value of the sample, but also the direction.

Both kriging and IDW, as well as some other interpolation techniques, are essentially weighted averages of samples, having the general form:

$$f(x) = \sum_{i=1}^N w_i D_i$$

Where w_i is the weight assigned to the sample at x_i with value D_i .

The weight assigned to a sample reflects the assumed similarity of the sample and the value at the interpolated point. A large weight indicates that it is

believed that the value at the interpolated point is similar to the sample, while a small weight indicates the opposite.

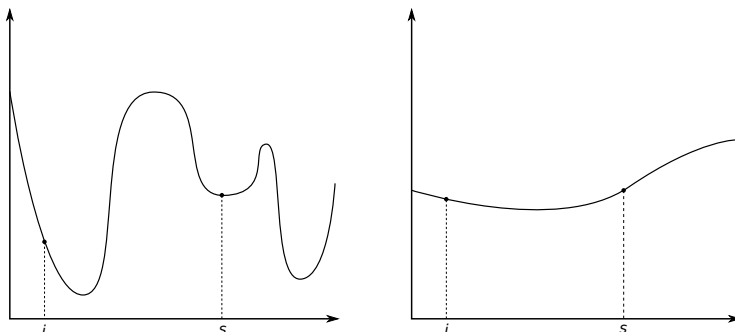


Figure 5.2: In the rapidly changing function on the left, the sample at s has little information about the value at the interpolated point i . In the slowly changing function on the right, the sample carries more information.

In general, it is natural to assume that similarity depends upon distance, so that samples close to the interpolation point should be assigned high weights. What should be regarded as close depends upon how rapidly the underlying function varies. If the underlying function changes rapidly, the values of distant samples are probably weakly related to the value at the interpolated point. Hence, close samples should be assigned higher weights relative to more distant ones. If, on the other hand, the underlying function changes slowly, more of the weight should be shifted to distant samples. This concept is illustrated in Figure 5.2.

In IDW, information about how the underlying function changes can be incorporated by adjusting the power parameter u or the radius r . Increasing u or reducing r assigns more weight to close samples.

In some cases, it might be known that the underlying function changes more rapidly in one direction than in another. An example of such a function is shown in Figure 5.4. If that is the case, the weight assigned to a sample should not depend only on the distance, but also the direction. A distant sample in the direction of rapid change should be assigned less weight compared to an equally distant sample in the direction of slow change.

In both IDW and kriging, this can easily be achieved by using an anisotropic distance metric, rather than the standard euclidean distance metric. The anisotropic distance between two points \mathbf{x} and \mathbf{y} can be defined as:

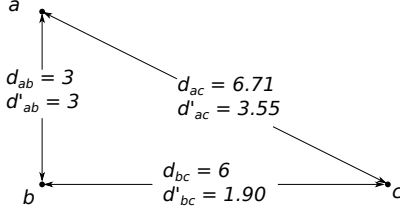


Figure 5.3: The euclidean, denoted by d_{xy} , and anisotropic, denoted by d'_{xy} , distance between three points, with $A = [0.1, 0; 0, 1]$. This A compacts the x-axis.

$$d_a(\mathbf{x}, \mathbf{y}) = \sqrt{\mathbf{d}'A\mathbf{d}}$$

Where $\mathbf{d} = \mathbf{x} - \mathbf{y}$ is the vector defined by the points, and A is a $N * N$ matrix specifying the anisotropy, with N being the dimensionality of the points. If A is an identity matrix, this is the same as euclidean distance. Using other matrices can "compact" or "expand" different directions.

This is best explained with a concrete example. Figure 5.3 shows three points, $a = (0, 3)$, $b = (0, 0)$, and $c = (6, 0)$. The euclidean distances between the points are $d_{ab} = 3$, $d_{bc} = 6$ and $d_{ac} = 6.71$. Using the anisotropic matrix

$$A = \begin{bmatrix} 0.1 & 0 \\ 0 & 1 \end{bmatrix}$$

The anisotropic distances becomes:

$$d'_{ab} = \sqrt{0.1 \cdot 0^2 + 3^2} = 3$$

$$d'_{bc} = \sqrt{0.1 \cdot 6^2 + 0^2} = 1.90$$

$$d'_{ac} = \sqrt{0.1 \cdot 6^2 + 3^2} = 3.55$$

Figure 5.4 illustrates the utility of anisotropic interpolation in a situation where the underlying function changes more rapidly in one direction than in another.

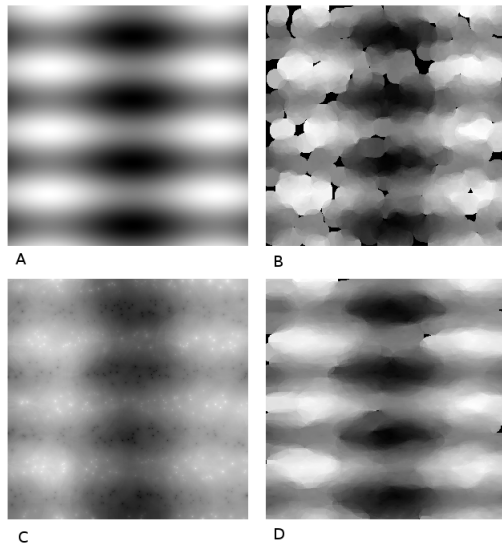


Figure 5.4: Illustration of anisotropic functions and interpolation. A is the original function, and B-D are reconstructions from 500 uniformly distributed samples using IDW with $u = 2$. In B and D, $r = 1$, in C, $r = 3$. In D, anisotropic distance is used, with $A = [1, 0; 0, 0.3]$. The results in D are clearly superior.

Chapter 6

Related Work

In this chapter, we describe some previous and related work in the fields of GPGPU computation and direct volume visualization, with a particular emphasis on visualization of unstructured data, and ray casting.

We have described previous work on generic direct volume visualization in Chapter 3.1.

Many computationally intensive tasks have been ported to the GPU, one recent review is [19]. At the Norwegian University of Science and Technology's High Performance Computing lab, it has, for instance, been used to perform 3D ultrasound reconstruction and visualization [34], simulating fluid flows in porous rocks [35] and simulating and visualizing snowfall [36] and avalanches [37, 38].

Recently, direct volume visualization has also been performed on the GPU. Early approaches, like those presented by Cullip and Neumann [15] and further developed by Cabral et al. [16] were essentially doing texture mapping, exploiting the GPU's ability to do this in hardware. As the GPU grew more flexible, it was also used for full volume ray casting. Stegmairer et al. present a flexible framework for GPU-based volume rendering in [11], based on ray casting. Crassin et al. have visualized billions of voxels on the GPU [39], using oc-trees combined with a mipmap like approach to avoid having to store the entire dataset in the limited GPU memory. These approaches used shader programs, more recently, Marsalek et al. [40] have shown that the higher level CUDA framework can be used without compromising performance.

The works presented so far has been based upon structured volume data in the form of voxels. Much work has also been done on unstructured data.

One main approach is performing resampling, that is, by superimposing a uniform grid on the spatial domain of the data, and interpolate values at the vertices, using the original data, to form voxels. The voxels are then rendered using standard techniques. This is done by Wihelms et al. [41] in the case where the original data were a curvilinear grid, and by Navratil et al. [42] to visualize the results of cosmological particle based datasets. Fraedrich et al. [43] take a slightly different approach, and resample the results of SPH simulations on a 3D grid fixed to the view volume rather than simulation domain, and uses GPU texture mapping to for visualization.

Much work has also been done in the case where the data is in the form of unstructured meshes, curvilinear grids or irregular cells, that is where the data is essentially deformed voxels, or cells in the shape of tetrahedra etc. Garrity [44] uses such data where the connectivity of the cells are known. Once the first cell a ray intersects has been found, subsequent cells can be found by checking which of the faces of the current cell the ray intersects, and moving to the corresponding cell. Giertsen [45] does not assume that the connectivity is known, or even that all the cells are connected. The intersections between a plane perpendicular to each scanline of the final image and the cells are used.

Finally there is the case of completely unstructured volume data, often referred to as point clouds. Here splatting, originally proposed by Westover [12] is popular, and is for instance used by Hopf and Ertl [46] to visualize the results of n-body simulations, with millions of particles. Ray casting based techniques are also used. Chen [47] visualizes point based volume objects using ray casting. At equidistant points along the rays, nearby points are found and used to evaluate a radial basis function. An octree is used to speed up the process of determining the nearby points. The leafs of the octree contain all the points which may influence rays passing through that cell. Hence, some points may be stored in multiple leaf nodes. A similar approach is taken by Kahler et al. [48] to render the results of SPH simulations on the GPU.

Chapter 7

Implementation

In this chapter, we describe how we implemented our method for visualizing 3D X-ray diffraction patterns.

7.1 Overview

As described in Chapter 2, the output of X-ray diffraction experiments is essentially a unstructured volume dataset, consisting of samples of the intensity of the X-ray diffraction pattern. We have implemented a method, based on volume ray casting, for visualizing such datasets.

Figure 7.1 illustrate how our method works to generate a single image. For each pixel in the output image, a ray is cast from the eye/camera, through that pixel, and into the volume. The intensity of of the volume is found at points along each ray. This is done by interpolation, using samples of the X-ray diffraction pattern close to each point. The intensities are then mapped to color and opacity values, and the colors and opacities along each ray are composited to find the color of the corresponding pixel.

Prior to this taking place, we naturally need to read the data, and also perform some preprocessing and filtering. We then build an acceleration data structure, in order to facilitate the efficient retrieval of those samples close to a point.

To summarize, the main steps of our algorithm are:

- Read, preprocess and filter data.

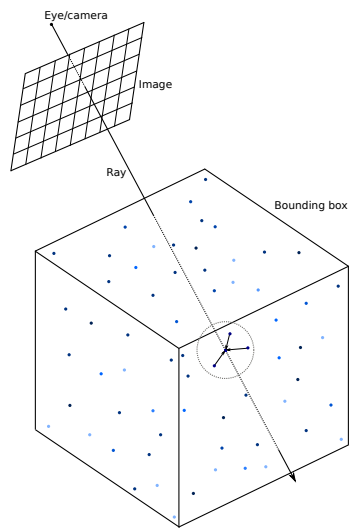


Figure 7.1: Overview of implementation. The input is a set of samples of a X-ray diffraction pattern, here shown as blue dots. Rays are cast from the eye/camera, through each pixel, and into the volume. The value of the diffraction pattern is estimated at points along the ray, by interpolating among neighbouring samples. Here, we show one such point, on one such ray.

- Build acceleration data structure.
- Create rays.
- Cast rays:
 - Find neighbouring samples of points along each ray.
 - Interpolate to estimate intensity at point.
 - Map intensities to color and opacity.
 - Combine all color and opacity values along each ray.

In the following sections, we will elaborate upon each of these steps.

7.1.1 Design Choices

As mentioned in the introduction, our main goal is high quality, artifact free, visualization, at the expense of performance. This goal motivated the following design choices.

Firstly, we decided to produce the visualization using the diffraction pattern samples directly. These samples have no spatial structure, making them difficult to manage. One obvious possibility would be to resample the diffraction pattern on a uniform grid to produce voxels. While the voxel approach almost certainly would be faster, we feared that it would lead to inferior results in terms of quality.

Secondly, we decided to use volume ray casting. It is not the fastest direct volume visualization algorithm, especially for unstructured data, where splatting is more popular. However, it is highly flexible, well documented and understood, and capable of superior image quality [49, 7, 43].

Thirdly, while there is some structure to the data, as described in Chapter 2, we have chosen to ignore this, and treat the samples as if they were taken at random locations. We made this decision because we saw no obvious way to exploit the structure of the data. At the same time, this approach made it easier to preprocess and filter the data, as our method made no assumptions about it.

Lastly, we have strived to make our implementation flexible, customizable and configurable. The settings that produce the highest quality images depends upon the dataset, what part of the dataset is viewed, and the specific aspects of the dataset the user wants to focus on. Hence, no single set of settings is universally best, and the user should be given as much control as possible. Importantly, this also includes the ability to alter or augment our implementation,

by, for instance, adding new interpolation algorithms. We have therefore taken great care to make our code flexible and readable.

7.2 Preprocessing and Filtering

Prior to visualization the data must be converted to an appropriate form. To reduce noise, and remove data that does not contribute to the final image, filtering is performed.

7.2.1 Input Data and Preprocessing

The raw output of X-ray diffraction experiments, is, as described in Chapter 2, a set of frames, along with a set of angles specifying the location of the sensor relative to the beam and material sample for each frame. The conversion of this data into a set of diffraction pattern samples on the form (x, y, z, i) where x, y, z specifies the scattering vector and i the intensity of one pixel is described in [4], and is not performed by our tool.

While x, y, z technically describes the scattering vector, they can be interpreted as specifying a point in 3D space. Under this interpretation, the input becomes a unstructured volume dataset.

It is important to point out that, while we in several figures in this chapter illustrate this dataset as being uniformly distributed in a axis-aligned box, this is generally not the case. Firstly, the convex hull¹ of the samples typically only occupy a small subsection of their bounding box. Secondly, the samples are not uniformly distributed within their convex hull, but rather organized in layers, as described in Chapter 2. The spatial distribution of the samples is illustrated in Figure 7.2.

The raw data is in double precision floating point format. Our implementation can easily be recompiled to use both single and double precision floating point numbers. Double precision numbers provide enhanced accuracy and might be necessary for some datasets and settings. However decreased memory consumption and improved performance (in particular on the GPU) can be achieved by reducing the precision. Since the output is images, small differences caused by reduced precision might not be detectable by the human eye. If needed, the raw data can therefore be converted to single precision.

¹The smallest polyhedron containing all the samples.

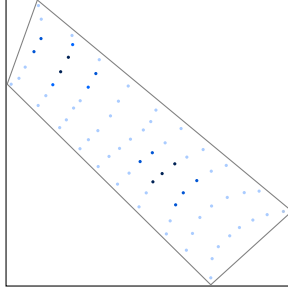


Figure 7.2: Distribution of samples, illustrated in 2D. The samples of the diffraction pattern are shown as blue dots, darker dots indicate higher intensity. The convex hull is shown in gray.

7.2.2 Filtering

To reduce noise, and remove samples that does not contribute to the final image, filtering is performed.

Threshold Filtering

The input datasets typically consists of large regions with low intensity, and smaller regions with high intensity. This is shown schematically in Figure 7.2, and in practice in Figure 9.2 and 9.1. For visualization purposes, the regions with low intensity should be transparent. By implicitly treating empty regions as transparent, samples with low intensity can simply be removed. This will reduce the size of the dataset, without affecting the output image. In our implementation, all samples with an intensity below a user specified threshold are ignored when the data is loaded.

Median Filtering

For various reasons, the data might contain noise, often in the form of a single sample with high intensity in a region with low intensity, or a single sample with low intensity in a region with high intensity. We use a variation of median filtering [50] to remove such noise.

For each sample, we find the median of it and its neighbouring samples intensity, and discard the sample if its intensity is significantly different from the median. In more detail, the neighbours of a sample are defined as those

closer to the sample than a user defined radius. How these points are found is described in Section 7.4. By assuming that the intensities follow a Poisson distribution, their standard deviation can be approximated as the square root of the median [51]. We discard a sample if its intensity differs from the median by more than a user specified number of standard deviations.

7.3 Volume Ray Casting

To visualize the diffraction pattern, we use volume ray casting.

7.3.1 Ray Creation

We use the standard pinhole camera model [52]. A detailed schematic overview of the camera and image plane is shown in Figure 7.3. The user specifies the position E , the direction \mathbf{f} , and the field-of-view θ of the camera. The user also specifies the resolution of the image. In order to uniquely define the image plane, we also need a vector \mathbf{u} defining up, and the distance between the camera and image plane. We use the vector orthogonal to \mathbf{f} which forms the minimum angle with the z -axis as up vector, and a arbitrary fixed distance to the image plane.

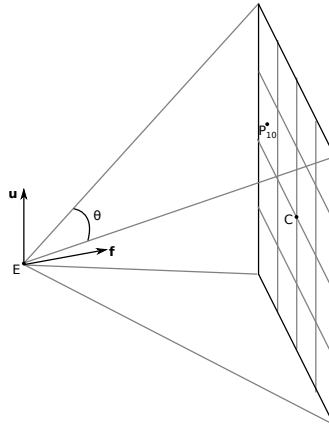


Figure 7.3: Schematic depiction of camera. Given the position E , direction \mathbf{f} and field-of-view θ of the camera, the up vector \mathbf{u} , center of image C and resolution, the location of the centers of pixels, such as P_{10} can be found.

With this information, the position of the center of each pixel in the image plane can be found. A ray is created for each pixel, with the direction of the ray being defined by the center of its pixel and the camera position.

7.3.2 Casting a Single Ray

We refer to the processing of a single ray to find the color value of its pixel as casting the ray. For each ray, starting at the camera, we move along the ray, and estimate the intensity of the volume at equidistant points. The points are separated by a user specified distance d , known as the step size. To estimate the intensity at a point, we first find all the samples whose distance to the point is less than a user specified threshold r , known as the search radius. To find these samples, we use range search, as described in detail in Section 7.4.

When the search has found the neighbouring samples, we estimate the intensity at the point, using interpolation. We have implemented both kriging and IDW interpolation. As described in Chapter 5, ordinary kriging, which is the variant we have used, requires the variogram, (or technically, the semivariogram) to be known. We have used a exponential semivariogram:

$$\gamma(x_i, x_j) = 1 - e^{-\frac{d(x_i, x_j)}{R}}$$

With d being the distance between the samples, and R is a user specified parameter. We support arbitrary values of the power parameter u for IDW interpolation, but have optimized for the common cases of $u = 1$ and $u = 2$. Finally, we support arbitrary, user specified, matrices for anisotropic distance calculation. Anisotropic distance is only used for the weights for IDW, and in the variogram for kriging, to decide which points to use for the interpolation, regular euclidean distance is used. Empty regions are treated as transparent, that is, if no samples are found, we set the intensity to 0.

Once the estimated intensity has been found, we map it to a color and opacity value. The color and opacity values of all the points of a ray are then used to evaluate the volume rendering integral, as described in Chapter 3.1, to find the color of the rays pixel.

In practice, we evaluate the integral incrementally, in a front to back manner, as described in [17]. Starting at the camera, for each point along the ray, we update the color using the formulae:

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i$$

Where C_i and A_i are the color and opacity of the i 'th point, C'_{i-1} and A'_{i-1} the accumulated color and opacity so far, and C'_i and A'_i are the new accumulated color and opacity. The initial value of both the color and opacity is 0, that is, $C'_0 = 0$ and $A'_0 = 0$. The final color of the pixel is $C'_n \cdot A'_n$, where n is the number of points along the ray.

7.3.3 Optimizations

To improve performance, we employ three common optimizations [53], which all aim to reduce the number of points at which the intensity must be estimated.

Firstly, rather than estimating the intensity at points along the entire ray, we use only the part of the ray within the bounding box of the samples. The intersection can be found using standard techniques [54].

Secondly, we use early ray termination. Rather than estimating the intensity at all n points along the ray, we stop when A'_i becomes sufficiently close to 1. A value of A'_i equal to 1 indicates that the part of the volume between the camera and the i 'th point is completely opaque. The color values at further points will therefore not contribute to the color of the pixel of the ray, so there is no point in continuing.

Thirdly, the filtering described in Section 7.2.2 leaves large regions of the volume empty. As described above, these regions are treated as transparent. Searching in these regions is therefore a waste of computational resources, ideally, empty space should be skipped. We have implemented a technique that reduces the number of searches in empty regions.

One way to do this is by taking advantage of the acceleration data structure, once a empty leaf node is encountered, one simply jumps to the end of it. This works well if voxels are used, but might lead to incorrect results for our method. Even if the leaf node of the search point is empty, there might still be samples within the search radius. Figure 7.4 illustrates this concept.

While it is possible to work around this problem, we have instead implemented a novel empty space skipping system. If the result of a search is empty, we increase the step size by a factor n . Each subsequent empty search result continues to increase the step size, until a threshold is reached. When a non-empty search is encountered, we move back to the previous point, reset the step size to its default value, and proceed. Our algorithm is illustrated in Figure 7.5.

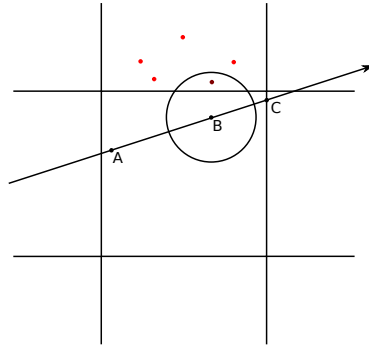


Figure 7.4: Illustration of why skipping a empty node might cause incorrect results. When A is reached and the empty node detected, we could skip it by jumping to C. This would lead to incorrect results, as B would be treated as transparent. Even though the node in the center is empty, its neighbour contains samples (indicated by red dots), some of which influence B

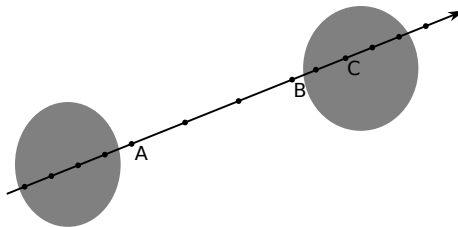


Figure 7.5: Illustration of our empty space skipping algorithm. The gray areas indicate regions with a high density of samples, while white areas are empty. The empty result at A will cause the step size to be increased. Once we reach a nonempty search at C, we retract to B, reset the step size, and proceed. The end result is that we search at all the points shown along the ray.

The ideal values for the threshold and factor depends, in the same way as the step size itself, upon the dataset, and are specified by the user. Great care must be taken to avoid setting the threshold to high, as the resulting coarseness of the sampling might miss small structures in the dataset.

7.3.4 Colors

Mapping the intensity of a point to a color and opacity value is done using a transfer function.

The intensity range of X-ray diffraction patterns is very large, typically several orders of magnitude. It is therefore common practice to use the logarithm of the intensity for visualization purposes. After taking the logarithm, we normalize by dividing by the logarithm of the maximum intensity of all the samples². That is:

$$I_{norm} = \frac{\log(I)}{\log(I_{max})}$$

This results in a number between 0 and 1.

Figure 7.6 shows a sample transfer function, mapping normalized intensities to colors.

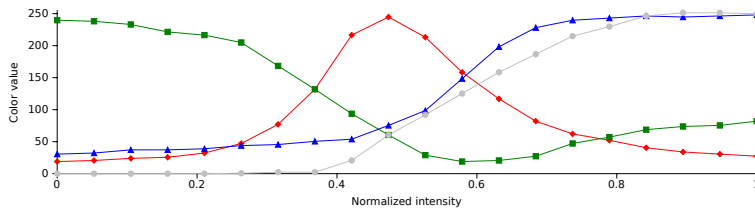


Figure 7.6: Example of a transfer function. The gray line is for alpha/opacity. A normalized intensity of 0.42 would result in the RGBA color (216,94,54,21)

We supports arbitrary, user defined, transfer functions. In more detail, the user is allowed to specify the red, green, blue and alpha values for an arbitrary number of normalized intensity values. The color of any intensity value can then be found by linear interpolation between the two nearest specified points.

For efficiency, we precompute a table with a high number of entries for the transfer function.

²Note that this normalization makes the base of the logarithm arbitrary.

7.4 Range Search

During the ray casting, we need to determine the intensity at points along each ray. This is done by interpolating among the samples close to each point. During filtering we remove a sample if it is significantly different from the surrounding samples.

In both cases, we have a fixed point, and wish to find all the samples close to that point. To make the intuitive notion of "close" more precise, we introduce the search radius, r , and define close samples as those having a distance to the fixed point less than r . We can visualize this problem as attempting to find those samples lying inside a sphere of radius r centered at the fixed point. This is essentially the range search problem from computational geometry [55].

In the following, we will refer to this sphere as the *search sphere*, or, in 2D examples, as the *search circle*.

The samples inside the search sphere can naively be found in $O(N)$ time, with N being the total number of samples, by computing the distance between the given point and each of the samples, and retaining those samples where the distance is less than r .

The performance can, however, be significantly improved by building an acceleration data structure. Numerous data structures for range search has been proposed in the literature [56, 57]. We have chosen the octree data structure, due to its conceptual simplicity and ease of implementation, intuitive and predictable structure and good performance. To fulfill our goal of creating a flexible implementation (cf. Section 7.1.1), we have structured our code so that this structure can easily be replaced.

7.4.1 Octree

The octree is a well known spatial subdivision data structure for range search [58]. Each node in the tree corresponds to a cube, and the children of a node are the eight octants of the cube. Figure 7.7 shows a sample octree.

The root node of our octree is the bounding box of all the samples. Leaf nodes contain those samples that lie in their corresponding cube, and may be empty if no such samples exists.

Construction

Pseudocode for construction of our octree is shown if algorithm 1. Starting with an empty root node, samples are inserted one at a time. The leaf node

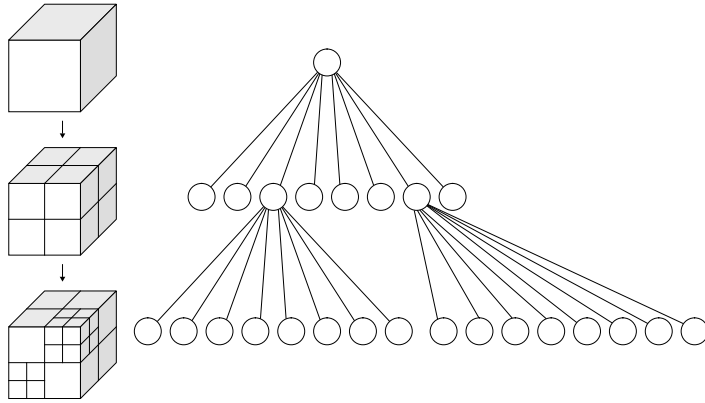


Figure 7.7: Octree on the right, with corresponding cubes on the left. Figure from [59]

of the sample is found by descending down the tree. If the leaf node contains more than 8 samples, we *split* it. That is, it is converted into an internal node, and the 8 samples are inserted into its new child nodes. Leaf nodes may contain more than 8 children if splitting them would cause the maximum height of the tree to increase beyond a certain threshold. This threshold is determined by the size of the search radius relative to the size of the bounding box, as described in the next section.

Algorithm 1 Construction of octree

```
function INSERTPOINT(Node n, Sample s, int d)
  if n.isLeaf then
    if n.numSamples < 8 or d > MAX then
      n.ADD(s)
    else
      n.SPLIT()      ▷ Create child nodes, and add old samples to them
      INSERTPOINT(n,s,d+1)
    end if
  else
    m = n.FINDCHILD(s)      ▷ Find child node of n containing s
    INSERTPOINT(m,s,d+1)
  end if
end function
```

Search

Since the computation of box-box intersections is simpler than box-sphere intersections, we use the bounding box of the search sphere when searching in the octree. The search will return all the samples in leaf nodes intersecting with this bounding box. Each of these samples must subsequently be investigated to see if it is inside the search sphere.

Pseudocode for range search with a box in an octree is shown in algorithm 2, and illustrated in Figure 7.8. At each node, we find the child nodes intersecting the search box, and search those nodes recursively. The search returns the samples of the leaf nodes of the tree where the intersection between the node and the box is nonempty. The combined volume of these leaf nodes is usually larger than the volume of the search box. Hence, the search may also return samples not lying within the search box, and by extension not in the search sphere. Each of the returned samples must therefore be checked to see if this is the case.

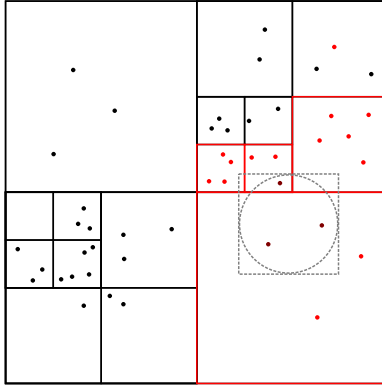


Figure 7.8: Searching in octree, illustrated in 2D. Even though we are only interested in the samples in the circle (marked with dark red), all the samples in those leaf nodes intersecting with the bounding box of the search circle will be returned. That is, all the red samples will be returned.

Algorithm 2 Range search in octree

```

function RANGESEARCH(Node n, Box b)
  if n.isLeaf and INTERSECTS(n,b) then
    return n.samples
  end if
  samples = [] ▷ Empty array
  for all c in n.children do
    if INTERSECTS(c,b) then
      samples.ADD(RANGESEARCH(c,b))
    end if
  end for
  return samples
end function

```

If the octree search returns many samples that are not in the search sphere, this last step can be costly. It is therefore desirable that the combined volume of the leaf nodes returned matches the search sphere as closely as possible. This can be achieved by reducing the size of leaf nodes. There is, however a trade-off, as reducing the size of the leaf nodes will increase the depth of the tree, making it

more costly to find the leaf nodes in the first place, and also increasing memory overhead.

In our implementation, we set the maximum tree height equal to:

$$h_{max} = \lfloor \log_2\left(\frac{R}{r}\right) \rfloor + 2$$

Where r is the search radius, and R is the smallest dimension of the bounding box of the samples. This means that the size of the smallest leaf nodes is between $\frac{r}{4} \times \frac{r}{4} \times \frac{r}{4}$ and $\frac{r}{2} \times \frac{r}{2} \times \frac{r}{2}$.

7.5 Parallelization

Ray casting and similar algorithms are well known to be embarrassingly parallel problems [22]. The processing of a single ray does not depend upon any of the other rays, so all the rays can be processed in parallel.

This can be done naively by spawning one thread for each processor core, and assigning an equal number of rays/pixels for each thread to process. However, the amount of work required varies significantly between rays. If a ray immediately enters a opaque region, it will be terminated quickly, while other rays may pass through the entire volume. Some rays do not intersect the volume at all, and requires almost no processing.

By dividing the rays/pixels evenly between the threads, some threads may finish significantly earlier than others, leading to wasted resources. To achieve better load balancing, we use the thread pool implementation developed for our earlier work, described in [21].

All the rays are placed in a work queue. Each thread in the thread pool will remove one or a few rays from the queue, process them, and then remove more threads until the queue is empty. This ensures that no threads will be idle while others are working for significant amounts of time.

7.6 Interactivity

In more practical terms, we have implemented our method as a C library with a corresponding API. Different client applications can then use this library to generate images, and subsequently display them, or save them to disk.

We have created one such client application⁴. It is based on OpenGL, and

³Assuming that the original bounding box is cubic.

⁴In addition to several simple ones, for testing and profiling purposes.

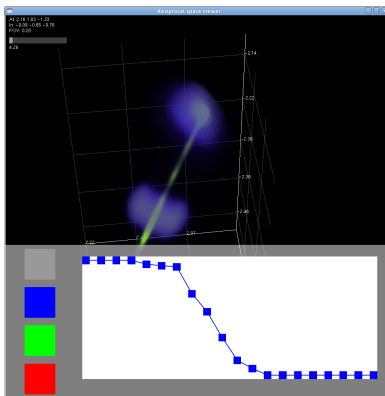


Figure 7.9: Screenshot of our client application, showing the built in transfer function editor.

allows users to interactively move the camera around to view the volume from different angles. It also allows the transfer function to be edited, and settings to be changed. A screenshot of the application is shown in Figure 7.9.

7.6.1 Incremental Update

Our primary concern has been to generate images of the highest possible quality. High resolution images can therefore not be generated at real-time or even interactive rates (see Chapter 9 for detailed timing results).

While it might be acceptable to let the rendering of the final, high quality image be an offline task, the ability to generate low quality previews at interactive rates might still be a desiderata. The ideal camera position, color transfer function and other rendering parameters is usually not known a priori. Finding these will often require significant experimentation, and being able to quickly see the effects of changes will reduce the time needed for this process.

We have implemented a system which achieves this effect. When the user moves the camera to a new position, or changes some setting, several images with increasing resolution are produced, and displayed as they become ready. Low resolution images require less rays, and hence less processing time. Based on the low resolution images, the user can decide if she wishes to move the camera to a new position, or wait for the high resolution images to become available.

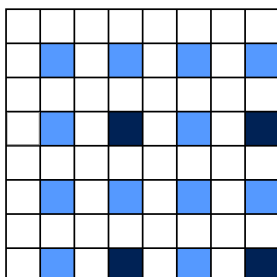


Figure 7.10: Incremental update by gradual resolution increase. By first processing the dark blue pixels/rays, then the light blue pixels/rays and finally the white pixels/rays, the resolution of the image is incrementally increased from 2×2 to 8×8 . Producing the first image is 16 times faster than the last.

We implement this system by simply changing the order in which the rays/pixels are processed. First, the rays necessary for the lowest resolution image are processed. Then, the additional rays necessary for the image with the second lowest resolution are processed, and so on. This is illustrated in Figure 7.10. This way information is made available as soon as its ready, without increasing the time needed to produce the final full resolution image.

Chapter 8

GPU Implementation

As mentioned in Chapter 7, ray casting is a problem that is easily parallelizable. There is a high number of rays, each of which can be processed independently, in parallel. Furthermore, processing each ray requires a lot of numerical computation.

As such, it is a problem ideally suited for the GPU, a platform designed precisely for highly parallel, compute intensive tasks. The fact that the output of ray casting is an image, makes the GPU an even better match, as the task of moving the finalized image to the GPU for display is made superfluous.

We have created an implementation of our visualization method where the ray casting is performed on the GPU. In this chapter, we describe how this was done. It should be noted that porting our method to the GPU was not our primary focus, as mentioned in Chapter 1. At the same time, tuning an application for the GPU is known to be a labour intensive task [26]. Our GPU version is therefore not as functional, or as optimized as it could be.

8.1 Overview

We have used NVIDIA's CUDA framework, introduced in Chapter 4, to port our implementation to run on the GPU.

We have created a kernel that process a single ray. N such kernels are then run on the GPU in parallel, with one thread for each ray/pixel of the output image. The kernel uses the same algorithm as described in Chapter 7, but it is, at the moment, less flexible. We currently only support isotropic IDW

interpolation, and some settings are hard coded.

CUDA allows the usage of a subset of the C programming language. Porting our application was therefore fairly straightforward. Some changes and modifications were, however, needed. In the following sections we will describe them.

8.2 Removing Recursion

While newer GPUs, such as the Tesla C2070 supports recursion, older models, such as the Tesla C1060, does, however, not [1]. We wanted to support as many devices as possible, to allow end users with older hardware to take advantage of our GPU version. The recursive octree search algorithm described in Chapter 7 was therefore changed.

We did this by replacing recursion with iteration in combination with a manually managed stack. Finding all the samples within a search box is done by initially pushing the root node of the tree onto the stack. At each step of the iteration, one node is popped. If it is a leaf node, its points are added to those returned. Otherwise, those child nodes of the popped node overlapping the search box are pushed onto the stack. The loop runs until the stack is empty. A pseudocode version of this algorithm is shown in 3.

Algorithm 3 Range search in octree using iteration

```
samples = [] ▷ Empty array
stack.PUSH(root)
while stack is not empty do
  n = stack.POP()
  if n.isLeaf then
    samples.ADD(n.samples)
  else
    ch = GETOVERLAPPINGCHILDREN(n, searchBox)
    stack.PUSH(ch)
  end if
end while
```

The depth-first nature of the algorithm ensures that the stack size is bounded by $O(D)$ where D is the maximum depth of the tree. The reason is that, at any given time, the stack only contains the nodes on the path from the root to the current node.

To avoid redundant calls to `getOverlappingChildren()`, the stack will also contain the children of the ancestors of the current node that will be visited later. Since each node has a maximum of eight children, and the size of a pointer to a node is 4 bytes¹, a more specific upper bound on the stack size is $32D$ bytes.

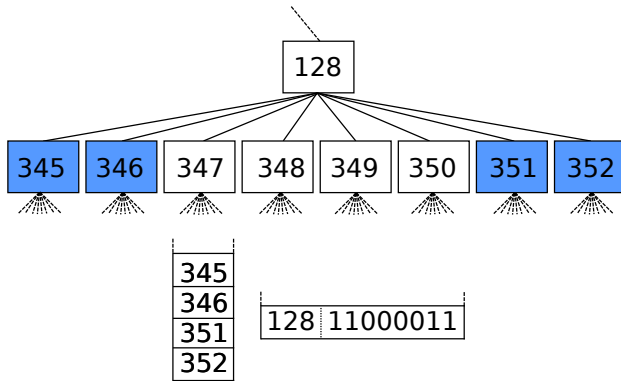


Figure 8.1: Stack optimization. On the top, a portion of an octree is shown. The shaded child nodes are to be pushed onto the stack. In the bottom left is the original stack, and in the bottom right is the optimized stack, after the shaded nodes have been pushed. The stacks grow downwards.

As mentioned, we push all the children of a node that overlaps the search box at the same time. Rather than pushing a pointer to each of these child nodes, we can push a pointer to the parent node, along with a description of which of its child nodes that are pushed. This idea is illustrated in Figure 8.1. The description can be encoded in one byte, where each bit indicates, whether a child is pushed. By reducing the size of the node pointer to 24 bits, we can combine the parent node pointer, and child descriptor in a single 4 byte integer. This way, the upper bound on the stack size is reduced to $4D$ bytes.

It should be noted that only 5 bits, and not 8 are needed to identify the children of a node, as some combinations (i.e. all involving exactly three child nodes) are illegal. Due to the significantly increased complexity, we did not take advantage of this opportunity for optimization. Furthermore, reducing the size of node pointers to 24 bits means that no more than 2^{24} nodes is supported. This has proved to be sufficient in practice.

¹We do not use actual pointers, but integer indices in an array.

8.3 Memory Considerations

Performing computations on the GPU requires the data to be operated upon to be transferred from host memory to GPU device memory as described in Chapter 4. We have adapted our tree data structure so that it occupies one consecutive block of memory, and updated the pointers, so they are all relative to the start of this block. All the samples are also placed in one consecutive block of memory, with the pointers from the leaf nodes of the tree to the samples being relative to the start of the samples block. This way, the tree and samples can easily be moved from main memory to GPU memory, and the pointers will still work, even if the address spaces are different.

8.3.1 Precision

Newer GPUs have support for double precision floating point numbers. However, as many GPU applications (in particular, traditional ones, like computer games) does not require double precision, support for single precision is better. NVIDIA's newest cards, based on the Fermi architecture, support twice as many single precision as double precision FLOPS [60]. Furthermore, the use of double precision numbers will lead to increased device memory and register consumption, both of which are scarce resources. Finally, texture memory does not support texels larger than 32 bits [1]. Storing the samples in texture memory is therefore only feasible if each of the four numbers constituting a sample is stored using single precision.

For these reasons, we have elected to primarily support single precision floating point numbers for the GPU version of our implementation. Double precision is only supported in a limited way, in particular, neither of the optimizations described below can be used in combination with double precision. The mechanisms needed to convert the raw data to single precision format were already in place, as described in Chapter 7.

8.3.2 Optimizations

As described in Chapter 4, memory operations are much more costly on the GPU. It is therefore important to optimize the memory access patterns. We have implemented two simple such optimizations.

Firstly, we realized that when a leaf node is found that overlaps the search box, all its samples will be accessed sequentially, to check if they actually are within the search radius of the search point, and if that is the case, be used in the

interpolation. Texture memory is optimized for such streaming access patterns. We therefore shuffle the order of the samples, so that all samples belonging to the same leaf node are placed together. We then store all the samples as a texture.

Secondly, we realized that the top levels of the tree will be accessed very frequently, all searches will include some of these nodes. By caching them in shared memory, access time is reduced. Therefore, all thread blocks will start by reading in the topmost nodes of the tree, before proceeding to the rendering proper. The number of nodes that can be cached depends upon the available shared memory. It is important to avoid using so much shared memory that fewer thread blocks can be scheduled simultaneously on the same streaming multiprocessor. Currently, we manually adjust the amount of shared memory used for the node cache.

Chapter 9

Results and Discussion

In this chapter we present and discuss our results. We will take a detailed look at the quality of the images our implementation produces, as well as its performance, both on the CPU and GPU.

9.1 Methodology

In this section, we will describe how we conducted our experiments.

9.1.1 Datasets

Two different datasets were used to test our tool. Both are actual X-ray diffraction datasets, and were provided by J. Fløystad¹:

27uc Diffraction pattern from a thin film of PbTiO_3 . Consists of 5 687 136 samples in a $0.238 \text{ \AA}^{-1} \times 0.226 \text{ \AA}^{-1} \times 0.393 \text{ \AA}^{-1}$ bounding box, with intensities ranging from 0 a.u. to 40005.5 a.u.²³. A large fraction of the samples have low intensity, using a filtering threshold of 10 a.u. removes 36.45% of the samples. Total size of raw data is 173.5 MB. A visualization of the dataset is shown in Figure 9.1.

¹Department of Physics, Norwegian University of Science and Technology

²1 $\text{\AA} = 1 \text{ \AA ngstr\o m} = 1 \cdot 10^{-10} \text{ m}$. The scattering vector is measured in inverse distance units, hence the ⁻¹

³a.u. = arbitrary unit.

00571 Diffraction pattern from $\text{Cu}[\text{C}_6\text{H}_4(\text{OH})\text{COO}]_2(\text{H}_2\text{O})_2$. Consists of 121 518 151 samples in a $3.89 \text{ \AA}^{-1} \times 3.92 \text{ \AA}^{-1} \times 4.04 \text{ \AA}^{-1}$ bounding box, with intensities ranging from 0 a.u. to 17389.8 a.u. A very large fraction of the samples have low intensity, a filtering threshold of 10 a.u. removes 99.98% of the samples. The dataset is therefore very sparse after filtering, compared with 27uc. The total size of the raw data is 3.62 GB. A visualization of the dataset is shown in Figure 9.2.

9.1.2 Testing Environment

The tests were performed on two different desktop PCs, machine A, equipped with a 2.8 GHz Intel Core i7 930 processor and NVIDIA Tesla C1060 GPU, and machine B, with a 3.2GHz Intel Core i7 970 processor and NVIDIA Tesla C2070 GPU. Detailed information about the hardware and software of the computers can be found in table 9.1.

	Machine A	Machine B
Hardware		
CPU model	Intel Core i7 930	Intel Core i7 970
CPU cores	4	6
CPU frequency	2.8 GHz	3.2 GHz
Memory	12 GB	24 GB
GPU model	NVIDIA Tesla C1060	NVIDIA Tesla C2070
GPU memory	4 GB	6 GB
GPU cores	240	448
Software		
OS	Ubuntu Linux 11.04	Ubuntu Linux 10.04
Linux kernel	2.6.38	2.6.32
GCC version	4.4.5	4.4.3
NVCC release	4.0	4.1

Table 9.1: Hardware and software of the two computers used to run our experiments. The number of CPU cores refers to physical cores. Both CPU’s have simultaneous multithreading (Hyper-Threading), doubling the number of logical cores.

Most of the experiments were performed on machine A, we will explicitly mention when machine B was used. In all cases, the compilation flag `-O3` was used, to enable maximum compiler optimization.

9.1.3 Measurements

Timing measurements were taken using the `clock_gettime` function with the `CLOCK_MONOTONIC` clock [61]. Unless otherwise noted, the timing results presented are for one 1024×1024 image. Only the time required for the rendering proper is presented. Time used to load data, build data structures etc. is not included, since several images can be generated once this is done. For the CPU version, 8 threads were used on machine A, and 12 threads on machine B, unless otherwise noted.

9.2 Overview

Figure 9.1 and 9.2 are representative of the results we achieve, the images were generated in 125.1 s and 4.75 s respectively.

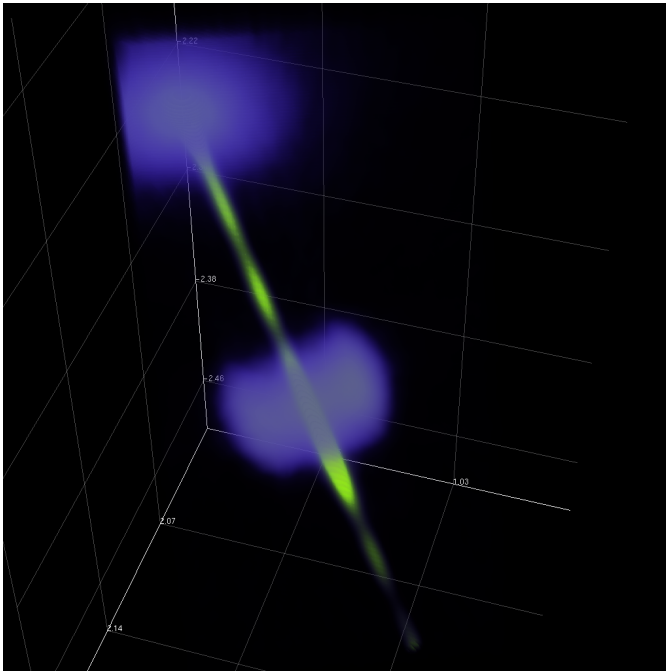


Figure 9.1: Visualization of the 27uc dataset, generated with our method.

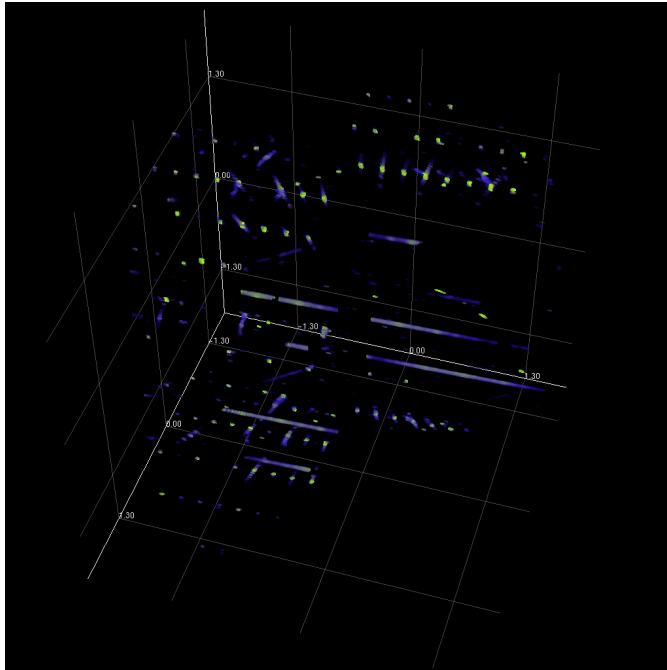


Figure 9.2: Visualization of the 00571 dataset, generated with our method.

9.3 Filtering

The threshold filtering proved to be highly useful, as it allowed a large fraction of the samples to be discarded, essentially without affecting the quality of the final image. The results of filtering is shown in Figure 9.3. The effects of filtering on performance is presented and discussed in Section 9.6.

In Figure 9.3a no filtering has been performed. The low intensity samples have simply been hidden by using a transfer function that makes them transparent (i.e. by assigning them a opacity of 0). These low intensity points are shown in Figure 9.3b. No filtering is performed there either, but the transfer function has been modified to show the low intensity samples. Finally, in Figure 9.3c, the same transfer function as in 9.3b has been used, but the low intensity samples have been removed prior to rendering, so that similar results to 9.3a are obtained. The two transfer functions used here are shown if appendix B, as transfer function f1 and f2.

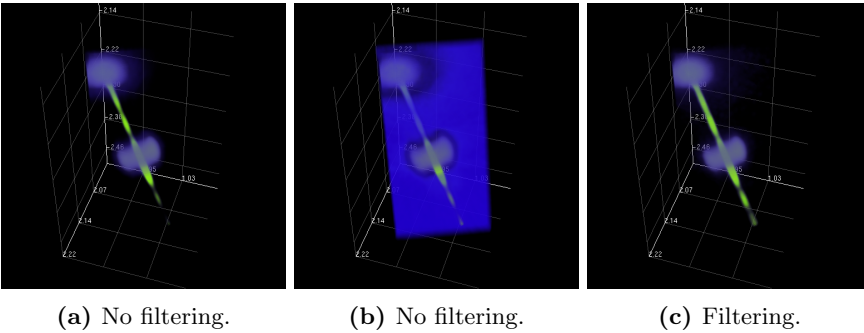


Figure 9.3: Results of filtering. In (a), no filtering is performed, but the transfer function assigns no opacity to low intensity samples. In (b), the transfer function is changed to show the low intensity samples. In (c), the same transfer function is used as in (b), but all low intensity samples are filtered away prior to rendering, yielding similar results as in (a).

In this case, a filtering threshold of 30 was used, which resulted in 88.93% of the samples being discarded. That is, Figure 9.3c is rendered with only 629 773 of the original 5 687 136 samples. Figure 9.2, showing the 00571 dataset, was rendered with a filtering threshold of 10, which resulted in 99.98% of the samples being discarded.

The appropriate threshold value is highly dataset dependent, and it might

also be desirable to use different thresholds for different images of the same dataset to emphasise different structures.

9.3.1 Median Filtering

The utility of median filtering turned out to be lower than expected. In particular, it turned out to be difficult to avoid false positives, that is, the filtering often removed samples that were not noise. We did therefore disable median filtering for our tests. It might, however, be more useful for other datasets.

9.4 Interpolation

In this section, we will present and discuss the qualitative differences between the different interpolation techniques, and anisotropic interpolation.

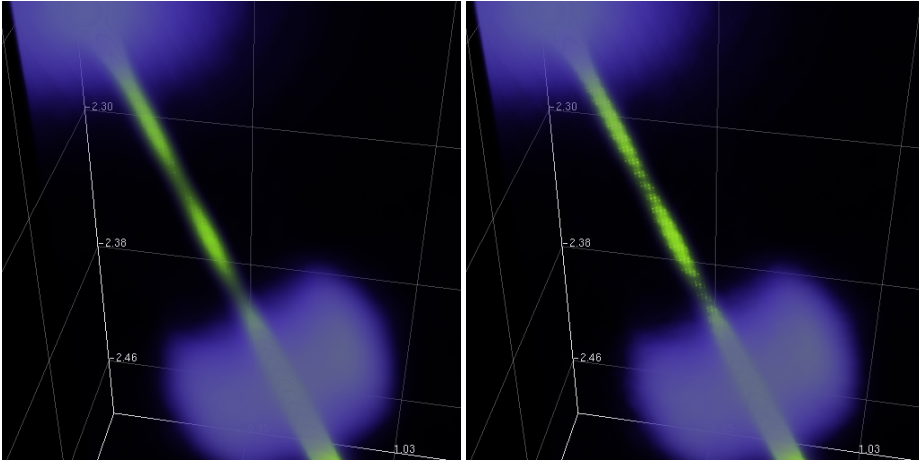
9.4.1 Interpolation Techniques

As described in Chapter 7, we have implemented two different interpolation methods, IDW and kriging. In this section, we will discuss the relative visual quality of the images produced using these techniques. Their performance is presented and discussed later, in Section 9.6.

Figure 9.4 shows three versions of the same part of the 27uc dataset, rendered using different interpolation techniques, Figure 9.4a was rendered with IDW with $u = 1$ (IDW1), Figure 9.4b IDW with $u = 2$ (IDW2), and Figure 9.4c kriging, using the exponential semivariogram described in Chapter 7, with $R = 0.0001$.

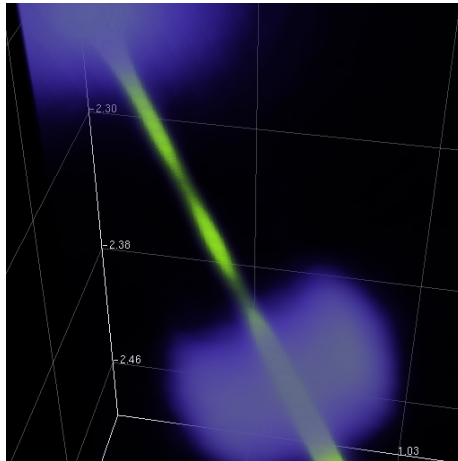
While the overall results are broadly similar, there are clear differences. The result of IDW2, in particular, differ from the two others. In the yellow-green rod connecting the two blue-white "spheres" several bright dots can be seen, these are barely visible in the IDW1 result, and can not be made out in the kriging result. Similarly, even ignoring the bright spots, the intensity does not vary smoothly along the rod in the IDW2 image, as it does in the two others. This is particularly noticeable in the rods low intensity, middle region.

Both of these artefacts can be attributed to the fact that IDW2 assigns more weight to close samples, and less weight to distant samples, compared to the other two methods. A ray passing close to a high intensity sample will therefore be highly affected by it, resulting in a bright pixel. A ray not passing so close to any high intensity samples will yield a comparatively duller pixel.



(a) IDW, $u = 1$.

(b) IDW, $u = 2$.



(c) Kriging.

Figure 9.4: Visual result of different interpolation methods.

Comparing IDW1 and kriging, we note that the results of IDW1, with the exception of the previously mentioned, barely visible, bright spots, are more smooth and visually pleasing than those of kriging. The highly complex nature of kriging interpolation makes it difficult to explain the cause of these results.

Finally, it is important to point out that kriging is a much more flexible method than IDW, because of the amount of freedom in selecting the variogram. Changing the variogram can give significantly different results. However, the poor performance of kriging (in terms of speed, discussed in more detail in Section 9.6) made it time-consuming and inconvenient to experiment with different settings. Other variograms might therefore give better results than those we have presented here.

9.4.2 Anisotropic Interpolation

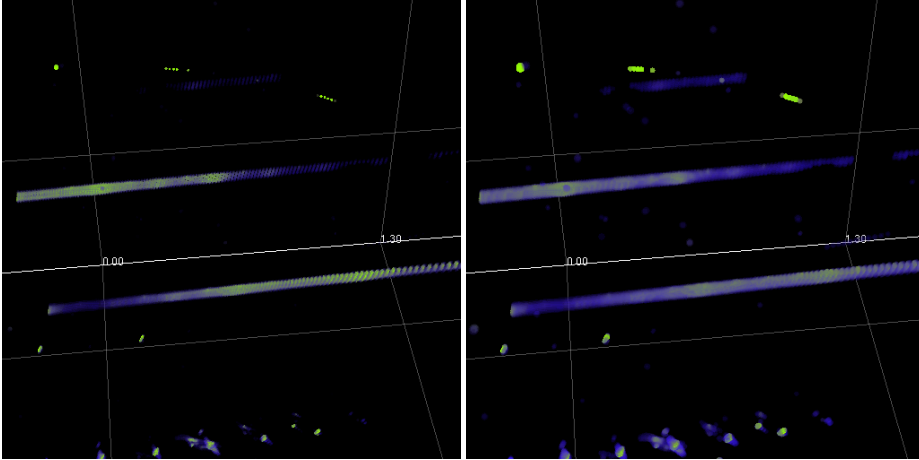
Figure 9.5 shows the effect of using anisotropic interpolation, and how it can be used to improve the image quality. The settings used to generate these images are shown in table 9.2. IDW interpolation with $u = 2$ were used in all cases.

Figure	Radius	Anisotropy
9.5a	0.005	$A = I_3$
9.5b	0.01	$A = I_3$
9.5c	0.005	$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Table 9.2: Settings used in Figure 9.5. I_3 is the 3×3 identity matrix.

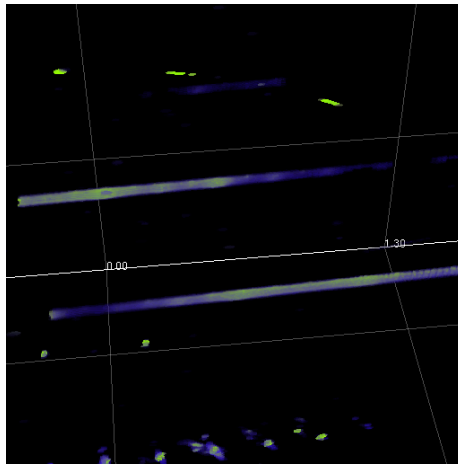
The images are from the 00571 dataset, zoomed in on two of the rod-like structures. Figure 9.5a uses a small radius, and no anisotropy. The resulting image has quite poor quality. In particular, the rightmost part of the rods appear discontinuous, as the search radius is too small to span the gaps between adjacent layers of samples. This problem can be mitigated by simply increasing the search radius. This is done in Figure 9.5b. While most of the artifacts of Figure 9.5a are removed, new ones are introduced. The increased radius increases blurring. The rods appear thicker, and the oscillation in intensity along the rods is less crisp.

Figure 9.5c combines a small radius with anisotropic distance measurement. The anisotropy used compacts distances along the rods. The result is that for interpolation points inside the rod, other samples inside the rod will be



(a) Radius: 0.005, no anisotropy.

(b) Radius: 0.001, no anisotropy.



(c) Radius: 0.005, anisotropy.

Figure 9.5: Effect of varying radius and anisotropic matrix. In (a), a small radius and no anisotropy is used. In (b) the radius is increased. In (c) the radius is the same as in (a), but anisotropic interpolation is used.

more heavily weighted. As we can see, the discontinuity artifacts are almost completely removed, without introducing the same amount of blurring, resulting in better image quality.

As described in Chapter 5, anisotropic interpolation is appropriate when the underlying function changes more rapidly in one direction than in others. In this case, the underlying function changes much more slowly along the rod than in directions orthogonal to it, so it should come as no surprise that the use of anisotropic distance improves the quality.

Anisotropic distance is, however, no silver bullet. In regions where the underlying function changes uniformly in all directions, the use of anisotropic interpolation will introduce artifacts. This can be seen in Figure 9.5c, the single points in the top and bottom of the image appear stretched, relative to the other figures.

9.5 Memory Consumption

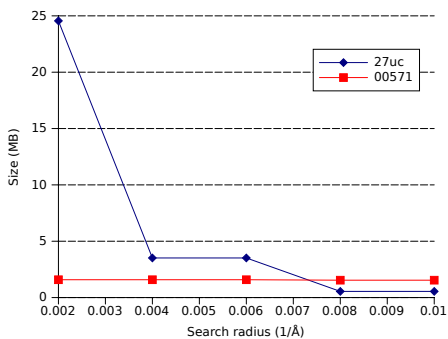
The raw data consumes considerable amounts of memory, as described in Section 9.1.1. However, threshold filtering will typically significantly reduce this. Furthermore, lowering the precision from double to single will naturally cut the memory consumption in half.

Turning from the raw data to the acceleration data structure, Figure 9.6 shows the memory consumption of the octree for different search radii, filtering thresholds and datasets. Note that this is only the memory used for the data structure, not the samples.

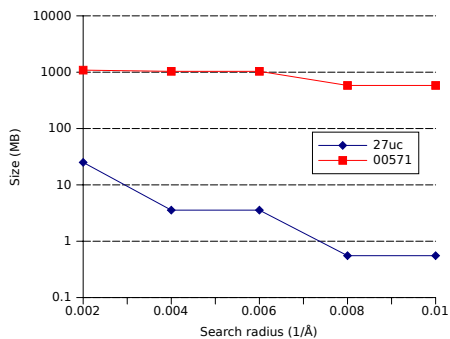
With a filtering threshold of 10, the memory consumption falls steadily for the 27uc dataset, but remains constant for the 00571 dataset as the search radius increases. Lowering the filtering threshold to 0.1 barely affects the 27uc dataset. In the case of the 00571 dataset, the lowered threshold significantly increases the memory consumption, and causes it to fall as the radius is increased.

The memory consumption of the octree depends upon two factors: the maximum depth of the tree, and its *completeness*, that is, how many of its branches have maximum depth. As explained in Chapter 7, we impose a maximum depth limit on the octree which depends upon the search radius (relative to the size of the bounding box). The maximum tree depths for the radii used here are shown in table 9.3. Completeness depends upon the density of samples. In dense areas, branches tend to be long, in less dense areas, they tend to be short. This is illustrated in Figure 9.7.

The 27uc dataset is fairly dense, with uniformly distributed samples, for both



(a) Filtering threshold 10.



(b) Filtering threshold 0.1.

Figure 9.6: Memory consumption of the octree data structure for different datasets, search radii and filtering thresholds. Note the logarithmic scale in (b).

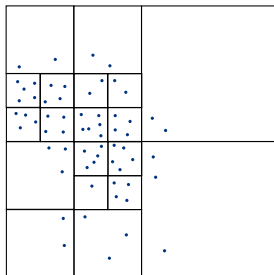


Figure 9.7: Branch length in octrees, illustrated in 2D with a quadtree. Here, nodes are split when it contains more than 4 samples. The tree is therefore deeper in the dense region on the left, compared to the less dense areas on the right. At the same time, the maximum tree depth is 3, so even though some level 3 leaf nodes contains more than 4 samples, they are not split.

	27uc		00571	
Radius	Depth	Max size	Depth	Max size
0.002	7	80MB	11	320 GB
0.004	6	10 MB	10	40 GB
0.006	6	10 MB	10	40 GB
0.008	5	1.25 MB	9	5 GB
0.01	5	1.25 MB	9	5 GB

Table 9.3: The maximum tree depth and maximum size for different search radii. The method for calculating the maximum tree depth is described in Chapter 7. The maximum size is the size of a complete tree, and is calculated using the formula: $S = n * \sum_{i=0}^D 8^i$, where D is the max depth, and n the size of a single node, in our case 40 B. Note that in practice, trees are rarely complete, these are pessimistic upper bounds.

the filtering thresholds shown here. The tree is therefore mostly complete, and the depth is limited by the maximum depth, not the number of samples. The memory consumption will therefore increase when the search radius is increased enough to allow the maximum depth to increase, as most of the branches of the tree will be expanded.

With a filtering threshold of 10, the 00571 dataset is not dense, with the exception of a few, small, high density regions. The tree will therefore be fairly incomplete, only a few of the branches are at full length. Increasing the maximum tree depth by increasing the search radius will therefore only cause a few of the branches to extend, and the memory consumption will remain close to constant.

Lowering the filtering threshold to 0.1 will include more samples, increasing the density, thus causing the tree to become more complete. The behaviour of the memory consumption as the search radius is increased therefore resembles that of the 27uc dataset. The increased completeness also causes the memory consumption to be on a overall higher level.

9.6 Performance

In this section, we will look at the performance of our implementation. The results stated in Section 7.1, of 125.1 s for 27uc and 4.75 s for 00571 are typical. To get more rigorous results, we created a movie of each dataset, by

rendering a high number of images at different locations. The results are shown in table 9.4.

Dataset	No. of Frames	Mean	Std. dev.
27uc	111	333.4 s	127.2 s
00571	101	11.9 s	2.70 s

Table 9.4: Mean and standard deviation of the rendering times for a high number of frames.

The high standard deviations illustrate an important point: the rendering time differs dramatically, depending upon the dataset, the settings used, and the scene rendered. In the remainder of this section we will look at how the performance of our implementation varies when different parameters are changed.

To better understand our results, we will first introduce a simple performance model for our ray casting method. We will then present timing results, and discuss them in the light of the performance model.

9.6.1 Performance Model

To more fully appreciate the results presented in this section, we will introduce a performance model. We will make our model as simple as possible, but be careful to mention the simplifying assumptions we make.

We use T to denote the time required to render a full image. To render a full image, we need to cast a ray for each pixel, so:

$$T = nT_r$$

Where n is the number of pixels/rays, and T_r is the time required to cast one ray. Here we assume that the same amount of time is required for all the rays, which is inaccurate.

To cast a single ray, we estimate the intensity at points along the ray. While different rays have different length in practice, we assume for simplicity that all have the same length L . Then, if the distance between estimation points is d , we need to estimate the intensity at $\frac{L}{d}$ points, so the time required for a single ray is:

$$T_r = \frac{L}{d}T_e$$

Where T_e is the time required for estimation at a single point, Here, we have ignore the time required to map the intensity to color and opacity values, and combine the colors and opacities at different points.

The estimation time consists of two parts: T_s , the time required to search and find the samples to be used during interpolation, and T_i , the time required for the interpolation itself:

$$T_e = T_s + T_i$$

As described in Chapter 7, range search in an octree consists of two steps: first, the acceleration data structure is used to find the samples in the search sphere. The search will typically also return some samples outside the sphere, so in a second step, all the returned samples must be checked. Therefore, the search time can be written as:

$$T_s = T_o + T_f$$

Where T_o is the time required for the search in the octree, and T_f the time required for the subsequent filtering.

Analyzing the time required for general range searches in octrees is complex. However, we always use the same search radius. Therefore, the range search is equivalent to one tree descent for each of the leaf nodes the search box might overlap. Since we have a fixed lower bound on the size of the leaf nodes, which depends upon the search radius, the number of such leaf nodes is always bounded by a constant. (This is explained in more detail in Chapter 7). Hence, the search time is proportional to the tree height:

$$T_o = O(\log(\frac{R}{r}))$$

where r is the search radius, and R the length of the sides of the bounding box.

The filtering time depends upon the number of samples returned. Assuming constant sample density D , the time is proportional to the volume of the search sphere:

$$T_f = O(Dr^3)$$

The time required for interpolation depends upon the method used, and the number of samples:

$$T_{i,IDW} = O(Dr^3)$$

$$T_{i,Krige} = O(Dr^9)$$

For IDW we only need to go through the samples and compute the distance to each sample. Hence, the time required is linear in the number of samples. Once again assuming constant sample density, the number of samples is proportional to the cube of the search radius.

Kriging involves computing the inverse of a $k \times k$ matrix, where k is the number of samples used, and the required time is therefore cubic in the number of samples, or proportional to the search radius to the power of 9.

Combining all of this, we get the following equation for the time required to render a full image using IDW interpolation:

$$T_{IDW} = O(n \frac{D}{d} (\log(\frac{R}{r}) + Qr^3))$$

Similar equations can be found for the other combinations.

9.6.2 Performance Results and Discussion

Here, we will look at how the rendering time varies when different settings are changed.

Search Radius

Figure 9.8 shows how the rendering time for several search radii. The rendering time increases with the search radius. The rate of increase does also grow with the search radius.

The fact that the rendering time increases with the search radius is expected. Increasing the search radius will increase both the time needed for searching and interpolating. The growth in the rate of increase is also expected, as the number of samples used during interpolation is proportional to the cube of the search radius (assuming constant density).

Step Size

Figure 9.9 shows the rendering time for several step sizes. As we can see, the rendering time decreases as the step size increases. The rate of decrease is also decreasing with the step size.

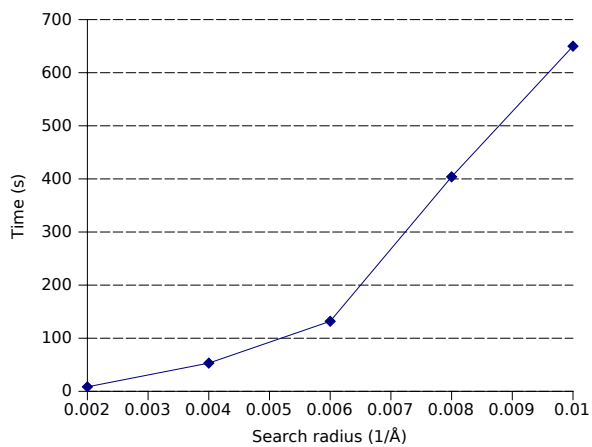


Figure 9.8: Rendering time with varying search radii.

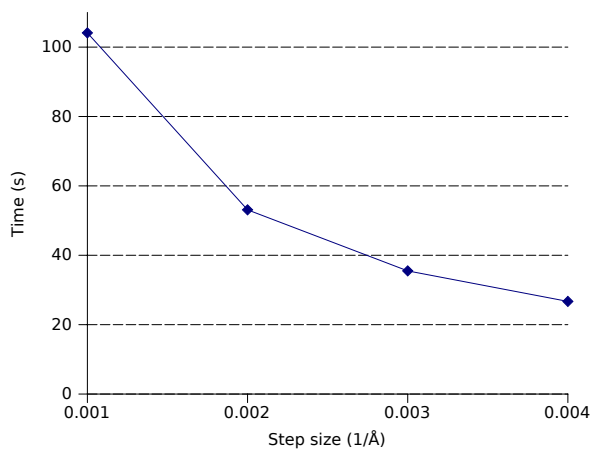


Figure 9.9: Rendering time with varying step size.

A decrease in rendering time as the step size increases is also expected. Increasing the step size is equivalent to reducing the number of points at which the intensity is estimated. The reduction in the rate of decrease is also expected as the number of estimation points is inversely proportional to the step size.

Interpolation Technique

Figure 9.10 shows timing results for the interpolation techniques mentioned in Section 9.4.1, for various search radii. As we see, the rendering time increases with the search radius. Furthermore, IDW2 is faster than IDW1, and both IDW techniques are significantly faster than kriging as the search radius increases.

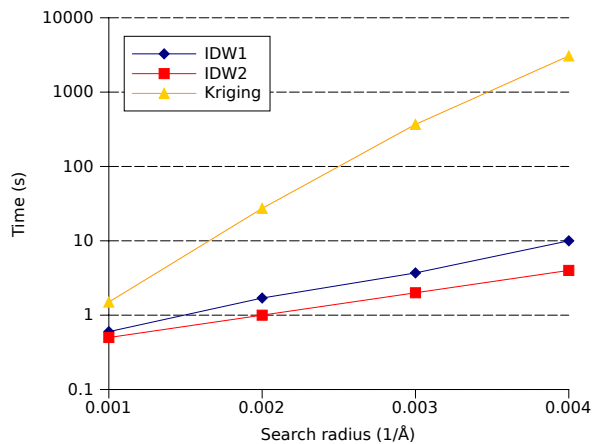


Figure 9.10: Rendering time with varying interpolation methods and search radii. Note the logarithmic scale.

The timing results of the different interpolation schemes are also as anticipated. Firstly, the rendering time with all the methods increase as the search radius is increased because more points are used for interpolation. Secondly, the rendering time with kriging increases much faster because its time complexity is $O(N^3)$ where N is the number of samples used, while the time complexity of IDW is $O(N)$. The fact that IDW1 is slightly faster than IDW2 is caused by the fact that no square roots, which is an expensive operation, needs to be computed.

Filtering

The performance impact of filtering is shown in Figure 9.11, both with and without our empty space skipping optimization. As an increasingly larger percentage of the low intensity points are filtered away, the rendering time decreases. Enabling the empty space skipping results in speedups between 3.13 and 3.69.

Filtering decreases the total number of samples n , which also causes the sample density D to drop⁴. Because of this, we expect the rendering time to decrease as more points are filtered away, which is consistent with what we observe. According to our model, decreasing the number of points should improve the performance due to two effects.

Firstly, the tree traversal should be faster because parts of the tree should be shallower. While the tree depth is depends upon the sample density, it is bounded by a size constraint on the leaf nodes relative to the search radius and bounding box (as described in Chapter 7). In this case, the maximum depth remained the same, but it was less complete, that is, several branches were shorter.

Secondly, the density is decreased, which reduces the time spent filtering the return of the octree range search, and the time spent interpolating.

Transfer Function

How the performance is affected by the transfer function is shown in Figure 9.12. Here, 512×512 images were rendered. The transfer functions used are listed in appendix B as t1 to t5, and only vary in how they assign opacity. The first transfer function assigns high opacity to medium and high intensity samples, while the other functions assigns decreasing amounts of opacity. It should be clear that equivalent results could be obtained by keeping the transfer function constant, and artificially changing the intensities of the samples, or changing the dataset or scene to one with samples with different intensities. As we can see, rendering scenes with semi-transparent objects is more time consuming than scenes with opaque objects, provided that early ray termination is enabled.

Our model is not sufficiently complex to explain the change in rendering time caused by changing the transfer function. This is because this effect is caused by the early ray termination optimization, described in Chapter 7, which we have not included in our model. Early ray termination stops rays once the accumulated opacity reaches a certain threshold, thus effectively reducing L . In

⁴More realistically, the density will only decrease in certain regions.

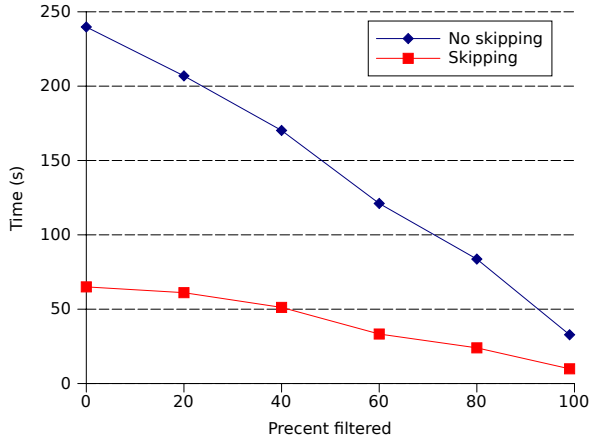


Figure 9.11: Rendering time with different filtering. In each case, the n percent of the samples with lowest intensity is discarded.

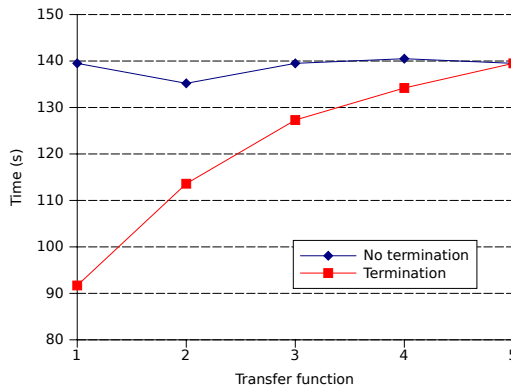


Figure 9.12: Rendering time with varying transfer functions.

a scene were a high degree of the samples have intensities which are mapped to high opacity values, many rays will be terminated prematurely. This will not be the case in a scene where most samples map to low opacity values, as few rays will accumulate the necessary opacity required to be terminated early. The main trend of the results we observe are therefore agreeing with our expectations. We also notice that the effect is less pronounced as we use increasingly "transparent" transfer functions. This is also as expected: once the scene is so transparent that a certain ray cannot be terminated before it leaves the volume, making the scene even more transparent will not affect that ray. Finally, disabling early ray termination will cause the rendering time to be the same for all transfer functions, which is expected, since no rays are terminated. Using early ray termination can result in a speedup of 1.54, when the most opaque transfer function is used.

9.6.3 Parallel Scaling

Figure 9.13 shows the speedup achieved as the number of threads is increased. These experiments were performed on machine A (with 6 cores).

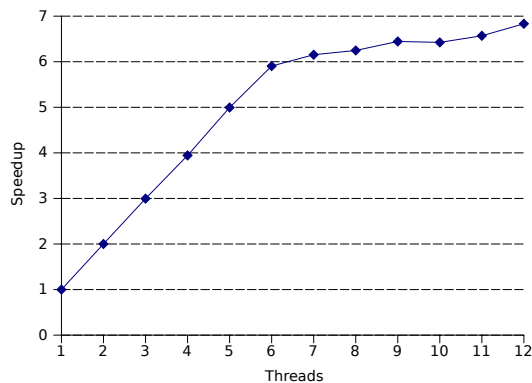


Figure 9.13: Speedup as the number of threads are increased.

As expected, our implementation scales exceptionally well as the number of threads are increased from 1 to 6. Beyond 6 threads we have more threads than physical cores, but we still see some improvement due to simultaneous multithreading (Hyper-Threading).

9.7 GPU

In this section, we will present and discuss results from the GPU version of our implementation. As already mentioned, these results are of a preliminary nature, as the GPU implementation is less polished than the CPU version.

9.7.1 GPU Compared to CPU

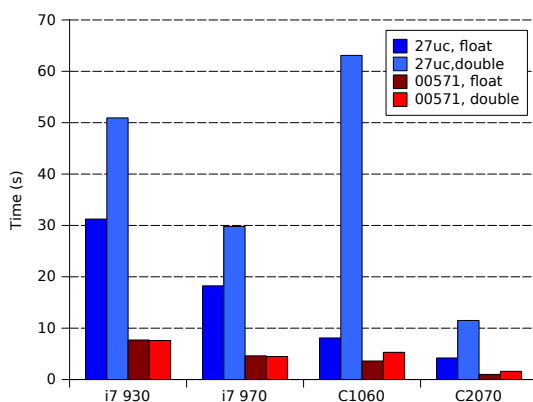


Figure 9.14: Rendering time on GPU and CPU, for different CPUs, GPUs, floating point precisions and datasets.

Figure 9.14 shows the rendering time on the CPU compared to the GPU, for different datasets and floating point precisions. Here, we have used both machine A and B, hence, we have results for two different CPUs and GPUs.

The first thing we notice is the significant speedups which are achieved when moving from the CPU to the GPU. Comparing single precision performance on the best CPU (i7 970) to the best GPU (C2070) we see a speedup of 4.64 for the 27uc dataset, and 4.62 for the 00571 dataset.

Secondly, we notice the poor performance of double precision floating point, in particular on the C1060 for the 27uc dataset, where single precision is 7.79 times faster than double precision. In general, there is a bigger difference between single and double precision for the 27uc dataset compared to 00571, and on the GPU, compared to the CPU.

The first result is expected. As we saw in Section 9.6.3, ray casting is a

highly parallel problem. It is also computationally intensive and therefore a ideal fit for GPUs. The improved performance is therefore as expected.

Furthermore, the poor performance of double precision is also as anticipated. The difference is more prominent on the GPU compared to the CPU, and the C1060 compared to the C2070 because GPUs have poor double precision support, especially on older GPUs such as the C1060 (as described in Chapter 8).

To explain the difference between the 27uc and 00571 datasets, we need to recall that, after filtering, 00571 is very sparse compared to 27uc. Most of the range searches will return no samples, so more time will be spent doing search than interpolation. The situation is reversed for the 27uc dataset, where more time is spent doing interpolation than search. Interpolation is more computationally intensive, involving expensive floating point operations such as multiplication, division and square roots. Searching is comparatively simple: the only floating point operations performed are comparison. Switching from single to double precision will therefore lead to a larger performance hit for the 27uc dataset, both on the CPU and GPU, and in particular on the older C1060.

9.7.2 GPU Optimizations

Figure 9.15 shows the speedup achieved when applying the memory optimizations described in Chapter 8. The effect of the texture optimization varies. On the C1060 with the 27uc dataset, we see a clear improvement, however, on the C2070 with the same dataset, using texture memory decreases performance slightly. For the other combinations of datasets and compute platforms, we see slight improvements. There is no effect of the node cache optimization, except on the C1060 for the 00571 dataset, where we observe a slight improvement.

As already mentioned, the 00571 dataset consists of fewer samples after filtering. Using texture memory for the samples might therefore cause less performance improvements, since there will be fewer memory accesses. The C2070 does, as opposed to the C1060 have L2 and L1 cache. These might serve the same purpose as using texture memory, which would explain why we don't see any improvement for 27uc on the C2070.

The rendering time of 27uc is dominated by interpolation. The node cache only speeds up the range search, and this might explain why it has little effect for this dataset. For 00571, we see some improvement on the C1060 as expected, but not on the C2070. That might be caused by the fact that the C2070 is equipped with L1 and L2 caches, which might makes our user managed cache unnecessary and superfluous.

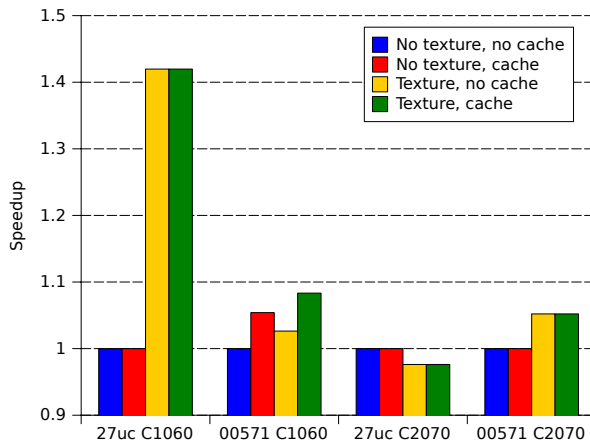


Figure 9.15: Normalized rendering time with different combinations of optimizations, datasets and GPU's. The normalization makes the effect of the optimizations more visible for each dataset/GPU combination. It is, however, not possible to make direct comparison between the dataset/GPU combinations.

Chapter 10

Conclusion and Future Work

X-ray diffraction experiments are used extensively to study the structure and properties of materials. Improved sensor technology has recently made 3D datasets common, but visualization of such datasets remains challenging, since the datasets are large, and unstructured.

The goal of this thesis was to investigate how high quality visualizations of 3D X-ray diffraction patterns can be generated in an efficient manner. We have developed a method based on volume ray casting, and implemented it on the CPU and GPU.

Our method differs from traditional ray casting algorithms by not using data in the form of voxels. Our input is a set of samples of an X-ray diffraction pattern. These samples are not taken on a uniform grid and therefore constitute an unstructured volume dataset. The method we have developed operates directly on these samples, rather than resampling them on a uniform grid.

The intensity of the X-ray diffraction pattern was estimated at points along the ray by interpolating among nearby samples. We have used an octree to facilitate efficient range search to find these samples. Two different interpolation techniques were implemented, inverse distance weighing, and kriging. Both of these techniques have been adapted to support anisotropic interpolation. To improve performance we have implemented standard ray casting optimization techniques such as early ray termination and empty space skipping. We have developed a novel, acceleration data structure agnostic algorithm for performing

empty space skipping, suitable for situations where voxels or similar representations are not used. Threshold filtering was used to remove samples with low intensity that would otherwise not contribute to the final image. In practice, we could often remove between 40% to 99% of the samples, greatly reducing memory consumption, and improving performance.

Volume ray casting is a highly parallel, computationally intensive problem, and is therefore ideally suited for the GPU. We have implemented a fully functional, but less flexible and optimized version for the GPU, using NVIDIA's CUDA framework. To improve performance we have used texture memory for the samples, and cached the topmost nodes of the octree in shared memory.

We have tested our implementation using actual X-ray diffraction data, consisting of up to over 120 M samples. Our method is capable of producing images of good quality. Of the implemented interpolation techniques, IDW with $u = 2$, in combination with anisotropic interpolation where appropriate, yielded the best results in terms of visual quality.

The rendering time varies significantly, from 5 s to over 200 s, for realistic settings, depending upon dataset, scene, and settings used. To better understand this variability, we have developed, and empirically tested a simple performance model. There was a good agreement between the predictions of the model and the observed rendering times. Our implementation scaled very well to more processor cores, with a speedup of 5.9 for 6 cores. For the GPU version of our implementation, we saw speedups around 4.6.

10.1 Future Work

Possible future work can broadly be divided into efforts to improve the visual quality of images, and efforts to improve the performance in terms of rendering time and memory consumption.

To achieve better image quality it could be fruitful to investigate more complex and sophisticated optical models, rather than the simple emission-absorption model we have used. Furthermore, it would be interesting to explore new interpolation techniques, or variations of those we have used, in combination with anisotropic interpolation.

Currently, the optimal settings and transfer function for a dataset must be determined manually, often through time consuming trial and error. It would therefore be useful to develop a system that could analyze the dataset, and automatically determine the optimal settings, or at least provide a good guess.

For certain datasets, the best settings varies in different parts of the dataset,

this is particularly true for anisotropy. Extending our method to allow different setting to be used in different parts of the dataset is therefore also an interesting possibility.

To improve performance, two approaches can be taken: the current algorithm could be further optimized, or a completely new or heavily modified version could be developed.

While our implementation is fairly well optimized for the CPU, we believe it is possible to improve performance even further. In particular, it would be interesting to see if data could be reorganized so that SIMD instructions could be used during interpolation, and if changing the order in which rays are cast could be optimized to better exploit caching.

More time could be spent optimizing our GPU implementation. The performance of GPU codes is highly sensitive to optimization, and is therefore a big potential for significant improvement. At the same time, the GPU version could be made as flexible and full featured as the CPU version. Another possibility would be extend the system to use multiple GPUs, or GPUs in combination with the CPU. This would include the development of sophisticated load balancing systems.

Performance can also be improved by changing or modifying the algorithm used. This might include using a different acceleration data structure, somehow organizing the samples prior to rendering, or performing resampling and preinterpolation. Using a completely different visualization algorithm is also a possibility. Splatting promises improved performance relative to ray casting, at the cost of flexibility and image quality. Comparing these methods for rendering X-ray diffraction data would be interesting.

Finally, it would be highly interesting to attempt to exploit the structure of the input data. We currently treat our input data as an unstructured volume dataset. The samples are, however, not actually randomly distributed, but rather organized on well defined curved layers. Taking advantage of this structure might result in big performance improvements, and potentially also improve image quality.

Bibliography

- [1] NVIDIA, *NVIDIA CUDA C Programming Guide*, 2012. <http://developer.nvidia.com/nvidia-gpu-computing-documentation> Accessed 30.04.2012.
- [2] J. Als-Nielsen and D. McMorrow, *Elements of Modern X-ray Physics*. Wiley, 2nd ed., 2011.
- [3] N. Kasai and M. Kakudo, *X-Ray Diffraction by Macromolecules*. Springer, 2005.
- [4] J. B. Fløystad, “Domain structures in PbTiO₃ thin films,” Master’s thesis, Department of Physics, Norwegian University of Science and Technology, 2010.
- [5] S. O. Mariager *et al.*, “High-resolution three-dimensional reciprocal-space mapping of InAs nanowires,” *Journal of Applied Crystallography*, 2009.
- [6] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3D surface construction algorithm,” *SIGGRAPH Comput. Graph.*, vol. 21, pp. 163–169, Aug. 1987.
- [7] C. Hansen and C. R. Johnson, *The Visualization Handbook*. Elsevier, 2005.
- [8] J. Wallis, T. Miller, C. Lerner, and E. Kleerup, “Three-dimensional display in nuclear medicine,” *Medical Imaging, IEEE Transactions on*, vol. 8, pp. 297–230, Dec 1989.
- [9] Y. Sato *et al.*, “Local maximum intensity projection (LIMP): A new rendering method for vascular visualization,” *Journal of Computer Assisted Tomography*, vol. 22, pp. 912–917, Nov.-Dec. 1998.

- [10] M. Levoy, “Display of surfaces from volume data,” *Computer Graphics and Applications, IEEE*, vol. 8, pp. 29–37, May 1988.
- [11] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl, “A simple and flexible volume rendering framework for graphics-hardware-based raycasting,” in *Volume Graphics, 2005. Fourth International Workshop on*, pp. 187–241, June 2005.
- [12] L. Westover, *SPLATTING: A parallel, feed-forward volume rendering algorithm*. PhD thesis, University of North Carolina - Chapel Hill, 1991.
- [13] P. Lacroute and M. Levoy, “Fast volume rendering using a shear-warp factorization of the viewing transformation,” in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques, SIGGRAPH '94*, pp. 451–458, ACM, 1994.
- [14] A. Van Gelder and K. Kim, “Direct volume rendering with shading via three-dimensional textures,” in *Volume Visualization, 1996. Proceedings., 1996 Symposium on*, pp. 23–30, 98, Oct. 1996.
- [15] T. J. Cullip and U. Neumann, “Accelerating volume reconstruction with 3D texture hardware,” *Radiation Oncology*, vol. 61, pp. 1–6, 1994.
- [16] B. Cabral, N. Cam, and J. Foran, “Accelerated volume rendering and tomographic reconstruction using texture mapping hardware,” in *Proceedings of the 1994 symposium on Volume visualization, VVS '94*, pp. 91–98, ACM, 1994.
- [17] M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski, “Advanced illumination techniques for GPU-based volume raycasting,” in *ACM SIGGRAPH 2009 Courses, SIGGRAPH '09*, pp. 2:1–2:166, ACM, 2009.
- [18] N. Max, “Optical models for direct volume rendering,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 1, pp. 99–108, Jun 1995.
- [19] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, “GPU computing,” *Proceedings of the IEEE*, vol. 96, pp. 879–899, May 2008.
- [20] J. Nickolls and W. Dally, “The GPU computing era,” *Micro, IEEE*, vol. 30, pp. 56–69, Mar.-Apr. 2010.

- [21] T. L. Falch, “Optimization and parallelization of ptychography reconstruction code,” project report, Department of Computer and Information Science, Norwegian University of Science and Technology, 2011.
- [22] P. Pacheco, *An Introduction to Parallel Programming*. Elsevier Morgan Kaufman, 2nd ed., 2011.
- [23] M. J. Flynn, “Some computer organizations and their effectiveness,” *Computers, IEEE Transactions on*, vol. C-21, pp. 948–960, Sept. 1972.
- [24] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS ’67 (Spring), pp. 483–485, ACM, 1967.
- [25] J. Gustafson, “Reevaluating Amdahl’s law,” *Communications of the ACM*, vol. 31, pp. 532–533, May 1988.
- [26] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPOPP ’08, pp. 73–82, ACM, 2008.
- [27] NVIDIA, *CUDA C Best Practices Guide*, 2012. <http://developer.nvidia.com/nvidia-gpu-computing-documentation> Accessed 04.05.2012.
- [28] E. Kreyzing, *Advanced Engineering Mathematics*. Wiley, 9th ed., 2006.
- [29] Wikipedia, *Trilinear Interpolation*, 2012. http://en.wikipedia.org/wiki/Trilinear_interpolation Accessed 06.06.2012.
- [30] D. Shepard, “A two-dimensional interpolation function for irregularly-spaced data,” in *Proceedings of the 1968 23rd ACM national conference*, ACM ’68, pp. 517–524, ACM, 1968.
- [31] N. Cressie, “The origins of kriging,” *Mathematical Geology*, vol. 22, pp. 239–252, 1990.
- [32] M. Tomczak, “Spatial interpolation and its uncertainty using automated anisotropic inverse distance weighting (IDW) - cross-validation/jackknife approach,” *Journal of Geographic Information and Decision Analysis*, vol. 2, no. 2, pp. 18–30, 1998.

- [33] C. Castano-Moraga, M. Rodriguez-Florido, L. Alvarez, C.-F. Westin, and J. Ruiz-Alzola, “Anisotropic interpolation of DT-MRI,” in *Medical Image Computing and Computer-Assisted Intervention MICCAI 2004* (C. Barillot, D. Haynor, and P. Hellier, eds.), vol. 3216, pp. 343–350, Springer Berlin / Heidelberg, 2004.
- [34] H. Ludvigsen, “Real-time GPU-based 3D ultrasound reconstruction and visualization,” Master’s thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2010.
- [35] E. O. Aksnes, “Simulation of fluid flow through porous rocks on modern GPUs,” Master’s thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2009.
- [36] R. Eidissen, “Utilizing GPUs for real-time visualization of snow,” Master’s thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2009.
- [37] Øystein. E. Krog, “GPU-based real-time snow avalanche simulations,” Master’s thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2010.
- [38] Øystein. Krog and A. Elster, “Fast GPU-based fluid simulations using SPH,” in *Applied Parallel and Scientific Computing* (K. Jnasson, ed.), vol. 7134 of *Lecture Notes in Computer Science*, pp. 98–109, Springer Berlin / Heidelberg, 2012.
- [39] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, “Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering,” in *Proceedings of the 2009 symposium on Interactive 3D graphics and games, I3D '09*, pp. 15–22, ACM, 2009.
- [40] L. Marsalek, A. Hauber, and P. Slusallek, “High-speed volume ray casting with CUDA,” in *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, p. 185, Aug. 2008.
- [41] J. Wihelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri, “Direct volume rendering of curvilinear volumes,” *SIGGRAPH Comput. Graph.*, vol. 24, pp. 41–47, Nov. 1990.
- [42] P. Navratil, J. Johnson, and V. Bromm, “Visualization of cosmological particle-based datasets,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, pp. 1712–1718, Nov.-Dec. 2007.

- [43] R. Fraedrich, S. Auer, and R. Westermann, “Efficient high-quality volume rendering of SPH data,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 16, pp. 1533–1540, Nov.-Dec. 2010.
- [44] M. P. Garrity, “Raytracing irregular volume data,” *SIGGRAPH Comput. Graph.*, vol. 24, pp. 35–40, Nov. 1990.
- [45] C. Giertsen, “Volume visualization of sparse irregular meshes,” *Computer Graphics and Applications, IEEE*, vol. 12, pp. 40–48, Mar. 1992.
- [46] M. Hopf and T. Ertl, “Hierarchical splatting of scattered data,” in *Proceedings of the 14th IEEE Visualization 2003 (VIS’03)*, VIS ’03, pp. 57–, IEEE Computer Society, 2003.
- [47] M. Chen, “Combining point clouds and volume objects in volume scene graphs,” in *Volume Graphics, 2005. Fourth International Workshop on*, pp. 127–235, June 2005.
- [48] Kahler *et al.*, “Simultaneous GPU-assisted raycasting of unstructured point sets and volumetric grid data,” *Volume Graphics*, 2007.
- [49] J. Huang, K. Mueller, R. Crawfis, D. Bartz, and M. Meissner, “A practical evaluation of popular volume rendering algorithms,” in *Volume Visualization, 2000. VV 2000. IEEE Symposium on*, pp. 81–90, Oct. 2000.
- [50] R. Gonzalez and R. Woods, *Digital Image Processing*. Pearson, 3rd ed., 2008.
- [51] R. E. Walpole *et al.*, *Probability and Statistics for Scientists and Engineers*. Pearson Prentice Hall, 8th ed., 2007.
- [52] E. Angel, *Interactive Computer Graphics*. Addison Wesley, 5th ed., 2009.
- [53] M. Levoy, “Efficient ray tracing of volume data,” *ACM Trans. Graph.*, vol. 9, July 1990.
- [54] A. S. Glassner, *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.
- [55] M. de Berg *et al.*, *Computational Geometry, Algorithms and Applications*. Springer Berlin Heidelberg, 3rd ed., 2008.
- [56] J. L. Bentley and J. H. Friedman, “Data structures for range searching,” *ACM Comput. Surv.*, vol. 11, pp. 397–409, Dec. 1979.

- [57] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.
- [58] R. A. Finkel and J. L. Bentley, “Quad trees: A data structure for retrieval on composite keys,” *Acta Informatica*, vol. 4, pp. 1–9, 1974.
- [59] Wikipedia, *Octree*, 2012. <http://en.wikipedia.org/wiki/Octree> Accessed 07.05.2012.
- [60] NVIDIA, *NVIDIA’s Next Generation CUDA Compute Architecture: Fermi*, 2012. http://www.nvidia.com/object/fermi_architecture.html Accessed 30.04.2012.
- [61] The Open Group, *clock_getres*. http://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_getres.html Accessed 17.05.2012.

Appendix A

RSV User Manual

This is the user manual for RSV, a tool using volume ray casting to visualize X-ray diffraction patterns.

A.1 Quick Start

If you are impatient, this should be enough to get you started:

To compile:

```
$ make
```

To run:

```
$ ./a.out
```

Use     and the arrow keys to move around.

A.2 Overview

This manual describes how to use RSV. We'll focus on how to use the tool, rather than giving detailed descriptions of its inner workings, interested readers should

check out the master's thesis¹ written about RSV for an in depth treatment.

A.2.1 Raycasting

Raycasting² is a technique to generate 2D images from 3D volume data, and is illustrated in Figures 1 and 2. As shown in Figure 1, for each pixel in the image, a "ray of light" is sent from the eye/camera, through the pixel, and into the volume. Samples of the volume will be taken at points along the ray, and the value at each sample is mapped to a color. Finally the colors for all the samples of the ray are blended to find the final color for the pixel.

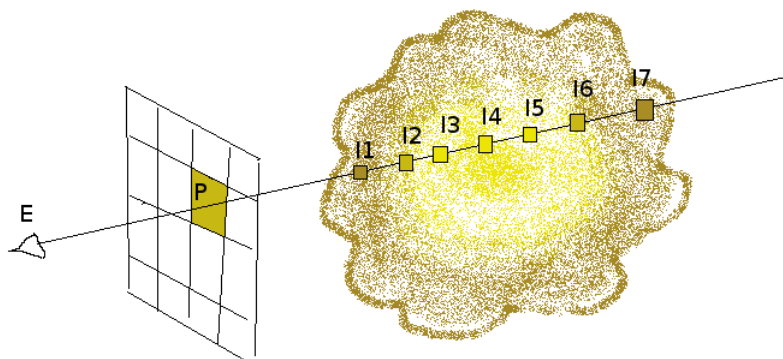


Figure A.1: High quality figure illustrating raycasting. A ray is cast from the eye at E, through the pixel P and into the volume. At the points I1 - I7 the value of the volume is found, and mapped to a color. The colors are then blended to find the final color for the pixel.

A.2.2 Files

This is an overview of the files used by RSV (after compilation):

a.out The executable.

config.txt Configuration file.

¹Thomas L. Falch, *3D Visualization of X-ray Diffraction Data*, Norwegian University of Science and Technology, 2012

²Not to be confused with the similar technique of ray tracing.

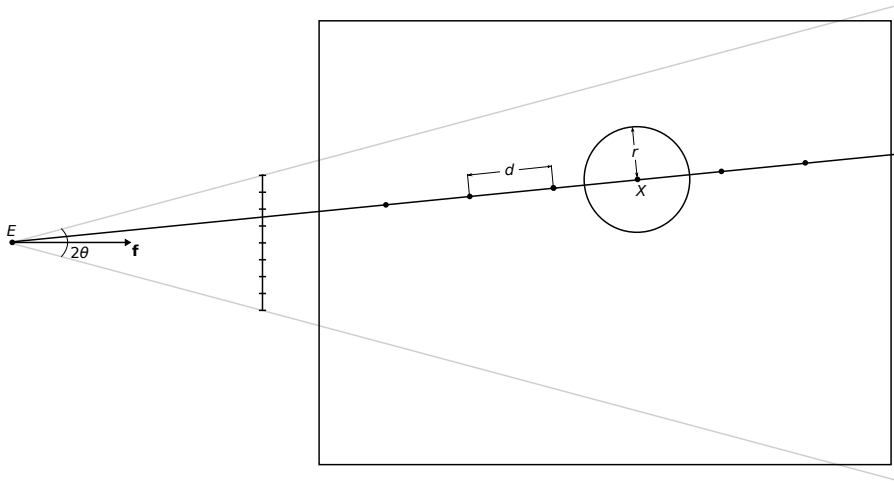


Figure A.2: Schematic overview of raycasting in 2D

color.txt Color configuration file. Optional. If not present, default colors will be used.

batch.txt Used to specify a batch of images to be generated. Only needed in batch mode.

moviemaker.sh Script to create movies from sets of images. Only needed in batch mode.

In addition, an input hdf5 file is obviously required.

A.3 Compilation

A.3.1 Dependencies

The following libraries are required in order to compile and run RSV:

- hdf5
- openGL
- glut (in particular, the freeglut implementation is required)

- lapack
- blas
- ffmpeg (for movie creation, not required for compilation)
- imagemagick (for movie creation, not required for compilation)

A.3.2 Compilation

To compile, it should be enough to use:

```
$ make
```

CUDA Support

To enable CUDA support, you need to download and install the SDK (and possibly driver) from NVIDIA's web pages: www.nvidia.com. Make sure you can compile and run some of the example programs before proceeding.

Then, replace the line:

```
cuda=false
```

in the makefile with :

```
cuda=true
```

And then:

```
$ make clean
```

```
$ make
```

Precision

RSV supports both single and double precision floating point numbers. Switching requires recompilation, and is controlled by the `float` flag in the makefile. For single precision, set:

```
float=true
```

for double precision set:

```
float=false
```

And then:

```
$ make clean
```

```
$ make
```

to recompile.

A.4 Use

A.4.1 Input File

The input hdf5 file can be specified in one of two ways. Either by using the `-f` option, ie:

```
$ ./a.out -f myfile.hdf5
```

Or by using the `DEFAULT_FOLDER` setting in the `config.txt` file, ie:

```
DEFAULT_FOLDER=/home/data/
```

If the default folder setting is used, RSV will search through the folder for hdf5 files. The first one found will be used.

A.4.2 Moving the Camera

Use **w** **s** **a** **d** **q** **z** to move the camera forwards, backwards, left, right, up and down, respectively. Use **←** **→** **↑** **↓** to rotate the camera to the left, right, up and down, respectively.

Use **x** to decrease the fov (zoom in), **c** to increase the fov (zoom out) and **X** to reset the fov to its default value.

In zoom mode, the mouse can be used to select an area to zoom in on. This is done by left-clicking and dragging. The area indicated by the opaque rectangle will be zoomed in upon. Zooming is not done by moving the camera, but by reducing the field of view, corresponding to theta in Figure 2. Use **□** to toggle between zoom and trace modes.

A.4.3 Grid

Use **g** to toggle the visibility of the grid. Use **n** to toggle the visibility of numbers on the grid. Other aspects of the grid can be controlled with settings in the configuration file.

A.4.4 HUD

Use **h** to toggle the visibility of the HUD. Note that the progress bar is an experimental feature, which might be wrong.

A.4.5 Color


Use **t** to toggle the display of the transfer function editor. Use the mouse to select color channels and the values of the different "bands". The image will only be updated when the transfer function overlay is hidden.

Use **y** to save the current color transfer function. **WARNING:** this will overwrite the contents of the color.txt file. Make a backup of this file if necessary.


A.4.6 Saving Images

Use **b** to save an image of what's currently displayed. The image will be saved as rsw_hh-mm-ss-dd-mm-yyyy.bmp, based upon the time of creation.

A.4.7 Trace Single Ray

In trace mode, a single ray can be traced, and the intensity, and accumulated color for each interpolated point will be dumped to a file named raydump.txt
WARNING: any existing file with this name will be overwritten. To trace a single ray, left-click on any pixel. Use  to toggle between zoom and trace modes.

A.4.8 Configuration File

Most of the parameters of RSV can be found in the config.txt configuration file. The file is read when RSV is started. Pressing  will cause RSV to reread the file, but some settings will be ignored, as they require RSV to be restarted.

Settings may be commented out (or removed completely), in which case default values will be used. (However, the file, even if empty, must be present).

The following list provides a list of the settings available, a description, default value, and whether changing the settings requires restart.

DEFAULT_FOLDER Unless the input file is specified with the -f option, RSV will look in this folder, and use the first hdf5 file found as input.
Default value: /home/data. Requires restart.

INTERPOLATION_RADIUS Corresponds to r in Figure 2. To estimate the intensity at x , all data points whose distance to x is less than r is used.
Default value: 0.002. Requires restart.

RESOLUTION The size of the images generated will be **RESOLUTION** * **RESOLUTION**. This value must be a power of 2 (ie 256, 512 etc).
Default value: 512. Requires restart.

INTERPOLATION_MODE The algorithm used for interpolation. The possible values are:

- 0 Kriging.
- 1 Inverse distance weighting with $p = 1$ (standard IDW).
- 2 Inverse distance weighting with $p = 2$.

See somewhere else for a more detailed description of these algorithms.
Default value: 1. Does not require restart.

NEIGHBOUR_MODE Determines which data points are used for interpolation. The possible values are:

0 Some number of close points. In particular, all the points in the leaf node of the interpolation point.

1 All points within INTERPOLATION_RADIUS of the interpolation point.

It may seem as if using 0 will cause the value of INTERPOLATION_RADIUS to be ignored. However, the value of INTERPOLATION_RADIUS influences the depth of the tree, and hence the size of leaf nodes.

Default value: 1. Requires restart.

STEP_SIZE Corresponds to d in Figure 2. The distance between interpolation points along a ray.

Default value: 0.05. Requires restart.

ANISO_MATRIX Specifies the matrix to be used for anisotropic distance measurement for interpolation. Row major.

Default value: 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0. Does not require restart.

EYE_INITIAL The initial position of the eye/camera. Corresponds to e in Figure 2.

Default value: Depends upon dataset. Requires restart.

FORWARD_INITIAL The initial direction of the eye/camera. Corresponds to f in Figure 2.

Default value: Depends upon dataset. Requires restart.

GRID_BASE The base of the grid lines. Possible values are:

0 Origo (ie (0,0,0)).

1 The corner of the bounding box with smallest coordinates.

Default value: 1. Does not require restart.

GRID_SPACING The distance between grid lines. Default value: Depends upon dataset. Does not require restart.

VERBOSE The amount of "usefull" output. Possible values 0-2. Higher value gives more output.

Default value: 1. Does not require restart.

FILTERING_THRESHOLD All datapoints with a intensity less than this will be ignored.

Default value: 0. Requires restart.

MEDIAN_FILTERING Whether median filtering should be used or not.
 Default value: 0. Requires restart.

NUM_THREADS The number of threads used.
 Default value: Depends upon processor, the number of threads used will be the same as the number of logical processors. Requires restart.

COLOR_BANDS The number of points to interpolate between for the color transfer function. If the color.txt file is used, the numbers must correspond.
 Default value: 11. Requires restart.

DATA_STRUCTURE The type of spatial subdivision datastructure used to accelerate the range searches. Possible values are:

- 0** Octtree.
- 1** Grid.

Default value: 0. Requires restart.
 NOTE: The grid datastructure is experimental.

STEP_FACTOR The factor with which the step size is increased during empty space skipping.
 Default value: 2.0. Does not require restart.

STEP_LIMIT Max step size, the step size will not exceed this value during empty space skipping.
 Default value: 0.04. Does not require restart.

OPACITY_THRESHOLD Rays will be terminated when this accumulated opacity is reached.
 Default value: 1.0. Does not require restart.

THRESHOLD_MEDIAN_FILTER Samples whose intensity is higher than this value times the median intensity of its neighbours will be removed during median filtering.
 Default value: 15. Does not require restart.

VARIOGRAM_RADIUS This is R in the semivariogram $1 - e^{-\frac{d}{R}}$.
 Default value: 0.00001. Does not require restart.

BATCH_SIZE During loading, samples are read from file in batches of this size, filtered, and inserted into the acceleration data structure. Useful when

the raw data size is larger than main memory (but the filtered data size is not).

Default value: 1e9. Does not require restart.

NOTE: Only a minimum of input validation is performed. Stupid values (like negative search radius) might cause undefined results (i.e. segfaults).

Syntax

The configuration file parser is, well, kinda crappy, so it's important that the syntax is correct. For most settings it is:

`NAME=value`

With the name in all caps, and no whitespace. The only exception is `ANISO_MATRIX`, whitespace is used to separate the values of the matrix (but not before the first value).

Empty lines, and lines starting with `#` are ignored.

A.4.9 Batch Mode

Batch mode is used to create a number of images, and convert them into a movie, without user intervention. To start RSV in batch mode, use the `-b` option.

Using batch mode requires the `batch.txt` file to be present. This file specifies the images to be generated. The syntax is:

```
xe ye ze xf yf zf
f
xe ye ze xf yf zf
f
xe ye ze xf yf zf
```

Odd lines specifies camera positions and directions, corresponding to E and f in Figure 2.(cf. `EYE_INITIAL` and `FORWARD_INITIAL`). Even lines specifies the number of frames between the positions. The camera will be smoothly moved and turned from one position to the next.

NOTE: The rotation algorithm requires the cross product of two subsequent forward vectors to have a nonzero length.

A.4.10 Single Image Mode

Single image mode is used to generate a single image. After the image is generated, it will be written to a file, and the application will be terminated. Incremental update is disabled, nothing will be displayed until the full resolution image is finished. To start RSV in single image mode, use the `-s` option.

A.4.11 Quitting

Use `Esc` to quit.

A.5 Development

RSV has been designed to be unusually easy to modify and extend. Here, an overview of the code is presented as an aid for such undertakings.

The file `simple.c`, parts of which are reproduced in listing is a minimal working program. Figure A.3 shows a callgraph for this program.

Compilation couldn't be made simpler:

```
$ make simple
```

To run:

```
$ ./simple
```

Running the program will cause an image to be generated and saved, but no GUI will be shown.

```
1 #include <...>
2
3 Ranges* ranges;
4 Root* root;
5 Raycreator* rc;
6 Ray* rays;
7 Display_color** images;
8 Transfer_Overlay* transfer_overlay;
9
10 void load_data(){
11     Loader l = init_loader(1e9);
```

```

12     root = create_root(l.ranges);
13     while(has_next(&l)){
14         root->insert_points(root->s, next(&l));
15     }
16     root->finalize(root->s);
17 }
18
19 void setup(){
20     rc = init_raycreator(&root->ranges);
21     rays = (Ray*)malloc(sizeof(Ray)*RESOLUTION*RESOLUTION);
22
23     //When incremental update is used, we need all these buffers
24     //Now we really only need the last one, but have to create all of
25     //them
26     images = (Display_color**)malloc(sizeof(Display_color*) * (int)(
27         log2(RESOLUTION) - 3));
28     int c = 0;
29     for(int res = 16; res <= RESOLUTION; res *=2){
30         images[c] = (Display_color*)malloc(sizeof(Display_color) * res
31             * res);
32         c++;
33     }
34     transfer_overlay = init_transfer_overlay(&root->ranges);
35 }
36
37 int main(int argc, char** argv){
38     load_config_file(0);
39     load_data();
40     setup();
41     create_rays(rc, rays);
42     trace_rays(RESOLUTION, 0, RESOLUTION*RESOLUTION);
43     int i = log2(RESOLUTION) - 4;
44     write_bmp(images[i], RESOLUTION, RESOLUTION, NULL);
45 }

```

Listing A.1: Parts of simple.c

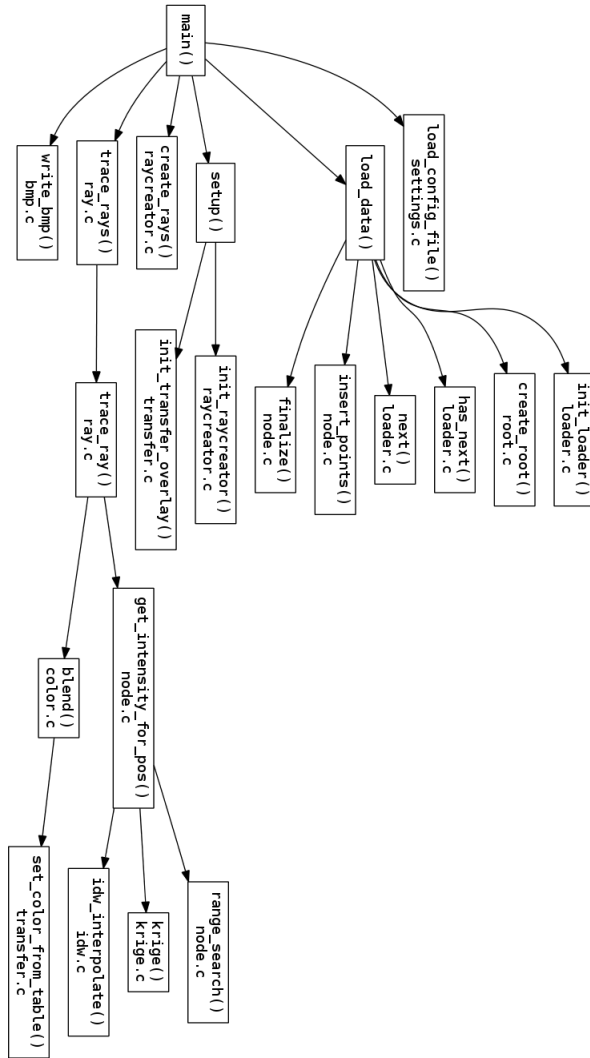


Figure A.3: Callgraph of the most important functions, and which files they are located in.

Appendix B

Transfer Functions

These are the transfer functions used in the filtering, and transfer function experiments in chapter 9.

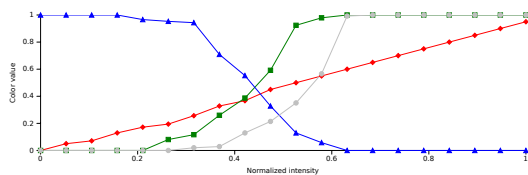


Figure B.1: Transfer function f1

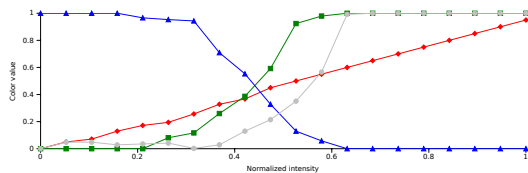


Figure B.2: Transfer function f2

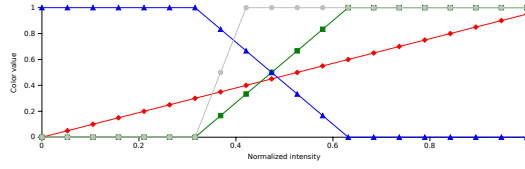


Figure B.3: Transfer function t1

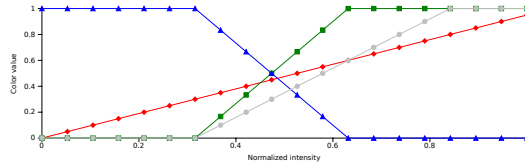


Figure B.4: Transfer function t2

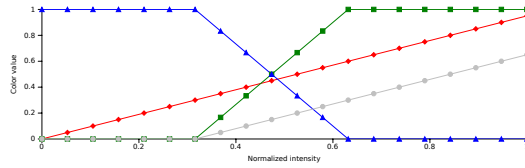


Figure B.5: Transfer function t3

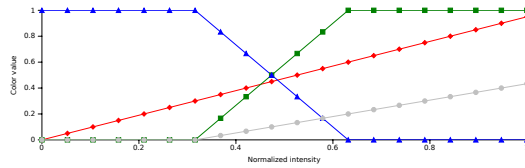


Figure B.6: Transfer function t4

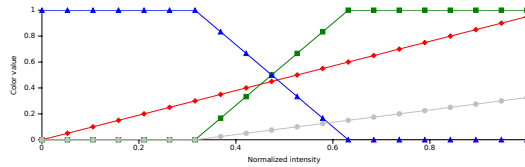


Figure B.7: Transfer function t5

Appendix C

Selected Source Code

Some selected parts of our source code is included here. The full source code, including Makefiles is located in the source code attachment upladed to the thesis submission system of the Department of Computer and Information Science. It is also freely available at <http://folk.ntnu.no/thomafal/master>.

```
1 Display_color trace_ray(Ray* ray, Root* root, int print){
2   if(ray->distance <= 0){
3     Display_color b = {0,0,0,0};
4     return b;
5   }
6
7   if(ray->distance < -1){
8     Display_color b = {255,0,0,255};
9     return b;
10  }
11
12  if(ray->color.r > -1 && print == 0){
13    return to_display_color(ray->color);
14  }
15
16  FILE* file;
17  if(print){
18    file = fopen("raydump.txt", "w+");
19  }
20
21  normalize_ray(ray);
22  Color output = {0,0,0,0};
23
24  Coord pos = ray->start;
25
26  double acc_distance = 0;
27  double local_step_size = STEP_SIZE;
28  while(acc_distance < ray->distance){
29
30    if(inside(pos, &root->ranges)){
31      real_t intensity = root->get_intensity_for_pos(root->s, pos);
32
33      if(intensity > 0 && local_step_size > STEP_SIZE){
34        acc_distance += local_step_size;
35        pos = add_scaled_Coord(pos, ray->dir, -local_step_size);
36        local_step_size = STEP_SIZE;
37      }
38      else if(intensity == 0 && (local_step_size * STEP_FACTOR) <= STEP_LIMIT){
39        local_step_size *= STEP_FACTOR;
```

```

40     }
41     else{
42         blend(&output, intensity);
43
44         if (print){
45             fprintf( file, "%f,%f,%f,%f,%f\n",intensity, output.r, output.g, output.b, output.a);
46         }
47         if (output.a > OPACITY.THRESHOLD){
48             break;
49         }
50     }
51 }
52
53 pos = add_scaled_Coord(pos, ray->dir, local_step_size);
54 acc_distance += local_step_size;
55 }
56 ray->color = output;
57
58 if (print){
59     fclose( file);
60 }
61 return to_display_color(output);
62 }

```

Listing C.1: From ray.c

```

1 void insert_point(Node* n, Point* p, Ranges r, int depth){
2     if(n->is_leaf){
3
4         //We insert the point in this node
5         if(n->num_children < 8 || depth >= max_depth){
6             if(n->num_children == 0){
7                 n->pointer = (void**)malloc(sizeof(void*) * 8);
8             }
9
10            //We need to expand the child array
11            if(n->num_children == n->is_leaf){
12                n->is_leaf *= 2; //We should check that it doesn't overflow...
13                n->pointer = (void**)realloc((void*)n->pointer, sizeof(void*) * n->is_leaf);
14            }
15            if(n->pointer == NULL){
16                printf("%d_NULL!\n", n->is_leaf);
17            }
18        }
19
20        n->pointer[n->num_children++] = (void*)p;
21
22        if(n->num_children > max_num_children){
23            max_num_children = n->num_children;
24        }
25    }
26
27    //This node is full, and should be split
28    else{
29        //Back up points
30        Point* points[8];
31        for(int c = 0; c < 8; c++){
32            points[c] = (Point*)n->pointer[c];
33        }
34        free(n->pointer);
35
36        //Overwrite to make new leaf nodes
37        n->pointer = (void**)get_new_nodes(node_list, 8);
38        real_t dx = (r.xmax - r.xmin)/4.0;
39        real_t dy = (r.ymax - r.ymin)/4.0;
40        real_t dz = (r.zmax - r.zmin)/4.0;
41
42        for(int c= 0; c < 8; c++){
43            ((Node*)n->pointer)[c].is_leaf = 8;
44            ((Node*)n->pointer)[c].num_children = 0;
45            ((Node*)n->pointer)[c].x = n->x + (dx * ((c < 4) ? 1 : -1));
46            ((Node*)n->pointer)[c].y = n->y + (dy * ((c%4 < 2) ? 1 : -1));
47            ((Node*)n->pointer)[c].z = n->z + (dz * ((c%2 == 0) ? 1 : -1));
48        }

```

```

49
50 //Insert existing points into new leafs
51 for(int c = 0; c < 8; c++){
52     int index = get_index(*points[c], r);
53     if(index == -1){
54         //No point left behind!
55         continue;
56     }
57     Ranges nr = get_ranges_for_index(index, r);
58     insert_point( &((Node*)n->pointer)[index], points[c], nr, depth+1);
59 }
60
61 //Insert new point into new leafs
62 int index = get_index(*p,r);
63 Ranges nr = get_ranges_for_index(index, r);
64 insert_point( &((Node*)n->pointer)[index] , p, nr, depth+1);
65
66 //This node is no longer a leaf
67 n->is_leaf = 0;
68
69 num.leaves += 7;
70 }
71 }
72 //This ain't no leaf, we just continue down
73 else{
74     int index = get_index(*p,r);
75     Ranges nr = get_ranges_for_index(index, r);
76     insert_point( &((Node*)n->pointer)[index] , p, nr, depth+1);
77 }
78 }

```

Listing C.2: From node.c

```

1 #ifdef TEXTURE
2 ..device.. real_t get_intensity_for_pos_full(Coord pos, Node* nodes, Node* node_cache, unsigned int* stack,
3 int base){
4 #else
5 ..device.. real_t get_intensity_for_pos_full(Coord pos, Point* points, Node* nodes, Node* node_cache,
6 unsigned int* stack, int base){
7 #endif
8
9     real_t intensity = 0;
10    real_t weight = 0;
11    short tos = 0;
12    unsigned char inc = 1;
13
14    unsigned int current_node = 0;
15    unsigned int b = get_covered_subnodes(pos, root.node);
16    stack[base + tos] = current_node | (b << 24);
17    tos += inc;
18
19    while(tos > 0){
20        current_node = stack[base + tos - inc] & 0x00ffffff;
21        b = (stack[base + tos - inc] & 0xff000000) >> 24;
22
23        if(b > 255){
24            return 0;
25        }
26
27        unsigned int t = 1;
28        while((b & t) == 0){
29            t = t << 1;
30        }
31        current_node += (unsigned int)log2((float)t);
32        b = b ^ t;
33
34        if(b == 0){
35            tos -= inc;
36        }
37
38        else{
39            stack[base + tos - inc] = (stack[base + tos - inc] & 0x00ffffff) | (b << 24);

```

```

40     Node n;
41
42     if (current_node < NODE.CACHE_SIZE){
43         n = node_cache[current_node];
44     }
45     else{
46         n = nodes[current_node];
47     }
48
49     if (n.is_leaf){
50         for (short c = 0; c < n.num_children; c++){
51 #ifdef TEXTURE
52             float4 point = tex1Dfetch(pointTexture, (long int)n.pointer + c);
53 #else
54             Point point = points[(long int)n.pointer + c];
55 #endif
56
57             real_t dx = pos.x - point.x;
58             real_t dy = pos.y - point.y;
59             real_t dz = pos.z - point.z;
60
61             real_t distance = sqrt(dx*dx + dy*dy + dz*dz);
62
63             if (distance < INTERPOLATION.RADIUS.D){
64 #ifdef TEXTURE
65                 intensity += (1/distance)*point.w;
66 #else
67                 intensity += (1/distance)*point.intensity;
68 #endif
69                 weight += (1/distance);
70             }
71         }
72     }
73     else{
74         b = get_covered_subnodes(pos, nodes[current_node]);
75         unsigned int temp = (unsigned int)nodes[current_node].pointer;
76         stack[base + tos] = (temp | (b << 24));
77         tos += inc;
78     }
79 }
80
81 if (intensity <= 0){
82     return 0;
83 }
84
85 real_t ratio = intensity/weight;
86 if (ratio <= 1){
87     return 0;
88 }
89
90 return log(ratio);
91 }
92
93 #ifdef TEXTURE
94 __global__ void kernel(Node* nodes, Display_color* image, Ray* rays, Color* colors, unsigned int* stack, int
stack_size){
95 #else
96 __global__ void kernel(Point* points, Node* nodes, Display_color* image, Ray* rays, Color* colors, unsigned
int* stack, int stack_size){
97 #endif
98     int i = blockIdx.x * blockDim.x + threadIdx.x;
99
100     __shared__ Node node_cache[NODE.CACHE_SIZE];
101
102     if (threadIdx.x < NODE.CACHE_SIZE){
103         node_cache[threadIdx.x] = nodes[threadIdx.x];
104     }
105     __syncthreads();
106
107     if (rays[i].distance <= 0){
108         Display_color b = {0,0,0,0};
109         image[i] = b;
110         return;
111     }
112
113     rays[i] = normalize_ray(rays[i]);
114

```

```

115   Coord pos = rays[i].start;
116   real_t acc_distance = 0;
117   Color output = {0.0,0.0,0.0};
118
119   while(acc_distance < rays[i].distance){
120     #ifdef TEXTURE
121       real_t intensity = get_intensity_for_pos_full(pos, nodes, node_cache, stack, i*stack_size);
122     #else
123       real_t intensity = get_intensity_for_pos_full(pos, points, nodes, node_cache, stack, i*stack_size);
124     #endif
125
126     output = blend_d(output, intensity, colors);
127
128     if(output.a > 0.99f){
129       break;
130     }
131
132     pos = pos + (rays[i].dir*STEP_SIZE_D);
133     acc_distance += STEP_SIZE_D;
134   }
135
136   image[i] = to_display_color_d(output);
137 }

```

Listing C.3: From raytrace_kernel.cu,label=shit3

Appendix D

Poster

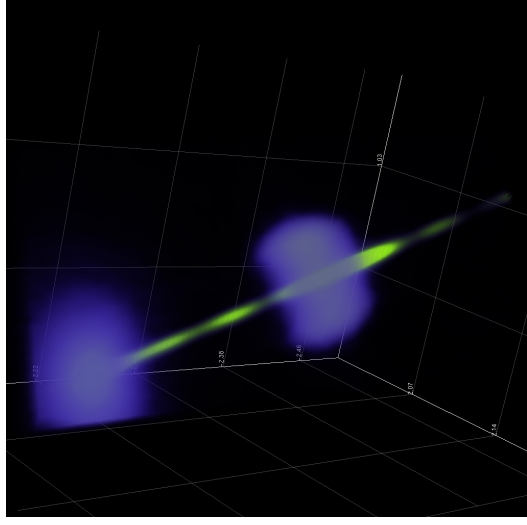
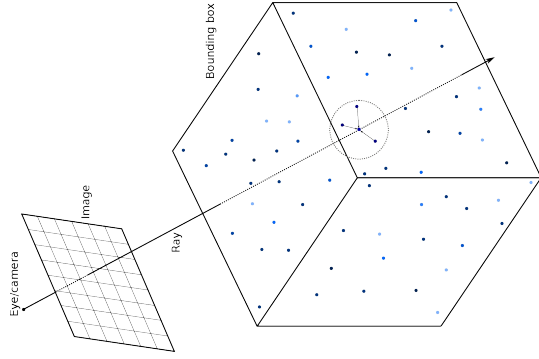
The poster shown on the next page, which summarizes this thesis, is to be displayed and presented by the author at the International Supercomputing Conference, in Hamburg, June 18-20 2012.

Using Raycasting to Visualize X-ray Diffraction Patterns

Thomas Løfsgaard Falch Supervisors: Anne C. Elster, Dag W. Breiby, Jostein Fløystad

Introduction

- X-ray diffraction experiments are used extensively in the sciences to study the structure and properties of materials.
- The result of these experiments are samples of the X-ray diffraction pattern, essentially a 3D unstructured dataset.
- 3D visualization is important, but remains a challenge.



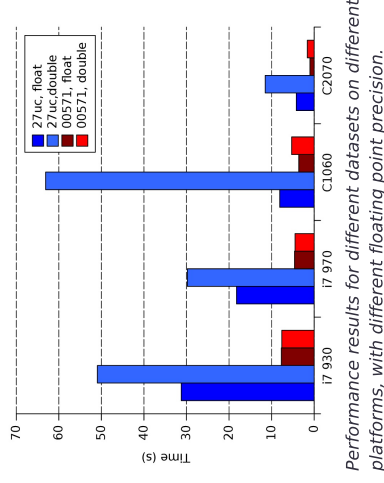
Example output. Diffraction pattern from thin film of PbTiO_3

Results

- Tested with real X-ray diffraction data, consisting of up to 120M samples.
- Rendering time varies widely, between 5 s and 200 s.
- IDW in combination with anisotropic interpolation yields best results.
- Speedup of 5.9 with 6 threads.
- Speedup of 4.6 on GPU, compared to 6-core CPU.

Conclusion and Future Work

- Our method can produce images of good quality, but is still slow.
- Future work includes further optimization, in particular for the GPU, and investigation of alternative visualization algorithms.



Method

- We use volume ray casting to visualize the diffraction patterns.
- No resampling. We operate directly on the unstructured samples.
- We estimate the intensity by interpolating among nearby samples, using either IDW or kriging, in combination with anisotropic interpolation.
- We have implemented our method for both the CPU and the GPU.



NTNU – Trondheim
Norwegian University of
Science and Technology



HPC-Lab
Computer & Info. Science
Norwegian University of
Science and Technology