Visualization of Fuel Cell Simulations

Niklas Röber
Otto-von-Guericke-Universität, Magdeburg

Diplomarbeit

nroeber@cs.uni-magdeburg.de

November 30, 2002

This work is dedicated to my parents.
Diese Arbeit ist meinen Eltern gewidmet.

Abstract

Visualization plays an important role in the analysis of scientific data sets. Here computer graphics can be used as a tool to extract information and to transform abstract data sets into meaningful images. While many good visualization techniques are known, the interactive display of huge time-varying data sets is still a challenging task.

This thesis develops a visualization pipeline that uses several different compression techniques to increase the rendering performance and to allow a more interactive visualization of large data sets. The data which is used for this is a numerical simulation of a fuel cell. This data set is multiparametric and consist of five scalar and one vector data set. In the second part of the thesis, some visualization techniques for the display of such data sets are discussed and evaluated on the fuel cell example.

Zusammenfassung

Die Visualisierung spielt eine grosse Rolle bei der Auswertung wissenschaftlicher Datensätze. Techniken und Methoden der Computer Graphik können hier effizient genutzt werden um abstrakte Datensätze in aussagekräftige Bilder zu verwandeln. Obwohl viele gute Visualisierngstechniken bekannt sind, ist die interaktive Darstellung grosser Datensätze noch immer schwierig.

Ziel dieser Dimplomarbeit war es einen Algorithmus zu entwickeln, der es unter Zuhilfenahme von Komprimierungstechniken ermöglicht grosse Datensätze interaktiv darzustellen. Der Datensatz welcher hierzu als Beispiel genutzt wurde ist die numerische Simulation einer Brennstoffzelle. Dieser Datensatz ist ausserdem multimodal und besteht aus 5 Skalaren und einem Vektordatensatz. Im zweiten Teil der Dimplomarbeit werden einige Techniken zur Visualisierung solcher Multiparameter Datensätze erläutert, und am Beispiel des Brennstoffzellen Datensatzes erklärt.

Acknowledgement

I would like to take the opportunity to say thank you to a few people who helped me in the writing of this thesis and throughout my studies. First of all, I would like to thank my family, my parents and my brother for their huge support, not just financially, but also for their motivation and encouragement.
Many thanks also to all who helped me with brainstorming, programming, writing or just talking, especially Torsten, Reza, Steve, Melanie, Ken and Ali to name a few. Special thanks also to the entire GrUVi lab, which was the best place for me to perform my research. The lab was full of support and everyone always had a minute to share. I will definitely miss this atmosphere.
Also many thanks to everyone from my home university in Magdeburg for their invaluable support to make this possible. I would like to name everyone, but I only have space for a few who most influenced me with their work, motivation and encouragement: my supervisors and tutors Thomas Strothotte, Floh and Roland, as well as Jörg, Maic, Stefan, Bert and Klaus Toennies.
A very special thanks to all my friends who motivated me and tried to distract me from focusing too much on my thesis, especially Steven and Martin.

Selbstständigkeitserklärung

Hiermit versichere ich, Niklas Röber (Matrikel 152946),die vorliegende Arbeit allein und nur unter der Verwendung der angegebenen Quellen angefertigt zu haben.

Vancouver, den 30. November 2002 Niklas Röber

Contents

Chapter 1

Introduction

Computers play an important role in today's society. They have proven to be very versatile and are used for many different applications. The main quality that makes computers so multitalented is the incredible speed which can be used to process the data. But this all would be useless, if one would not be able to look at the data. A computer can really only add numbers and works with binary data which is very unusual for humans to look at.
Visualization and scientific visualization in particular deals with the *translation* of this binary information into a graphical representation. Vision is our strongest sense and helps us to orient ourselves. Visualization can be basically divided into two groups, information visualization and scientific visualization. While the border between is very blurred, information visualization deals with abstract n-dimensional data sets while scientific data sets are usually 3- or 4-dimensional and have a spatial reference. One every day example is the visualization of the file structure on the various hard drives in a computer. The general focus in visualization is the extraction and display of features from a given data set. Figure 1.1 shows two examples from information and scientific visualization. The left one shows a file browser under Linux, while the right one displays the flow of air into an aircraft engine [Gro].
The huge processing power of high performance computers can be used to simulate complicated processes using mathematical approximations. These simulations can help to reduce the number of expensive constructions and experiments to a minimum. They are also very attractive and invite users to change some parameters which would not be possible in practice. These simulations can be used to increase the efficiency and to save time and resources. The difficult part with these simulations is the visualization of the experimental results. Here computer graphics can be used as a tool to display complicated information.
The focus for this thesis is the visualization of fuel cell simulations. The two most challenging attributes of these data sets are their size and multi-

parametric nature. Currently, the technology and algorithms used are not fast enough to produce the final simulations, but the realistic dimensions of the data set will be around $250 \times 250 \times 1000 \times t$. Additionally for all sampling points, five scalar values and one vector value exist which represent attributes like the concentration of oxygen or hydrogen and flow information. Chapter 2 has a more detailed introduction on the data sets and to fuel cell technology. In the following two sections the research goal and the layout

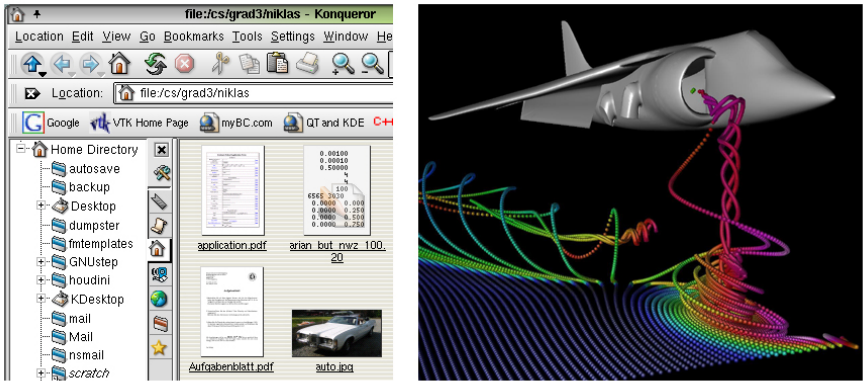

Figure 1.1: Examples for information a), and scientific visualization b)

of the thesis are described in more detail. The next section represents the intention of the research and describes what qualities could be used for an efficient and expressive visualization of fuel cell simulations. The following section describes the structure and the organization of this work.

1.1 Intentions

The current visualization of the fuel cell data is a slice extraction of one of the six data sets and a display using contour lines. This slice extraction can only be performed along the z-axis. The goal of the research and this thesis is the improvement of this simple visualization and to search for more appropriate display techniques.
Because of the nature of the fuel cell data set, the research goal can be divided into two parts. The first one is the development of fast visualization techniques for huge time-varying volumetric data sets. Here the current available volume rendering techniques have to be surveyed and evaluated for their applicability. Also compression techniques might be useful in order to reduce the amount of data. To eventually create a tool which can be used with other data sets as well, the used compression techniques must be able to compress the data set without any loss of information. This is

very important for medical data where physicians need uncompressed images without artifacts. The focus for this task is on fast visualization with good image quality.
Additionally, multiparameter visualization techniques have to be evaluated for their usefulness for the fuel cell data set. Possibly not all data sets have to be diplayed at the same time. Maybe a few are characteristic enough to display most of the contained information. Here the focus is on the development of a simple tool which can be easily used to display the fuel cell data using common visualization techniques. Further research has to be performed to find more appropriate visualization techniques which can adequately represent the contained information.

1.2 Structure

This thesis is organized in eight chapters. The first Chapter, this one, gives an introduction into the topic and explains the research goal and lays out the outline of the thesis.
In the second Chapter some sample data sets are discussed. The interesting part here is where these data sets originate, what they contain and what the goal for the visualization is. In contrast to all data sets which are described in Chapter 2, the fuel cell simulation data sets are the main application for this thesis and are discussed in more detail. Some information about the functionality of a fuel cell will be presented as well.
Chapter 3 is the theoretical introduction into the topic. Here some basic ideas of volume rendering, signal- and wavelet theory are explained which are required later in Chapters 4 and 5. This Chapter is also used to compare some existing techniques in volume rendering and to evaluate these with regard to their applicability to the visualization of fuel cell simulations. The second part of Chapter 3 is used to introduce signal theory and wavelet compression, which are both needed for the later developed technique for fast volume visualization.
Chapter 4 describes a method which was implemented by using some compression ideas from Chapter 3. The main principle of this algorithm is to use a more efficient lattice to store the data and some lossless/lossy compression techniques to further shrink the size of the data set. For the final visualization of the data set common graphics hardware of current PCs is used to render huge data sets at interactive rates. Here, Chapter 4 explains the algorithm in more detail and gives also some references to previous and related work. In the beginning of Chapter 4 more efficient grids are introduced for static and time-varying data sets which allows to store the same amount of information with fewer samples. In the following sections some pre-segmentation and compression techniques are discussed which additionally help to reduce the data size. In the end, some visualization techniques

are presented for volume rendering using current texture mapping hardware. Chapter 5 in contrast deals with the multiparametric nature of the fuel cell data sets. Here some techniques are described in theory and some selected methods have been implemented. The focus here is to create meaningful visualizations which help in the process of gathering information. Some of the presented techniques are well known and some are new. Chapter 5 presents a wide variety of possible techniques for multiparameter visualization. All discussed methods are compared with respect to their applicability for visualizing the fuel cell data set. Even though this data set is of main interest for Chapters 4 and 5, all described methods and techniques can be used with other data sets as well.
Achieved results are presented in Chapter 6. This Chapter is divided into three parts, where the first two sections present qualitative and quantitative results from Chapter 4 and the last section describes the achieved results from Chapter5. Finally some conclusions are drawn about the usefulness of the developed ideas.
Chapter 7 presents some screenshots and describes some more details about the actual implementation. Here also some code examples for important parts of the program are shown and explained.
The last Chapter summarizes the work and compares the achieved results with other existing techniques. In the end of Chapter 8 some ideas are presented for future improvement and development.

Chapter 2

Applications

Even though the main application for this thesis is the fuel cell data set, which will be discussed in more detail in Section 2.2.1, efficient and fast visualization is necessary for the display of all data sets. The goal of this Chapter is to motivate the need for good visualization by presenting sample data sets which are used in this thesis. Most of the data sets discussed here originate either from volvis.org [Mei00] or from the National Library of Medicine [Set].
Usually data sets are sampled on a regular grid. These are the only data sets which are considered in this thesis. Some very large data sets are sampled on irregular grids where a grid definition is needed in order to visualize the contained information. Because the rendering of these data sets is more complicated and computationally expensive, the current implementation focuses on regular data sets only.
In the first section of this Chapter some medical data sets are discussed while the following section focuses more on general scientific data sets. Also in Section 2.2.1 a detailed introduction to the fuel cell simulation data is given with background information of how fuel cells actually work.

2.1 Medical Data

In medical imaging, visualization is needed on a daily basis to diagnose patients and to treat illnesses. Since the discovery of the X-Rays in 1895 by Wilhelm Conrad Röntgen, medical science has developed several very efficient imaging techniques. These techniques have evolved in the last decades and are able to scan down to fractions of a millimetre. The data sets which are generated using these imaging techniques need to be reconstructed first. Depending on the imaging technique used, the format of the data can range from 8 bit up to 32 bit. Most of these medical imaging techniques are

also capable of creating 3- or 4-dimensional volumetric data sets. The most commonly used ones are:

- computed tomography (CT),
- magnetic resonance imaging (MRI),
- positron emission tomography (PET),
- single photon emission computed tomography (SPECT), and
- Ultrasound (US).

While CT and MRI are used to capture anatomical information, PET and SPECT are utilized for functional imaging. Also MRI can be used for functional imaging by measuring the differences in the oxygen concentration in particular tissues. All these techniques can also be *abused* to create other scientific data sets, as can be seen in the next section.
The medical sample data sets used for the visualization and compression for the work in this thesis are:

- visible human male (MRI) ($256 \times 256 \times 512$)
- visible human male (CT) ($512 \times 512 \times 1877$)
- visible human male (Photographs) ($2048 \times 1216 \times 1877$)
- dynamic kidney study (dSPECT) ($90 \times 90 \times 80 \times 64$)
- mouse embryo (Ultrasound) ($256 \times 256 \times 256$)
- unc brain (MRI) ($256 \times 256 \times 145$)
- head (MRI) ($128 \times 128 \times 128$)
- aneurism (MRI) ($256 \times 256 \times 256$)

The visible human data sets are created and sponsored by the National Library of Medicine [Set] for scientific research. Three different modalities are available, but have to be preprocessed prior the rendering.
The dynamic SPECT study shows the reconstruction of a human kidney which was imaged using the SPECT technique. This data was reconstructed differently to reveal the temporal behaviour of the kidney function. In this time-varying data set one can observe the washout of the radioactive tracers in the kidney.
Additionally, three other data sets have been used. One is a 3D ultrasound volume of a mouse embryo and the other two are MRI scans of the brain and the head of a human male.

2.2 Scientific Data

Besides the medical data sets from the previous section, also other scientific data sets were used which shall be discussed in this Section. Some of these data sets are created using simulations or are created synthetically by eval-

uating mathematical equations, but most are scanned using either CT or MRI imaging. Most of these data set are already resampled to 8 bit. The data sets are:

- Vienna Christmas tree (CT) ($512 \times 512 \times 999$)
- Marchner Lobb (various sizes)
- frog (MRI) ($500 \times 470 \times 138$)
- engine (CT) ($256 \times 256 \times 128$)
- lobster (MRI) ($301 \times 324 \times 56$)
- nucleon ($41 \times 41 \times 41$)
- statue leg (CT) ($341 \times 341 \times 90$)
- tomato (MRI)($256 \times 256 \times 64$)
- bonsai tree (CT) ($256 \times 256 \times 256$)
- skull (CT) ($256 \times 256 \times 256$)

The first one is a Christmas tree which was created by the computer graphics department of the Technical University of Vienna and used in various styles on a Christmas card [TUW]. The tree was scanned using a CT scanner and exist in three different resolutions. The Marschner-Lobb data set [ML94] was created to evaluate the quality of different interpolation filters. A time-varying version of the original function was implemented and is used as a test data set throughout the thesis. The remaining data sets are from volvis.org [Mei00] and represent several commonly used data sets in volume rendering. These data sets are either simulations, or are acquired using CT or MRI.

2.2.1 Fuel Cells

This section is used to discuss the fuel cell simulations and the actual fuel cell in more detail. Here, first a theoretical introduction about the working principle is given, followed by an analysis of the simulation data.
A picture of a fuel cell can be seen in Figure 2.1, which shows a transportable fuel cell from Ballard Power Systems [Sys01]. A fuel cell is an electrochemi-

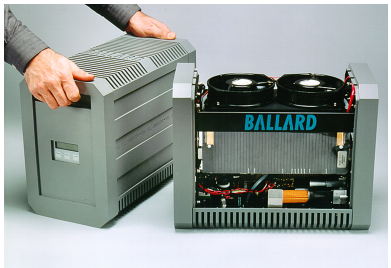

Figure 2.1: Transportable fuel cell

cal energy conversion device which converts hydrogen and oxygen into water,

heat and electricity. The proton exchange membrane fuel cell (PEMFC) uses one of the simplest reactions of any fuel cell. Figure 2.2 shows the principle of such a fuel cell. On the anode, hydrogen is dispersed into the fuel cell and the hydrogen molecules release electrons. On the cathode side, oxygen is lead into the fuel cell and electrons are conducted from the external circuit to the catalyst. Here hydrogen and oxygen are combined to water. The electrolyte is the proton exchange membrane. This membrane conducts positive ions, but blocks electrons. The catalyst supports the reaction and

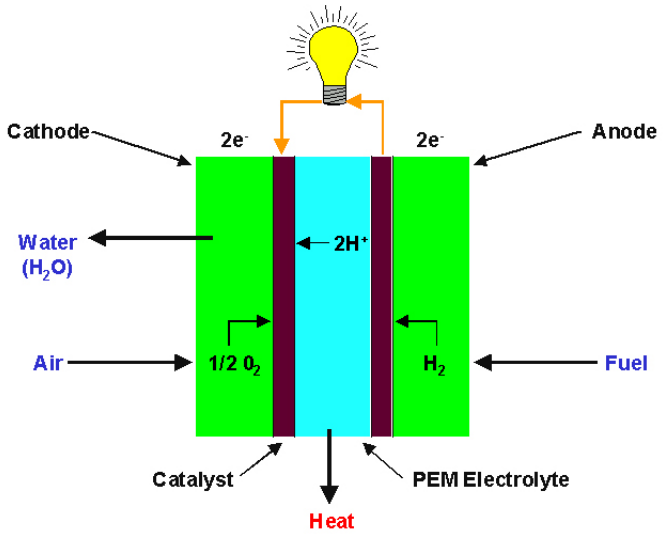

Figure 2.2: Principle of a PEM fuel cell

is built of a thin coat of platinum. The actual chemical reaction can be seen in Equation 2.1. Hydrogen is split on contact with the catalyst into H^+ ions and electrons. The electrons are conducted through the anode where they can be used to power electrical devices, such as a computer.

$$\begin{array}{ll} Anode & 2H_2 = 4H^+ + 4e^- \\ Cathode & O_2 + 4H^+ + 4e^- = 2H_2O \\ NetReaction & 2H_2 + O_2 = 2H_2O \end{array} \tag{2.1}$$

The reaction of a single fuel cell produces about 0.7 volts. Many fuel cells are combined in order to get the voltage to a higher level. The advantage of PEM fuel cells is that they operate on low temperature, which makes them usable for a lot of applications. Fuel cells can be used to power cars and busses which would pollute the air only with water. They can also be used for transportable devices, as can be seen in Figure 2.1.
The fuel cell simulation data set which is used in parts of this thesis is a simulation of such a PEM fuel cell. The actual data set represents only a small part of a real fuel cell. These simulations are used to perform research

about the efficiency of fuel cells and to find weaknesses in current designs. The fuel cell simulation data consists of the following parameters:

- concentration of oxygen O_2,
- concentration of hydrogen H_2,
- pressure p,
- temperature T,
- velocity (vx, vy, vz), and
- concentration of water H_2O.

The actual size of a simulation of these data sets is around ($250 \times 250 \times 1000 \times t$). However, the computation of these simulations is very difficult and time consuming. For the research in this thesis, only a small representation of the actual data set could be used. The size is ($11 \times 13 \times 101$).

Chapter 3

Fundamentals

I would like to use this section to discuss some pre-requisites and to explain basic and fundamental principles related to the topic of the thesis. This chapter is not intended to present previous and related work only, as this will be discussed throughout the thesis. Whenever applicable and known, references to previous or similar work are given.
The motivation behind this chapter is to explain some required ideas and to develop an understanding of how the different topics which are covered by the thesis are linked together by this work.
The main focus in this thesis is on possible techniques for the visualization of huge data sets. The fuel cell data, which was discussed in the previous chapter, is used as an application and example for these methods. What makes this data challenging is that it is a huge and multiparameter data set. Hence techniques have to be explored for fast visualization of volumetric data sets, as well as methods for multiparametric display. Here Chapter 4 explains techniques for fast visualization of huge volumetric data sets, develops a new algorithm, and shows how existing methods can be improved in rendering speed and less memory consumption. Chapter 5 is dedicated to the multiparametric nature of the fuel cell data and shows existing as well as new or not yet explored methods to display more than one data set at once using these methods.
The next sections in this chapter discuss fundamental principles which are needed for the first part of the thesis. In the first section volume rendering is introduced and different rendering methods are compared based on rendering speed and accuracy. Here one can find a general overview of visualization techniques available for volumetric data sets. The focus here is slightly set to direct volume visualization using texture mapping hardware, but this will be discussed in more detail as part of Chapter 4.
The second part pays attention to compression techniques and in particular to wavelets. There is also a very short introduction to signal processing which is of huge interest for both, wavelets as well as for lattice theory.

Wavelets can be used to compress volumetric data sets to make them more manageable for interactive visualizations. In these sections one can find the required math as well as explanations for the basic principles of wavelet and wavelet theory with the focus set to image compression. The sections on signal processing and wavelet theory will be helpful to understand Chapter 4 where a different lattice is introduced to sample the data more efficiently. Also some more advanced rendering techniques for volume data sets are shown and explained. This requires knowledge of the actual rendering pipeline.
For the discussed rendering techniques the three most important qualities are the possible rendering speed, the image quality and the memory consumption of the algorithm. Also some techniques might require special hardware to run which would be a drawback. The speed issue is very important to render huge data sets interactively. For this reason, speed will be the main focus for the thesis. Nevertheless, image quality is important too, but it would be nice to have a method which allows to gradually switch between different Levels-of-Detail.
For the compression not only the final compression ratio is of interest. Also and especially for medical data the resulting artifacts and the image quality are of interest too. Some of the data will be decompressed on the fly, e.g. for time varying data sets, the decompression has to be fast too in order to achieve interactive update rates.
In the end of this chapter a short summary is given which compares the discussed techniques with one another. Also some conclusions are drawn about which methods or algorithms would fit best in the implementation of a fast volume visualization tool for huge data sets.

3.1 Volume Visualization

Volumetric data sets are common in many scientific applications and engineering. For instance, many medical imaging methods, like CT, MRI, or SPECT/PET, can produce three- or four-dimensional data sets. These data sets are usually reconstructed on a slice by slice basis. A common method to visualize this data is to simply display it also on a slice by slice basis. A more intuitive way is to display the whole 3D data set at once to effectively convey information within the volumetric data set. The idea of volume rendering is to put all slices from one volume on top of each other which results in a 3D image stack with the dimensions of the volume. Pixels in 3D are called voxels and they have a spatial influence in all three directions. This volume can now be rendered using different rendering techniques which shall be explained shortly in a little more detail.
Other applications for volume rendering include numerical simulations like the fuel cell data where physical properties like temperature, concentration

or pressure are modelled. Using volume rendering one can easily gain an overview of the entire data set which allows one to more quickly conclude about the efficiency of the simulated fuel cell design. If there are inconsistencies somewhere in the model, they can be easily found in the visualization with the proper transfer function.
Other areas where huge volumetric data sets need to be visualized are geological data sets for oil or resource exploration. Here one is interested in finding particular densities or structures which again can be easily discovered when one is able to look *within*. These structures can then be displayed and analyzed using volume rendering.
This section is divided into several parts, each of them explaining a different method of volume rendering. The first section deals with classification techniques which are very important to highlight specific regions in the data and to mask out uninteresting background information. The section after is not really a part of volume rendering, but shows how to work with volumetric data sets in 2D and how to interpolate oblique slices. The third part shows how to extract contour information for display and the last section finally explains the most commonly used techniques for direct, fuzzy volume rendering.

3.1.1 Classification

Classification is very important for every type of volume rendering and is a critical step in producing meaningful volume rendered images. Most data sets simply represent density information which needs to be interpreted in order to produce expressive images which shall aid in the understanding of the data. The information how to classify the different densities or portions of the volume is used to determine their contribution to the final image. Using these usually pre-defined object material properties in a proper way

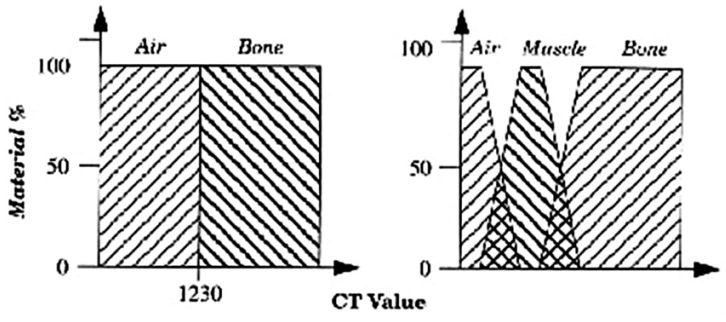

Figure 3.1: Transfer functions

one can easily distinguish between these different objects inside a volume. A big influence has the chosen ray function which shall be discussed later

in this section. In short, classification and therewith transfer functions are responsible for mapping the information a voxel represents into different values, such as material, colour and transparency.
Figure 3.1 shows two simple transfer functions on how to classify different densities in a CT data set. The first transfer function simply uses a predefined threshold and assigns everything below the threshold as air and everything above as bone. A better solution is shown on the right side of the image where another type of tissue (muscle) is introduced and where the transition from one tissue to another is smoother.
The most commonly used transfer functions in volume rendering are for colour and opacity mapping.

Colour

Colour is very important in order to get significant answers from volume rendered data sets. Without colour, it could be hard to look at and interpret these artificial images. For the CT example in Figure 3.1, it would be difficult to distinguish between the different kinds of tissue just by their opacity value, but with a proper colour table assigned to the volume it is easy to differentiate bone from soft tissue and air. Colour transfer functions

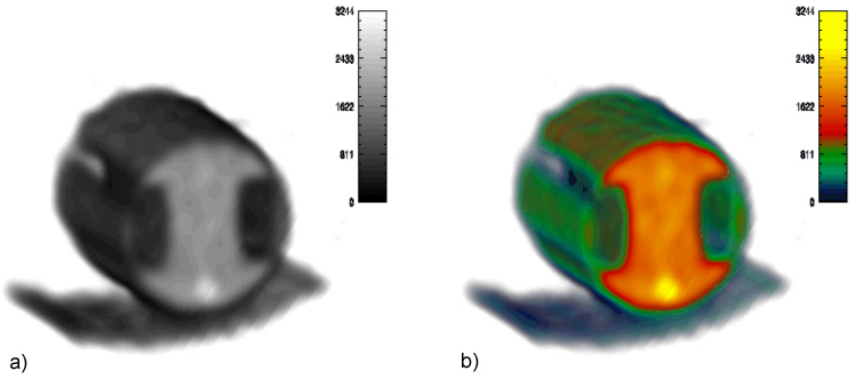

Figure 3.2: The dynamic heart phantom volume rendered without(a) and with an applied colour table(b)

can be defined by using three independent transfer functions which map scalar values into red, green, and blue. Often these three components are pre-processed into one colour lookup table, which specifies the colour for a given gray tone. Care must be taken by the definition of the colour table to avoid hard edges within one class of the volumetric data set. These edges can produce unintended visual artifacts in the rendered image which eventually lead into misinterpretations. Figure 3.2 shows an example of a volume rendered phantom with and without a colour transfer function applied to it. Recent research has shown that there are better solutions than just using

the RGB space for colour mapping. Interactive spectral volumetric rendering techniques [BMDF02] can be applied to reveal the inside of a volume in a way which would not be possible using standard colour mapping techniques. Instead of using three colour components, spectral rendering deals with up to 31 different components. These techniques can be used to visualize tomatoes, bonsai trees and engines in a completely new way [fSSSFU02]. Figure 3.3 shows two images from a frog data set. The image only needs to be rendered once, the light sources can be changed in real-time without the need of rerendering the entire scene.

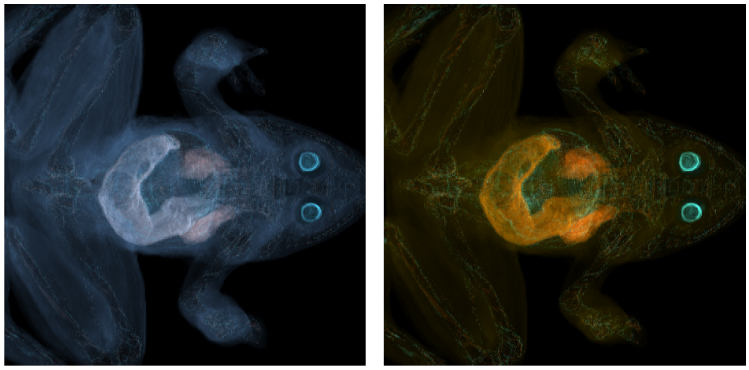

Figure 3.3: Spectral volume rendering with two different light sources

Opacity

Opacity, or better translucency, is used to make the volume transparent such that one can look inside. Without opacity transfer functions a volume rendered image would simply look like an opaque rendered block with texture applied to its sides, or in other words useless. But the fact of being able to look within and change the opacity for each voxel makes volume rendering so powerful and a favoured visualization technique. An example for different opacity transfer functions applied to the same volume can be seen in Figure 3.4.
Opacity transfer functions can also be used to make a pre-selection of the volume-of-interest. In the CT example one can, for instance, apply a transfer function which blends off everything except bone structures. As the result only what is classified as bone would be visible in the final image. Other tissues are simply transparent and are not visible. Classifying a volume based on scalar values alone is often not capable of completely isolating an object of interest. A technique introduced by Levoy [Lev88] adds the gradient magnitude to the specification of a transfer function. Using this technique an object in the volume is specified by a combination of scalar value and gradient magnitude. This is useful to avoid the selection and

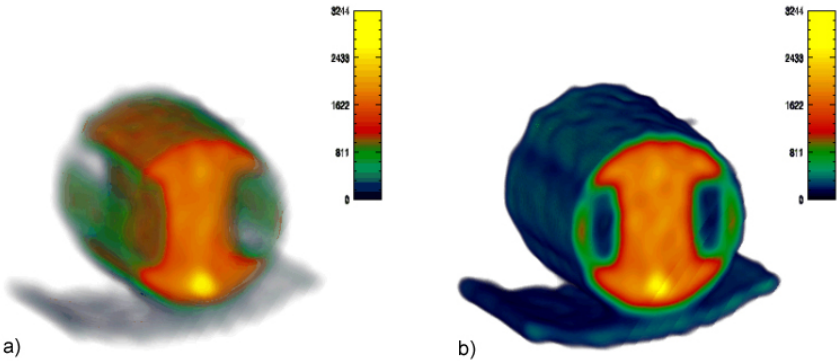

Figure 3.4: The dynamic heart phantom volume rendered with two different opacity tables

eventually enhanced rendering of uninteresting homogeneous regions within a volume and highlighting only regions that change a lot. In a certain way this rendering scheme enhances what the human vision system is most sensitive for: edges.
Figure 3.5 shows a CT scanned foot rendered with this technique. The sharp changes from air to soft tissue and from soft tissue to bone are clearly visible, but the homogeneous regions inside are almost transparent.

Figure 3.5: Volume rendered foot with gradient magnitude transfer function

3.1.2 Volume Slicing

The classic approach to display volumetric data sets is to simply visualize them in 2D slice by slice. This is also the natural way for most of the data sets, because they are often reconstructed from tomographic measurements

in a slice by slice manner. To view the data on a slicing basis is still very common in medical imaging. This can be done with either orthogonal slices which are parallel to a coordinate planes or by using oblique slices. Most physicians are trained to look and interpret such slices for decades. Hence they are very familiar with them and often prefer this viewing scheme instead of 3D volumetric methods. Another reason is that volume rendering still requires a lot of hardware resources to generate *pretty* images at interactive rates.
Figure 3.6 shows a screenshot from a typical medical visualization program displaying the volume data set, in this case from a dSPECT study, using orthogonal volumetric slicing [RMC+00]. Care must be taken when in-

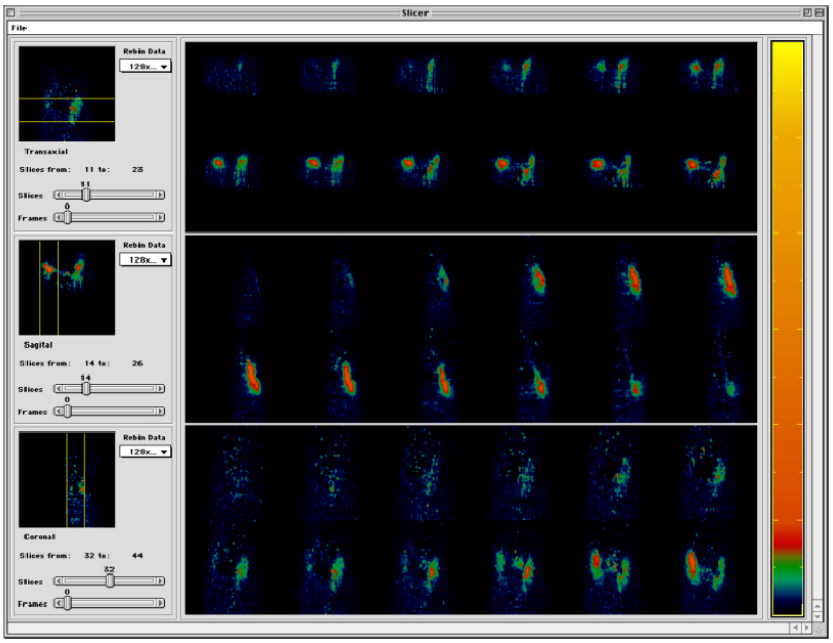

Figure 3.6: Slicing of volume data sets in medical imaging

terpolating oblique slices to not introduce artifacts in the extracted images which also could result in misinterpretations. While direct volume rendering is capable of generating nice looking images with a great context, it is sometimes more efficient to quickly browse through the 2D slices to find the information needed. Depending on the field of application either of these methods might be desirable.

3.1.3 Contouring

Contouring can be thought of as an extension to colour mapping. When we see a surface coloured with data values, the eye often separates similarly

coloured areas into distinct regions. When we contour data, we are effectively constructing the boundary between these regions. These boundaries correspond to contour surfaces of constant scalar value, also called an iso-surface. Examples of iso-surfaces include constant medical image intensity corresponding to body tissues such as skin or bone. The Marching Cubes al-

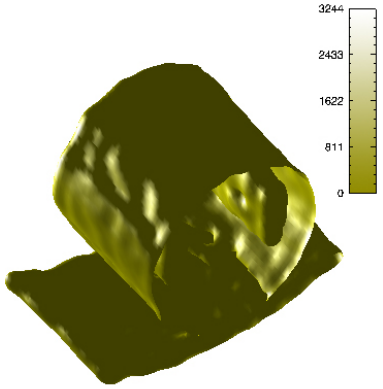

Figure 3.7: The dynamic heart phantom rendered as iso-surface from gray level 25

gorithm by Lorensen et.al. [LC87] is often used to compute such iso-surfaces. It is based on a divide-and-conquer technique which treats each cell independently. The basic assumption is that a contour face can only pass through a cell in a finite number of ways. A case table is constructed that enumerates all possible topological states of a cells given combinations of scalar values at that cell point. The number of topological states depends on the number of cell vertices, usually eight for cubed voxels in standard cubic grid. A vertex is considered inside a contour if it's scalar value is larger then the threshold which is used for the contour surface. Figure 3.7 shows an example where the Marching-Cubes algorithm was used to contour the dynamic phantom data set.

3.1.4 Direct Volume Rendering

Direct volume rendering is the technique which is most often thought as being just volume rendering. Here an image is generated by using the whole or parts of the volumetric data set. Different methods have been developed, where each of them has their own advantages and disadvantages in terms of speed and accuracy. The most common techniques are ray-casting, shear warp, splatting, Fourier domain volume rendering, volume rendering using special hardware and volume rendering using texture mapping hardware. These techniques can be basically divided into image-order and object-order volume rendering.

In an image-order method, rays are cast for each pixel in the image plane through the volume to compute the pixel values, while in an object-order method the volume is traversed, typically in a front-to-back or back-to-front order, with each voxel processed to determine its contribution to the final image.
Volume rendering uses different ray functions to determine the contribution of each voxel to the final image. For example, maximum intensity projection looks for the brightest voxel along the ray and only this one will contribute to the final pixel colour. Another ray function uses a compositing method called alpha blending to determine the pixel colour. It traverses through the volume and adds the contribution of each passed voxel to compute the colour of the pixel. Figure 3.8 shows a volume which was rendered using alpha blending (a) and the maximum intensity projection(b).
Classification must be performed to assign colour and opacity qualities to regions within the volume. Volumetric models can also be defined to support shading which enhances the 3-dimensionality of the data set. Raycasting

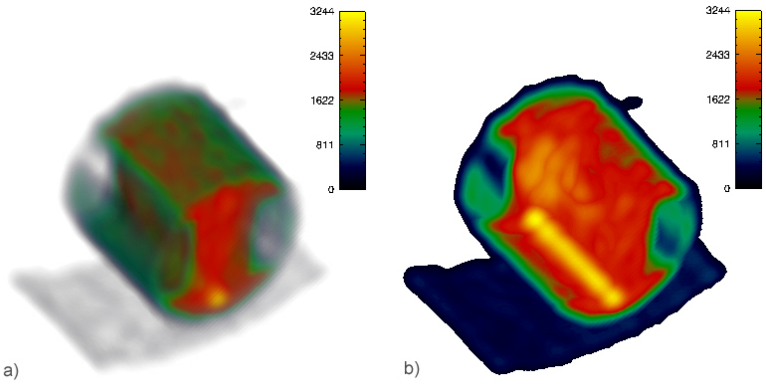

Figure 3.8: The dynamic heart phantom volume rendered with alpha blending(a) and the maximum intensity projection(b)

[Lev88] is the classic volume rendering technique and performed in software. Here rays are cast from a viewpoint through the volume to determine the contribution of each voxel to the final image. Different ray functions can be used in order to enhance certain parts of the volume. The simplest one is alpha blending which generates semi-transparent volumes. The maximum intensity projection only evaluates the *brightest* voxel, while x-ray images are produced by building the average of all voxel which where visited. Improvements to the rendering speed can be made by early ray termination where the ray is not further traced when a certain opacity, often a value around 0.95 is reached. Ray casting generates very good looking and clear images, but the software implementation is one of its biggest drawbacks. A hardware implementation of this algorithm is available as PCI card for the

PC (VoluemPro) [PHK+99]. This card is available with up to 512 MB of *texture memory* and allows fast rendering of volumes that fit into the available memory space.
The shear-warp algorithm [Lac95] is an extension and improvement to the ray casting algorithm. Because it is difficult and computationally expensive to trace a ray through a volume arbitrary this algorithm first shears the volume according to the current rotation, evaluates all rays in parallel in the sheared volume and warps the final image in order to remove the distortions. The overhead of shearing the volume and warping the images is smaller than the gain in speed through evaluating the parallel rays. The resulting images are a little bit blurred due to the additional warping operation.
Splatting [Wes90] is a fast volume rendering techniques where each voxel is represented by a footprint which is then splatted onto the viewing plane. The reconstruction kernel of all splats are computed in pre-processing which saves time while rendering the images. Other optimizations are that only those voxels are splatted which are above a given threshold. Other improvements were made by using commonly available OpenGL graphics hardware to splat the voxels more efficiently [KM01]. Images created using the splatting algorithm usually look a bit blurred due to the splatting technique.
Figure 3.9 [MHB+00] shows some images which were generated using the most often used volume rendering techniques. Even though it is a little outdated and some techniques have improved a lot, it shows that some differences are noticeable in the image quality. The fastest known volume ren-

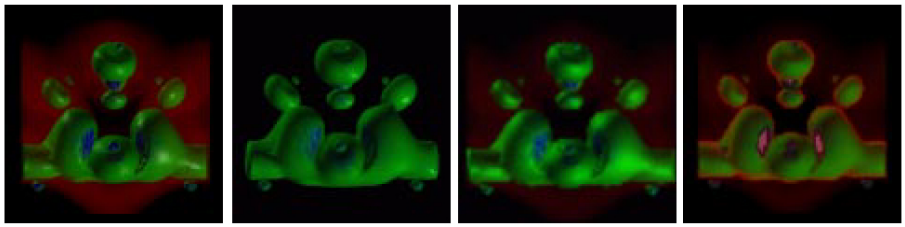

Figure 3.9: Comparison of raycasting (a), splatting(b), shear-warp(c) and texture mapping(d)

dering technique is the Fourier domain volume rendering [TL93]. Here, the entire volume is transformed into the frequency domain in a pre-processing step. Using the Fourier projection slicing theorem [Lev92] allows to extract slices out of the 3D Fourier volume which are then re-transformed into spatial domain to generate the output image. Because one only needs to extract a 2D slice out of the 3D Fourier volume, this algorithm has a complexity of $O(N^2 log(N))$ versus all other algorithms, which have a complexity of $O(N^3)$. One drawback is that this algorithm only allows one to create X-Ray looking images, but recent research shows that efficient shading is possible using spherical harmonics [ESMM02]. Other improvements are

depth cueing with aid in the lack of depth perception, but occlusion is still a problem within Fourier based volume rendering.
The volume rendering technique which was used for the implementation of this thesis is volume rendering using texture mapping hardware [CCF94]. Here either 2D textures and object-oriented slices or 3D textures and viewport-aligned slices are used to resample the volume to evaluate the rendering integral. The volume data set is transferred to the available texture memory and then used as texture with alpha-blending enabled. This texture can be sliced either object-aligned(2D) or viewport-aligned(3D), depending on the available hardware capabilities. Usually 3D textures give better results, but some sampling artifacts still occur. Volume rendering using texture mapping hardware is very fast and the image quality has improved enormously over the last years. This is due to the availability on common consumer graphics hardware since a few years. Chapter 4 has a more detailed discussion of this technique and Chapter 7.3 shows additionally some implementation details.

3.2 Signal Theory

Everything related to imaging, image processing or sampling can be studied via signal processing. A signal is simply a function that carries some information. This signal can be as simple as a 1-dimensional straight line and as complex as a n-dimensional mathematical function. Usually a signal changes over some set of spatiotemporal dimensions.
A simple 1-dimensional example of a signal is $f(t)$ which represents a function that is changing over time. A real world example would be an audio signal as a collection of various tones of different audible frequencies that vary over time. Then the signal is the amplitude of each frequency at each moment in time, or a time-varying signal.
Other signals can also vary over space like an image or video. Here, the signal is simply $f(x, y)$ and varying over the two spatial dimensions x and y. For monochromatic images the signal represents the amount of light at the position (x, y). RGB or RGBA images consist of three or four dependant channels, each representing either the red, green, blue or the alpha signal.
For the application of 3D volume rendering we use 3- or 4-dimensional data sets. In 3D, it is just the extension of the 2D image with an additional spatial dimension $f(x, y, z)$, and when also varying over time or another parameter with one spatial and one temporal dimension $f(x, y, z, t)$.
Signals can have very different quantities. Audio signals have an amplitude at each time step, images or volumes have an intensity at each position. Medical images can represent the attenuation of x-rays or the hydrogen density. Other signals can be seen as vectors, like RGBA for the four different channels in an RGBA image or volume.
Signals are most often continuous which have to be discretized and in order

to work with them digitally in a computer. Also the (natural-)continuous signal range needs to be discretized because of the limited precision and storage capacity of computers. All samples are stored with finite precision from infinite precision in the continuous domain. The discrete representation of a continuous signal will generally introduce some artifacts, aliasing, in to the data.
The accuracy of the digital representation depends on two qualities; sampling frequency and the number of bits used for the quantization. The spacing of discrete values from a continuous signal is called sampling a signal at discrete locations. The sample frequency describes how often the signal will be sampled and hence how well the sampled signal can approximate the original, continuous version. The spacing of discrete values in the range of a signal is called the quantization and describes how many different possible values the sampled signal can represent. Even though, sampling and quantization are independent from each other, both play a significant role in how much the sampled signal deviates from the original continuous signal. The quantization defines the precision of the sampled signal, while the sampling frequency controls the temporal or spatial accuracy of the discrete signal.
The Nyquist-Shannon sampling theorem [Sha49], a fundamental theorem of digital signal processing, states that a digital signal can not unambiguously represent signal components with frequencies equal or above half the sampling frequency. This frequency is called the Nyquist frequency. Frequencies above the Nyquist frequency can be observed in the discretized signal, but their frequency is ambiguous. That means, a frequency component with frequency f cannot be distinguished from another component with frequency $2f$ and other harmonic frequencies if f is larger than the Nyquist frequency. To avoid this problem, most analog signals are low-pass filtered by the Nyquist frequency before converting to the digital representation. If the sampling frequency is less than this limit, then frequencies in the original signal that are above half the sampling rate will be aliased and will appear in the resulting signal as lower frequencies. Therefore, an analog low-pass filter is typically applied before sampling to ensure that no components with frequencies greater than half the sample frequency remain. This is called *anti-aliasing*.
Signal processing and signal theory are very important for understanding wavelets and lattices. Wavelets are discussed briefly later in this Chapter while lattice theory, especially with focus on closest sphere packings, is discussed and explained in Chapter 4.1.

3.3 Compression

One challenging problem in computer science is the amount of data which needs to be processed, analyzed and stored. Larger data sets require more

memory and processing power to analyze the data sets and to extract some information. The visualization of huge data sets, which can be built by simulation or scanning with today's technology, is still a difficult task.
To be able to handle huge data sets, which can exceed the scale of giga- or terabytes, one needs solutions which allow one to store the data in a more efficient way. This section explains some basic compression techniques for the wavelet transform. Wavelets are a good tool for compression as they can be used to locally decorrelate the signal into low and high frequencies bands and this information can be further used to select the level of compression and to encode this data in a better, more efficient way. Other advantages are the good image quality and the high compression ratios which can be achieved. Compression, and compression using wavelets in particular, work

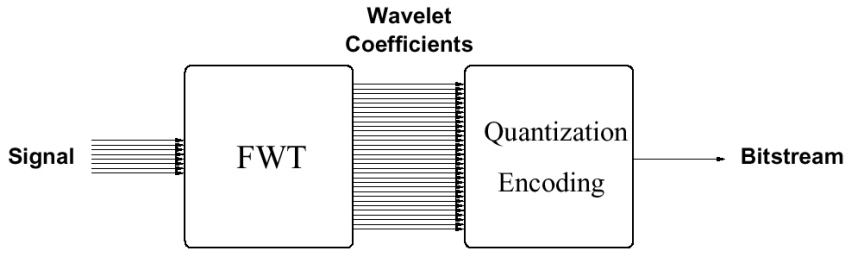

Figure 3.10: Principle of wavelet compression

such that the signal is decomposed using the fast forward wavelet transform into low and high frequencies. The high frequencies are the detail information which is needed to reconstruct the original signal from the low pass version. One quality of these detail coefficients is that they can be stored more efficiently using a bitstream encoding techniques, like run length encoding. Figure 3.10 shows the principle of compression using wavelets. An incoming signal is decomposed using the fast wavelet transform (FWT) and the resulting detail information is quantized and encoded into a bitstream. The step where the data is really *compressed* is called quantization. Using some thresholds, one can define how many details are needed. Another very helpful side effect from using wavelets is that while one transforms the data set one is also building a multiresolution version of the data, which can be exploited for Level-of-Detail rendering.
In the next section first the run length encoding scheme is introduced which is used to encode the high frequencies. After this two wavelet decomposition techniques are discussed. The classic approach on the example for Haar wavelets and a newer algorithm, called Lifting Scheme which allows a integer to integer wavelet decomposition.

3.3.1 RLEncoding

Run-Length-Encoding (RLE) is a simple and straightforward technique to reduce the amount of memory it takes to store repeatative data. The basic principle is that it uses stings to encode the runs of the same character. A typical data string can be seen in Example 3.3.1:

abcddddddcbbbbabcdef

Example 3.1: Data sequence

The example shows a string of a length of 20 characters composed out of *a* through *f*. Each character occupies one byte of memory which makes 20 bytes for the whole string. However, there are two parts in the string which are build out of one character only, namely *b* and *d*. These runs can be stored using two bytes only. The first indicates how many letters follow and the second one shows which letter was used. The data sequence from Example 3.3.1 can be stored using 14 bytes only as can be seen in Example 3.3.1:

abc6dc4babcdef

Example 3.2: RLEncoded sequence I

The example above is very simple and limited. It is not possible to encode strings with numbers because one could never tell which number indicates a run length and which is a literal. Another problem is that if one would encode using this scheme, also the runs of one would be encoded by two bytes. This could easily lead to even bigger data sets not speaking of compression. The difficulty is to tell when a run starts and when a literal sequence begins. A common approach is to use only 7 out of 8 bits to indicate the run length. If the length is positive then it shows that the following byte is repeated that many times. If it is negative, then it indicates a literal sequence which is as long as the negative number. The data sequence from Example 3.3.1 was encoded using this principle and is shown in Example 3.3.1. To store this sequence 17 bytes are needed.

-3abc6d-1c4b6abcdef

Example 3.3: RLEncoded sequence II

In this example we saved only 3 bytes, but as the frequency and length of the repeating characters increases the compression ratio gets better and better. In the worst case, see the left image in Figure 3.11, RLE will not compress the data, instead it will produce a bigger file. Every 127 bytes an additional byte will be inserted to indicate a literal sequence.
In the best case, however, 128 bytes can be compressed to two bytes only, resulting in a compression ratio of 64. An example can be seen in the right image of Figure 3.11. For these reasons, RLE is most often used on grayscale

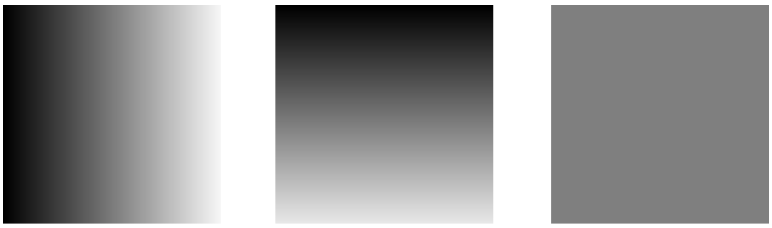

Figure 3.11: RLE Example

or black and white images where long runs are more likely than in high colour images such as photographs where usually every pixel differs from the last one.
The three images in Figure 3.11 demonstrate the extreme cases for the RLE scheme. The size of all of these images is 100×100. The uncompressed image is 10000 bytes. The left image shows the worst case scenario, where every pixel differs from the one before. Here, every 127 bytes an additional byte is inserted which results in a *compressed* size of 10100 bytes. The image in the middle, however, shows a better compression result. It has long runs and can be compressed to 5317 bytes. The last image shows the best case where 128 bytes can be compressed to two only. This results in a compression ratio of $64 : 1$ and the whole image needs only 200 bytes to store now.
To avoid the worst case scenario, the algorithm could run in the horizontal and vertical direction first and then the decides which results in the best compression result. The additional information of which compression was used could be stored in a one bit header, two bit for volumes.

3.3.2 Wavelets

Wavelets are a mathematical tool to hierarchically decompose signals and functions. Using wavelets allows one to describe a function in terms of a coarse resolution, plus detail information ranging from broad to narrow. Wavelets can be applied to images, curves [FS94] or surfaces [GC95] offering an elegant technique for representing the levels of detail as needed.
Wavelets have their roots in approximation theory and signal processing [Dau88], but they have been also applied to many problems in computer science. Applications in computer graphics include image editing [BBS94], image compression [DJL92], and image querying [JFS95].
The next section starts with the simplest form of wavelets, the Haar basis. The following sections are used to explain the one-dimensional wavelet transform and basis functions for the 1D Haar wavelet and how it can be used to compress discrete 1-dimensional functions. After this, a more generalized case is discussed and it is shown how the algorithm can be extended to work with n-dimensional functions.

1D Haar Wavelet

The Haar basis is the simplest wavelet basis and a good choice for an introduction in wavelet theory. In this section it is first shown how a discrete function can be decomposed using simple Haar wavelets. Later this decomposition will be described using the proper basis functions for this wavelet type. In the end of this section it will be shown how this can be further utilized for a straightforward compression technique.
To demonstrate the wavelet decomposition and reconstruction, an accompanying example will be used throughout this section. Example 3.3.2 shows a typical 1-dimensional function with four pixel values:

$$[\,1,\ 7,\ 4,\ 0\,]$$

Example 3.4: 1D image

This simple "image" can be represented in the Haar basis by computing the wavelet transform. All pixels will be averaged pairwise to yield the next lower resolution image. The missing information can be stored as detail coefficients which are used later to restore the original image. Lossless wavelet compression is only possible when using the integer wavelet transform. Even if one would keep all the detail coefficients it would not be possible to have a really lossless compression because of the limited precision of computers. Example 3.3.2 shows the complete decomposition of Example 3.3.2.

Resolution	Averages	Detail
1	$[\,1,\ 7,\ 4,\ 0\,]$	
2	$[\,4,\ 2\,]$	$[\,-3,\ 2\,]$
3	$[\,3\,]$	$[\,1\,]$

Example 3.5: Wavelet decomposition

The complete wavelet transform for the 1-dimensional Haar basis of Example 3.4 is:

$$[\,3\ 1\ -3\ 2\,]$$

Example 3.6: Wavelet transform

To recursively restore the original image one would start with the lowest resolution level and compute: $[\,3+1,\ 3-1\,]$ which yield the next higher resolution $[\,4,\ 2\,]$.
To retrieve the original image and to compute the highest resolution one would also include the next higher detail coefficients and compute: $[\,4+(-3),\ 4-(-3),\ 2+2,\ 2-2\,]$. And finally one would have recovered the original image.
The way the wavelet transform was computed, by recursively averaging and differencing detail coefficients, is called a filter bank. A filter bank is a set

of analysis and synthesis filters to decorrelate and reconstruct the signal. If the filter bank is appropriate, one can recurse and built a multiresolution pyramid. No information has been gained or lost in this process. The original image had four coefficients, and so does the transform. Also, given the transform, one can reconstruct the image to any resolution by recursively adding and subtracting the detail coefficients from the next lower resolution. Storing the image as wavelet transform, rather than the image itself, has a number of advantages. For instance, a lot of detail coefficients are usually close to zero. Truncating these details introduces only small errors in the reconstructed image, depending on the used threshold. This results in lossy image compression, which is one of the major applications for wavelets.
Compression using wavelets usually results in better image quality and compression ratios than standard JPEG which operates on a local basis. Whereas the compression using wavelets works global where a given threshold is used to truncate detail information from all points. One drawback is that everything remains in the floating point domain if one is not willing to introduce artifacts due to rounding. Smooth wavelets avoid the jagged artifacts common in DCT JPEG.

Basis Functions In the last section the principles and the idea behind wavelets where explained using a simple discrete function. This section is used to describe the theory and the mathematics behind wavelet theory. Instead of thinking of images one can also visualize them as piecewise-constant functions on the half-open interval $[0, 1)$. A simple one-pixel image can be seen as a function which is constant over the entire interval $[0, 1)$. Here V^0 is defined as the vector space for all these functions. A two-pixel image has two constant pieces over the two intervals ranging from $[0, \frac{1}{2})$ and $[\frac{1}{2}, 1)$. The vector space containing these two functions is called V^1. If one would continue using this scheme, the space V^j will include all piecewise-constant functions defined on the interval $[0, 1)$ with constant pieces over each of the 2^j equal subintervals.
Every one-dimensional image with 2^j pixels can be seen as an element, or vector, in V^j. Because all functions are defined over the unit interval, every vector in V^j is also contained in V^{j+1}. This can easily be seen if one would describe a piecewise-constant function with two intervals as a piecewise-constant function with four intervals. Now each interval in the first function corresponds to a pair of intervals in the second function. The vector spaces V^j are nested, that means,

$$V^0 \subset V^1 \subset V^2 \subset ... \subset V^j. \tag{3.1}$$

The mathematical theory of multiresolution analysis requires this nested set of spaces V^j. What remains is to define a basis for each vector space V^j. The basis functions for the spaces V^j are called scaling functions, and are

denoted by the symbol ϕ. A simple basis for V^j is given by the set of scaled and translated *box* functions (Haar basis):

$$\phi_i^j(x) := \phi(2^j x - i), \qquad i = 0, ..., 2^j - 1, \tag{3.2}$$

where

$$\phi(x) := \begin{cases} 1 & \text{for } 0 \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

Now an inner product has to be defined for the vector spaces V^j. The *standard* inner product,

$$\langle f \mid g \rangle := \int_0^1 f(x)g(x)dx, \tag{3.3}$$

for the two elements $f, g \in V^j$ would serve well for the decomposition of Example 3.3.2. Now a new vector space W^j can be defined as the orthogonal complement of V^j in V^{j+1}. Here, W^j will be the space of all functions in V^{j+1} that are orthogonal to all functions in V^j under the chosen inner product defined by Equation 3.3. Informally, one can think of the wavelets in W^j as a means for representing the parts of a function in V^{j+1} that cannot be represented in V^j. A collection of linearly independent functions $\psi_i^j(x)$ that are spanning W^j are called wavelets. These basis functions have two important properties:

1. The basis functions ψ_i^j of W^j, together with the basis functions ϕ_i^j of V^j, form a basis for V^{j+1}.
2. Every basis function ψ_i^j of W^j is orthogonal to every basis function ϕ_i^j of V^j under the chosen inner product.

The detail coefficients which were introduced earlier are in fact coefficients of the wavelet basis functions. The wavelets corresponding to the box basis are known as the Haar wavelets, which are given by:

$$\psi_i^j(x) := \psi(2^j x - i), \qquad i = 0, ..., j^j - 1, \tag{3.4}$$

where

$$\psi(x) := \begin{cases} 1 & \text{for } 0 \leq x < \frac{1}{2} \\ -1 & \text{for } \frac{1}{2} \leq x < 1 \\ 0 & \text{otherwise.} \end{cases}$$

Now these ideas can be applied to the "image" from Example 3.3.2. The original image $\mathcal{I}(x)$ can be expressed as a linear combination of the box functions in V^2:

$$\mathcal{I}(x) = c_0^2\phi_0^2(x) \; + \; c_1^2\phi_1^2(x) \; + \; c_2^2\phi_2^2(x) \; + \; c_3^2\phi_3^2(x). \tag{3.5}$$

The coefficients $c_0^2, ..., c_3^2$ are simply the four original pixel values [1 7 4 0]. In terms of basis functions in V^1 and W^1, $\mathcal{I}(x)$ can be rewritten using pairwise averaging and differencing:

$$\mathcal{I}(x) = c_0^1\phi_0^1(x) \; + \; c_1^1\phi_1^1(x) \; + \; d_0^1\psi_0^1(x) \; + \; d_1^1\psi_1^1(x). \tag{3.6}$$

And finally, $\mathcal{I}(x)$ can be expressed as a sum of basis functions in V^0, W^0, and W^1:

$$\mathcal{I}(x) = c_0^0\phi_0^0(x) \; + \; d_d^0\psi_0^0(x) \; + \; d_0^1\psi_0^1(x) \; + \; d_1^1\psi_1^1(x). \tag{3.7}$$

The four coefficients of Equation 3.7 are the Haar wavelet transform of the original image in Example 3.3.2 for the basis V^2.
The Haar basis possesses an important property known as orthogonality, which is not always shared by other wavelet bases. An orthogonal basis is one in which all of the basis functions are orthogonal to one another. Orthogonal filters are good since the reconstruction filters can be obtained easily by transposing the decomposition matrix. This is explained later in this section. Besides all the advantages of separable filters, they tend to have a favoured direction which can also be seen in the low pass versions of the signal. The solution are non-separable filters which are more difficult to find and to apply.
Another important quality is the normalization step. A basis function $u(x)$ is normalized if $\langle u \mid u\rangle = 1$ is true. The Haar basis can be normalized by replacing the earlier definitions of $\phi_i^j(x)$ and $\psi_i^j(x)$ with:

$$\phi_i^j(x) := 2^{\frac{j}{2}}\phi(2^j x - i) \tag{3.8}$$

$$\psi_i^j(x) := 2^{\frac{j}{2}}\psi(2^j x - i); \tag{3.9}$$

where the constant factor of $2^{\frac{j}{2}}$ is chosen to satisfy $\langle u \mid u\rangle = 1$ for the standard inner product. With these modified definitions, the new normalized coefficients are obtained by multiplying each old coefficient with superscript j by $2^{\frac{-j}{2}}$.
For the Example 3.3.2, the unnormalized coefficients $[\,1\,7\,4\,0\,]$ become the normalized coefficients $[\,1\,7\,\frac{4}{\sqrt{2}}\,0\,]$.

Compression The idea of compression is to replace an initial data set $f(x)$ using some techniques or algorithms with a smaller one, e.g. $\tilde{f}(x)$. In general, there are two different forms of compression, without any loss of information and lossy compression. A function $f(x)$ can be expressed as a weighted sum of basis functions $u_1(x), ..., u_m(x)$:

$$f(x) = \sum_{i=1}^{m} c_i u_i(x). \tag{3.10}$$

The data set in this case consists of the coefficients $c_1, ..., c_m$. In terms of compression, one would like to find another function which approximates $f(x)$, but which uses fewer samples than $f(x)$. With a pre-defined error

tolerance ϵ (lossless compression $\epsilon = 0$) the approximating function can be described as:

$$\tilde{f}(x) = \sum_{i=1}^{\tilde{m}} \tilde{c}_i \tilde{u}_i(x). \tag{3.11}$$

with $\tilde{m} < m$ and $\|f(x) - \tilde{f}(x)\| \leq \epsilon$. In general one would attempt to construct a set of basis functions $\tilde{u}_1, ..., \tilde{u}_m$ that would provide a good approximation of $f(x)$ with simply fewer coefficients.
One problem is the ordering of the coefficients $c_1, ..., c_m$ which has to be done in a way that for every $\tilde{m} < m$, the first $\tilde{m}$ elements of the sequence give the best approximation of $\tilde{f}(x)$ to $f(x)$ by using the L^2 norm. The solution is easy if the used basis is orthonormal as it is for the normalized Haar basis. Let σ be a permutation of $1, ..., m$, and let $\tilde{f}(x)$ be a function which uses the coefficients corresponding to the first $\tilde{m}$ numbers of the permutation σ:

$$\tilde{f}(x) = \sum_{i=1}^{\tilde{m}} c_{\sigma(i)} u_{\sigma(i)}. \tag{3.12}$$

Then the square of the L^2 error in this approximation is:

$$\begin{aligned} \left\| f(x) - \tilde{f}(x) \right\|_2^2 &= \langle\, f(x) - \tilde{f}(x) \mid f(x) - \tilde{f}(x) \,\rangle \\ &= \left\langle \textstyle\sum_{i=\tilde{m}+1}^{m} c_{\sigma(i)} u_{\sigma(i)} \;\middle|\; \sum_{j=\tilde{m}+1}^{m} c_{\sigma(j)} u_{\sigma(j)} \right\rangle \\ &= \textstyle\sum_{i=\tilde{m}+1}^{m} \sum_{j=\tilde{m}+1}^{m} c_{\sigma(i)} c_{\sigma(j)} \langle\, u_{\sigma(i)} \mid c_{\sigma(j)} \,\rangle \\ &= \textstyle\sum_{i=\tilde{m}+1}^{m} (c_{\sigma(i)})^2 \end{aligned} \tag{3.13}$$

The error that is introduced is $\langle u_i \mid u_j \rangle = \delta_{ij}$. If the basis is orthonormal one can conclude that, in order to minimize the error for any given $\tilde{m}$, one needs to find a σ that satisfies $\mid c_{\sigma(1)} \mid \geq ... \geq \mid c_{\sigma(m)} \mid$. The best choice for σ is the permutation that sorts all the coefficients in order of decreasing magnitude.
In Figure 3.12 [SDS96] a one-dimensional function was compressed using the L^2 Haar wavelets. One can see the different compression levels which were achieved by truncating the coefficients with the smallest value. Several different methods of thresholding exist. When using hard thresholding, the coefficients are simply set to zero based on a user or algorithm defined cut-off value:

$$\tilde{c}_i = \begin{cases} 0 & \text{for } c_i < t \\ c_i & \text{for } c_i \geq t \end{cases} \tag{3.14}$$

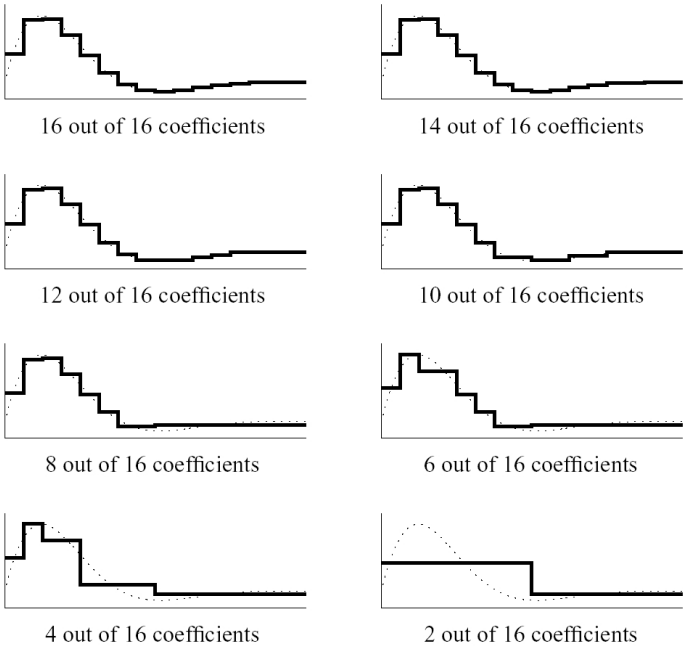

Figure 3.12: Wavelet compression using different amounts of detail information

Soft thresholding shrinks all the detail coefficients towards zero for a given threshold. This method is often used in biomedical imaging and used for noise reduction in MRI data sets. The method can be described as:

$$\tilde{c}_i = \begin{cases} c_i - t & \text{for } c_i \geq t \\ 0 & \text{for } |c_i| \leq t \\ c_i + t & \text{for } c_i \leq -t \end{cases} \tag{3.15}$$

Quantile thresholding simply discardes as many detail coefficients as is stated by the given percentage. The rule for quantile thresholding is:

$$\tilde{c}_i = \begin{cases} 0 & \text{for } c_i < p \\ c_i & \text{for } c_i \geq p \end{cases} \tag{3.16}$$

Here p is the p-quantile of all wavelet coefficients.

Extension to nD

In preparation for volume compression, we need to generalize Haar wavelets to n dimensions. Here only the extension to two dimensions is shown, but also explained how it can be further applied for the use in n dimensions. Again, this in only for the linear-separable Haar basis which works in one dimension at a time. These kind of filters can introduce some sampling artifacts due to a preferred directions. Better results can be achieved by using

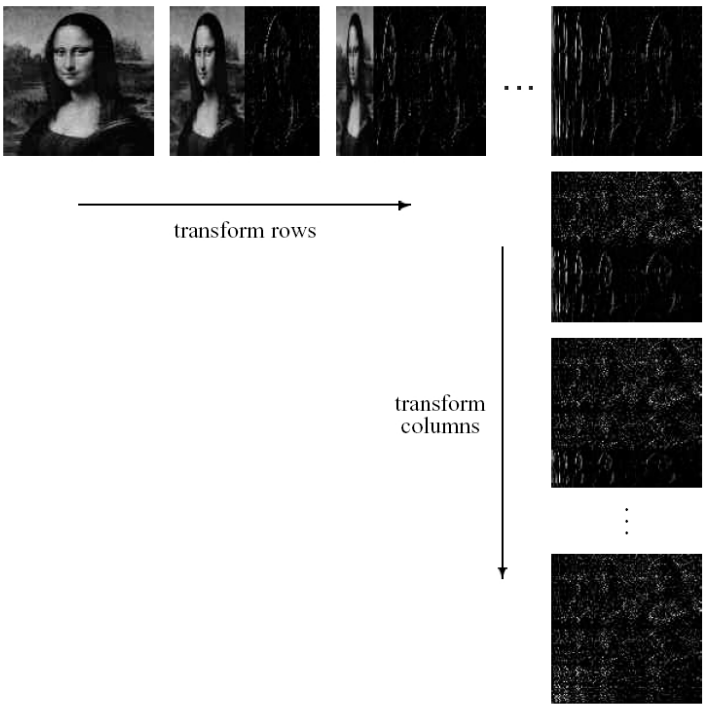

Figure 3.13: Standard wavelet decomposition
of a 2-dimensional image

non-separable filters. There are basically two ways to use linear-separable wavelets in order to transform the pixel values within a 2D image. Each one is a generalization to two dimensions of the 1-dimensional wavelet transform described in the previous sections. The standard decomposition of an image, applies the 1-dimensional wavelet transform to each row of the image. This operation results in an average value for each row with detail coefficients for each row. In the next step, each of these transformed rows is treated as if they were themselves an image and another 1-dimensional wavelet transform is applied to each column. The resulting values are all detail coefficients except for a single overall average coefficient. Figure 3.13 [SDS96] demonstrates the principle of standard decomposition.
The second two-dimensional wavelet transform is called the non-standard decomposition. Here the algorithm alternates between operations on rows and columns. First, one step of horizontal pairwise averaging and differencing on the pixel values is performed for each row of the image. Next, vertical pairwise averaging and differencing on each column is performed on the results from the previous step. To complete the transformation, this process is recursively repeated only on the quadrant containing averages in both directions. Figure 3.14 [SDS96] shows an example of non-standard decom-

position. These two methods of decomposing a 2-dimensional image yield

Figure 3.14: Non-standard wavelet decomposition of a 2-dimensional image

to coefficients that correspond to two different sets of basis functions. The standard decomposition of a 1-dimensional wavelet basis consists of all possible tensor products of 1-dimensional basis functions. When one starts with the 1-dimensional Haar basis for V^2, it would result in the 2-dimensional basis for V^2. If the standard construction to an orthonormal basis is applied in one dimension, the result is an orthonormal basis in two dimensions.
The nonstandard construction of a 2-dimensional basis proceeds by first defining a 2-dimensional scaling function:

$$\phi\phi(x,y) := \phi(x)\phi(y) \tag{3.17}$$

and three wavelet functions,

$$\begin{aligned} \phi\psi(x,y) &:= \phi(x)\psi(y) \\ \psi\phi(x,y) &:= \psi(x)\phi(y) \\ \psi\psi(x,y) &:= \psi(x)\psi(y). \end{aligned} \tag{3.18}$$

The levels of scaling are now denoted with a superscript j and horizontal and vertical translations with a pair of subscripts k and l. The non-standard basis consists of a single coarse scaling function:

$$\phi\phi^0_{0,0}(x,y) := 2^j \phi\phi(x,y) \tag{3.19}$$

along with scales and translates of the three wavelet functions $\phi\psi$, $\psi\phi$, and $\psi\psi$:

$$\begin{aligned} \phi\psi^j_{kl}(x,y) &:= 2^j \phi\psi(2^j x - k, 2^k y - l) \\ \psi\phi^j_{kl}(x,y) &:= 2^j \psi\phi(2^j x - k, 2^k y - l) \\ \psi\psi^j_{kl}(x,y) &:= 2^j \psi\psi(2^j x - k, 2^k y - l). \end{aligned} \tag{3.20}$$

The constant 2^j normalizes the wavelets to give an orthonormal basis.
Both, the standard as well as the non-standard wavelet decomposition can be extended to n dimensions by either computing the transform for each dimension separately or be alternating between them. Here the standard decomposition is easier to use, because it simply requires to perform a 1-dimensional transform for each dimension before the next one is computed. On the other hand, it is more efficient to compute the non-standard decomposition. For an $m \times, m$ image the standard decomposition needs $4(m^2 - m)$ operations while the non-standard decomposition requires only $\frac{8}{3}(m^2 - 1)$.
In order to be able to additionally use the wavelet transformed data in a multiresolution environment, like level of detail volume rendering, the non-standard wavelet decomposition has to be used. This transforms all dimensions for each resolution level which yields a factor of 2 for the downsampled image or volume.
Another specification for a wavelet transform is the support of each basis function. This means the portion of each functions domain where the function is non-zero. All non-standard Haar basis functions have square support, while some standard basis functions have non-square support. Depending upon the application, one of these choices may be preferable to the other.
Because of finite floating point precision, all these methods are not able to lossless restore an original signal from its low resolution counterparts. This is one major disadvantage of standard wavelets. Another disadvantage is that if one does not want to additionally introduce artifacts due to rounding of the low and high pass parts from floating point into the integer domain, the lower resolution representation as well as the detail coefficients have to be stored as floating point numbers. This is a little bit silly if the original data was given in the integer domain like most images and volumes are. The next section discusses a different technique for wavelet filtering which allows to avoid these problems.

3.3.3 Lifting Scheme

The Lifting Scheme is a very efficient implementation of the wavelet transform which does not rely on the existence of the Fourier transform for the given problem [Swe95]. Wavelets which are generated using lifting are called second-generation wavelets in contrast to the standard wavelets which are called first-generation. Second-generation wavelets are more general and are able to represent all classic first-generation wavelets which have a biorthogonal basis. In fact, all wavelets which can be found using the Cohen-Daubechies-Feauveau machinery [CDF92] can also be constructed using the lifting scheme. The decomposition of a classic wavelet filter into the lifting scheme can be obtained by using the Euclidian Algorithm [DS98]. Also, second generation wavelets are not necessarily translates and dilates of one function and can also be applied to problems where a Fourier transform can not be used as a construction tool, like curved surfaces or on irregular grids [SS95].
Another advantage of the lifting scheme is that it can be easily extended to perform a pure integer to integer wavelet transform. It allows a real lossless wavelet decomposition by avoiding the existing rounding problems. Data which is given in the domain of bytes or shorts now does not need to be transformed into four byte float values. This way huge amounts of storage space can be saved. Also the inverse transform is always easy to find, as it is as simple as reverting the order of operations and switching $+$ and $-$. Other benefits are that it allows a faster implementation, still $O(n)$, by using similarities between both, the high and the low pass filters. It also allows an in-place calculation of the transform which saves additional memory costs.
The next sections explain the principle of lifting and discuss how the forward transform works and how the inverse transform can be found. It also demonstrates how it can be used to perform an integer to integer wavelet transform.

Dual and Primal Lifting

Wavelets are used to decorrelate a signal into different multiresolution levels. At each resolution step, the function is split in the low and high frequencies. These high and low pass parts of the signal are obtained by applying wavelet filters to the signal. The lifting scheme is a very efficient implementation of these filter operations. The transform using the lifting scheme can be split into three parts:

- splitting the signal using the lazy wavelet in the odd and even samples,
- then predicting the even samples from the odd samples and storing the prediction error as the even samples(dual lifting), and

- updating the odd samples to maintain the bias of the low frequency part (primal lifting).

Formally this would be represented as follows:

- split $\lambda_{j+1} \rightarrow (\lambda_j, \gamma_j)$,
- dual lifting $\gamma_j - P(\lambda_j) \rightarrow \gamma_j$, and
- primal lifting $\lambda_j + U(\gamma_j) \rightarrow \lambda_j$.

Here, γ_j and λ_j are the odd and even samples of the original signal for the current wavelet level j. To continue with Example 3.3.2:

$$\lambda_{j+1} = [\,1,\, 7,\, 4,\, 0\,]$$

Example 3.7: Lifting scheme

represents a one-dimensional discrete signal of length 4. This signal can now be split using the lazy wavelet into two parts:

$$\lambda_j = [\,1,\, 4\,]\, \gamma_j = [\,7,\, 0\,]$$

Example 3.8: Split

The lazy wavelet only divides the signal and creates two functions, one with the odd and another one with the even samples of the original signal. The lifting is now performed in two steps, primal and dual lifting. In the dual step λ_j is used to predict γ_j and the prediction error is stored in γ_j. The prediction is dependent on the used wavelet, but for the simple Haar box example the prediction for Example 3.3.3 is:

$$\gamma_j = [\,2.5,\, 4\,]$$

Example 3.9: Prediction

Care has to be taken on the boundary of a data set. There are mainly two different possibilities of how to treat boundaries. The first one assumes a periodic signal where the given function is periodically extended at the boundary. The other option is to assume that the signal is symmetric where the function is mirrored at the border. Usually the later one gives better results and is used more often.
Now these γ_j are replaced by the error of this prediction:

$$\gamma_j = [\,4.5,\, -4\,]$$

Example 3.10: Update

The new γ_j represent the wavelet coefficients, or in other words the detail information which is necessary to reconstruct λ_{j+1} from λ_j. In this example the coefficients seem to be very high, but this is due to the chosen example. Usually one can assume a correlated function where pixel values can be interpolated from their neighbours without introducing huge artifacts. In order to avoid aliasing, λ_j needs to be adjusted in a way that the average of

the $\lambda_{j,k}$ coefficients are the same for all resolutions, i.e. the mean value of the image has to be the same,with k as the current wavelet level:

$$\sum_k \lambda_{-1,k} = \frac{1}{2} \sum_k \lambda_{0,k} \tag{3.21}$$

This is performed when primal lifting the $\lambda_{-1,k}$ with the help of the detail coefficients $\gamma_{-1,k}$:

$$\lambda_{0,k} = \lambda_{-1,k} + \frac{1}{4}(\gamma_{-1,k-1} + \gamma_{-1,k}). \tag{3.22}$$

Now, the final low and high pass version of Example 3.3.3 using integer lifting is:

$$\lambda_j = [\,1,\, 4\,]\; \gamma_j = [\,7,\, 0\,].$$

Example 3.11: Lifting wavelet decomposition

These steps can now be repeated by iteration on λ_j in order to create a multiresolution version of the original signal.
The next section shows how easily the inverse transform can be found and how the algorithm can be adapted to perform an integer to integer transform.

Inverse Transform

One of the great benefits of using the lifting scheme is that the inverse transform can be easily derived from the forward transform. As the forward transform can be split into three simple steps, the inverse transform can too. Each of these steps is invertible and they just need to be executed in the opposite order. The inverse transform can be found by simply reverting the order of operations and changing every + into a − and vice versa:

- inverse primal lifting $\lambda_j - U(\gamma_j) \rightarrow \lambda_j$,
- inverse dual lifting $\gamma_j + P(\lambda_j) \rightarrow \gamma_j$, and
- merge $(\lambda_j, \gamma_j) \rightarrow \lambda_{j+1}$.

First, the λ_j need to be updated in the inverse direction so that λ_j contains the values of the odd λ_{j+1} samples. After this, γ_j is determined by using the *updated* λ_j and the detail coefficients currently stored in γ_j. Now in the final merging process, the odd and even samples are put together and one yields the next resolution level λ_{j+1}. To reconstruct λ_{j+2} one needs the next detail coefficients from γ_{j+1} and repeats all these steps until the original resolution is reached.
While both parts - λ_j and γ_j - still need floating point computation to accurately reconstruct the function, the next section will explain how the lifting scheme can be easily converted into an integer to integer wavelet transform.

Integer Wavelet Transform

Wavelets have many applications besides image, video, or volume compression. But most data sets are given in the integer domain and is undesirable to first convert them to floating points and then do the forward wavelet decomposition and store the low and high frequencies also as floating points. Of course, because one wants to compress the data sets and eventually do lossy compression, the floating points could be rounded to the next integer and stored in this way. But this would introduce additional artifacts and would not allow one to do lossless compression even if one would keep all the detail coefficients.
The lifting scheme can aid for this problem. The advantage that the inverse transform is the exact reversal of the forward transform can be exploited to achieve a real lossless integer to integer wavelet transform which would allow a perfect reconstruction of the original signal. Since divisions by 2 are performed during the lifting step when Haar is used, the prediction values $\mathcal{P}(\lambda_j)$ as well as the update values $\mathcal{U}(\gamma_j)$ are floating point numbers. These numbers can be rounded to the next possible integer and then the dual and primal lifting step look like this:

- dual lifting $\gamma_j - \{\mathcal{P}(\lambda_j)\} \rightarrow \gamma_j$, and
- primal lifting $\lambda_j + \{\mathcal{U}(\gamma_j)\} \rightarrow \lambda_j$.

The place where rounding occurs is marked by the curly braces. The inverse transform looks similar:

- inverse primal lifting $\lambda_j - \{\mathcal{U}(\gamma_j)\} \rightarrow \lambda_j$, and
- inverse dual lifting $\gamma_j + \{\mathcal{P}(\lambda_j)\} \rightarrow \gamma_j$.

Here one can see that in both cases the $\mathcal{P}(\lambda_j)$ or $\mathcal{U}(\gamma_j)$ are either added or subtracted. That means that if one uses a deterministic method to round $\mathcal{P}(\lambda_j)$ and $\mathcal{U}(\gamma_j)$ there will be no error in the reconstruction of the signal and all the low and the high frequency parts can be stored as integers.
This is a great advantage for the lifting method which allows now to perfectly reconstruct a signal while still storing only integers. This is very important, especially when dealing with huge data sets. Other benefits are the in-place calculation and a fast decomposition and reconstruction of the signal.

3.4 Conclusions

All the techniques which were discussed so far are well known and more or less often used in practice. In order to be able to interact in realtime with the visualization, i.e. change the viewpoint or other render relevant parameters, the used technique must be very fast. Some volume rendering algorithms already fail at this point because they are too slow. Raycasting has a very high image quality, but is too slow for the rendering of large

data sets. The fastest volume rendering technique is Fourier domain volume rendering which also allows to gradually change the image quality to gain faster interaction, but one huge drawback is that no occlusion effects can be used and that the images only have an X-Ray character. Another very fast technique, which was chosen for the implementation, is the exploitation of 3D texture mapping hardware. Even though, this technique needs special hardware, it becomes available for more computers with the introduction of new consumer graphics hardware [*n*vi01] [ATi01]. The image quality is also very good as this technique simulates the raycasting approach. One drawback is the available texture memory which can be extended by successively rendering parts of the volume. A short summary of the advantages and disadvantages of these techniques can be seen in table 3.1: Several methods

	Ray casting	Splatting	Shear-Warp	3D Texture
Sampling rate	free	free	fixed	free
Sample evaluation	point based	averaged	point based	point based
Interpolation	tri-linear	Gaussian	bi-linear	tri-linear Spline
Rendering	post-classification	post-classification	pre-classification	pre- and post-classification
Acceleration	early ray termination	early splatt termination (hardware)	RLE	hardware
Precision	float	float	float	8bit (float)
Voxels visited	all	relevant	relevant	all

Table 3.1: Volume rendering techniques

can be used to make the data more manageable. Compression techniques, as the discussed Haar wavelet, can be employed to store the data more efficiently on the hard disk and in main memory. Here another advantage of wavelets is the multiresolution approach. Together with volume rendering using texture mapping hardware, this feature can be used to have an adaptive Level-of-Detail for each brick. Due to finite precision rounding, data which was compressed using standard wavelets can only be reconstructed lossy. Even if all detail coefficients are kept, some information will be missing. The lifting scheme on the other hand allows the wavelet decomposition and a lossless reconstruction. This is very valuable, especially for medical data where a precise visualization is necessary. Hence this technique was chosen for the implementation.
Signal processing and lattice theory can be used to resample the given data set onto another grid which stores the data more efficiently. Here the BCC lattice for 3D and the D_4^* lattice for 4D are used in the implementation. A big advantage of this is that the number of samples is directly related to the

execution time. Fewer samples means faster rendering with no loss in image quality.
All these methods are covered in more detail in Chapter 4. Some selected methods are implemented which are described in Chapter 4 as well as in the Chapters 6 and 7. Chapter 5 discusses possibilities and constraints of multiparameter techniques which can be used to visualize the fuel cell data.

Chapter 4

Realtime Visualization

The ability to interact with an object is of major importance in order to find out what it is and what features are present. As a Chinese proverb says: *A Picture is worth a thousand words*, it could probably be extended to *Interaction is worth a thousand pictures.* In computer graphics, the feeling of looking at a 3D object comes from the interaction with it, the possibility to look from different angles and maybe under different lighting conditions. This enables one to integrate over all pictures and to create a mental model of this object. This is very important in order to know what one is looking at. While this is easy for small data sets, it becomes even more difficult for big data sets. Some modalities of the visible human data set [Set] occupy several gigabytes of storage space. While large hard disks are available, main memory and the processing power is still limited to handle these data sets in real time.

All of the volume rendering techniques which were covered in the last chapter can not directly be used to render these data sets. A large number of optimizations have to be implemented in order to get slow interaction. Processing power and main memory is too limited to use software based rendering techniques like ray casting [Lev88]. Even though, the focus is on fast rendering, image quality has also a big influence. Fourier domain volume rendering [TL93] is a very efficient implementation, but the image quality lacks real depth and occlusion information. Special hardware methods like the Volume Pro [PHK+99] can not be used because of the limited on board *texture memory*. The techniques which could be extended to handle large volume data sets are Splatting [Wes90], Shear-Warp [Lac95] and volume rendering using texture mapping hardware [CCF94]. Even though texture memory is limited for OpenGL hardware accelerated volume rendering, this technique was chosen as the basis for the volume rendering algorithm. A special bricking scheme can be used to *extend* the OpenGL texture memory [vGK96]. This straight forward bricking method is extended in this thesis and in addition compression techniques and a different lattice were used to

reduce the storage space and to increase the frame rate.
This Chapter describes one part of the main work for this thesis. A rendering system with several improvements was built that allows one to interactively visualize volumetric data sets. Interaction is available for exploring the current time frame from orbiting around the volume, changing transfer functions- and rendering parameters as well as by switching to a different time frame.
The method to achieve these results can be divided into two steps, a pre-processing and the final rendering step. A rough outline of the pre-processing step is:

- transform data into BCC or D_4^*,
- subdivide data (split and merge),
- compress each brick depending on its importance, and
- save the data.

The rendering step can be described as:

- load data,
- determine the visibility for each brick,
- uncompress remaining bricks to required resolution level,
- sort bricks from back to front, and
- render bricks from back to front and perform classification and shading in hardware.

In the pre-processing step, the entire data set is resampled onto a more efficient grid which helps to save a huge amount of data without impairing the frequency domain. Hence, this is some sort of lossless compression technique. For 3D data sets, the BCC lattice and for 4D the D_4^* lattice was used. Using these lattices, one only needs 70 percent of the samples for 3D and only 50 percent of the samples in 4D to represent the same information. Section 4.1 explains this in more detail.
For the final rendering of the data set OpenGL hardware is used to perform the visualization in hardware. Here the capabilities of 3D textures of new graphics board are exploited to simulate ray casting in hardware [Lev88]. The data is loaded as 3D texture into the texture memory and then sampled at viewport-aligned slices. These slices are alpha blended from back to front and the resulting images look similar to those generated by software ray casting. Because of the limitation of available texture memory, 128 MB, volumes which are bigger need to be rendered in smaller pieces. Here a bricking algorithm was developed that subdivides the volume into parts of similar importance and that will fit into texture memory. To create these volumes, the original volume is subdivided into small cubes and for each cube the information content is derived. After this, cubes with an importance below a certain threshold are discarded and not used, all other are

merged into bigger bricks of similar importance.
After the subdivision, the 3D/4D bricks are compressed using wavelets and the lifting scheme. This allows not only to save additional memory costs when saving the data and working with it in main memory, wavelets also build a multiresolution version of the data set. To speed up the decompression, wavelets are only used up to the third or forth level. For time-varying data sets, each 3D volume is stored separately in order to not have to decompress the entire data set when visualizing only one time frame. Currently only the mean, the contrast and the entropy are used to determine the importance of a cube.
As said above, the final rendering is accomplished using OpenGL acceleration. To perform shading and classification some advanced features of modern graphics accelerators, such as texture shaders and register combiners, are used. This hardware accelerated volume rendering not only allows direct volume rendering, also other techniques, like maximum-intensity-projection or x-ray images, can be simulated using some special blending functions.
This chapter is organized to follow the rendering pipeline from the preprocessing to the final visualization. The first section explains in detail how storage space and hence also render time can be saved by using a more efficient grid (Section 4.1). It proves that by using the BCC lattice, respectively the D_4^* lattice, 30 or even 50 percent of the samples are sufficient to represent the same amount of information. After this, the next section shows how coherency in the data can be used to increase the compression ratio and the render speed without introducing too many artifacts (Section 4.2). Multiresolution and compression benefit from this coherency and are explained one section later (Section 4.3). Also, level-of-detail rendering is used to decrease the rendering time. Finally, in the last section, volume rendering using texture mapping hardware is explained (Section 4.4). This section also describes which methods were used to extract gradient information and which techniques are used to achieve good image quality through shading and classification.

4.1 Body-Centred-Cubic Grids

Since the beginning of science, researchers have always observed nature and tried to adapt some of these ideas for other applications. The observation of nature can also be very valuable in the field of data compression. Bees build their honeycombs using hexagonal cells, and as a result they achieve a minimum expenditure of wax. Many crystals also exhibit hexagonal spacing to achieve a minimal energy state.
In computer science one huge problem is the size of the data sets. This is an acute problem not only within volume visualization. With consistently increasing processing power and memory capacity, the calculation of more

detailed simulations is possible which consequently results in bigger data sets. Medical equipment is evolving as well and allows now more precise scans of the human body than ever before. But these data sets sometimes require storage capacities of gigabytes or even terabytes. New techniques to aid in the handling of such big data sets are needed. One method is to simply use a more efficient lattice to sample and store the data set. A lattice is a special kind of a grid which is constructed by using one point cell, the origin, and base vectors which are used to construct all the other grid points. As a result, lattices can be described as matrices. A grid itself is a possible arrangement of sample points and can not always be described like a lattice. The most often used lattice is the *CC-Lattice*, or Cubic Cartesian lattice. Its advantage is the simplicity in storage and accessing data values because the indices are inherently included in the lattice. The matrix which constructs a 3D CC Grid can be seen in Equation 4.3in the next section.
A disadvantage of the cubic grid is that it is not very efficient in storing these samples. One of the most efficient lattices for 3D space is the BCC lattice, or Body Centred Cubic lattice. The matrix which describes the BCC lattice can be seen in Equation 4.7 in the next section. BCC lattices are well known from crystallography [Jac91], chemistry [Wel84] and solid state physics [AM76], but rather unknown in computer science. Hexagonal grids are also known in mathematics and refer to the closest sphere packing problem [Slo98]. Equation 4.1 shows the relationship of a sample matrix for

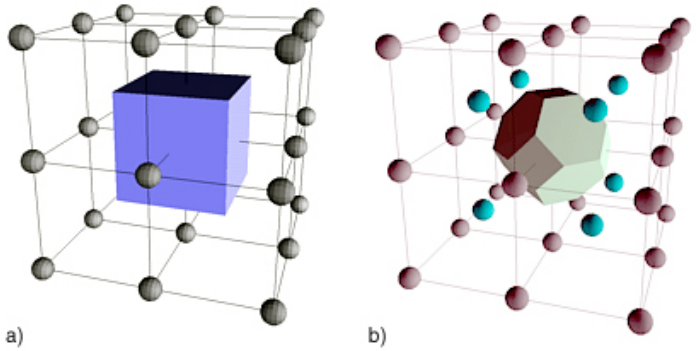

Figure 4.1: Delauny regions of the CC lattice a), and the BCC lattice b)

a lattice in the spatial domain and the corresponding frequency replicator matrix. Here V denotes the sample matrix for the spatial domain and U the replicator matrix in the frequency domain. I is the identity matrix.

$$UV^T = 2\pi I \tag{4.1}$$

These two matrices are directly proportional which means that if one would be able to pack the samples in one domain closer together, they can be taken further apart in the other domain. This refers to the closest packing of spheres or spectra in the frequency domain. One requirement is that the spectra in the Fourier space need to be isotropic and (hyper-)spherically bandlimited. Because the spectra are stored more densely in the frequency domain, the samples are further apart in the spatial domain, and hence one can represent the same amount of information with less samples without impairing the frequency domain. The BCC grid is the most optimal sampling grid in terms of the Shannon theorem [Sha49], and is the best grid to implement discrete mathematical morphology algorithms [Ser82]. Its symmetry helps to simplify the definition of algorithms like Marching Cubes [IHR96] and the relation to optimal sphere packing makes it interesting for tomographic image reconstruction as well as lossless compression. Lossless compression is only warranted if the data is reconstructed and sampled onto a BCC lattice and spherically bandlimited. Data sets which are sampled on the CC lattice can be resampled onto the BCC lattice (Section 4.1.4).
Figure 4.1 shows the basic neighbourhood relations for a CC and a BCC grid. It can be seen that a point in the BCC grid has more neighbours than a point in the CC grid. This allows for finer line tracing, like the Bresenham algorithm, than it is possible in the CC lattice, Figure 4.2. Hexagonal grids

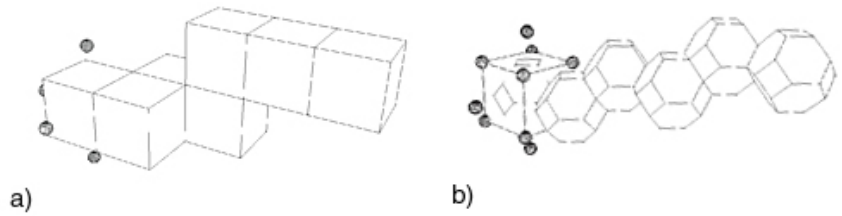

Figure 4.2: Bresenham for the CC lattice a), and the BCC lattice b)

are also used in imaging and image processing. Here the advantage is on sampling non-axis parallel lines. Staunton et.al. [SS89] have shown that image processing operators on hexagonal grids are computationally more efficient, and as accurate, as their square counterparts.
The first time the BCC grid was used for scientific visualization was for ray casting medical image data sets [IHR97]. Ibáñez et.al. adapted a raytracer to work with the BCC lattice by using a customized Bresenham algorithm for hexagonal grids. Later, body centred cubic grids were also applied to Westovers Splatting algorithm by Teußl et. al [TMG01]. They adapted a splatting algorithm for the BCC lattice and showed that this implementa-

tion speeds up the rendering process by the same ratio as less samples are needed to represent the original signal in the CC lattice. Other recently for the BCC grid adapted visualization algorithms are iso-surfaces using Marching Hexahedra [CTM02] and the Shear-Warp volume rendering [SM02].
The next sections explain in more detail how optimal sampling in 3D and 4D work. It also proves that the sampling is correct and how an optimal lattice can be constructed for the nD case. Interpolation and resampling, i.e. how to transform a given CC data set into BCC or D_4^*, are discussed as well.

4.1.1 Optimal Sampling in 3D

A basic requirement for functions that are sampled to or into the BCC lattice is that they are isotropic and hyperspherically bandlimited. This guarantees that the frequency response is restricted to hyperspheres. To reconstruct the original continuous signal, the samples in the spatial domain need to be close enough so that the aliased spectra in the frequency domain do not overlap. For optimal sampling, the number of samples which fulfill this condition is minimal and is known as the Nyquist frequency [Sha49]. To sample optimally in higher dimensions, aliased spectra in the frequency domain have to be packed as close as possible. This is known as the closest sphere packing problem [Slo98].
For any bandlimited waveform, there is an infinite number of possible choices for the periodicity matrix U and the appendant sampling matrix V. The most often used sampling lattice is the rectangular Cartesian grid, where the sampling matrix is simply diagonal. Equation 4.3 shows such a matrix for the 3D CC Grid. Sampling a function can be seen as the mapping from indices to the actual sample position:

$$\begin{pmatrix} x \\ y \end{pmatrix} = V \cdot \begin{pmatrix} i \\ j \end{pmatrix} \tag{4.2}$$

Here i and j are the indices of the sample point and x and y is the corresponding sample position. The matrix V is the sampling matrix for the CC lattice:

$$V_{rect3D} = \begin{bmatrix} T_1 & 0 & 0 \\ 0 & T_2 & 0 \\ 0 & 0 & T_3 \end{bmatrix} \tag{4.3}$$

For the regular Cartesian grid applies $T_1 = T_2 = T_3$ and is usually equal to 1.0. But regular rectangular sampling in 3D is by far not optimal. An optimal sphere packing for arbitrary packing structures in 3D is not known, but there exist a few packing structures for 3D, which all have the same packing density less than regular Cartesian grid [CS88]. One example is the face centred cubic grid (FCC). The hexagonal close packing (HCP) which

has similar qualities as the FCC lattice can not be described with a matrix. The replicator matrix for the 3D FCC grid can be used in the frequency domain to achieve a more dense packing:

$$U_{FCC} = U_{FCC}^T = \begin{pmatrix} u & 0 & u \\ 0 & u & u \\ u & u & 0 \end{pmatrix} \tag{4.4}$$

The FCC lattice consists of a cubic cell with additional sampling points at the centre of each cube side (face). The kissing number, i.e. the number of how many neighbouring spheres a sphere touches, is 12. Figure 4.3 c) shows an FCC cell with the appropriate distances between the sample points. FCC

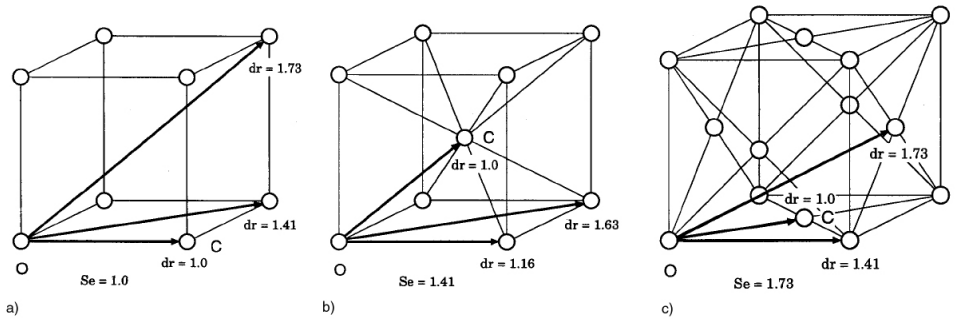

Figure 4.3: CC lattice a), BCC lattice b), and FCC lattice c)

grids are found in the real world in fruit stands or in piles of cannon balls on war memorials. By transforming Equation 4.1 and by plugging it into 4.4 one yields:

$$\begin{aligned} U^T V &= 2\pi I \\ V &= 2\pi (U^T)^{-1} \\ V_{BCC} &= \tfrac{1}{2} \begin{pmatrix} T & -T & T \\ -T & T & T \\ T & T & -T \end{pmatrix} \end{aligned} \tag{4.5}$$

with $T = \frac{2\pi}{u}$. As can be seen from Figure 4.3 b) a BCC lattice consist of a cubic cell with one additional sampling point in the cell centre. The BCC lattice can also be interpreted as two interwoven cubic grids. Here applies:

$$T_{1_{CC1}} = T_{1_{CC2}}, \quad T_{2_{CC1}} = T_{2_{CC2}}, \quad \text{and } T_{3_{CC1}} = T_{3_{CC2}}, \tag{4.6}$$

with $T_{k_{CCl}}$ as the k^{th} base vector for the l^{th} CC lattice.
While some of the base vectors of Equation 4.5 are negative and rather unintuitive to use, a different yet better set of base vectors is [CS88]:

$$V_{BCC} = \frac{1}{2} \begin{pmatrix} T & 0 & \frac{1}{2}T \\ 0 & T & \frac{1}{2}T \\ 0 & 0 & \frac{1}{2}T \end{pmatrix} \tag{4.7}$$

The 3D Fourier transform $\mathcal{F}$ of a spherically, Cartesian bandlimited signal has the feature:

$$\mathcal{F}(\omega_1, \omega_2, \omega_3) = 0 \quad with\ \omega_1^2 + \omega_2^2 + \omega_3^2 \geq W^2. \tag{4.8}$$

Because the data set is bandlimited, W is the maximum representable frequency, the Nyquist frequency. To guarantee that the replicas in the frequency domain do not overlap, $u = \frac{2\pi}{T}$ must be larger or equal to $2W$ for regular Cartesian sampling [Sha49]. Calculating the sampling matrices from these periodicity matrices, one yields:

$$V_{rect3D} = \begin{bmatrix} \frac{\pi}{W} & 0 & 0 \\ 0 & \frac{\pi}{W} & 0 \\ 0 & 0 & \frac{\pi}{W} \end{bmatrix}, \tag{4.9}$$

with:

$$|detV_{rect3D}| = \frac{\pi^3}{W^3}, \tag{4.10}$$

and for the FCC lattice, u must be equal to $\sqrt{2}W$ which results in:

$$|detV_{BCC}| = \frac{\pi^3}{W^3}\sqrt{2}. \tag{4.11}$$

The ratio between V_{rect3D} and V_{BCC} now is:

$$\frac{|detV_{rect3D}|}{|detV_{BCC}|} = 0.707, \tag{4.12}$$

which proves that one needs only as many as 70.7 % sampling points to display the same amount of information than the 100 % for the regular grid. The sampling distance in the third dimension decreases by $\sqrt{2}$ while the samples are $\sqrt{2}$ further apart in the other two dimensions..
A big advantage of using the FCC grid in Fourier and the BCC grid in the spatial domain, is that the indexing and storage scheme for the BCC grid is similar to the one which is used for regular grids. All data can still be stored in a 3D array with an inherent indexing scheme. The even slices remain at their position and the odd slices need to be shifted by $\frac{1}{2}$ in x and y directions. Here one can again see that the BCC grid can be described as two interwoven regular Cartesian grids, one with the even and another one with the odd slices. This quality can also be exploited for rendering 3D BCC data sets using texture mapping hardware.

4.1.2 Optimal Sampling in 4D

For the 1-dimensional case the ratio of the efficiency of the cubic lattice to a hexagonal lattice is 1.0. That means both lattices are equal. This is not a surprise, however in 2D the efficiency is about 0.86 and for 3D already at 0.707. This decreases further with higher dimensions and shows how inefficient the CC lattice is. This section explains how the spherical lattice for the n-dimensional case can be found and proves this as well as the efficiency ratio for the 4-dimensional case. In 4D a sampling matrix

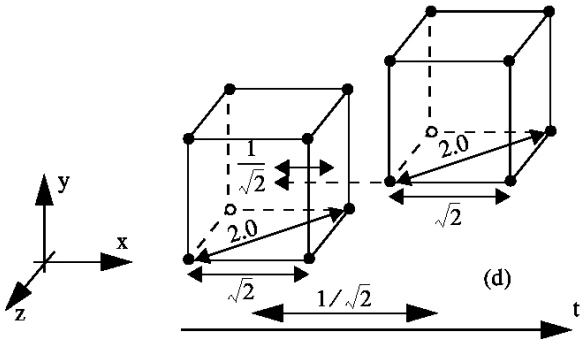

Figure 4.4: D_4^* lattice

can be used that allows to save 50 percent of the original samples while assuming that the spectra are hyperspherically bandlimited. In general, the lattice for the n^{th} dimension can always be constructed by taking the $n-1$ base lattice, e.g. the $n-1$ CC lattice, and offsetting and interleaving it for the n^{th} dimension. The 4D hexagonal lattice can be seen as several 3D CC grids which are offset by $\frac{T}{\sqrt{2}}$ in all 4 directions x, y, z and t. Similar to the BCC grid, the 4D grid can also be seen as been built by two 4D CC grids which are interleaved by $\frac{T}{\sqrt{2}}$ in x, y, z and t. Figure 4.4 shows the *4D BCC* lattice where the two 4D CC cells have been pulled apart along the time axis.
The sampling matrix for a 4D regular Cartesian grid is as follows:

$$V_{rect4D} = \begin{bmatrix} T_1 & 0 & 0 & 0 \\ 0 & T_2 & 0 & 0 \\ 0 & 0 & T_3 & 0 \\ 0 & 0 & 0 & T_4 \end{bmatrix} \tag{4.13}$$

As for the 3D case, it also applies for the 4D Cartesian grid that usually $T_1 = T_2 = T_3 = T_4$ with T_i the sampling distance for n. For the regular CC grid, typically $T_i = 1.0$ is used. As for the like the 3D CC grid, this lattice is not very efficient for sampling and storing data sets. The densest sphere

packing for 4D is know as the checkerboard lattice D_4. The sampling matrix V for the D_4 lattice is:

$$V_{D_4} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ \end{bmatrix} \tag{4.14}$$

Here the centres for the spheres are all the points u_1, u_2, u_3, u_4 which add to an even number. For instance $1, 0, 1, 0$ is allowed while $0, 1, 0, 0$ is not. This principle gave the name to the lattice. The kissing number is 24 and the minimal distance between the centres is $\sqrt{2}$. This is minimal for 4D.
While Equation 4.14 is not very intuitive, a better one exists in the dual space D_4^* [CS88] which can be derived from:

$$M^* = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \cdots & \frac{1}{2} \\ \end{bmatrix}. \tag{4.15}$$

With the proper scaling factor one yields the D_4^* sampling matrix:

$$V_{D_4^*} = T\sqrt{2} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \end{bmatrix}. \tag{4.16}$$

In general all other sampling matrices for nD can be derived from M^*. Again, the D_4^* lattice can be considered as build by two 4D CC grids which are offset by $\frac{T}{\sqrt{2}}$ in x, y, z and t. The sampling distance along t is decreased to $\frac{1}{\sqrt{2}}$. So we actually have $\sqrt{2}$ more slices along the time axis, but at the same time, each cube has $\frac{1}{\sqrt{2}^3}$ less samples because the sampling distance in x, y and z is increased. With $\sqrt{2}$ more samples along the 4^{th} dimension, the *efficiency ratio* of the D_4^* lattice over the standard regular CC lattice becomes:

$$\frac{\sqrt{2}}{\sqrt{2}^3} = \frac{1}{2} \tag{4.17}$$

This proofs that by sampling a time-varying data set into D_4^* one can get away with 50% of the samples without damaging the frequency content. This is, of course, only true if the data is hyperspherically bandlimited, which can

be assumed for most data sets. While we still have only 3D regular textures and the texture memory is rather limited and does not support hexagonal sampling yet, one has to think on how to slice the D_4^* lattice in order to extract 3D volumes which can be rendered. There are several possibilities which will be covered in the next section.

4.1.3 Slicing D_4^*

If no 4D visualization techniques are used, for examples see Chapter 5.6, one has to slice the D_4^* lattice in order to extract either a 3D volume or a 2D slice prior to the visualization. To extract a *slice*, several techniques can be used. The most simple one would be to evaluate the plane equation for $\Re^4$:

$$ax + by + cz + dt + e = 0 \tag{4.18}$$

to extract an arbitrary oriented hyper slice. In Equation 4.18 a, b, c and d represent the normal of the plane in 4D space and e is the distance from the origin. Using higher order interpolation, one can extract an arbitrary 3D volume and display it through volume rendering or other visualization techniques. Because most 4D data sets are time-varying and hence all volumes are topologically identical and refer to the same location in space, it is more intuitive to slice perpendicular to the 4th dimension, in this case the time axis t. If one considers that the D_4^* lattice is constructed by taking the $n-1$ CC lattice and offsetting them in the 4^{th} dimension, one could come up with a simpler and less computationally expensive method. Additionally the 4D data set could be stored in several 3D arrays and each of these arrays can be compressed separately. This is very convenient for huge data sets which can occupy several gigabytes of memory. These 3D arrays represent data which is actually sampled into D_4^* but which is decomposed into several 3D CC lattices.
The data which is extracted is actually sampled into 3D BCC. For slicing D_4^* two different methods can be used. The most simple one would be to directly render the 3D CC volumes which where built by the resampling into D_4^*. The size of each of these volumes is only:

$$\frac{1}{\sqrt{2}^3} = 0.35, \tag{4.19}$$

of the original size of one volume/time-frame. This is because the sampling density has been decreased in all three dimensions. However, in the 4h dimension are $\sqrt{2}$ more samples/volumes.
A different approach which would result in better image quality is to first reconstruct a BCC grid out of the three neighbouring CC volumes. Because all volumes have a shift in time by $\frac{1}{2}$, first the second BCC volume has to be interpolated from the two CC volumes at $t-1$ and $t+1$. This reconstructs

a 3D BCC volume at time step t.
The interpolation is straight forward and higher order interpolation schemes, like cubic or spline interpolation can be used. If one would apply a different weighting to these slices, one would be able to interpolate arbitrary in time which would result in a signal that is continuous over time. Hence the transition between two time frames is smooth and this would yield a continuous animation over time.
The next section explains how interpolation can be performed and how the data sets can be transformed into a hexagonal grid.

4.1.4 Resampling and Interpolation

This section explains how the interpolation in the BCC grid is performed and which filters can be used for hexagonal lattices, especially for the 3D BCC lattice. It also explains how regular Cartesian data sets can be resampled onto hexagonal grids.
Sampling and interpolation can be performed in the same way than on the Cartesian grid, except that some different weighting schemes have to be used due to the different grid topology. Geometrical transformations on discrete grids can be computed directly for only a few cases, like rotation about $90° \cdot i$ for $i \in N$. To sample a function on arbitrary positions, the function values in between have to be interpolated which can be realized using a standard convolution [KOPR97]:

$$f(x,y,z) = \sum_l \sum_m \sum_n f(l,m,n) h_{3D}(x-l, y-m, z-n), \tag{4.20}$$

with h_{3D} being the interpolation mask in 3D.
Figure 4.2 shows the Delauny neighbourhood for both, the 3D CC and the 3D BCC lattice. The Delauny neighbourhood for the cubic grid is a cube. That means that each point in the CC lattice has eight nearest neighbours which need to be included and weighted in the interpolation scheme. The evaluation of this cube implies the computation of a cubic polynomial. Common PC rendering hardware for 3D textures only support tri-linear interpolation between samples which is not perfect, but sufficient in most cases. The Delauny neighbourhood for the BCC lattice is a tetrahedron. Here each point has only four nearest neighbours and hence only these need to be considered for the interpolation. However, it results in better image quality when the surrounding 8 or 14 points are considered as well.
Sampling corresponds to a copying of the spectra in frequency domain. The original signal can be reconstructed using an ideal *box* filter in the frequency domain. The corresponding filter in the spatial domain is the *sinc* filter [OS75], Equation 4.21, which can not be used in practice because of its in-

finite range. Hence the goal is to find a filter that approximates the sinc filter.

$$sinc(x) = \frac{sin(\pi x)}{\pi x} \tag{4.21}$$

The easiest solution is the nearest neighbour filter which simply uses the closest sampling points. The result is a somewhat blocky signal. Better interpolation can be achieved by using a linear interpolation or interpolating higher order polynomials which weight the sampling points by distance and really *interpolate* at the new sampling position. Very good results can be achieved using the Lagrange polynomial [KOPR97]:

$$k_{Ln}^{k}(x) = \begin{cases} \prod_{i=1}^{k}(1 - \frac{x^2}{i^2}) & \text{for } 0 \leq |x| \leq \frac{n}{2} \\ 0 & \text{else} \end{cases}, \tag{4.22}$$

with k the order of the polynom and n the number of supporting points. For very high orders ($k \geq 500$) the difference between the sinc filter and the interpolation polynom is neglectable. The disadvantage is that this interpolation is computational very expensive. Cubic filters were used for the resampling of the regular data sets onto the BCC or D_4^* lattice.
When resampling a signal from one lattice to another one, the same principles for interpolation apply. First one tries to reconstruct a continuous signal from discrete sampling points and then resample the signal on different sampling positions. The same filters which were used for interpolation can be applied here. Because this resampling is done only once in a pre-processing step higher order filters can be used to achieve a good approximation.
For the interpolation step that needs to be performed during the rendering process, a faster implementation has to be used. Because volume rendering is performed in hardware, the interpolation has to be done in hardware as well. For 3D textures tri-linear interpolation is used. For rendering BCC textures, two texture units have to be used and the interpolation works only on the texture unit where the texture is assigned to. Hence in the interpolations step, not all sampling points can be considered for the interpolation in one texture unit. This 4th linear interpolation step is performed when the two textures are blended together by using multi-texturing and alpha blending.
Chapter 4.4 discusses the sampling which is performed during the rendering in more depth, while Chapters 7.2 and 7.3 give some details for the implementation.
Once the data is resampled on the BCC lattice some other pre-processing steps like the computation of the multiresolution version and data compression can be invoked. The next section explains how spatial and temporal coherency in the data can be used to achieve better compression results and an increased rendering speed.

4.2 Coherency

The use of more efficient lattices helps to reduce the data size and to store the data in a more economic way. Often this is not enough and the data set is still larger than the memory which is available on the graphics hardware. With the 128 MB available on some of the GeForce series graphic boards, one could store data sets up to a size of 512^3 if they are stored with one byte per sample point. The data can be allocated as a simple 3D texture with one component. If one needs to include shading as well, then one would have to allocate a 3D texture with four components, three for the normal or gradient and one for the intensity value. This significantly decreases the available texture memory space. Farther all buffers, like frame- and z-buffer have to be deducted from the available memory because these buffers are defined on the same memory. This shows that further improvements are necessary in order to render huge data sets.
Data sets that do not entirely fit into texture memory can be split into smaller bricks and rendered separately. One actually has three cases to render data using texture mapping hardware. The first one is that the data fits entirely into texture memory and one has only one single brick. This is the easiest and also the fastest solution because the data can be kept in texture memory and does not need to be reloaded every frame. If the data is bigger, it can be split into several bricks of the largest texture size that can be allocated. This is the simple and often used approach for huge data sets [vGK96]. For a more efficient subdivision of the data, two forms of coherency can be used to improve this simple solution.
The first one is spatial coherency, which uses information from neighbouring bricks and tries to merge these regions together that have a similar importance or entropy. This results in bigger bricks which reduces the number of texture loads. Additionally, one can effectively use Level-of-Detail to render each brick using its own unique LoD. Empty regions can be skipped which decreases the number of texture loads and the amount of data that has to be transferred to the graphics card.
The other form of coherency is the temporal behaviour. Here an error metric is used for time-varying data sets to decide if the texture of a given brick needs to be changed once the user selects a different time frame. If some of the textures can remain, these bricks do not need to be updated and a faster response is the result if a new frame is chosen.
The next section explains the bricking method in more detail and what problems or artifacts might occur when a multiresolution representation of the volume is used. The following two sections after, discuss the use of spatial and temporal coherency and how the importance of a brick can be computed.

4.2.1 Bricking

Bricking is a technique that is used to render volumes using texture mapping hardware that exceed the limit which is defined by the available texture memory. Because of internal definition, all textures that can be used with OpenGL need to be of 2^n in each dimension. Even though recently a new extension was introduced that enables one to use textures without this restriction, the hardware still allocates the next power of 2 texture size. The advantage of this extension is that only the data that is needed has to be transferred over the AGP bus. For huge textures that just exceed the next smaller texture resolution, the speed up can increase by a factor of nearly eight for 3D textures. Unfortunately this extension is not available yet for 3D textures. Textures have to be zero padded to the next power of 2 before transferring them to the graphics hardware [Kil01].
Volume rendering using 3D textures was introduced by Cabral et.al. [CCF94] in 1994. The first computer which supported this in hardware was an SGI with the Infinite Reality Engine I. The available texture memory was rather limited and only 4MB in size. In order to be able to use bigger textures van Gelder et.al. [vGK96] developed a solution called bricking which simply divides large textures and renders each brick individually.
Because the volume rendering algorithm simulates the over operator by using alpha blending [CCF94], each brick is sliced and composed from back to front. In order to render the bricks correctly, they have to be sorted prior to the rendering in a back to front manner. For uniform bricks which all have the same size, the order is inherent in the brick's location. One only needs to store all the different cases which can occur and while rendering, chose the one depending on the camera position which is most parallel to the viewing plane. A similar technique was used by Kilthau et.al [KM01] for fast cache coherent splatting of volumetric data sets.
For uniform bricks which do not all have the same size, e.g. one is 256^3 and another one is 64^3, the ordering can be simply done by using the angle between the camera and each brick centre. The brick which has the largest angle is farthest away and rendered first, while the one with the smallest angle is rendered last. The sorting can be done by using Bucketsort [Sed88] which has an average complexity of $O(n)$.
For non-uniform bricks, the sorting can be rather difficult, because one would actually have to sort over all 8 corners of each brick in order to find out which brick needs to be rendered first. Additionally the absolute ordering of the bricks along the x, y and z direction has to be included to use the topology or brick structure in the sorting. Then these arrays can be used in some sort of sweeping algorithm where a sweeping plane from the back of the data set is passed through the volume which decides which brick gets rendered first. Existing bricking algorithms subdivide the entire volume, regardless of what it contains into the biggest possible texture chunks and then render each of

these textures in a back to front manner. This approach could also be described as greedy or brute force bricking, because it does not pay attention to areas which are empty or contain unimportant data such as air or noise. The bricking algorithm which is used for this implementation goes a little further. It can handle uniform as well as non-uniform bricks of different sizes and different resolution. It also incorporates the nature of the data by taking the spatial and the temporal coherency of the data set into account. These two qualities will be explained in more detail in the next two sections. Figure 4.5 a) shows an example of how a volume might be divided into eight

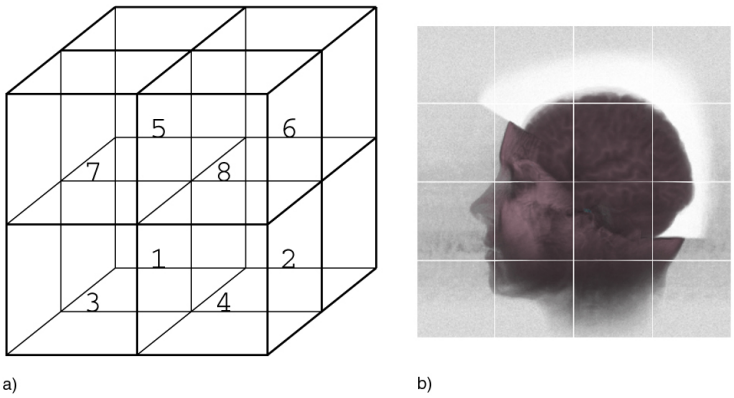

Figure 4.5: Simple bricking a), and artifacts b)

bricks. Here the bricks are ordered in the same way as they will be rendered. The farthest brick (1) will be rendered first. In order to avoid interpolation artifacts the bricks must be loaded with boundary voxels which are not used for rendering, but for proper interpolation while resampling the texture. If one would just divide the texture by half and not use overlapping boundaries, the interpolation at the border would be wrong and clearly be visible as white lines at the brick border, Figure 4.5 b). A more memory efficient solution is to load the textures overlapping. Figure 4.6 shows the principle in 2D. The texture is only sliced, or better sampled, to the centre of the

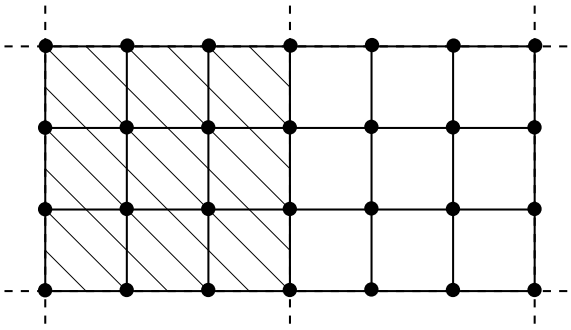

Figure 4.6: Overlapping textures

outer voxels as can be seen in Figure 4.6. The slicing border of the two bricks is shown as dotted lines. Only the slices for one brick are displayed. As can be seen from Figure 4.6, the two bricks have the same border, or an overlapping texture. To avoid artifacts, the next texture has to be shifted by one voxel size towards the origin. The n^{th} texture is moved by $n-1$ voxels towards the origin.
If multiresolution and Level-of-Detail are used, the bricking algorithm introduces artifacts at the brick boundary. If simple averaging is used to determine the next lower resolution, then in the next resolution level, the bricks do not correctly overlap because inner points were included to compute the next resolution level. Figures 4.7 a) and b) clarifies the problem.
Additionaly, as can be seen in Figure 4.7 b) the sample position of the lower resolution volume is shifted towards the brick centre. To avoid these artifacts, one could include more boundary voxels which would require more texture memory and would not result in an smaller texture. Hence LoD would be useless. One goal in the design of this rendering method was to focus on speed to handle huge data sets at interactive rates. Although good image quality is another goal, some compromises had to be made towards the perfect solution. The correct sampling position for the lower resolution bricks is used and the texture is rendered with the *GL_CLAMP* command that continues the signal which is at the border to infinity. The described

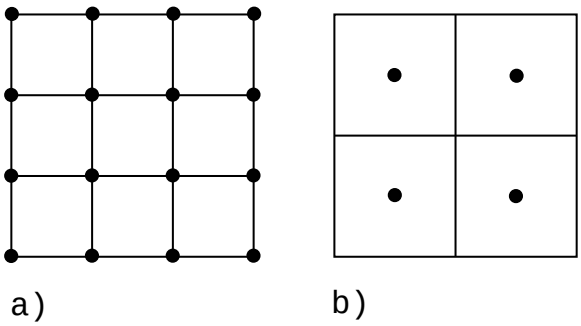

Figure 4.7: Multiresolution bricking - a) full resolution, b) half resolution

methods work well for regular grids and can directly be applied to BCC grids. The only difference is a modified lattice topology. One could either work with both CC cubes or just with one and treat the second one similar to the first one. For the rendering both CC cubes can be rendered separately using two texture units and then blended together using register combiners. The next section discuss how the data set can be efficiently subdivided by using spatial coherency to compute the importance of particular regions in the data.

4.2.2 Spatial Coherency

Most data sets do not only consist of interesting regions. There are usually some parts within the volume in which one is not interested in or which even disturb in the visualization. Medical data sets for instance sometimes contain areas with only air, noise or artifacts which are due to the reconstruction. Figure 4.8 shows the engine data set volume rendered using texture mapping hardware with diffuse shading and orthographic projection. One can clearly see the reconstruction artifacts due to the CT acquisition. These parts of the data are not only unpleasant, they also hinder in the process of gathering information from behind because this noise hides the real interesting data. Of course, one could use transfer functions to suppress the noise and simply blend it off. The disadvantage of this technique is that other structures in the data set with similar qualities (gray tone) are blended off as well. A more efficient solution would be to simply exclude the noise from rendering. This way one can increase the rendering speed twice, due to data that now does not need to be transferred to the graphics card and does not need to be rendered. Currently there are no techniques which automatically

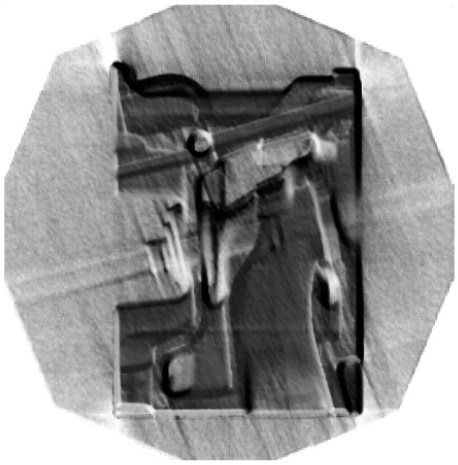

Figure 4.8: *Empty* regions around the engine data set

segment the data in a pre-processing step and use only *interesting parts* for the final rendering. The advantage of segmentation is that on one hand unnecessary data can be removed and on the other hand the rest of the data is analyzed and rendered with a unique Level-of-Detail suitable to represent the information of this brick. If the entire data set fits into the texture memory, then this method does not need to be used. Here, all data can simply be loaded in the texture memory once and then used for rendering. What one still might want do is to segment the data before loading it as

texture to remove the noise by zero padding these areas in a pre-processing step which only contain noise, air or reconstruction artifacts.
To combine texture based volume rendering with a multiresolution approach, usually an octree structure is used which sets the highest LoD for bricks close to the camera and the lowest LoD to bricks which are farthest away [BNS]. The Time-Space-Partitioning Tree (TSP) introduced by Shen et.al. [SCM99] [ECS00] also uses some spatial and temporal coherency to decide which LoD to use. But this algorithm is also based one an octree structure. Figure 4.9 shows an example [ECS00]. The goal for the new bricking algo-

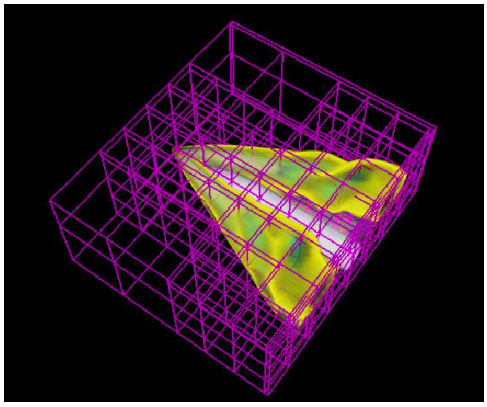

Figure 4.9: Octree structure in volume rendering

rithm was to remove unimportant data, and to merge the remaining parts into bigger bricks, which, of course, have to be small enough to fit in the available texture memory. The algorithm merges smaller bricks with similar importance together to reduce the number of texture loads and to effectively use Level-Of-Detail. The complete bricking algorithm can be roughly divided into three steps:

- split data set into small cubes ($\approx 16^3$),
- compute importance for each cube,
- merge similar cubes together into *homogenous* bricks.

The first step simply homogenously divides the data into many small cubes of sizes between 4^3 and eventually 128^3. This size can be controlled by a parameter and depends on the size of the actual volume. All cubes are created with overlapping borders because this is a requirement for the rendering (Section 4.2.1).
After this step is done, for all cubes the importance is computed. For static data sets, a gaussian smoothed version of the data set is used to compute the importance. For time-varying data, some of the time-frames are accumulated together which results on one hand in a smoothed volume and on the other hand in a volume that represents most of the time features. This

smoothing is important to not highlight noise in the data which one actually wants to remove. The importance is computed by looking at several qualities of the data set, like the mean value, the contrast, the maxima and minima as well as the entropy:

$$\begin{aligned} &mean = \bar{g} = \tfrac{1}{L \cdot M \cdot N} \textstyle\sum_{l,m,n} f(l,m,n), \\ &contrast = q = \tfrac{1}{L \cdot M \cdot N} \textstyle\sum_{l,m,n} (f(l,m,n) - \bar{g})^2, \\ &entropy = H(p) = \textstyle\sum_{g \in \underline{G}} p(g)\, log_2\, (\tfrac{1}{p(g)}) \\ &\text{with} \qquad p(g) = \tfrac{h(g)}{L \cdot M \cdot N}. \end{aligned} \tag{4.23}$$

Here L, M and N are the dimensions of the brick and $f(l,m,n)$ is the density value of the volume at position l,m,n.. The entropy is computed by using the relative frequency $p(g)$ of the gray tone g. One application for the entropy is image processing where it is used to determine the information content of a certain region. The importance is then computed as:

$$importance_i = mean * contrast * entropy, \tag{4.24}$$

and scaled between zero and one. For time-varying data, also the local importance, the importance for each time step is computed and is used as an error metric for temporal coherency.
All this information is used to decide if the current sub-brick contains important information which needs to be rendered or not. If this information will be used for the rendering, then the importance is also used to determine the resolution/compression level for this brick. If the importance, contrast and mean are below a certain threshold then this sub-brick is not used for rendering. Otherwise all other sub-bricks are merged together to form larger textures, to reduce the amount of texture loads. The merging procedure can be seen in Figure 4.10. Here the double hatched areas are the already merged bricks, and the single hatched areas are examined for merging. The merging step starts at one corner of the volume ((x_0, y_0, z_0) and decides if the current brick can be merged with the neighbouring one on the x axis ((x_1, y_0, z_0)), Figure 4.10 a). If so, then the algorithm tries to merge the next two bricks on the y axis ((x_0, y_1, z_0) and (x_1, y_1, z_0)), Figure 4.10 b). If this succeeds as well then the z axis is analyzed and the next four bricks are merged if possible ((x_0, y_0, z_1), (x_1, y_0, z_1)), ((x_0, y_1, z_1) and (x_1, y_1, z_1)), Figure 4.10 c). This continues until the maximum texture size is reached or if no more bricks can be merged together. As can be easily seen, this algorithm allows one to create uniform as well as non-uniform bricks. All textures have to be of size 2^n. E.g. if the volume is divided into cubes of the size of 8^3, then the next bigger uniform brick would be of size 15^3. The smaller texture is due to the overlapping textures. Now, to merge the next level, two additional sub-bricks have to be merged in all directions to yield a 29^3 texture. One

problem with this merging is that the textures are not 2^n anymore, because they are initially defined with an overlapping border. To solve this problem, the remaining space in the texture can be filled with zeros which allow a nearly 100% compression, but they have to be transferred to the graphics card. In order to render correctly, the texture coordinates as well as the clipping planes have to be adjusted. One problem that might occur is that

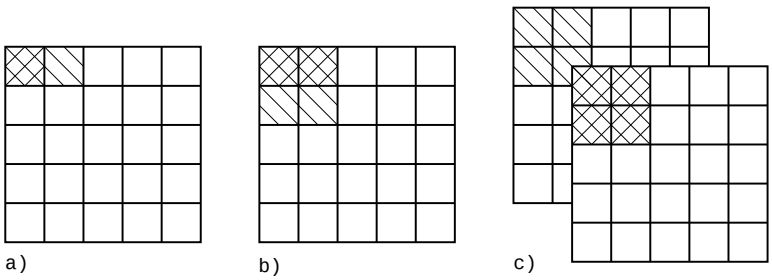

Figure 4.10: Merging textures

a cube will be classified as unimportant, even though the border or just a few voxels belong to an edge of an interesting object. The absence of these voxels in the visualization might be noticeable. In order to improve the algorithm, voxels at the border are weighted higher than those in the middle. Also the importance information from neighbouring sub-bricks can be taken into account. Because the textures overlap at the brick border, all border voxels are still rendered, even if the brick is classified as not interesting.
This pre-segmentation has lots of room for improvement. One is the question on how to efficiently subdivide the volume and how to differentiate between important and unimportant. A better feature extraction might use wavelet filters as is described by Machiraju et.al [GMR97]. Also the merging step could be improved as the one which is described here does not globally analyze the data set and only merges in one direction.
As with the bricking, the subdivision can directly be applied to BCC grids. One only has to change some weights when filtering the data set and computing the importance for each brick.

4.2.3 Temporal Coherency

The bricking process from the previous section also includes the temporal information to create bricks that are also *uniform* in time. The temporal information is used in addition to determine the temporal behaviour of a brick or a data set in time. An example would be the movement of gray tones or other changes in the data set. This temporal behaviour can be used for the update procedure to decide if a given texture needs to be changed or is still valid by a given ϵ to describe the information contained in the volume at this point in time. For instance if the information content of a brick does

not, or just slightly, change from one frame to another, the texture does not need to be updated and hence some update time is saved which results in a faster response of the visualization.
Therefore in a pre-processing step for each brick that will actually be included in the rendering, the local (time) importance is computed and stored in an error matrix. Some thresholds are used to decide if the current texture can also be used to represent the signal of the next time frame. Then only those bricks are updated which textures have to be changed.

4.3 Multiresolution and Compression

Once the data is decomposed into bricks using spatial and temporal coherency it can be further compressed by using wavelets. The wavelet decomposition catches two birds with one stone. One is the compression that allows to store the data in a more efficient way and the other is that at the same time a multiresolution version of the data set is constructed. Multiresolution and a specific Level-of-Detail for each brick allow to gradually switch between two foci for the rendering: image quality or the speed of interaction.
As described in the previous section, the entire data set is subdivided into small bricks. Each of these bricks has its own unique importance which tells how much the contained information will contribute to the overall image. If possible, neighbouring bricks are merged together to build bigger textures to keep the total amount of bricks small. These bricks are either in the BCC lattice, if only one single time frame is available, or in the D_4^* lattice if the data set is dynamic and varies over time. These bricks are now compressed using either wavelets for cubic grids, or wavelets for hexagonal grids.
If the data set is sampled into the regular cubic lattice then each volume, for static data sets only one, is compressed and stored using 3D wavelets for cubic lattices. This way, only the current frame is uncompressed in memory and used for rendering. All other time frames are efficiently encoded and remain either in main memory or on hard disk. Which wavelet is finally used is not important for this technique, but very important for the image quality and the resulting compression ratio. In order to be able to use lossless compression, the lifting scheme, see also Chapter 2.3.3, has to be used which also allows to compress in the integer domain.
For static BCC data, the bricks are compressed using hexagonal wavelets for the BCC lattice. If the volume is dynamic then 3D wavelets for cubic grids are used because the D_4^* lattice can be decomposed into $\sqrt{2}t$ 3D cubic grids. These volumes can be independently compressed and stored as each of them represents a different time frame. If a BCC volume needs to be extracted, either two or three of these volumes need to be decompressed and the BCC grid will be interpolated as described in the Sections 4.1.3 and 4.1.4.

Using either lattice and/or wavelets, the wavelet decomposition can be performed in a way that high frequencies (details) can be discarded using a user specified threshold. This threshold can be set from numerically lossless to visually lossless and for best compression ratios also to lossy compression. The data is stored in different arrays dependent on high or low frequencies. The low frequency volume might be further decomposed to a certain level, while the high frequency volume is stored using a bitstream method like Huffman Coding or Run-Length-Encoding [Sal98]. For this application the RLE algorithm was chosen because of its high speed and good compression ratios for detail information. The final low frequency volume will not be compressed because the compression ratios would not be as good as for the high frequency volumes. Another advantage is that when switching to a new time frame, a low resolution version of the volume can be rendered immediately without the need of decoding it first.
The use of wavelets to compress volumetric data sets is not new, but not heavily explored for volume rendering. First results were described in [PSM02] and more recently with better results in [IP02] and in [GWGS02]. Wavelet compressed data sets can also be rendered in the wavelet domain. This technique is similar to Fourier based volume rendering and also suffers from the same problems, like depth cueing and occlusion. But it can also be used together with other techniques like ray-casting or splatting [GDH97].
In the following sections, first the used wavelet method for the cubic lattice is discussed followed by the BCC lattice. After this the Run-Length-Encoding algorithm is explained and it is shown how the volumes are stored and organized.

4.3.1 Wavelets for Cubic Grids

Wavelets were already discussed in detail in Chapter 3.3. The goal of this and the following section is to highlight specific problems for the decomposition of volumetric data sets. This section discusses in detail volumes which are sampled in Cartesian space, while the next section covers the volumes which are sampled on the BCC lattice. Time-varying data sets which are resampled to the D_4^* lattice are also decomposed using regular Cartesian wavelets, because the single volumes for each time frame are sampled into Cartesian space. See Section 4.1.2 in this Chapter for more details.
Data sets which consist of more than one volume are decomposed separately frame by frame. Thus, only 3D volumes need to be considered. In Cartesian space, one has three axes of symmetry. As discussed in the introductory section in Chapter 3 data sets which are two- or higher-dimensional can be decomposed in two ways, using either separable or non-separable filters. Linear separable filters which are easier to use, but can introduce some directional artifacts, because the data set is decomposed in a pre defined

direction. Here, non-separable filters which up- and down-sample all axes simultaneously yield better results. But these filters are more complex to use and computationally more expensive.
Another important quality are the used wavelet decomposition method and the filter used. In order to guarantee the possibility of real lossless compression, the integer wavelet decomposition using the lifting scheme was chosen for this implementation. The filters used are the simple Haar wavelet as well as several variations of the Cohen-Daubechies-Feauveau wavelet filters [CDF92] which shall be discussed here briefly. For the final implementation, the Waili package [UVWJ+98] was used.
The integer wavelet transform (IWT) is a particular wavelet transform that maps integers to integers in order to obtain a lossless decomposition. The theory of IWT was developed after Cohen, Daubechies and Feauveau introduced the bi-orthogonal basis for perfect reconstruction filters [CDF92]. Bi-orthogonal basis functions are an alternative to orthogonal basis for decomposition. By removing the orthogonality constraint, they give flexibility to design filters that can be used for integer wavelet transform.
Bi-orthogonal wavelets are a family of wavelets that are obtained when the dilation equation is applied on two different scaling functions. This results in two sets of basis functions Φ_n and $\tilde{\Phi}_n$ which are bi-orthogonal. Therefore one can decompose/reconstruct a signal as:

$$f = \sum < f,\ \Phi_n > \tilde{\Phi}_n \tag{4.25}$$

The derivation of wavelet basis are similar. The integer transform is taken from the fact that using the appropriate bi-orthogonal filters, one can round the coefficients after the analysis step, but still can reconstruct the original signal by reverting the operations in the synthesis step.
In addition to Haar wavelets, also CDF filters with higher vanishing moments were used. CDF filters are the optimal bi-orthogonal wavelet filters in terms of number of vanishing moments. The vanishing moments correspond to smoothness of the filter response in frequency domain, or in other words, if the wavelet has p vanishing moments, the low pass filter and its $p-1$ derivations aere zero at $\omega = \pi$. The higher the vanishing moment are, the higher is the quality of the filtered images.
However, by increasing the number of vanishing moments, the spatial support of the filters increases. This means that singularities in the signal affect a larger number of coefficients. The image quality rises, but the compression ratio drops down. Comparisons and image examples can be seen in Chapter 6. If $p, \tilde{p}$ correspond to vanishing moments of the primal and dual wavelets, the CDF proved that any bi-orthogonal wavelet has a support of at least $p + \tilde{p} - 1$. CDF wavelets achieve this lowest support and are therefore optimal in the tradeoff of quality and compression. In conclusion, they are the best choice for the wavelet compression in the chosen approach.

The wavelet decomposition depth and the used compression ratio are different for each brick and depend on the brick dimensions as well as the importance of this brick, Section 4.2. After the volume is decomposed and the detail coefficients are thresholded the last low resolution volume is stored uncompressed on the disk and all high frequency data is encoded using RLE, section 4.3.3.

4.3.2 Wavelets for BCC Grids

While the last section explained the decomposition of wavelets for the Cartesian data sets, this section explains what problems can occur when creating a multiresolution representation of a volumetric data set that is sampled onto the BCC lattice. These hexagonal wavelets are only needed for 3D BCC data sets. Time-varying hexagonal data sets are sampled to the D_4^* lattice which single time frames are sampled to Cartesian space. The BCC lattice has, in contrast to the 3D CC lattice, nine axes of symmetry. This makes it more difficult for the use of linear separable filters, as one has to chose three out of nine axes. Because of these preferred axes, one would introduce directional artifacts in the up- and downsampled versions of the data set. The possibilities for filtering BCC data sets are:

- linear separable filters on three chosen directions,
- linear separable filters applied independently on each CC volume,
- linear separable filters for the entire BCC volume,
- butterfly interpolation scheme for 3D BCC, or
- the use of non-separable filters.

The easiest solution would be to use linear separable filters and perform either of the first three methods. Then simply Cartesian wavelets can be used with a different indexing scheme. However, one would introduce some artifacts, due to the different sampling density on selected axes. The most artifacts will occur when three axes are arbitrarily chosen for the up- and down-sampling. Better results can be achieved when the two CC volumes are treated independently and standard wavelet filters are used for the three axes. The disadvantage here is that the direct neighbouring points can not be used as they are *sampled* into the second cube. Here the most simple approach would be to treat the BCC lattice as one CC lattice. Every odd row is shifted by $\frac{1}{2}$ in x and y and one also uses a denser sampling in the z direction. If either of the linear separable filters is used, all wavelet filters from the Cartesian space can directly be applied to BCC data sets.
For the implementation the third approach was chosen because it allowed the use of existing wavelet filters which were discussed in the previous section.
For better image quality non separable filters or a butterfly interpolation

scheme can be used. The butterfly interpolation [DL90] is used to avoid directional artifacts by still using linear separable wavelet filters. Here one uses several axes for the interpolation to avoid artifacts in the lower or higher resolution data set. Hexagonal wavelets are used in medical imaging for image processing on 2-dimensional images [Per96] [SL98]. Here the butterfly scheme is used in two dimensions to avoid artifacts due to a preferred direction.

4.3.3 Encoding and Storage

Once the bricks are decomposed into high and low frequencies, most of the resulting data can be encoded in a more efficient way. The wavelet decomposition is only performed two or four levels deep, depending on the original brick size. The last low frequency volume is not encoded and directly saved to the disk. All detail information which is needed to reconstruct the original resolution is thresholded to achieve a better compression ratio. Then these high frequency volumes are encoded using the run length encoding mechanism and saved to the disk. The used RLE algorithm is the same one as described in Chapter 3.3.1. In the current implementation only one direction is used for encoding. As a 3D volume could be traversed in three possible directions, an eventually better path for encoding could be found and stored with two additional bits.
If the data is loaded, either from the beginning or from selecting a new time frame, the lowest resolution volume, which was not compressed, is read in and directly used as texture to achieve a fast response time. The texture is first loaded as simple density texture without normal information. If shading is desired and the normals are not pre-computed, then they are computed in a different thread with lower importance than the rendering thread. Also the next texture resolution is computed using an independent thread. Once the computation of the gradients and the next texture resolution is completed, the texture is updated and the two threads are terminated. Then the next texture/gradient level is reconstructed until the required Level-of-Detail is reached.
If the normals were computed in pre-processing, then they are downsampled to the current texture size and loaded as RGBA texture where the RGB values represent the gradient in x, y and z.
All the detail coefficients from one resolution level can be encoded into one file. The last low resolution volume is not encoded for faster access. Depending on the number of wavelet levels a brick is decomposed to, there are $n + 1$ files to store per brick and per time frame. This can result in a huge number of files, depending on the data set size, the resolution, and the number of time frames needed, but this is necessary to guarantee that each brick has access to its data immediately. Because the algorithm accesses

only at one point a data file, a better solution can be used where all data, high and low frequency is stored in one file and accessed via data pointers. This would result in one file per brick and time step.
Also, one additional header file is stored for the entire volume which contains all the information that is needed to initialize the bricks and to setup the rendering properties. In the initialization step, for each brick the smallest resolution level, i.e. the low resolution volume of the last wavelet decomposition, is loaded and rendered. Then the update mechanism, as described earlier in this section, starts until the required resolution level for all bricks is reached.
To keep the entire data set reconstructed in main memory can require a lot of space and might not be feasible on smaller systems. An alternative would be to reconstruct each brick before rendering. This would increase the rendering time, but one would be able to use this method also on smaller computers with less main memory. One of the largest data sets for the visible human male ($1700 \times 950 \times 1877$) requires 3 GB of space for the density data alone. If shading is needed, then this data set would require uncompressed roughly 12 GB for the maximum resolution. Using the BCC lattice one would *only* have to store 8.5 or 2.1 GB of data which is still too much for most computer systems. Smart subdivision and different resolutions for certain parts will also decrease the memory costs, but this might still be too big for most workstations. Here a compromise would be to only store the second resolution level in main memory and then reconstruct the higher resolution on the fly if necessary.
If the data set is dynamic, i.e. several volumes exist over time, then before the new texture is loaded and reconstructed the difference between the two time frames is analyzed using a previously computed error metric, Section 4.2.3. If both time frames vary only in minor differences, the old texture might be kept and further used. Only if the differences are noticeable, the old texture is discarded and the new texture is reconstructed.

4.4 Volume Visualization using Texture Mapping

The final step in the visualization pipeline is the rendering of the volume data onto the screen. Besides the many techniques available, volume rendering is still a challenging field, especially when dealing with huge time-varying data sets. As described in Chapter 3.1., some methods are software based and some are accelerated through special hardware. The software techniques are usually more accurate, but slower then the hardware accelerated methods. The most fundamental operation in volume rendering is the sampling of the volumetric data set to evaluate the volume rendering integral, Equation 4.26.
The data which is already sampled on a discrete grid, either regular CC

or hexagonal BCC, needs to be re-sampled onto another discrete grid, depending on the viewing direction. These resampling locations as well as the interpolation kernel have to be chosen carefully in order to achieve high quality images.
All direct volume rendering algorithms evaluate or approximate the volume rendering integral:

$$C = \int_0^D c(s(\vec{x}(t)))e^{-\int_0^t \tau(s(\vec{x}(t')))dt'}dt \tag{4.26}$$

In this equation C is the colour for a pixel which shoots the ray $\vec{x}$ into the volume and which is integrated along $\int_0^D$. Here D is the farthest point that will be sampled by this ray. The colour at a certain position is determined by using the current colour at the current place $c(s(\vec{x}(t)))$ which is integrated by the absorbtion term to the position of emission $e^{-\int_0^t \tau(s(\vec{x}(t')))dt'}$.
Ray casting, as described in Chapter 3.3 is a software implementation that simply evaluates this integral and traverses the volume from front to back. While ray casting is software based and not available for interactive exploration of huge data sets, a hardware based solution was chosen for this implementation. Alpha blending, which is used for texture accelerated volume rendering, approximates the volume rendering integral for all rendered slices:

$$C_i' = C_i + (1 - A_i)C_{i+1}'. \tag{4.27}$$

Alpha blending is performed from back to front and a new colour C_i' is computed by using the colour from the previous slice C_{i+1}' and multiplying it by the current alpha value A_i and adding the current voxel colour C_i. Hence the blending function which is used for compositing can be described as *(GL_ONE, GL_ONE_MINUS_SRC_ALPHA)*. See also Chapter 7.3 for a more thorough discussion of the actual implementation.
In volume rendering using texture mapping hardware the volume data is first loaded into the texture memory. Slicing planes are successively arranged through the volume on which the volume data, i.e. the texture, is sampled. These slices are rendered from back to front and composed through alpha blending [CCF94].
Two major algorithms exist for volume rendering using texture mapping hardware. One uses three stacks of 2D textures and six predefined sets of slices, Figure 4.11, the other one uses 3D textures and view aligned slices, Figure 4.12. When using the 2D textures, only the stack which is most perpendicular to the viewing direction is used, and is switched when the angle between the camera and normal of the slices exceeds 45°. Figure 4.11 [HKERS02] demonstrates the principle when using 2D textures: The disadvantage for 2D textures is that one needs three times as much data because three copies of the texture are needed for either (x, y), (x, z) or (y, z) slicing planes. Artifacts are introduced due to slices that are not

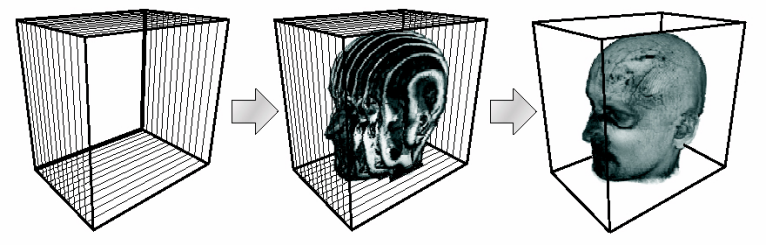

Figure 4.11: Volume rendering with 2D textures

aligned with the viewport. These artifacts are most visible when the angle between the slices and the viewport is greatest. Also popping artifacts occur when switching from one set of textures to the next one. However, the big advantage is that 2D textures are available on all 3D graphics cards in today's computers.
The next step for volume rendering using graphics hardware is the use of 3D textures. When using 3D textures, the slices can be laid arbitrary in the volume and hence be always parallel to the viewport. This approach mimics the ray-casting algorithm when alpha blending is used. The hardware can perform tri-linear interpolation within the volume for each location where the volume is resampled. Figure 4.12 [HKERS02] demonstrates the use of 3D textures in volume rendering: The volume rendering pipeline starts

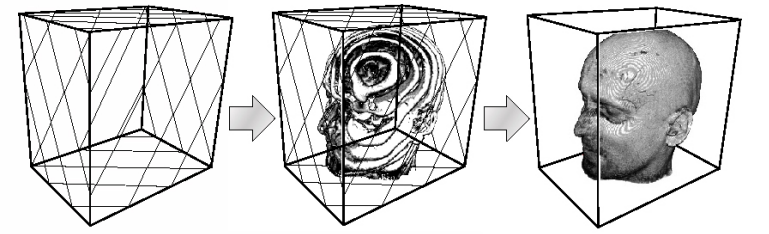

Figure 4.12: Volume rendering with 3D textures

with loading the data into the texture memory. If shading is needed, the gradient for each voxel is stored in the Red, Green and Blue component of the texture. With the intensity stored in the Alpha component, the final texture is defined as:

$$Tex_{3D} = (N_x, N_y, N_z, I) \tag{4.28}$$

Now, this texture is loaded into the texture memory and resampled where the slicing planes cut through the data. Depending on the classification used the gradient as well as the intensity are used to determine the colour and the opacity. Various methods can be used which classify the data using the given transfer functions. Shading techniques can be applied to further enhance the visualization. A more detailed discussion on this topic can be

found in Section 4.4.6.
Other blending functions allow to simulate different rendering techniques. The most common one as described earlier in this Section is alpha blending. Different blending equations can be chosen to simulate x-ray images or the maximum-intensity projection technique. The alpha test can be used to simulate non-polygonal iso-surfaces.
To interact with the data, i.e. rotate, zoom or translate, the texture matrix stack is used. Every texture unit has its own matrix which can be used to apply matrix transformations like rotation. Before the texture is resampled, the texture is re-oriented and then sliced parallel to the viewing plane. The advantage of this method is that the position of the sample slices does not change and that they can be reused for every frame if the sampling distance remains the same. The texture unit either repeats or clamps the texture at its border. In order to have only one copy of the volume, clipping planes have to be used to clip off the ghosts.
To reduce the texture size, the S3TC compression algorithm which is supported in hardware can be used [Bro00]. The compression ratio is 4:1 for RGBα textures. Nevertheless, this compression technique is lossy and the artifacts are visible, especially when storing gradient information in the RGB channel. But this technique can be used in addition to LoD when high frame rates are necessary, for instance when interacting with the data. This compression can also be used to extend the limitations which are set by the available texture memory. Now data sets up to a size of 512^3 can be used with gradient information without bricking.
When the bricking technique is used to render data sets which do not fit into the available texture memory, neighbouring bricks have to be loaded with overlapping textures. Because back to front compositing is used, all bricks have to be sorted before rendering. More detailed information about bricking can be found in Section 4.2.1.
Additional clipping with arbitrarily shaped clipping volumes can be easily performed in hardware and is a valuable addition for the rendering algorithm, see Chapter 5.3.5. The clipping can be either performed using a clipping volume or on a per-fragment basis [WEE02].
While the interpolation is an important step in order to reconstruct the original signal, tri-linear interpolation produces visible artifact. Even though tri-linear interpolation works fine for most data sets, higher order interpolation filters can be used, even in hardware to achieve better resampling results and higher image quality. Hadwiger et.al. [HTHG01] implemented a technique which allows to use arbitrary reconstruction kernel to resample the volume.
A disadvantage for using 3D textures is that they still have a limited availability, even though there are becoming available on more consumer graphics hardware from $n$vidia [$n$vi01] and ATI [ATi01]. But sooner or later 3D textures will be available on most graphics workstations. Reasons for choosing

3D textures for the volume rendering task were their increasing availability as well as the good image quality and the high performance rendering.
The following sections are dedicated to different parts of the rendering pipeline in more detail. It starts with a description on how to slice BCC data sets and which changes had to be applied to the original algorithm. Section 4.1.2 explains in theory how visibility determination can be utilized to speed up the rendering of huge data sets. The next sections describe how Level-Of-Detail is used and how the visualization of time-varying data can be performed as well as how non-polygonal iso-surfaces are *extracted* and visualized. Then classification and shading, most important for every volume rendering technique, are discussed and at last, an overview and comparison of different proxy geometries is given in order to render perspective images.

4.4.1 Slicing BCC Grids

Unfortunately current OpenGL implementations do not yet support the BCC lattice in hardware. This might change in the future when the game industry realizes the usefulness of the BCC lattice. The implementation should not be difficult as it would be just a matter of indexing which could be as easy as simply linking two cubic textures together. The advantage of the BCC grid in computer graphic is not only that it allows to store the information more efficiently, but also fewer operations have to be performed because fewer samples are used. This additionally decreases the processing time while still having the same image quality.
The BCC grid which is either given by itself or which originates by slicing a D_4^* data set can be understood as two interleaving CC grids which are offset by $\frac{1}{2}$ in x, y and z. Hence the easiest way to render BCC grids using current texture mapping hardware is to use multitexturing and two texture units for the two CC data sets. Figure 4.13 explains the principle. Here the BCC grid is *decomposed* into two regular cubic grids and for each data set one texture unit with its own texture matrix stack is used. Each texture is then loaded into either texture unit one or texture unit two. The *outer* cube is sliced in the same way as it was described for regular Cartesian grids in the beginning of this Chapter. The second cube is translated in texture space by $\frac{1}{2}$ in all three directions. For both texture units 3D textures are enabled and alpha blending is activated. During rendering, or sampling, these two textures are sliced and blended together on the same slicing plane using multitexturing and register combiner.
One problem is that the textures are clipped in polygonal space, dotted lines in Figure 4.13, using the six clipping planes OpenGL supplies. This works very well for the one texture that is aligned with these slicing planes. The other texture, that basically builds the inner cell points, Figure **??**, can not be sliced properly and interpolation artifacts will occur. If both textures

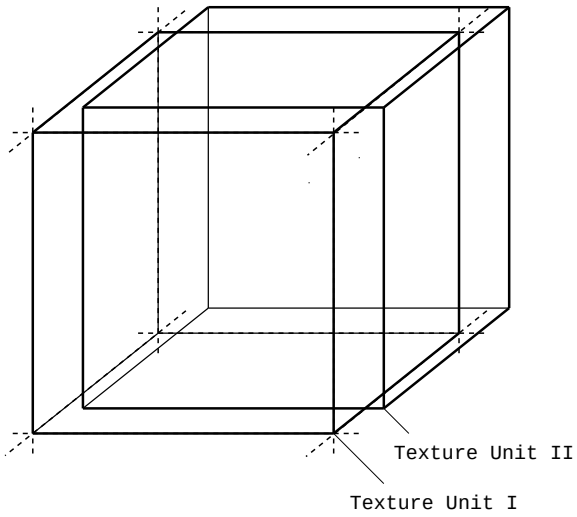

Figure 4.13: Volume rendering of a BCC lattice

are of the same size, then the samples on three sides of the *inner* texture are not used, but they are used for interpolation. Because there are not enough samples on the other side, the texture must be clamped to reduce the artifacts to a minimum. These artifacts have a constant texture space length which is $\frac{1}{2}$ a voxel in size. The other problem is that when resampling

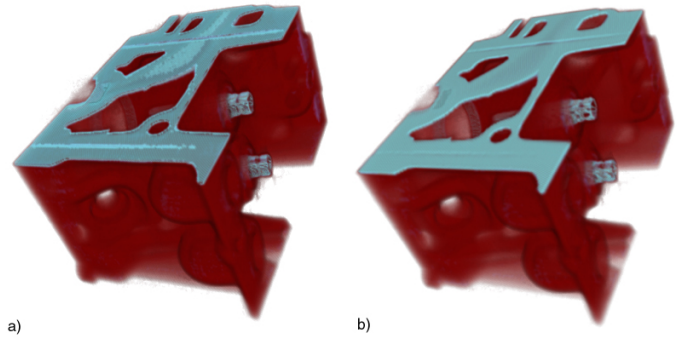

Figure 4.14: Comparison CC lattice a), and BCC lattice b)

the volume and interpolating on arbitrary slices, not all neighbours can be considered for the interpolation, because they are loaded into the other texture unit. But both will be sampled on the same slice and blended together. Each texture unit performs tri-linear interpolation for each of the cubic data sets. The forth linear interpolation is performed in the blending step when the two textures are blended together.
Figure 4.14 shows a comparison of the engine data set rendered on the CC

lattice 4.14 a) and on the BCC lattice 4.14 b).
Another drawback is that when using this approach, already two out of four available texture units are spend. This makes it more challenging for finding methods for good classification and shading within one single rendering pass. Sections 4.4.4 discusses this topic in more detail.

4.4.2 Visibility Determination

Even though hardware accelerated volume rendering is a very fast technique, the major bottleneck is the size of the available texture memory and the transfer of the texture to the graphics hardware. The bricking method allows one to render volumes that do not entirely fit into the texture memory. The price is that this technique uses several rendering passes, one for each brick. The most time is spent by transferring the data from main memory to the texture memory as well as for resampling and blending. The more slices, the more expensive. Hence the goal is to reduce the amount of data that has to be sent over the bus and also to decrease the number of texture loads to reduce the number of rendering passes needed to render the data set. These goals are partially fulfilled by the used bricking algorithm, see section 4.2, which segments the data set before rendering and merges similar bricks together. Additionally this bricking algorithm assigns a unique LoD to each brick which allows the use of a smaller texture and fewer slices.
The rendering time could be further decreased by adapting some techniques of visibility determination which are used in other volume rendering algorithms. In ray casting for instance the data is sampled from front to back by casting a ray through the volume. Here the colour and opacity values are accumulated to determine the final pixel colour. The ray is usually only traced until the opacity reaches a value around 0.95. The ray traversal can be terminated at this point, because the missing voxels have a too small influence to contribute to the final pixel colour.
A similar technique could be used for volume rendering using texture mapping hardware. While visibility determination is not needed for small data sets where only one or a few bricks are used. As described in Section 4.2.1, the main bottleneck of the bricking algorithm is the transfer of the volume data to the texture memory. For huge volumes, like the visible human data set, a visibility determination for each brick prior the rendering would be very useful to decrease the rendering time. Even if only a few bricks do not need to be rendered, the increase in rendering performance would be noticeable.
For a simple visibility determination, first the bricks can be clipped against the viewing frustum. Only bricks which are at least partially in the viewing volume are further considered, all others are discarded. Next a low resolution version of the entire volume which remains in the texture memory

for the whole time is rendered in the back buffer. Now the depth and the alpha image from this volume are read back into main memory to be further processed using image processing techniques. While this is a very expensive task, maybe other ways can be found to perform everything in hardware or to only partially need to transfer these buffers back to main memory. The alpha image is analyzed to see if there is any value above 0.95. If so, then the minimal depth value for this *opaque region* is determined from the z-buffer. All remaining bricks are sorted and those who are closer then this minimal depth are not further processed, but classified as need to be rendered. All other bricks which are possibly behind an opaque region are further examined to check if they need to be rendered or not. This could be done projecting the shape of the brick onto the viewing plane to check what pixels it covers and to decide if this brick need to be rendered or not. Only the shape of the brick has to be projected, not the actual data. After all bricks are checked for their visibility, the remaining bricks are re-ordered, not sorted, as the order of rendering did not change, and then rendered from back to front as usual.
This technique could help to increase the rendering performance for huge data sets where the opacity transfer function allows opaque regions. If the opacity can not be higher than 0.95 then the second part of this method can be skipped, but the clipping against the viewing volume is always advantageous.

4.4.3 Level of Detail

Level of Detail is very important for rendering huge data sets at interactive rates. When rendering huge volumes using the bricking technique, then every frame each brick texture needs to be loaded from main memory into the texture memory. Even though the AGP bus is very fast, it is still too slow for transferring that many textures several times a second over the bus. Depending on the system bus speed, the peak for transferring data to the graphics card with AGP4x is roughly 1 GB per second. But this is just a theoretical value and impractical because the CPU, the memory and other system components interfere which reduces the bus speed.
Therefore one goal is to minimize the texture loads and the total amount of data that needs to be transferred to the graphics card. One point to achieve this is by pre-processing the data and checking for relevant data, see also Section 4.3. Most data sets can be reduced in size when using this method. It is important to not only reduce the amount of texture memory that is needed, but also to group textures with similar information together into larger bricks to reduce the number of texture loads.
Texture compression, like the S3TC [Bro00] algorithm which is supported in hardware can be used prior sending the data to texture memory to de-

crease the texture size in main and texture memory and the time it takes to transfer the texture to the graphics card. With the S3TC compression enabled, textures which originally do not fit in the texture memory because of their size might fit now and bricking is not necessary anymore. However, because this compression is not lossless, the image quality will degrade. The compression ratio is 4:1 for RGBα textures.
All these techniques are related to LoD, they can help to increase the overall performance of the rendering. Level-of-Detail means that during rendering it is checked how much a particular brick contributes to the image. Level-of-Detail can be used in two situations. First, bricks which are far away in the back of the data set and have a pixel to voxel ratio smaller than 0.5 can be rendered with a lower texture resolution. Also bricks which entirely represent a more homogenous region can be rendered with a lower texture resolution and less slices. This way the image quality might decrease slightly, but the performance increases. At each frame the necessary Level-of-Detail is recomputed for every brick. One factor for the LoD determination is the distance from the camera, the other factor is derived from the information which is actually stored in the brick. This importance information is precomputed for each brick (Section 4.2).
When a different level of detail is used, i.e. a texture with only a quarter resolution, then only half the slices are necessary to resample this data. Usually the number of slices which are needed to resample the signal is directly taken from the dimension of the data set, i.e. a 256^3 volume is sampled with 256 slices. The next lower resolution would be 128^3 and here one only needs 128 slices to sample the signal. Sometimes the data is oversampled and intermediate slices are used to increase the image quality. If the same opacity transfer function is used for the smaller resolution texture, then this transfer function has to be adjusted. The opacity when blending 128 slices have to be the same as with 256 slices. Also when different LoDs are used between neighbouring bricks, opacity artifacts can occur because of *overlapping* slices. Figures 4.15 a) and b) show the two possible cases that can occur. Figure 4.15 a) shows the case where a lower resolution volume is viewed through a higher resolution volume. The area which is marked is rendered with too much opacity. Figure 4.15 b) shows the opposite case. Here the marked area is too transparent. A solution to this problem was found by Weiler et.al. [WWH$^+$]. Usually downsampled images or volumes look smoother compared to the original volume when they are viewed in the same resolution as the original data set. This is due to the low-pass filter which was applied to the original volume to built the next lower resolution LLL volume. Because the human vision system is most sensitive to edges, image processing which is either used in a pre-processing step, or with future hardware in real-time can help to improve the image quality. Possible techniques for edge enhancing are unsharp masking and anisotropic diffusion [Loh98]. But these techniques have to be applied carefully in order to not

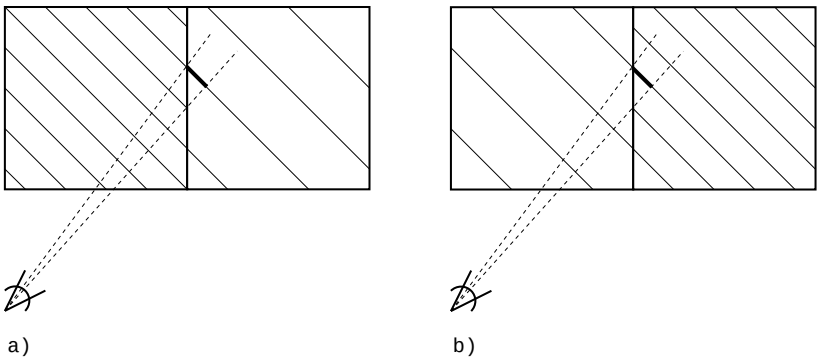

Figure 4.15: Level-of-Detail artifacts

change the bias of the volume.
The multiresolution representation of the volume is computed by using wavelets. The wavelet filters used are described in more detail in Sections 4.3.1 and 4.3.2.

4.4.4 Time-varying Volumes

Often volumetric data sets are only static and not varying over time. But some data sets, like the fuel cell simulation, are animated and several volumes exist. Section 5.4 and Section 5.6 in the next Chapter explain some basic principles as well as some more advanced possibilities to visualize the temporal component of such data sets. For this application the simple approach was chosen which allows use of a slider bar to change the current time frame. For volume rendering using texture hardware, this means that the current texture is discarded and a new one has to be loaded. Several optimization techniques can be used which shall be discussed shortly.
If the data sets are resampled to the D_4^* lattice, then $t \cdot \sqrt{2}$ CC time frames are created. Each of these time frames is only 0.35 of the original size of one time frame. This might vary because OpenGL requires the textures to be 2^n and sometimes these smaller volumes have to be padded with zeros. Now, one could either visualize each of these time frames separately or reconstruct a BCC grid out of the three volumes neighbouring in time. See Sections 4.1.3 and 4.1.4 for more details.
The bottleneck of hardware accelerated volume rendering is the transfer of the volume data into the texture memory. For quickly browsing through the single time frames, several optimization schemes can be used. One could simply have a low resolution version of each frame resident in the texture memory and when using the slider to move through time, the current volume is selected and displayed. If the user selects one time frame, then the high resolution version is loaded from disk and setup for rendering. The advantage is that this technique is highly interactive and allows a fast se-

lection of the volume data set in time. The disadvantage is that, depending on the data dimensions, quite a lot of texture memory is used to keep all the time frames in memory. Here, a better solution is to keep only a few time frames on the hardware. When moving through time the neighbouring volumes are blended together using multitexturing and register combiners. These volumes are low resolution representations of the different time frames and because for example only every 10th volume is used, one can save up to 90 percent of the data, compared to the technique before. Because these volumes are only used to specify a time frame, high resolution is not necessary and additionally the S3TC hardware texture compression [Bro00] can be used.

4.4.5 Iso-Surfaces

Besides direct volume rendering, another very important and often used method to visualize volumetric data is to generate iso-surfaces and therefore analyze the different layers of the data. As described in Chapter 3.1, iso-surfaces are usually computed by using the Marching Cubes algorithm [LC87] which builds a polygonal data set. The amount of triangles generated by the Marching Cubes algorithm can be very high, especially for large data sets. Also the computation of an iso-surface can be very time consuming. Figure 4.16 compares a polygonal and a non-polygonal iso-surface of the engine data set. While the iso-value of the non-polygonal iso-surface can be changed in realtime, the standard Marching Cubes version still takes considerably more time to extract the next one. The standard iso-surface

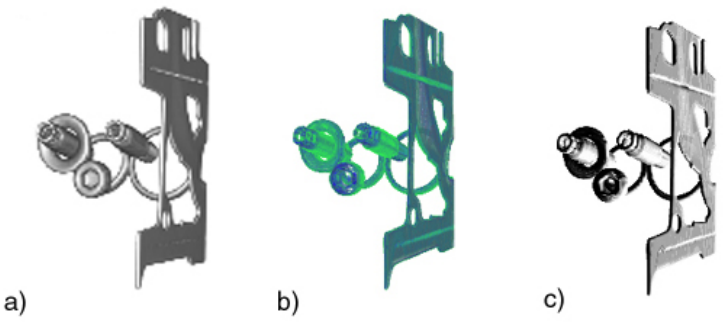

Figure 4.16: Polygonal, a), and non-polygonal iso-surface from the engine data, b) and c), of iso value 170

was generated and visualized using the Visualization Toolkit [SMLS98]. When texture mapping hardware is used for volume visualization, the *Alpha Test* can be used to allow only fragments which are above a certain threshold. This can be specified by setting the alpha test function to *GL_EQUAL*

which allows only fragments that have a certain iso-value to pass and to be rendered. These non-polygonal iso-surfaces can also be shaded. Blending using the *GL_BLEND* command has to be disabled in order to construct real iso-surfaces. However, in some cases the image quality is better when blending is enabled, Figure 4.16 b). The volume is still rendered from back to front, but the slices are now composited through the alpha test. In order to achieve satisfying images, the number of slices must be increased. Otherwise artifacts due to undersampling will occur. If *GL_GEQUAL* is used instead of *GL_EQUAL* for the depth test, the number of slices can be reduced considerably. Now all samples pass the depth test which are at or above the iso-value.
If these non-polygonal iso-surfaces are rendered without illumination and blending, the resulting image would just show the black silhouette of the iso-surface. Whereas shading and illumination can add local detail through normal vectors and show characteristics of the surface. The next section explains which techniques can be used for shading and classification for volume rendering based on texture mapping hardware.

4.4.6 Classification and Shading

Classification is a very important step for volume rendering in order to categorize the different regions in a data set. Here usually two different transfer functions are applied to the data set, one for colour mapping and another one to specify the opacity for each voxel, see also Chapter 3.2. Classification is used to visually group regions which belong together or to differentiate between diverse materials. Shading can be used to enhance the 3-dimensional impression of the visualization. We know shading from our every-day experience and use it to recognize shape information and also to conclude about material qualities of an object. Shading is well known from polygonal modelling and uses the phong illumination model:

$$I_{Phong} = I_a k_a O_a + \sum_i [((N \cdot L_i) I_i k_d O_d) + ((R \cdot V)^n k_s)] \tag{4.29}$$

The Phong illumination model can be split into three parts, the ambient, the diffuse and the specular term. The diffuse and the specular term are computed for every light source. In equation 4.29 k_a, k_d and k_s are the lighting coefficients for the ambient, the diffuse and the specular light. L is the light vector, N the normal vector , R the reflection vector and V is the view vector. O_a and O_d are the ambient and the diffuse object colours.
Shading can be incorporated into volume rendering in a variety of ways. In order to include shading, first the normal vector has to be computed for each voxel. The normal can be computed either during pre-processing or on the fly. The pre-processing is not recommended for huge data sets, because

this would increase the storage requirements by a factor of four. Several different methods to compute the gradient information have been proposed with different advantages in speed and accuracy. Because the gradient will be computed on-the-fly, a simple and effective method is needed. The method of central differences yields good results:

$$\begin{aligned} g_x &= \frac{f(x+\Delta x,y,z)-f(x-\Delta x,y,z)}{2\Delta x} \\ g_y &= \frac{f(x,y+\Delta y,z)-f(x,y-\Delta y,z)}{2\Delta y} \\ g_z &= \frac{f(x,y,z+\Delta z)-f(x,y,z-\Delta z)}{2\Delta z}. \end{aligned} \tag{4.30}$$

Here Δ usually is the size of one voxel. On the border of the volume the gradient has to be computed using forward- or backward differencing or by clamping the volume. While gradients have to be normalized before they can be used, the gradient length can be used to initially setup the opacity transfer function to create a gradient magnitude opacity transfer function [Lev88]. To avoid artifacts due to noisy data sets, the volume has to be smoothed prior to the gradient computation. Especially data sets from medical reconstructions are sometimes very noisy.
If the data set is too big to be kept in main memory for the whole time, the gradients can be computed for the next lower resolution volume and if the highest resolution is needed they can be simply upsampled using a linear interpolation technique. However, this would result in artifacts, because the interpolated gradient length would not be 1.0. Depending on the shading technique used this artifact will also occur from interpolation when resampling the volume. The gradients can also be compressed using wavelets and decompressed on-the-fly, depending on the needed resolution level, if the normals are computed in pre-processing.
Fake phong shading can be used on systems with very small texture memory where shading is needed. Here the gradients are pre-computed using Equation 4.30 and then quantized to a set of N predefined gradient vectors which are stored in a lookup table and accessed via an index. The resulting image quality depends on the total number of directions, and for $N = 256$ good results where achieved by Kilthau et. al. [KM01]. Because all gradients are quantized to one of N directions, the resulting image might have some directional aliasing. The creation of good normal vector quantizers is essential to minimize the impacts of this principal problem on image quality. The big advantage is that the texture is 50 percent smaller than using the approach described in Equation 4.28.
The classification in hardware accelerated volume rendering can be divided into two different groups. The first is pre-classification and the second one is called post-classification. The reference point for pre and post is the time

when the volume data is interpolated and resampled.
The simplest classification approach is to classify the volume before sending the texture to the texture memory and to load a $RGB\alpha$ volume as 3D texture. Here the volume is classified right in the beginning and then sampled. The disadvantage is that the interpolation is performed on this already classified volume. This results in coarser looking images. A slightly better, while more memory efficient approach is to just load the intensity volume and to classify while sampling the volume using OpenGL lookup tables. Here the two transfer functions for colour and opacity are combined and loaded as one 1-dimensional RGBα colour lookup table. These values are assigned to the texture before the volume is sampled. Here the intensity is just a pointer into the lookup table. The main advantage besides the memory efficiency is that now one can interactively change the lookup tables which results in the ability that the transfer functions can be changed in real-time without re-loading the entire volume. The disadvantage is that the volume is still pre-classified which can be seen in the image quality. Figure 4.17 shows two data sets which where classified using OpenGL lookup tables. Shading can

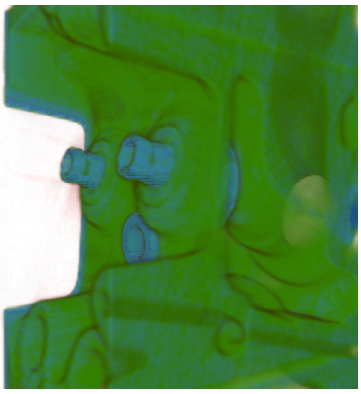

Figure 4.17: Classification using lookup tables

be performed in a variety of ways. In order to be able to use shading, the normals for each voxel have to be loaded with the texture. The texture is then loaded, as described in the beginning of this section as (N_x, N_y, N_z, I) where N_x, N_y and N_z represent the normal and I is the density value which can be pre-multiplied by the normal length to achieve gradient magnitude rendering. One method to incorporate shading is by using the OpenGL extension *texture_env_dot3* [PGG01]. This extension performs diffuse shading by using only one light source. While here the normals are interpolated during the resampling, there length is not necessarily 1.0. This causes artifacts which can be seen as bumps in the images. The register combiners can also be programmed directly to support more than one light source for diffuse and specular illumination [RSEB$^+$00]. Figure 4.18 shows two examples for classification using the gradient length. Better classification results

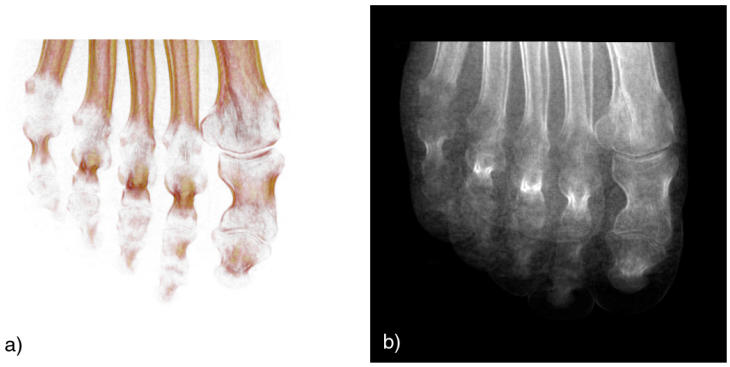

Figure 4.18: Gradient magnitude classification a), and X-Ray b)

can be achieved when post-classification is performed. Here the colour and the opacity values are looked up in a dependent texture after the volume is resampled at the slices. When using this technique, the data set is loaded as luminance texture. After the sampling, the interpolated density values are used as texture coordinates to lookup a 1-dimensional, previously computed transfer function [MHW99]. This results in higher image quality as can be seen in Figure 4.19: A further extension of this technique is pre-integration

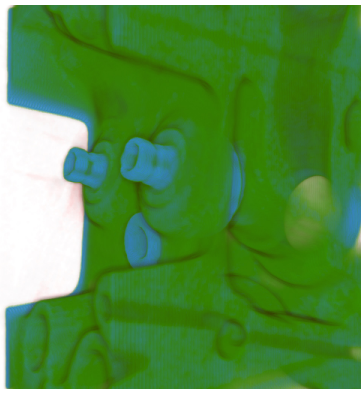

Figure 4.19: Post-classification

which samples the volume not only at the slicing planes, but also includes the values between two slices by pre-integrating them [EKE01]. This is done in a pre-processing step when the dependent texture is computed. During the rendering, the volume is sliced in slabs where each slices has a front- and back-texture which are also classified using a dependent texture lookup. Image artifacts which occur when the volumes is sampled with too few slicing planes are eliminated. The image quality which can be achieved using this method is very high. Another advantage is that this image quality can be produced with less slicing planes.

A disadvantage of all these techniques is that the normal vectors are still interpolated during the resampling. The used normals are not normalized which results in visible bumps on the volume surface. Also the light direction can not be changed when only dependent textures are used. Here a better solution is to employ cube mapping for shading [MGW02]. In this method, the first texture unit is used to sample the data, using either of the pre-classification methods. Then from the second to the forth texture unit the gradient is loaded and in the third texture unit a cubemap which is used for diffuse lighting is applied. In the forth texture unit a specular cube map is accessed by computing the reflection vector, the normal, which points into the cube map. The big advantage is that this method does not rely on normalized normals and hence perfect phong shading is used. Another benefit is that using cube maps as many light sources can be used without an increase in processing time. In addition, the light sources can be of any shape or colour and the cube maps only need to be recomputed when the relative distance between two lights changes. To include post-classification as well, either more texture units are needed, or the register combiners can be used to perform simple classification.
Cube maps can also be used for environment mapping and to specify a reflection map [HKERS02]. This can be used to highlight some material properties of the displayed object.
Shadows and self-shadowing can also be included into the rendering algorithm which increases the image quality and results in more realistic images. This can be done by computing a *new* shadowed volume in the frame buffer [BR98]. This volume has to be recomputed every time the light direction changes. Another method which extends the volumetric lighting model by including light attenuation to produce volumetric shadows and the subtle appearance of translucency [KPHE02].
All these techniques can add valuable contributions to increase the 3-dimensional impression of the volume rendered data set. Classification has to be performed in order to categorize the different regions. All other techniques, like shading, reflection mapping and shadowing increase the image quality, which can be valuable in order to extract interesting features. Also, the graphics hardware is still too limited to perform all of these techniques together in a single rendering pass. This is very important, especially for large volumes where bricking has to be used.

4.4.7 Proxy Geometry

Usually direct volume rendering is not related to any polygonal surfaces. But the only primitive that current graphics hardware can render is polygonal data. Because this hardware is so highly specialized, it can render up to several million polygons per second. This ability, together with some

more advanced texture mapping techniques can be *abused* for direct volume rendering. Here the 3-dimensional density field is sampled by the slicing planes that are laid in the volume. Along these slicing planes, the current texture is sampled and interpolated and all these slices are then blended together from back to front. The geometry where the data is resampled is called *proxy geometry*, because it only serves as a location where the volume can be sampled and it has no relation to the data itself. For 3D textures,

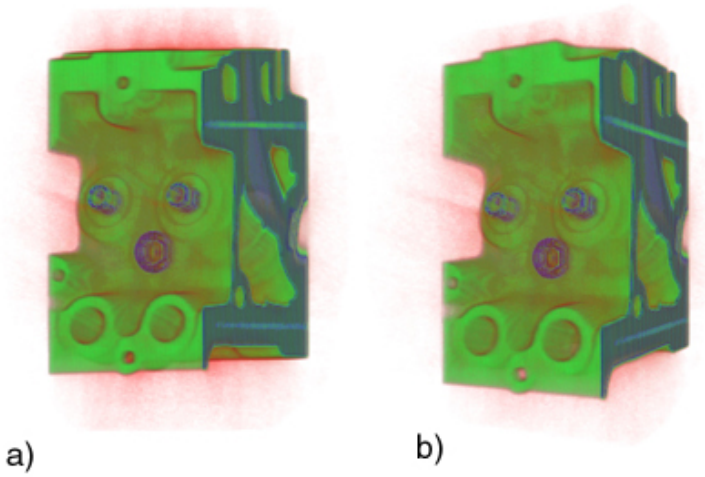

Figure 4.20: Orthographic, a), vs. perspective projection b)

virtually every polygonal structure can be used to sample the signal, but in order to minimize the slicing artifacts and to reduce the number of sampling planes, viewport aligned slices are prefered. The most commonly used ones are simple quads defined by 4 vertices. These work well for orthogonal projections. But for perspective projection, the distance between successive samples used to determine the colour of a single pixel is different from one pixel to the next one. Here the solution is to use spherical slicing shells [BGK+99] instead of planer slices. These spherical shells are spheres with the centre at the viewpoint and which are clipped against the view frustum. Now it is guaranteed that each pixel on this shell has the same constant sampling distance. The main drawback for using spherical shells is that they are more complicated to setup than simple planar slicing planes. But often artifacts which are due to planar slicing in perspective projection are hardly noticeable and planar slices might suffice for most applications.
Figure 4.20 compares a rendering of the engine data set with orthogonal projection (left) and perspective projection (right). For the perspective projection, only simple quads were used to resample the volume.

4.5 Conclusions

The visualization of scientific data sets is a difficult task. Here the appropriate techniques have to be chosen and setup in order to get quantitative results. This task becomes even more challenging for huge, time-varying data sets. The fuel cell data set is only one example, see Chapter 2. With the increase in computing power and finer instruments for measuring, simulations as well as data scans become larger and more difficult to handle. New techniques have to be developed in order to visualize these huge data sets interactively. The focus in this Chapter was to extend an existing rendering algorithm and to develop new ideas which can be used to achieve this goal. Section 4.1 reviewed the BCC lattice and explained the 4D counterpart. These lattices can be used to store the data sets more efficiently than the Cartesian lattice by simply resampling the data and adapting existing rendering algorithms. The 3-dimensional BCC lattice only needs 70 percent of the original samples, while the 4-dimensional D_4^* lattice can represent the entire data set with 50 percent samples less. For huge data sets 30 or even 50 percent less samples seems to be a lot, but in order to be able to really interact with the data in realtime, the remaining data has to be compressed. Before the data is compressed, the entire volume is divided into small bricks and each of them is analyzed to decide which Level-of-Detail is needed to display the inherent information (Section 4.2). This can vary from zero, the brick is discarded, to one which renders the brick in full resolution. Bricks with similar *importance* are grouped together. After this subdivision, each brick is compressed using either Cartesian or Hexagonal wavelets (Section 4.3) and the detail information is stored using an RLE scheme. Depending on the importance of the brick and the globally specified compression level, some of the detail coefficients are discarded prior to the encoding. Now, the pre-processing step is finished and the data can be rendered using texture mapping hardware (Section 4.4). Here, some principles of volume rendering using texture mapping hardware are explained as well as some more advanced features of current graphics accelerators.
A short summary of the developed algorithm could be described as:

$$\begin{aligned} &\textit{A new Lattice} + \textit{Coherency} + \textit{Compression} + \textit{Hardware} \\ &= \textit{Fast Visualization} \end{aligned}$$

These techniques allow to store the volumetric data sets more efficiently on hard disk and because fewer samples are used, the rendering time increases as well. A more detailed conclusion and also the achieved results are presented in Chapter 6. Chapter 6 shows conclusions and explains the results from this Chapter but also the ones from the next Chapter in more detail. The results are enhanced by images as well as some numbers if applicable. In the end of Chapter 6 some conclusions are drawn and the usefulness of each technique is evaluated.

Chapter 5

Multiparameter Visualization

In the last Chapter some methods and techniques for fast volume rendering of huge time-varying data sets were developed. These methods work well for unimodal data sets where only one scalar value, i.e. density or attenuation, is given for each voxel. But some data sets have additional properties and can consist of several scalar and vector values for each sample point. Data sets which have more then one scalar value are called multimodal and are more difficult to visualize than unimodal data sets. Some of the techniques which can be used for unimodal data sets are also applicable for multiparameter data sets. A huge problem for the display is the high density of information which has to be displayed without cluttering the visualization. This Chapter will give a short summary on how to efficiently work with such data sets and also provide a theoretical overview of possible visualization techniques available for multiparameter data sets in general. In the following sections one can find some well known methods as well as some new techniques to visualize multivariate data sets. Some selected methods are implemented to prove their applicability. They are described in more detail in the single sections of this chapter, as well as in Chapter 7 where some implementation details are presented.
The accompanying example for the different techniques throughout this Chapter is the fuel cell data set from Chapter 2.2.1. The used fuel cell data set is relatively small in size ($14 \times 15 \times 100$) compared to the final resolution ($250 \times 250 \times 1000 \times t$) which will also vary over time t. Unfortunately the algorithms and the technology to generate these huge data sets are not available yet. It already took over four days on a large multi-processor computer to create the small, static simulation above. Computer scientists and mathematicians are currently working on improvements to the algorithms, but until then, the small data set can be used instead. All of the presented

techniques can be used in general with other data sets as well, but most methods will be described using the fuel cell as an example.
By recall from Chapter 2.2.1, the fuel cell data set consists of several scalar data values and one vector value:

- concentration of oxygen O_2,
- concentration of hydrogen H_2,
- pressure p,
- temperature T,
- velocity (vx, vy, vz), and
- concentration of water H_2O.

The next section starts with the definition of multimodal data sets and explains some basic properties and how they impact the form of visualization. The following two sections describes some obvious techniques and how standard visualization tools can be adapted to support these methods (Sections 5.2 and 5.3). Sometimes multimodal data sets can also vary over time. Here, Section 5.4 gives some examples on how to deal with these special cases. The third last section explains why it is important to give context information during the visualization and shows some techniques useful to focus on some specific parts in the data only (Section 5.5). Multimodal data or time-varying data can also be seen as data sets in higher dimensions. The next section demonstrates and discusses some methods which can be used to visualize higher dimensional data sets (Section 5.6). The last section of this chapter, Section 5.7, employs non-photorealistic rendering techniques to picture properties of multiparameter data sets and to aid in familiar experiences for understanding this data.

5.1 Terms and Definitions

In this section some vocabulary will be introduced and the main ideas behind multiparameter visualization are explained. The term *multi* means more than one and multiparameter or multimodal simply means that the given data set contains more than one dependent data value for a single sampling point. Important for the visualization task are three questions [SM00]:

- What is the data range of the variables?
- What is their spatial reference?
- And what is their time(-varying) aspect or reference?

The range and the type of the data set describes which possible values a variable can take. This information is also used for the design of the transfer function which maps the information onto a graphical primitive which can be displayed on the screen. The spatial reference, if the data has one, shows

how the data is organized and structured in $\Re^n$. The local frame of reference is part of the observation space. To minimize loss of data when mapping from one space to another, it is necessary that the number of features as well as the number of data sets remain constant. When projecting from a higher space to a lower space ambiguities, in the data representation can occur.
Besides the proper visualization of the local frame of reference, also the sphere of influence for each data point is important. The influence can be either:

- point based,
- local (neighbourhood), or
- global (for the whole spatial reference system).

The sphere of influence has a huge impact of the type of visualization. The mapping can be either point based, like direct volume rendering, line oriented, like stream- or streaklines, or global using surface elements. The fuel cell simulation has (strictly speaking) a point based influence, but is sampled on a regular grid at discrete points and therefore has a more local sphere of influence. However, both types of visualization, local and point based are applicable for the visualization of this data. On the other hand, the vector data set which shows the flow information of the gases has a local reference and here line based visualization techniques, like streamlines, are applicable.
A common pitfall in multiparameter visualization is that the huge number of graphical primitives can clutter the image and make the visualization meaningless. One goal therefore is to find suitable representations which on the one hand can represent the structure and the information contained in the data, and on the other hand allow easy interpretation of the visualization and navigation through different data sets. This is not a trivial task and the methods and techniques described in this Chapter address this problem.

5.2 Classic Techniques

Probably the easiest way to visualize multi-parameter data sets is to simply visualize only one parameter/data set at a time. When using these classic techniques, the existing visualization methods for unimodal data sets can be used directly. The programs need only a minor modification so that the user is able to select the current data set and time frame. Everything else stays the same: the user has still the possibility to interact with the visualization as well as to change some display specific parameters. Figure 5.1 shows a screenshot displaying the fuel cell data set. Here the temperature data is selected and direct volume rendered using alpha blending. A simple extension to this approach is to choose different visualization techniques for each parameter or a special colour coding scheme to have more than one parameter displayed in a given visualization. Figure 5.2 shows an example for this

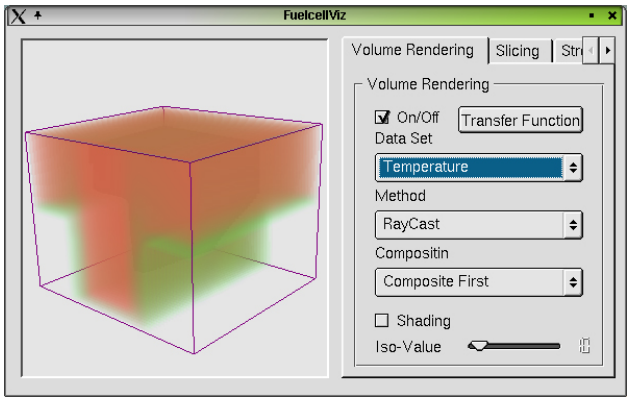

Figure 5.1: Simple *multi*parameter visualization

more complex multiparameter visualization. Here the oxygen concentration data set is blended together with the flow information. This technique is also known as the *Layer Concept* and heavily used in cartography where different sets of layers are blended together, like streets, vegetation or height-isolines. However, these techniques are limited to a few parameters only. If too many

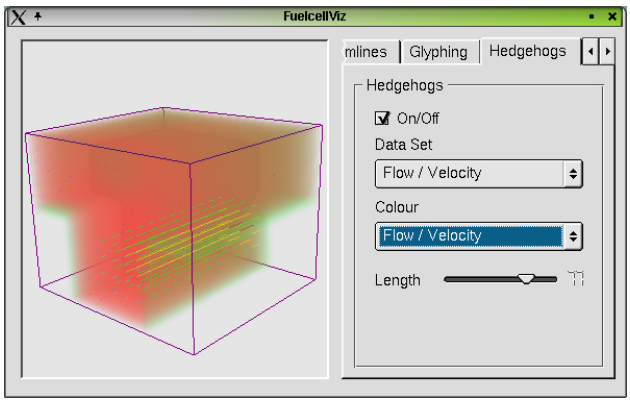

Figure 5.2: Multiparameter visualization with two data sets

different techniques are used or too many data sets are visualized, the resulting visualization can be too cluttered and eventually even misleading. Some more advanced techniques can be applied to circumvent or to reduce these problems. The next section gives some ideas and some examples in classic multiparameter techniques which can be useful for visualizing the fuel cell data set.

5.3 Multiparameter Techniques

While the last section presented only a simple extension to the unimodal data visualization, this section covers some more advanced methods and techniques to handle multiparameter data sets. First, when speaking about multiparameter visualization, one needs to think about what the important information is. Often some parameters are correlated and do not need to be visualized *twice* if one value can be derived from another one. This is dependent on the given data set and needs to be decided prior to visualization. The fuel cell data set has correlation to some degree but not enough to skip one of the data sets entirely. Figure 5.3 shows two iso-surfaces blended

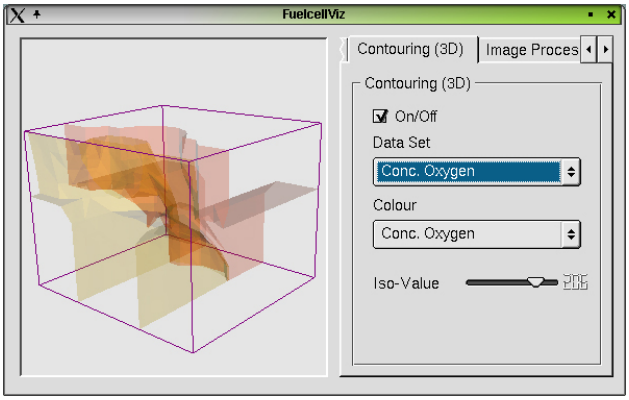

Figure 5.3: Two iso-surfaces, yellow - oxygen, red - hydrogen

together. The yellowish one shows the oxygen concentration, the reddish one the concentration of hydrogen. Both iso-surfaces are similar, but one is pointing in the direction of the flow, and the other is pointing the opposite way. Also the pressure is, as can be assumed, higher behind the tip of the flow (as is the reaction temperature). To conclude, correlation is applicable for the fuel cell data to some degree. One needs to decide what is a good mixture to visualize together. With all the correlation of the fuel cell data in mind, A useful selection might be to direct volume render the temperature together with an iso-surface of either the concentration of oxygen or hydrogen and extend this with hedgehogs or glyphs showing the flow information. Figure 5.4 depicts this combination. In this image the temperature visualizes the speed and the strength of the reaction and hence also shows where the highest concentration of water will be. The hedgehogs display the flow information of the gas mixture, and the gas concentration can be derived from the iso-surface which is mapped to oxygen. The pressure is highest where the temperature is high and behind the tip of the flow.
Also uncertainty information could be included to clarify that this data is sampled on a regular grid at a certain sampling distance and that no contin-

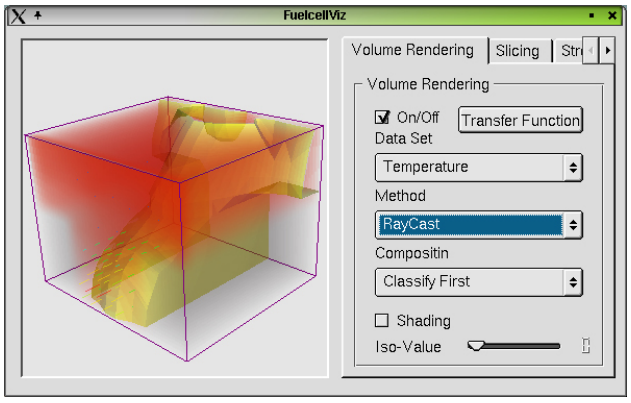

Figure 5.4: Selected fuel cell visualization

uous signal is visualized. Even though this is not a real application for the fuel cell data, it might be useful for downsampled data sets to enhance the loss of information there. It also allows the viewer some more freedom in concluding or formulating hypotheses about the displayed data. The human visual system is very good at integrating visible primitives together which makes also the field of NPR very interesting for these kind of techniques. See also section 5.7 for a more thorough discussion on this topic.

5.3.1 Scatterplots

Scatterplots have a long history in visualization. Their origin is mainly information visualization, where they were used to display multiparameter data sets. In principle they combine every parameter with one another to create different (panel-)images. From these images one can relate a set of parameters and also compare these information pieces to the global behaviour and the underlying nature of the data set. Most often scatterplots work on 2D images where two different 1-dimensional data sets are used. But they can also be extended to 3D or higher dimensions. In medical imaging they are used to split a 3D volume into three 2D image stacks that can be viewed sequentially. Figure 5.5shows such an application from medicine [Cor].
Here the 3D volume is visualized through a set of slices. In this example, only one set of slices is displayed, but all three (axial, coronal and sagittal) can be used together as can be seen in Figure 3.6 on page 19. Each set of slices shows as much 2D slices as possible for the given resolution. If they do not entirely fit on the screen, one can select different ones using a scrollbar [RMC+00].
This example can also be extended to higher dimensions, like displaying all time-frames of a volume that vary over time. One could also swap some axes in the display and show (x, y, t) volumes over z. This would show the changes over time of one slice in a volume. By moving through z, different

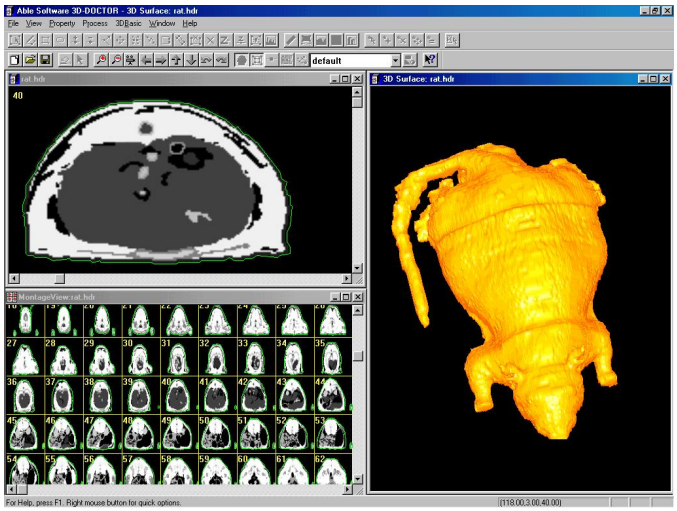

Figure 5.5: Screenshot from medical visualization software

slices are selected and displayed with the appropriate time information.
For the fuel cell data, scatterplots can be used as in medical imaging for simply slicing the volume data and displaying it along the coordinate planes. This might give more precise results in comparing one data set with another one. One could also easily go down one dimension further and slice the volumes along lines. Then two different volumes can be directly compared in one single image. The common factor that is used is the position of the data.

5.3.2 Hierarchy

The principle of hierarchical techniques is the arrangement of the data set into a hierarchical structure. The goal is to show global properties of the data set as well as some selected detail information. A hierarchy can be used in two different ways [SM00]:

- hierarchization in presentation space, and
- hierarchization in data space.

The first one creates a hierarchy in the presentation space by subdividing the 2- or 3-dimensional space into subspaces, while the second one establishes a hierarchy on different levels of the dependent variables.
Worlds-within-Worlds is the name which was used by [FB90] to subdivide the 3-dimensional representation space. This technique uses for every data point a local coordinate system to visualize more than one parameter. This method can be nested to a certain degree. For too many layers the visualization can easily get cluttered and disordered. It usually works with user

interaction to specify a point of interest, at which position the next layer is generated and displayed. Which geometric primitive works best to map the data in the local coordinate systems is dependent on the application and the data set. It can also include iso-surfaces and direct volume rendering if the last set of dimensions is not three. *Worlds-within-Worlds* are very effective when using stereoscopic devices in order to have a better understanding of the location where the tree branches. An example for the fuel cell data might be that the visualization starts with displaying the topology of a fuel cell, probably by showing a wireframe of the area where the chemical reaction takes place. With a 3D cursor, the user can interact with the data and display more information on interesting locations. Here classic visualization techniques, like streamlines, glyphs or iso-surfaces can be used.
A hierarchization in data space subdivides the data sets and builds a hierarchy depending on characteristics of the data values. One direct example is a multiresolution representation where first a course representation is displayed which can be refined by branching to the next level. Another example for the fuel cell data would be to start with the temperature data which indicates the magnitude of the reaction and then ramify to the concentrations of oxygen, hydrogen and water. Here for the rendering, also classic visualization techniques can be used.

5.3.3 Shadow Projection

A very intuitive and easy to implement technique for multiparameter visualization would be a modified shadow projection algorithm [KDG99]. Here the centre of the visualization is the context object, which provides one with global context information that all data sets have in common. This information can have a structural nature, like the silhouette or an iso-surface of the physical topology. It could also represent a mixture of some parameter channels, but it must provide a global understanding of the entire data set. Around this context visualization, several planes can be drawn, each one displaying a different parameter or data set. They could also depict all the same data set, but using different transfer functions or different visualization methods for each plane. The normals of the projection planes are inverted, so that the planes which are facing towards the viewer are not displayed. Figure 5.6 shows an example which explains the principle for the fuel cell data set. On the three sides visualizations of different data sets are shown, while the centre displays the oxygen concentration of iso-value 225. In this example the context object is an iso-surface of the physical structure of the fuel cell. A cube with inverted normals surrounds the visualization and density maps of the different parameters are projected onto the cube's inner faces. While rotating either the cube alone or together with the context object, different data sets will be diplayed and the projection angle changes.

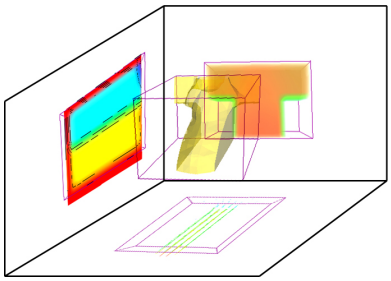

Figure 5.6: Modified shadow projection

To support the user's orientation while interacting with the visualization, a 3D cursor or icon would be helpful when rotating or moving the objects. A coordinate cross would rest anywhere in the context object while the projected 2D coordinate cross also marks its position in the projection space.
Originally, this technique was developed for shadow projections to enhance the 3D understanding and to better perceive the 3-dimensional structure of the visualized object in the 2D image.
A related approach is discussed in Section 5.5.2 where the focus is on detail and context visualization.

5.3.4 Probing

Probing is not a technique by itself, but often used as interaction method to let the user experience the data set. By probing the data sets, users can step through the data in whole or in part and receive more detail information at points of interest. The probe can be any geometric primitive which maps the data into visible form. One example of probing is in the hierarchical subdivision of the presentation space where the user has to specify at which location this subdivision should branch (see also section 5.3.2).
Another example is when the user can places a rake into a flow field to generate streamlines which originate from this rake and follow the flow field. By moving through the field, the user will get an understanding of the flow by integrating over the different visualizations. Other geometric primitives like glyphs can be used as well to map information from the current location into visible properties.
Another approach is described in the next section where a special lens is moved through the data set to evaluate some algorithms in specific regions. Additionally a 2- or 3-dimensional coordinate cross can be used to highlight the current position-of-interest in the data and to extracts detailed information at this position.

5.3.5 Special Lenses

As described in the previous section, *special lenses* can be anything from defining a region-of-interest (which is handled differently then the rest of the volume) to methods which extract and visualize information for the affected region. These lenses, sometimes also called magic lenses, or regions-of-interaction are for playing around with the data and evaluating some special *things* for these regions. These things can perfrom a wide variety of functions, such as:

- enhancing or suppressing a region,
- enhancing or suppressing some features,
- choosing a different style of visualization,
- blending together data sets or choosing a different data set,
- interacting with the region differently,
- cut out the region,
- render only this region,
- change parameters for the visualization, to
- change the Level-of-Detail/resolution, and more.

All these lenses are used to either enhance or suppress something or to add some additional information. These lenses/regions can be of any size or shape. They can be 2-dimensional for images and 3- or higher dimensional for 3D objects which might also vary over time or which have more than one parameter. A good choice is a sphere or a circle because of its isotropic character. An example for a simple lens would be a sphere which either

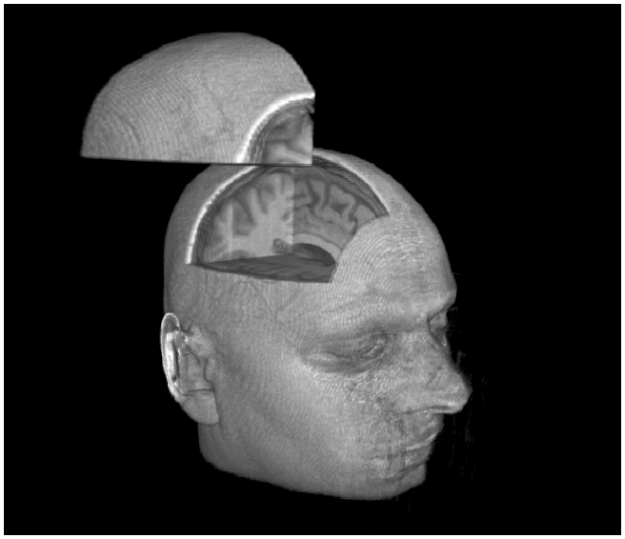

Figure 5.7: Interactive volume clipping

enhances or suppresses the entire selected region by changing the (opacity) transfer function. If the opacity in the region is set to zero, the lens can also be used to clip portions of the volume to look on details behind. An efficient implementation of this technique using OpenGL hardware is already available and was published by Weiskopf et. al. [WEE02]. Figure 5.7 shows an example of this technique.
A variation of this is when the region, or part of it, is rendered differently then the rest of the visualization. For instance the entire data set is rendered using direct volume rendering, but a mobile spherical region is displaying an iso-surface using a given threshold. This way the global volume is present using volume rendering and some mobile ROIs can be explored in a different rendering style [SHER99].
Important for multiparameter visualization is that these regions can also show different variables. For the fuel cell data, one could render the overall visualization using the temperature data set and have some additional mobile spherical regions which display the concentration of oxygen or the amount of water. These regions can now be moved around and to see values at different locations. Figure 5.8 shows an example for the fuel cell data set where an iso-surface for the concentration of oxygen is visualized together with a volume rendered sphere of the temperature data set which additionally shows some flow information. Some of these techniques are also

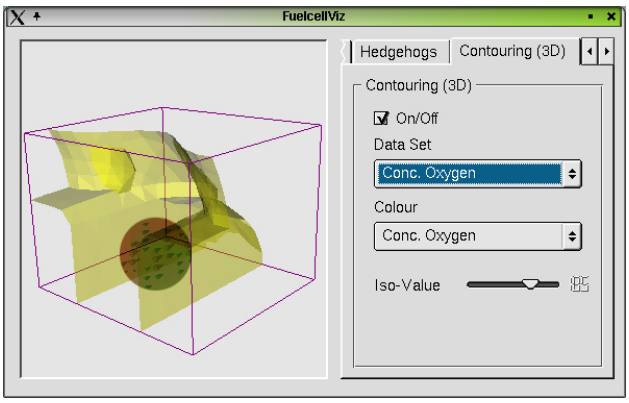

Figure 5.8: Interactive combination of different data sets

discussed in Section 5.5 *Focus and Context*. Another example is to use the magic lens as a *real* lens. Here the region can be distorted using fisheye views to magnify the region inside the lens centre while still displaying the rest of the data set. An example can be found in [CCF97].
For huge data sets, like the visible human, another method could also help to increase the rendering speed. If the entire volume is rendered with low resolution, the region inside the sphere-of-interest can be rendered in higher resolution and quality. In combination with an eye tracker this could be

very powerful and improve the overall performance. But even without, when pushing the sphere manually with a mouse the viewer's focus rests on the sphere, and hence only this region needs to be rendered in full resolution.

5.3.6 Customized Glyphs

Glyphs are 2- or 3-dimensional icons which can be placed inside the data set to visualize some information at a given point. The usage of glyphs is very popular for flow visualization where these icons are placed in the flow field and point into the direction of the flow. Other parameters can also be mapped to the icon like size or colour. Figure 5.9 shows an example of a dynamic SPECT kidney study where glyphs were used to visualize the flow inside the kidney [TRM+01]. The glyph, in this example a cone, is pointing into the flow direction and the colour and the size are mapped to the magnitude of the flow. In this image, only the left kidney is shown with the ureter and the aorta abdominalis. A problem with too many glyphs is that the visualization can get cluttered very fast and the resulting image will be meaningless. Taking several neighbouring glyphs together one can reassemble the global information contained in the flow field while still seeing details from each single glyph. Glyphs can also be specially designed to

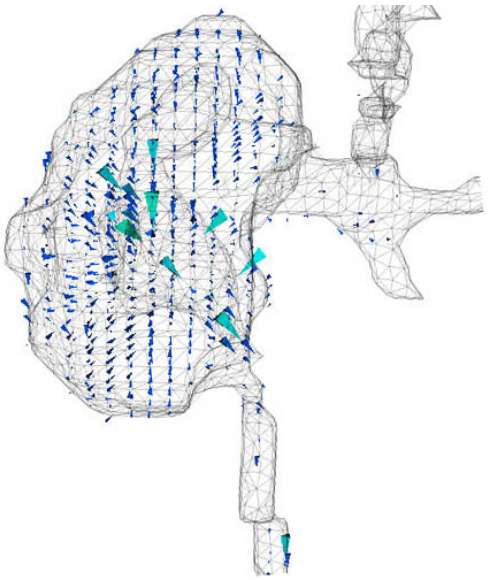

Figure 5.9: Glyphs in flow visualization

fulfill the specific visualization needs of a given data set. Here one needs to decide what information is important. This plays an important rule for multiparameter data sets where the visualization of lots of data does not necessarily makes sense.
For the fuel cell data set, we have six scalar values, including the vector

magnitude, and one vector for each data point. A simple customized glyph might be a 6-sided cone which would be able to represent all data values, but the visualization would be difficult to read and some information may not even be visible due to occlusion effects.
Classic glyphs include also the star shaped glyph which is used to present scalar information only, but it can also be adapted to display vector information. An example for these glyphs can be found in [SM00].
When vector information needs to be displayed, animating the glyph might help as well. Here recent studies have shown that a user is only capable of perceiving as many as five different sets of motion. Also, moving icons may distract too much, so the use of animating glyphs in a visualization needs to be carefully considered. Speedlines, lines behind the object that is in motion, and motion blur can help to depict motion in motionless pictures [MSS99].
The Jumping Jack Icon is a special kind of glyph where a stick or line figure is used as glyph [Srd]. The arms and legs can vary depending on the current data point. The big advantage of the Jumping Jack icon is that the viewer tends to assemble the icons together and global trends become immediately visible.

5.4 Time-Varying

In scientific visualization *time-varying* data sets can be divided into three groups. The first group contains data sets which are not time-varying and show only static information, but which might have a link to a specific point in time, e.g. a snapshot. Data sets from the second group are time-varying, but they are sampled only at discrete time steps and called semi-static. The third group represents data sets that are continuous over time and these are referred to as dynamic data sets. The most often used data sets are from the first and the second group. The third one is only available from continuous simulations or if one interpolates additional samples between two discreet time points. Even though the second one is not really dynamic, it is usually referred to as dynamic.
To visualize time-varying data sets (i.e. the cases two and three) one can use several well known methods which all have their advantages and disadvantages. Some of them are even applicable to static data sets. A frequently used method is animation. Here first an image using a specific visualization technique is created and saved. Then the next time frame is loaded and with the same technique another image is generated. The camera can remain at the same position for every time frame or can be animated and moved through the visualization space. All these single frames are then composed into an animation video. This simple technique can be varied in a number of ways. One is to alter the visualization technique or to just

vary some visualization specific parameters. The animation of parameters can also be used for static data sets. An example is the visualization of the different layers of a data set by animating and browsing through different iso-values. Other visualization parameters, like thresholds for streamline integration length, can be varied as well. Figure 5.10 demonstrates both techniques with three frames each. The first, Figure 5.10 a), is an example for a parameter animation from the flow magnitude of the fuel cell data set, while Figure 5.10 b) display a time-varying kidney study over time. Usually

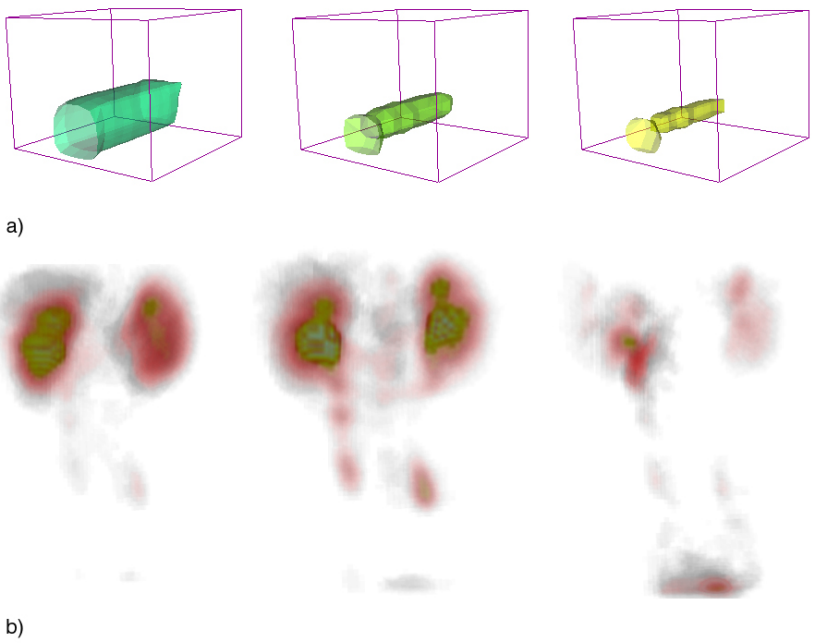

Figure 5.10: Animations: over time a), parameter b)

every parameter that is used in the visualization can be animated. This includes colour and opacity for the transfer functions, but also the graphical primitives themselves. Their shape can be altered and they can be morphed between two shapes. With all these possibilities, special care has to be taken to not overkill the visualization with too much motion.
Time-varying data can also be displayed without any animation by simply using the colour of glyphs to visualize the deviation over time. This would allow one to easily identify regions which have a high temporal gradient.
Besides the glyph technique which visualizes the deviation to give an idea of the temporal behaviour of a ceratin object, some techniques can visualize the complete time spectrum in one image. An example is to simply switch the time with one spatial axis. Now one 2D slice is visualized over time. To not lose the context information, two visualizations can be used

concurrently, one showing the position of the slice in 3D space and the other displaying the slice changing over time (see also Section 5.3). Additionally, higher dimensional display techniques can be utilized to visualize four or more dimensions. See Section 5.6 for a more detailed discussion on this topic.
All these techniques can be used for the fuel cell data set. For the current setup of the fuel cell design, animations and complex visualization are not necessary (yet). With complexer structures and a more realistic design, see Chapter 2, animations and the visualization of the temporal component become invaluable to detect vortices and structural weaknesses in order to improve their efficiency.

5.5 Focus and Context

When dealing with multiparameter data sets, usually more then one data set or parameter is displayed in the same visualization. The resulting image can easily get cluttered and appear more confusing than helpful. Most visualization algorithms or multiparameter techniques can be modified so that one area or one data set is in focus. The rest of the information can be visualized in a different way and serve more as context or background information. Figure 5.11 shows an example for a kidney data set: This visu-

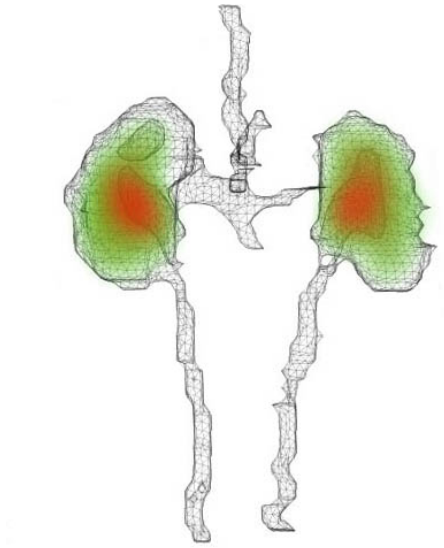

Figure 5.11: Focus and Context example

alization shows the concentration of radioactive tracers from nuclear imaging in a dynamic SPECT kidney study [TRM+01]. The concentration of the labelled pharmaceuticals is displayed using volume rendering and context in-

formation, the anatomical structure, is displayed as wireframe which shows the outline of the kidney as well as the blood and urine vessels.
Easily changing the focus and redirecting it to another position if very important to highlight specific regions of the data set. This can be done by using an interactive lens or other 3D cursor which were described earlier in this chapter in Section 5.3.5.
It is usually the best to show all the available information at once, but for most data sets this is not practical because of the huge amount of data. Here, focus and context techniques can help. They are not only useful for multiparameter data sets, but also for static data where these techniques can be used to navigate and explore large data sets.
The next two sections give a more detailed overview of what weighting techniques can be applied and which methods can be used to set one part in focus and the other one out of focus. The second section gives some more insight in ongoing research on how detail information can be displayed without destroying the reference object.

5.5.1 Weighting

Focus and Context means that one part of the data is enhanced in the visualization and that some other information is given in order to provide structural or context information. The advantage is that one portion of the data set is highlighted and detailed enough to work with, but at the same time one is still able to evaluate this data on a global scale.
One possibility to achieve this is to apply different weights to the used graphics primitives to differentiate between objects of focus and objects of context. Colour and opacity can be varied to enhance or suppress some regions, like standard transfer functions. Also, usually not the entire data set has to be in focus. An interactive lens (see also Section 5.3.5) can be used to have only some parts of the data set in focus, the part inside the lens. Other visualization specific parameters like size, motion or transparency can be used in addition to distinguish objects in and out of focus.
Another idea would be to use focus in the photographic meaning and represent the visualization using some depth-of-field method [KMH01]. Here the interesting parts of the data set are in focus and sharply visible, while the surrounding data is slightly blurred and not in focus. This immediately highlights the important parts in the middle while still giving some context information. For this technique, an interactive lens could be, even literally, used for interaction. This is similar to the first x-ray slicing which was used before the CT technology was developed in 1973 by Hounsfield [Hou73]. Here a film was used and placed at a special position, so that the organ of interest was imaged clearly but all the other information which was below or above the slice of focus was blurred. Only this blurring made it possible

to observe a clear slice inside the body. This technique has is to be mistaken with the classic x-ray invented by Wilhelm Conrad von Röntgen in 1895. The very classic approach, depicted in Figure 5.11, shows the data of interest surrounded by either a wireframe or a translucent surface which visualizes structural information, like boundaries or the size of the data set. These weighting methods can be applied to all parameters which can influence the visualization. Segmentation techniques can be used prior the mapping to further enhance the visualization.

5.5.2 ExoVis

ExoVis is a framework to develop new *focus/context* techniques and to evaluate their applicability. This work is ongoing research by Melanie Tory [TS02] and part of her Ph.D. thesis. *Exo* is derived from Greek and means *outside* or *external* and this is actually what ExoVis is doing. It visualizes the detail information (focus) outside the reference object (context) by linking them together in a way that the immanent relationship is clearly visible. Two different widgets were introduced, one to visualize 2-dimensional slices and another one to render 3D callouts. Figure 5.12 shows an example for

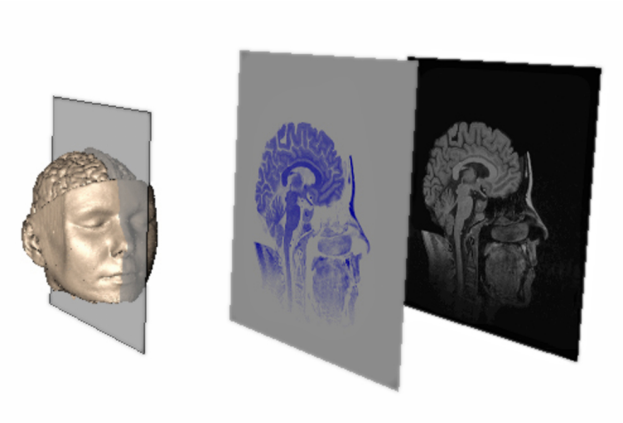

Figure 5.12: ExoVis - 2D widget

the 2D slices. Here one semi-transparent slice is put as a placeholder inside the volume to mark the spatial position of the detail slice. The plane where the slice will actually be displayed is put in parallel to the placeholder in the visualization space. This helps to relate the view to its placeholder. Also several copies of the same plane can be generated, each showing the same data with different transfer functions or different modalities from a multi-parameter data set. This allows one to easily compare different data at the same position and to make conclusions about similarities or dependencies between these data sets. The placeholder can be rotated and translated

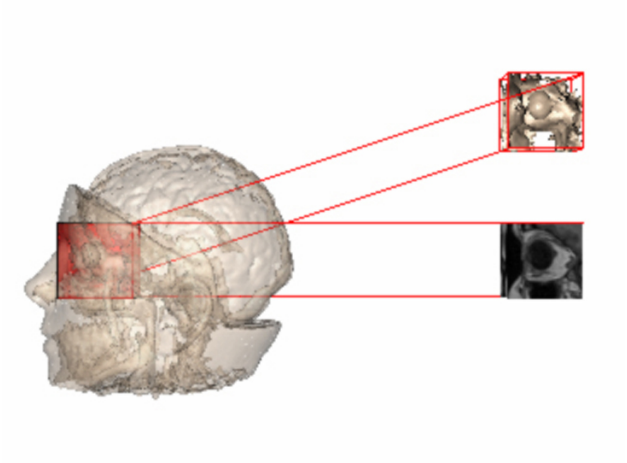

Figure 5.13: ExoVis - 3D widget

through the object. The slice will automatically be updated. The 3D callout widget is displayed in Figure 5.13 and is a natural extension to the 2D widget. In this example two copies of one callout region are made. One shows an iso-surface and the other one a semitransparent rendered volume. The same qualities that apply to the 2D counterpart apply to the 3D version as well. The advantage of both techniques is that one has the entire object of context and can visualize this using whatever display technique is appropriate. The placeholder shows the exact position of where the detail information is originating.
One problem that might occur is that when one visualizes too many of the 2D slices or 3D callouts, the image can get cluttered and too much information will be displayed. Here the solution could be that the visualization space for all detail images or volumes can be made smaller so that all detail data sets fit on the display and can be seen without overlapping. Then a mouse over could be used to activate a zoom-in where the current slice or volume is moved to closer distance and scaled to its original resolution. This would be a classic focus/context technique and similar to the use of a fisheye lens. The other solution would be that all detail objects are equally spaced around the object and that they are only visible when the object normal is pointing towards the camera. This method is similar to the shadow projection method which was described earlier in this chapter in Section 5.3.3.
When focusing on the visualization of fuel cell data, one first needs to decide what context information will be used. The simple answer would be structural layout which could be rendered as semi-transparent wireframe and used with all other data sets. Another possibility would be to use a data set that describes the strength of the chemical reaction, like the temperature data, together with the pressure and the flow information. Now either or all of these can be used as context for the input of the reaction, i.e.

the concentration of O_2 and H_2, or the output H_2O. After this, one has to decide what weighting scheme is appropriate. To visualize more than one data set, opacity and colour indexing could be used. The techniques from the last section are very useful to extract one slice from the volume to use as reference to evaluate all the available data sets. One would gain a very good understanding of the reaction as one could see a reference object and detail information at the same time.
So far only 2- and 3-dimensional visualization techniques were discussed. The next section focuses on higher dimensional visualization techniques which can be used to display 4D time-varying data sets in one single image to directly derive some temporal behaviours of the data set.

5.6 Hyperspace

One drawback of most visualization techniques is that they are rather limited for visualizing higher dimensional data sets. The temporal features of a data set can only be observed through animation and visualizing all time frames successively. A technique which would be able to show all this 4D information in one picture would be very helpful to immediately reveal the temporal behaviour. While most time-varying data sets are represented as 3D cubes which are varying over time, the use of the hyper space would enable one to see everything in one picture.
The first publication about hyper space was the book "Flatland" from Abbott [Abb94] where he describes the experiences of a 2-dimensional rectangle, called A.SQUARE, and his discoveries in a 3D environment. Figure 5.14 shows a sketch of his house. The book describes the journey of A.Square who

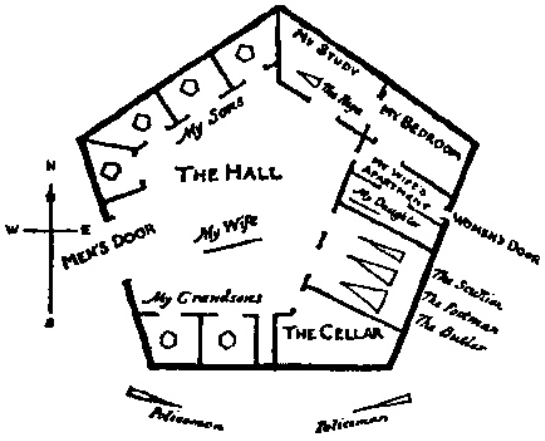

Figure 5.14: The house of A. Square

is a resident and a mathematician of the 2-dimensional Flatland. Residents

of Flatland have a different number of sides depending on their social status. Woman are represented as thin lines which represents the lowest shape in 2D. Through his discoveries he experiences Pointland (1D), Spaceland (3D) and the 4th dimension. Abbott choose a story in a lower dimension so that his readers could imagine that the same procedure applies also from our 3-dimensional spatial universe to a 4-dimensional one.
One understanding of our existing universe is that it actually has not only four dimensions, but ten. While six of them are *curled up* so that they do not influence our space, the other four dimensions build the 3-dimensional space with one temporal dimension as we know it [Kak84].
If a human enters a 2-dimensional world (i.e. a plane) the world's inhabitants would not be able to perceive nor understand the third dimension. They would only see 2-dimensional slices of the human which would not necessarily have to be connected. For instance, if a hand intersected with the 2D world, it would be only visible as five circles. They are just 2D projections out of 3D space. The same applies also in the other direction, when going from 3D to 4D space. A 4D being entering our three spatial dimensions would be sliced in 3D and might appear as several blobby 3D objects which are connected in the 4th dimension. Another advantage for the 4D inhabitants would be that they could entirely look into our body and even perform surgery without opening the body. This idea becomes more clear when thinking about how we perceive the 2D space. In 0D, only one object can exist, a infinitesimal

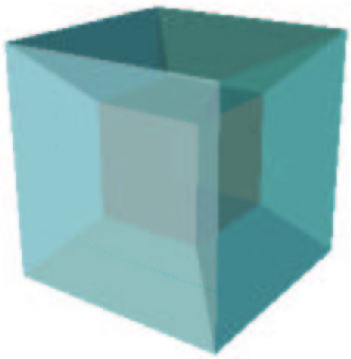

Figure 5.15: 4D hypercube

small point. In 1D, the point can be extend orthogonal to itself and is now a line. If this line is extended again orthogonal to itself one obtains a square and for 3D a cube. To find out what the shape of a cube, or tesseract in 4D is, one has to glide this cube orthogonal to itself. Because we only have 3 spatial dimensions, we can not really do this, but Figure 5.15 [RA] shows the principle of the tesseract or hypercube which is the extension of a regular cube by another spatial dimension. A hypercube is composed out of 16 vertices which are connected by 32 edges which build 24 faces or 8

bounding cubes. The 4 edges on each corner are orthogonal to each other. We can perceive the tesseract only as shadow projection from 4D into our 3D space. Figure 5.16 [RA] shows the principle for the projection from 3D to 2D. Because we already loose one dimension from the projection, the best visual impression of the hyperspace could be perceived when 3D stereoscopic devices are used. To exploit these qualities for scientific visualization, several

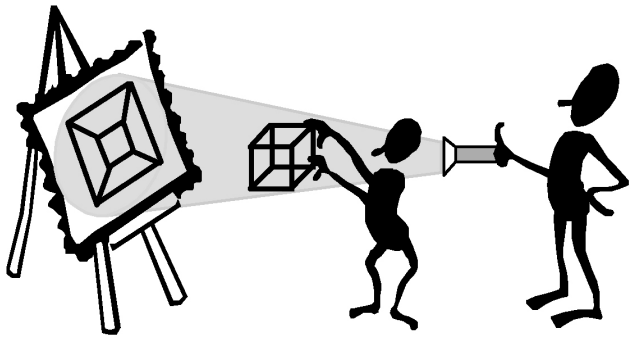

Figure 5.16: Projecting from 3D to 2D

techniques can be applied which shall be discussed in the next three sections. For the shading of 4D objects 4D light sources have to be used [HH]. If just 3D lights were used instead, the entire data set could not be lit. An example is to light a 3D surface with a 2-dimensional thin line. Only parts of the surface would be visible and perceiving the object shape would be rather difficult. Other techniques were developed which extend the idea of displaying 2D images as 3D height maps to four dimensions. Now 3D volumes can be visualized in the same intuitive way [HH92].
The next three sections give some little insight of how volume rendering, iso-surface visualization and slicing can be performed on 4-dimensional data sets.

5.6.1 4D Volume Rendering

Some work in visualizing so-called *hyper volumes* was done by Bajaj et.al. [BPRD98]. They implemented a splatting algorithm which allowed the visualization of n-dimensional data sets by re-orienting as many coordinate axes as needed in 3D space. Interaction includes stretching or rotating all axes to enhance one or several axes specifically. Figure 5.17 shows a screenshot of an example image of a 5D interaction energy scalar field of a chemical compound. Here red denotes attraction, blue repulsion, and green free movement. A different approach, which is not heavily explored yet, would be to visualize the 4D volume as a tesseract and to project it into 3D space. The data must be (re)sampled onto a 4D hypercubic lattice. While orbiting

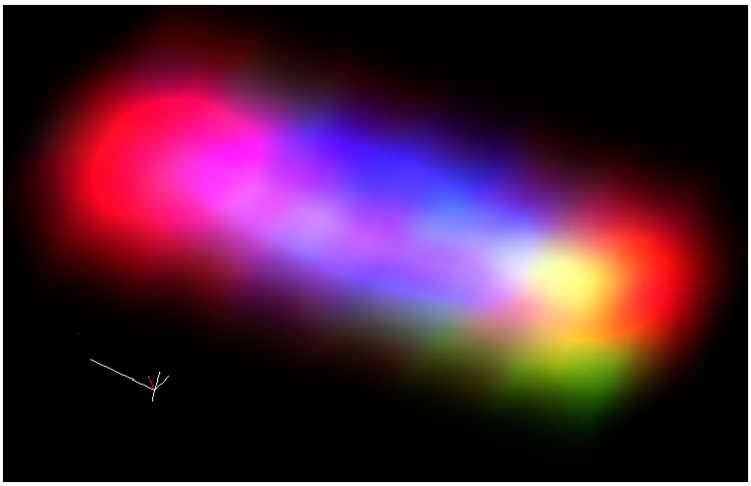

Figure 5.17: 5D interaction energy scalar field

around and rotating the tesseract, one should be able to detect temporal patterns in all dimensions in the visualization, if they exist in the data. However, the interaction with the data in 4D would not be as intuitive as in 3D. The number of rotations for higher dimensions grows quadratically with the order of the dimension. In 4D six different 4D rotations, around the xy, yz, xz, tx, ty and the tz axis exist. If the hypercube is projected using perspective projection, usually along the direction of the positive time axis, all points will be scaled accordingly to their position in time, e.g. the 4th dimension. The perspective projection can be written as:

$$f(x, y, z) = ((\frac{x}{t+c}), (\frac{y}{t+c}, (\frac{z}{t+c}), \tag{5.1}$$

with t as the position in time and c as an offset. Figure 5.18 [MSO] shows a $4 \times 4 \times 4 \times 4$ euclidian hypercubic lattice which is displayed in perspective along the positive time axis t. In Figure 5.18 the blue cube which is closest

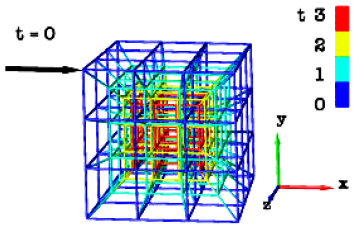

Figure 5.18: 4D hypercubic lattice

in time is rendered largest versus the red cube, as farthest away is rendered smallest.

5.6.2 Iso-Surface Extraction in Higher Dimensions

The extraction and the visualization of iso-surfaces is a valuable tool to identify features in the data and to display the different layers of a data set. The problem with most visualization techniques when it comes to higher dimensional data sets ($n > 3$) is that the data has to be first downsampled into $\Re^3$ before it can be used for the visualization. For time-varying data this is usually done by simply visualizing one frame at a time. But the drawback here is that features which are contained in the time domain are sometimes difficult to detect and eventually lost.
Weigle et.al [WB] used a technique which first generates a 4D iso-surface that satisfies $f(x, y, z, t) = C$ which is then used to re-triangulate new iso-surfaces at interpolated time steps. A recursive contour meshing algorithm is used to extract the iso-surfaces from 4D data. The final 3D slice is extracted by using a second constraint that locates points which lie on the same time on this iso-surface. Another advantage is that this method also allows for multiresolution mesh generation which helps one to quickly browse through the surface in 4D.
A similar method developed by Praveen et.al. [PWC] extends the Marching Cubes algorithm from Lorensen et.al [LC87] to extract iso-surfaces in any dimension. This is done by automatically generating a lookup table for the iso-surface and its corresponding triangulation for all possible states of the hypercube in a n-dimensional regular grid. After the n-dimensional iso-surface is computed, it can be sliced either perpendicular to the time domain, for time-varying data sets, or arbitrarily to obtain a 3D data set. This data set can now be rendered using standard OpenGL graphics techniques. The advantage is that features which evolve over time or in higher dimensions can now be observed more easily.

5.6.3 Slicing in 4D

Slicing a n-dimensional object usually results in a $(n-1)$-dimensional object which displays the information contained at this *slicing plane*. The slice of a 4D data set is a 3D object. Usually 4D time varying data sets are visualized using some sort of time slice technique where the user can step through the data set in a multi-frame animation. Here the 4D data set is sliced along the time axis and sampled at discrete points in time which yields the original frame at this time step. But slicing could also be performed in a way that several $(n-2)$-dimensional objects are extracted. In 3D this would result in three lines which intersect each other at the position in space where they are extracted (focal point). In nD, this would result in n^2 2D slices which would also intersect each other at the given point in space/time.
Van Wijk et.al. [vvL93] developed a technique they call *Hyperslice* which allows one to extract 2-dimensional slices from a n-dimensional data set.

Here the slices are shown as a matrix of orthogonal 2-dimensional slices. Figure 5.19 shows a screenshot of the Hyperslice technique applied to a 4-dimensional potential function. In general, this technique is a variation of

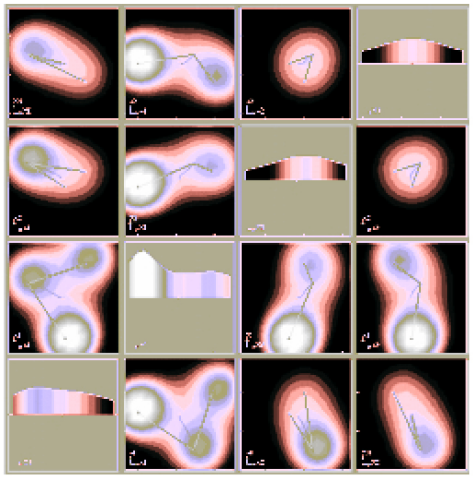

Figure 5.19: Hyperslice of a 4D potential function

the Scatterplots, which were discussed earlier in Section 5.3.1. This technique does not project the entire data set, instead, a selection is made and the result is displayed as 2D images. By moving the focal point, or point of interest, one can navigate through nD space.
This technique demonstrates an easy to implement and use method for viewing higher dimensional objects ($n > 3$). Disadvantages are that only a selection of the entire data is shown and that the number of slices grows quadratically, which makes a visualization for very high dimensions difficult.
The Hyperspace can be used in a variety of ways for the fuel cell data set. It could be used to visualize the temporal behaviour of the single data sets, and some data sets could also be combined in the visualization. The hyperslice method could be particulary useful for this.
Unfortunately only little research has been done so far in the field of higher dimensional data visualization. One drawback still is that four- or higher-dimensional data sets can be very huge in size which makes handling and realtime processing very difficult. With future hardware improvements, this might change.

5.7 Non-Photorealistic Visualization

Since the beginning of computer graphics, people were driven by the goal to generate images that were indistinguishable from real photographs. This

movement still exists in a number of fields in computer graphics. It is most noticeable in the computer game industry which tries to make their simulations and renditions as realistic as possible. Figure 5.20 shows a screenshot of current game which demonstrates the possibilities of today consumer graphics accelerators. While these pictures often attempt to look perfect

Figure 5.20: Screenshot from "Burnout 2: Point of Impact"

and want to simulate the real world through a camera's eye, some other computer artists try to achieve a similar effect for simulating hand drawn looking images. Non-photorealistic rendering of graphics models (i.e. polygonal data) has gained more in recent years, but it is still rarely used for the visualization of volumetric data sets. These techniques try to simulate the process of generating hand drawn looking images. One advantage of non-photorealistic rendering (NPR) is that it stimulates the human visual system in a way that appears to be natural. These non-perfect images leave some room for personal interpretation. Here, the images can be simplified by removing unimportant information. In a different way, NPR can also be used to explicitly point to certain characteristics in the data by enhancing selected structures. The human eye is very good at guessing from imperfect information and when a sketchy or fuzzy image is shown it tries to integrate over the whole image to grasp the *global picture*. By exploiting this quality, not the whole data set needs to be displayed, a low resolution version might suffice to get the same interpretation as with a high resolution version. As a result, NPR can also be used as a *compression* technique to reduce the size of the data set to what is really necessary for the interpretation. This would be very helpful in volume rendering where huge data sets could be rendered faster when this quality is used.
In a similar way NPR can be employed to specifically enhance or suppress

some features or parts of the data. For instance, homogenous regions can be drawn such that one can assume and rely on the law of continuity. Here not all information has to be presented.
But this is only one advantage. NPR can also be used to discuss unfinished or unclear projects or objects. An architect might want to show a sketchy version of his idea first to invite discussions. All this would be very helpful for medical or scientific visualization as well, where one wants to make an opinion about the data and to formulate a hypothesis about the underlying information. Here NPR visualization can help to provide prominent features of the data without showing too many details in order to provoke some discussion and to deflect from too obvious solutions which might led in a wrong direction. Also, because one is not directly working on the data set and eventually even using a low resolution version, reconstruction artifacts become negligible.
Recently, with the availability of new and fast evolving graphics hardware, scientists are interested in achieving these results in real-time, which would allow more natural interaction.
The following two sections are a short overview of possible techniques which can be utilized for scientific visualization, especially for fuel cell simulations in order to enhance their expressivity and to leave more freedom for conclusions.

5.7.1 NPR for Volume Visualization

While volumetric data sets are a little different from polygonal data, there are some similarities one can take advantage of in order to adapt some of the existing NPR algorithms to visualize volume data sets using NPR techniques. One important feature which can be used for non-photorealistic rendering is the gradient information. It can be approximated by using central differences (see Chapter 4.4.5) or by using more advanced techniques to achieve better results. In most cases the central differencing works well on a previously smoothed data set. This gradient information can be used together with the density value and more advanced transfer functions to perform volume illustrations [ER01]. Volume illustrations can include the enhancement of silhouettes and internal boundaries as well as tone shading and the use of special depth cuing. Figure 5.21 shows some examples.
The first image shows a CT scanned thorax using standard colour and opacity mapping. In the second image, the silhouettes and the boundaries are enhanced, while in the third image a tone shading was used in addition. When using the last technique, surfaces which point towards the light source receive a warm colour tone, while all other surfaces are coloured using a cooler tone. In both images the edges are enhanced which makes the image appear sharper and more detailed. It uses the fact that the human visionary system

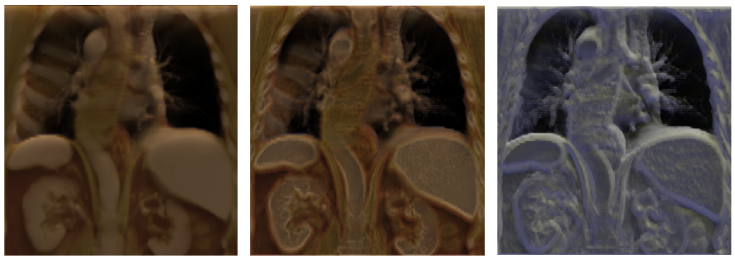

Figure 5.21: Volume illustrations

is most sensitive to edge information.
Additionally, before the data sets are used for any kind of volume rendering, some image pre-processing could be used to perform edge enhancements, like unsharp masking where the gradients are elongated and let the edges appear sharper. This gradient information can also be used to build a com-

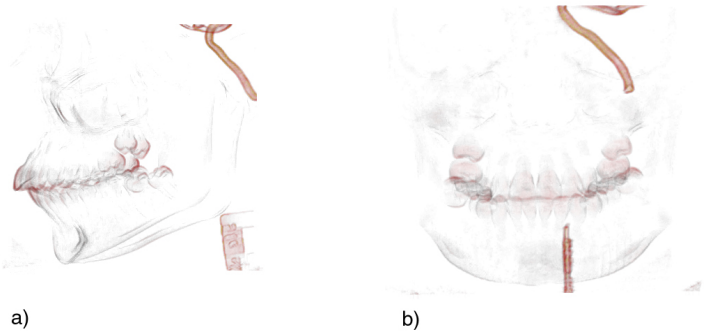

Figure 5.22: Charcoal line drawing skull data)

pletely different type of renderer by evaluating the gradient information and interpreting it as lines. One would be able to build a line renderer, similar to existing solutions for polygonal space [SR98]. A first step in this direction by creating a stipple based volume renderer was done by Ebert et. al. [LME+02]. They used the gradient information in a variety of ways together with the original density signal to create renditions in a stippling manner for volumetric data sets.
All existing polygonal NPR techniques are directly applicable if one first extracts iso-surfaces from either the intensity value alone or from a combination with the gradient magnitude. This data could be used to build iso-surfaces depending on the importance in the data and on which some NPR techniques can be evaluated. Also cartoon shading could be applied where simply a colour ramp with three or four different colour values is used to classify the data. This transfer function can be either applied for the entire data set, or to some regions only. Also faux shading which is a sim-

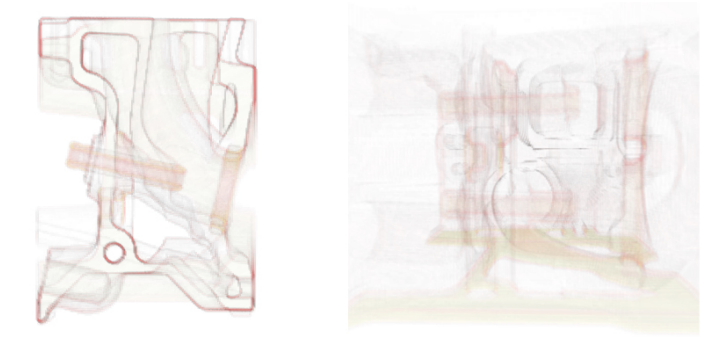

Figure 5.23: Charcoal line drawing engine

ple modification to the transfer function, could be used to create silhouette edges. The engine data, for example, consists of three different materials: one is noise (which can be skipped) and two different kinds of metal. Here two cartoon colour transfer functions together with an silhouette and edge enhancement can be used to achieve cartoon shading for volumetric data sets.
Figures 5.22 and 5.23 shows two example for NPR like volume rendering. Here the gradient magnitude is used together with a special transfer function to render charcoal looking images. The focus for Figure 5.22 was on the teeth.
Some of these techniques can be implemented using hardware acceleration and OpenGL. Lum et. al. [LM02] showed an interactive rendition using silhouettes, tone shading and depth cuing. In order to achieve interactive rates they implemented a parallel algorithm which runs on several computers.
For the fuel cell simulations or for multiparameter data sets in general, NPR can be used to create new visualizations by combining the extracted features of different data sets and combining them into a new scheme. Another advantage of using NPR techniques for huge data sets is that lower resolution data sets can be used to store the feature data only. For instance, if only edge information is used, then only the gradients have to be stored to a certain magnitude together with the spatial information and the intensity value for these points. Huge *compression ratios* can be achieved using this technique.

5.7.2 NPR for Flow Visualization

The possibilities of non-photorealistic rendering for flow visualization are explored only briefly in the scientific community. Tory applied some ideas

like tone shading and shaded timeline strips [Tor01].
Animation can be used to visualize the motion and even the strength of the flow field. Depending on the type of visualization, either objects can be animated or textures. Both techniques would give a good understanding of the global flow field, but one has to be careful with not to animate too much at once.
Here, motion can be depicted without animation by using speedlines. Masuch et.al. [MSS99] showed how this can be effectively used. These speedlines can be attached to glyphs which usually visualize flow parameters, like direction or magnitude, to enhance their expressiveness and to additionally show the path of motion and hence the flow field. By virtually integrating over the visualization, one can perceive a good understanding of the global flow field. Instead of speedlines, one can also use motion blur and include the glyph's position from previous frames in the visualization. Here the visualization might integrate over five different time frames which are blended and weighted together. The most current frame is weighted to contribute most to the final image.
Figure 5.24 shows a classic flow visualization from a car repair manual [Hoc97]. Here the flow of coolant through an engine is visualized through arrows. For 2-dimensional visualization and eventually also for 3D visual-

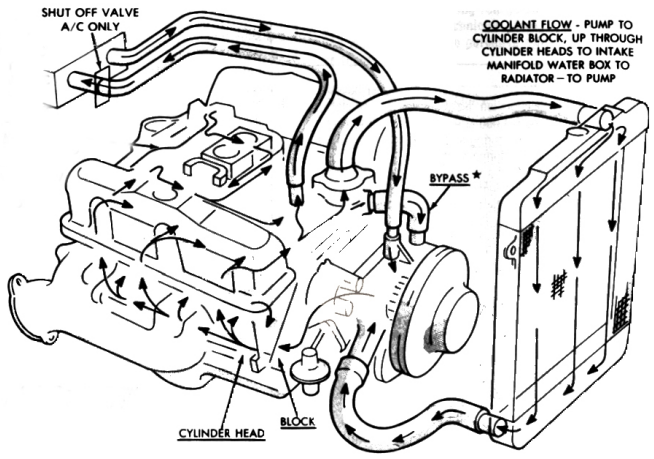

Figure 5.24: Some Flow NPR

izations, painting and drawing techniques can be employed which use the flow field as input for the stroke length and direction. The result would be either sketches of the flow field, or even paintings. One example visualizing this technique is the painting from Eduard Munch "Der Schrei". While he definitely did not use any scientific flow data, the result is similar to the described method and also to line integrated convolution [CL93].

5.8 Conclusions

The two most challenging problems for the visualization of the fuel cell data set are the size and the multi-parametric nature. While the last chapter developed a new method and extended some existing techniques to be able to interactively explore large volumetric data sets, this chapter investigated the possibilities to interact with more than one data set at a time. Multiparameter visualization is a difficult topic and differs for each data set. The advantage of including more than one data set in the visualization is that sometimes features are only visible when one can directly compare several data sets with each other.
This chapter explained some existing multiparameter techniques and developed some new ideas which were shown to be useful for the visualization of the fuel cell data set. In Section 5.2 some classic techniques where presented which can directly be applied to the fuel cell data. These methods, which are derived from unimodal data visualization have some limitations and can only be used to a certain degree. This led to the next section where some standard multiparameter techniques were presented. In this section, first the data set was analyzed and it was shown which parts of the fuel cell data are important for the visualization and which data sets can be skipped due to inherent coherency. The next section highlighted some specific problems for time-varying data sets. Here not only do several data sets have to be visualized, but also the temporal component and the development over time has to be considered for some data sets. The ability to focus on some regions of a data set while still displaying the other parts as context information is important. Which techniques are available and can be used for the fuel cell data are described in Section 5.5. An option which is not heavily exploited yet in scientific visualization are higher dimensional visualization techniques which were described in Section 5.6. Here the basic principles are explained and how they can be utilized to use more than three dimensions in a visualization. The last section was dedicated to non-photorealistic rendering techniques which can be applied to several problems from highlighting specific parts in the data to data compression.
One focus in this chapter, besides the presentation of useful techniques for the fuel cell data set, was to develop techniques and ideas which can be used to increase interactivity with the data sets. Here, some methods make use of special Level-of-Detail for the data to increase the rendering speed. Other methods, like NPR or hyperspace techniques, try to *compress* the available information in the data set to make the visualization easier to understand.
The next chapter explains the results from this chapter and also from Chapter 4 in more detail and highlights the advantages, but also the disadvantages of the methods described here. The results are fortified by images as well as

some numbers if applicable. In the end of Chapter 6 some conclusions are drawn and the usefulness of each technique is evaluated.

Chapter 6

Results and Conclusions

While the end of Chapter 4 and Chapter 5 already presented some conclusions for the given chapter, the intention of this chapter is to present the results in a more global context. This chapter will also be used to evaluate some of the work for applicability to the given problems and compare the results with other existing techniques.
The main topic of the thesis was the visualization of fuel cell simulations. The difficulty for the visualization task is that the data sets can be huge in size and multiparametric. These two problems were discussed and analyzed in Chapter 4 and Chapter 5. Here the overall focus was on fast, expressive visualizations with high frame rates and good image quality.
Chapter 4 combined some existing technologies and developed a new approach to visualize huge data sets on standard workstations at interactive rates. In the beginning a better lattice was introduced which allows one to store the samples more efficiently than the current CC lattice, (Chapter 4.1). The BCC lattice was used for static 3-dimensional data sets and allowed to save 30 percent of the samples, while for time-varying data sets 50 percent of the samples could be discarded with the use of the D_4^* lattice. The resampling of a data set usually introduces some minor blurring, but the advantage of the hexagonal lattice is that even with fewer samples the frequency content is not damaged. After the hexagonal resampling, the data set is analyzed for spatial and temporal coherency (Chapter 4.2). Unimportant regions within the data set are discarded, where the entropy and the mean value of this brick are below a certain threshold. After this segmentation step, the remaining data is merged into larger bricks of the size of the available texture memory. This is performed in a way that homogenous regions are merged together. The homogeneity criterion here is the importance, the information content of the brick. After this merging, a multiresolution representation of each brick is created using wavelets (Chapter 4.3). Lossy compression results in a better compression ratio, but the frequency content is damaged and artifacts might be visible in the visualization. For real

lossless compression, the integer wavelet transform was used. However, the degree of the compression ratio can be adjusted to also allow lossy compression for better compression ratios. After this pre-processing, the data can be used for display (Chapter 4.4). Here common texture mapping hardware is used where the data is loaded as 3D texture and rendered from back to front. The data can be categorized using pre- and post-classification and several different visualizations methods can be selected.
Chapter 5 was dedicated to the multiparametric nature of the fuel cell data set. Here some existing techniques for the visualization of multiparameter data sets were evaluated and there applicability for the visualization of the fuel cell data was explored. Some new ideas were developed which can be used to make the visualization more expressive and efficient. In the beginning of Chapter 5 some general multiparameter techniques are discussed, Chapters 5.2 and 5.3, and examples were presented on how these techniques can be used to visualize the entire, or parts of the fuel cell data set. A very effective and universal tool are the different *lenses* which were covered in Chapter 5.3.6. Standard possibilities to display time-varying data sets were presented in Chapter 5.4, while some more advanced techniques for higher dimensional data visualization where discussed in Chapter 5.6. Applicable for multiparameter data sets as well as static data sets is the use of *focus and context* techniques which were introduced in Chapter 5.5. These methods are very important to visualize local features of the data set while still showing global context information. Non-photorealistic rendering techniques are discussed in the end of Chapter 5. Here some methods are discussed which can improve the expressiveness of the visualization as well as been used to compress the contained information. This makes these techniques very attractive for huge data sets, like the fuel cell simulations.
This Chapter is divided into two sections, which both present results and comparisons for Chapter 4 as well as for Chapter 5. The first section shows some achieved qualitative results and presents some screenshots from Chapter 4 and explains the different available rendering techniques. After this, some qualitative comparisons between the hexagonal and the Cartesian lattice as well as with compressed and uncompressed data sets are presented.
The second part of this chapter discusses quantitative results and shows running times of the algorithms and how many frames per second can be achieved using these techniques. Here also some comparisons are shown in the improvement of the memory efficiency by using the hexagonal lattice and wavelet compression techniques. In the end of this Cchapter some conclusions are drawn for the multiparameter visualization of the fuel cell data which were discussed in Chapter 5.

6.1 Qualitative Results

This section presents some qualitative results from the techniques developed in Chapter 4 and Chapter 5 and is divided into two groups. First some results for general image quality are presented which demonstrate the achieved rendering quality for the various visualizations and rendering styles. One focus for the algorithm which was developed in Chapter 4 was to visualize huge data sets. Here some compression techniques have been used which are applied in a pre-processing step. While all of the techniques can be used for lossless compression, minor blurring can occur due to the resampling. Some parameters are also adjustable to perform lossy compression with better compression ratios. This second section will be used to demonstrate the achieved image quality, depending on the compression technique used.

6.1.1 General Image Quality

The renderer which was described in Chapter 4 can not only be used to render huge data sets at interactive rates. Also several different rendering techniques and possibilities to classify the volumetric data were implemented. In the following section, some of these techniques will be presented and compared with existing algorithms.
The volume renderer which was implemented and which is also described in Chapter 4.4 is able to handle Cartesian as well as both hexagonal data sets. These data sets can be rendered with the focus either on accuracy or speed. Figure 6.1 shows a screenshot of the application. Different methods of classification and shading have been implemented as well as several rendering styles. The data set can be rendered as:

- direct volume rendering,
- iso-surface (with and without opacity),
- gradient-magnitude,
- maximum intensity projection, and
- X-Ray.

Independent of the classification technique used, the transfer functions for colour and opacity can easily be defined using a dialog widget. In the following, some results are presented to demonstrate the high quality of the visualizations. More information about the rendering can be found in Chapter 4.4 and also some implementation details in Chapter 7.3. The classification can be performed in several different ways. The simplest technique for volume rendering with texture mapping hardware is pre-classification using either RGBα volumes or a luminance volume and OpenGL lookup tables (Section 4.4.6). Figure 6.2 shows a rendering of the engine data set together with the dialog which is used for classification. The reconstruction artifacts are

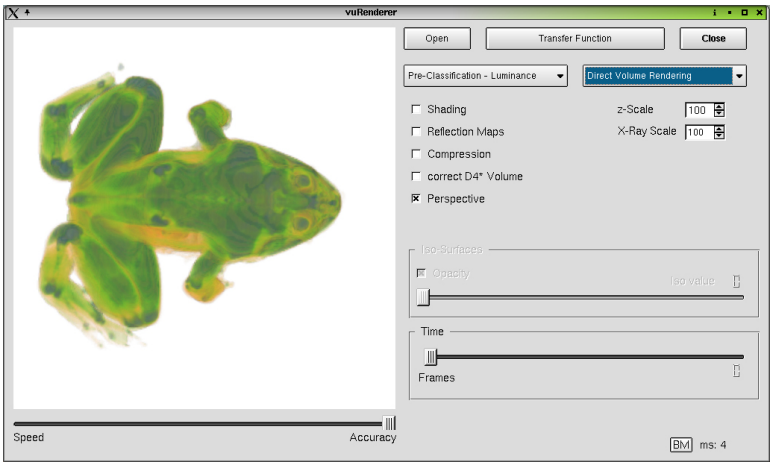

Figure 6.1: vuRenderer with frog data set

coloured red, the engine itself is green and the parts which are a made of a different alloy are depicted blue. Better classification can be achieved when

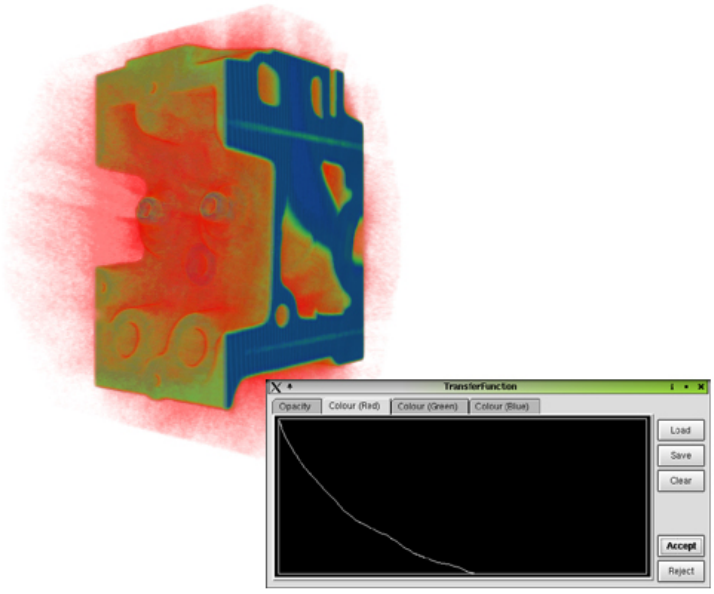

Figure 6.2: Pre-classification of the engine data

using texture shader and register combiner to perform post-classification. Figure 6.3 shows two images of the frog data set which were classified using dependent textures. Shading can be used to extend this technique and to aid in the 3D impression of the rendering. Figure 6.4 shows a rendition of the engine which was shaded using register combiner (Section 4.4.6). The

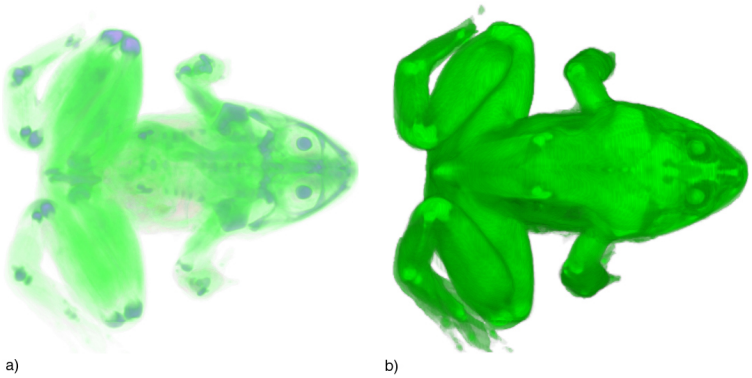

Figure 6.3: Post-classification of the frog data

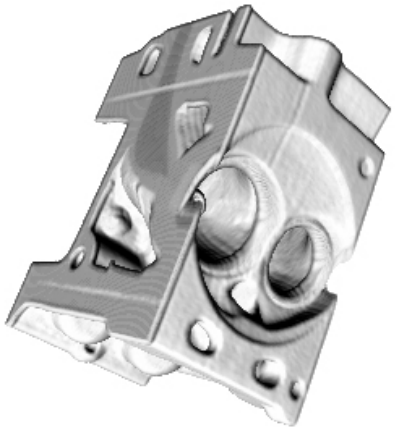

Figure 6.4: Shading of the engine data

gradients which are used for shading can also be used to generate an opacity transfer function which is dependent on the length of the gradient. This is very useful to highlight those regions in the data which change most, i.e. where the gradient is largest. Figure 6.5 shows two examples of this method with the engine data set and the tomato. This technique can be used with pre-processing as well as with dependent textures. The latter technique requires one additional 3D texture with the gradient length information to perform the computation within the register combiner [MGW02]. With this gradient magnitude classification and an appropriate transfer function, NPR like volume rendered images can be produced, as can be seen in Figure 6.6. Here the foot data set was rendered using this technique. By using different blending equations and enabling the alpha test, other rendering techniques like iso-surfaces, X-Ray, or the maximum-intensity projection

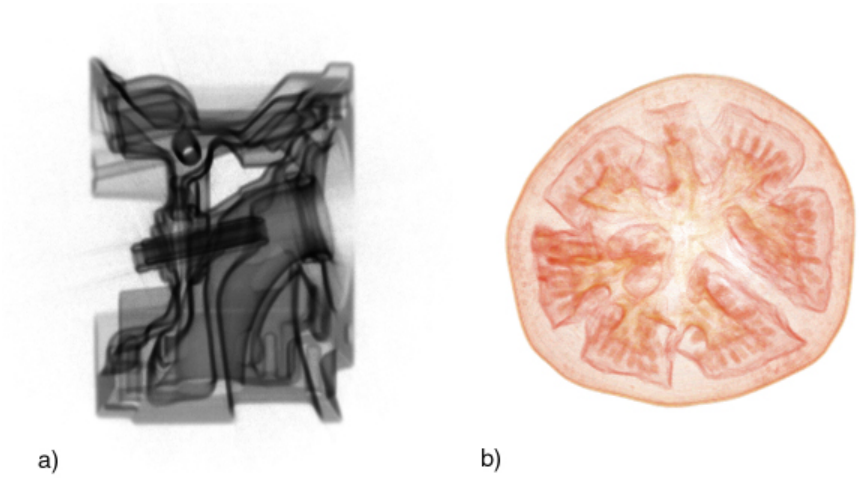

Figure 6.5: Gradient magnitude rendering, engine a) and tomato b)

Figure 6.6: Non-photorealistic volume rendering of the foot

can be simulated. In the following each of these techniques shall be presented and discussed using a few examples. Non-polygonal iso-surfaces can be rendered by enabling the alpha test which only allows fragments which are at or above a certain threshold to pass (Section 4.4.6). Figure 6.7 shows the engine data set rendered with an iso-value of 170 and the silicon data set rendered with an iso-value of 40. Both images are rendered with shading enabled. Especially for medical data sets the display of the data as X-Ray or maximum intensity projection is very useful. With these techniques regions inside the data set which have a high value become immediately visible. The two techniques can be simulated by using different blending equations for

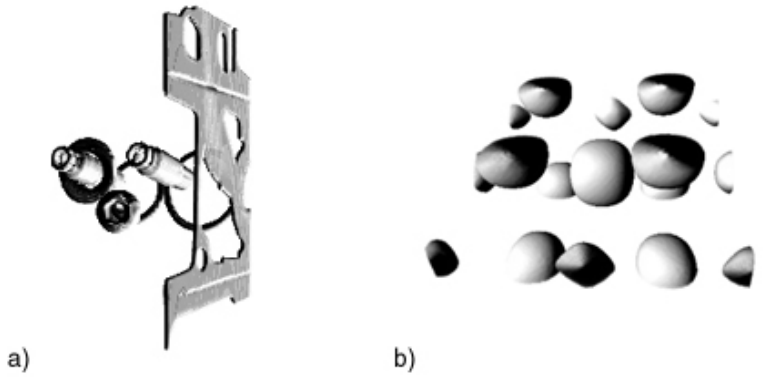

Figure 6.7: Iso-surfaces, engine a) and silicon b)

the compositing of the sampling slices (Section 4.4.6). Figure 6.8 shows the UNC-Brain rendered using X-Ray and maximum-intensity projection.

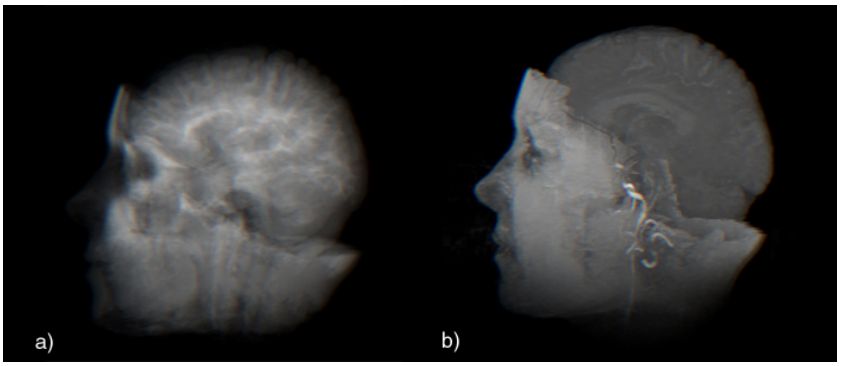

Figure 6.8: UNC Brain as X-Ray a) and using maximum intensity projection b)

6.1.2 Compression Quality

The last section only covered general image quality which can be achieved by rendering standard medium sized Cartesian data sets. In order to handle huge data sets, some techniques have been employed to compress the data sets in size. These methods can be divided into two groups. The first performs a resampling of the entire data set onto a more efficient lattice (Section 4.1) and the second one is a multiresolution compression technique (Sections 4.2 and 4.3). With the resampling of Cartesian data sets onto the BCC lattice in 3D and the D_4^* lattice in 4D, one can save 30, or 50 percent of the samples without impairing the frequency domain. This resampling is performed lossless when the signal is hypercubic bandlimited. Some minor

smoothing might occur due to the resampling. While the focus for the visualization is on speed, some minor quality artifacts may be observed. This depends on the goal of the visualization and can be influenced during the creation of the data set as well as during the rendering. The second optimization step is the pre-segmentation of the data and the multiresolution compression using wavelets. Both techniques can be adjusted to perform a real lossless compression of the data set.
In the following some quality comparisons are presented for different data sets using these compression techniques. First a comparison of the BCC lattice with the CC lattice is presented. Figure 6.9 shows the lobster data set rendered in CC and BCC. The first image, Figure 6.9 a), is rendered using

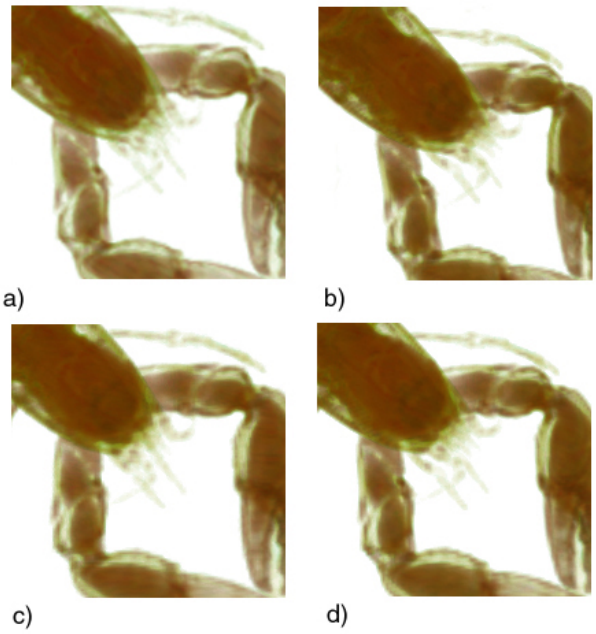

Figure 6.9: Lobster, a) CC, b) BCC, c) half, d) one

the CC lattice, while all other images are produced using the lobster data set resampled onto the BCC lattice. Here Figure 6.9 b) shows the data set in the full resolution BCC with both cubic textures interleaving by $\frac{1}{2}$ in all directions. Figures 6.9 c) displays the BCC data set rendered using only half the resolution of the BCC lattice with one of the two textures, while Figure 6.9 d) renders the data with all two textures, but without interleaving.
If the data is time-varying, then the D_4^* lattice can be used. Figure 6.10 shows a comparison of the CC lattice with the D_4^* lattice. It shows the kidney data set rendered at frame 49 for the CC lattice and 69 for the BCC lattice. In this example, Figure 6.10 a) is generated using the CC lattice with standard pre-classification. Figure 6.9 b) is rendered using the D_4^* lattice.

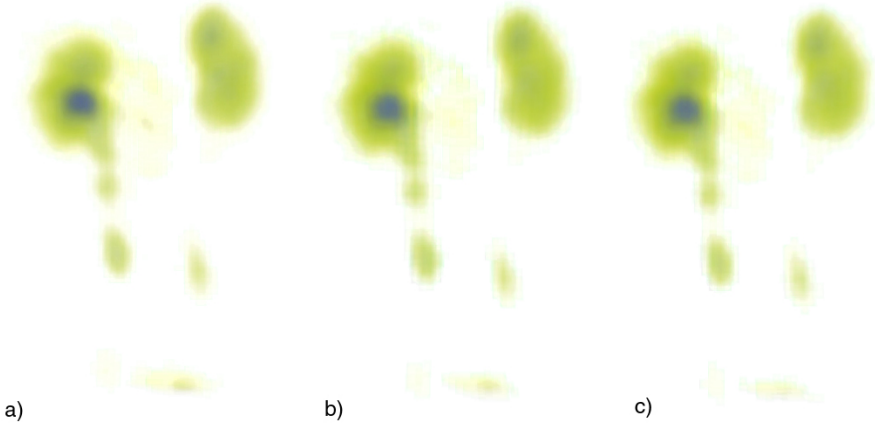

Figure 6.10: Kidney data frame 49/69, a) CC, b) D_4^*, c) BCC

The used texture is eight times smaller, because the next smaller possible texture size could be used. This results also in a eight times faster interaction if the time slider is used and a new frame is selected. For comparison, Figure 6.9 c) shows the same time frame, but correctly rendered as BCC data set. Here the two time frames from before and after are included as well and cubic interpolation is used to ensure that both textures are at the same point in time (Chapter 4.1). This second cubic texture is also shifted by $\frac{1}{2}$ in all directions. Further to the hexagonal lattices, wavelets were used

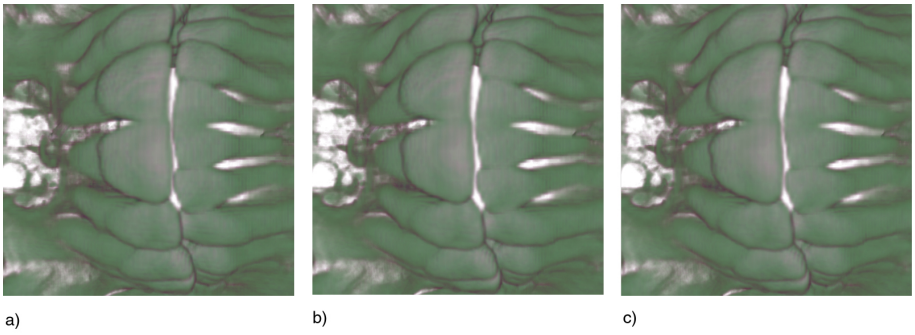

Figure 6.11: Skull data, a) (1,1) $T = 1.5$, b) (2,2) $T = 1.5$, c) (4,2) $T = 1.5$

to build a multiresolution representation of the data set as well as to compress the data to store the data more efficiently on disk. Figure 6.11 shows a comparison for the different compression quality that can be achieved by using different wavelet filters. The skull data set was soft thresholded and rendered using the following wavelets (1,1) and with (2,2) and (4,2) (see Chapter 4.3.1). The decomposition depth was one level. The differences for

this compression ratio are not very big, but the image quality increases from a) to c). This can be best seen around the teeth section. These wavelets also have a different storage requirement, as can be seen in the next section, where quantitative results are presented.
The wavelet technique which was used allows a lossless as well as a lossy compression of the data. Figure 6.12 shows a comparison of different compression ratios, from lossless, Figure 6.12 a) to a threshold of $T = 25.0$. The appendant memory requirement for the storage can be seen in the next section.

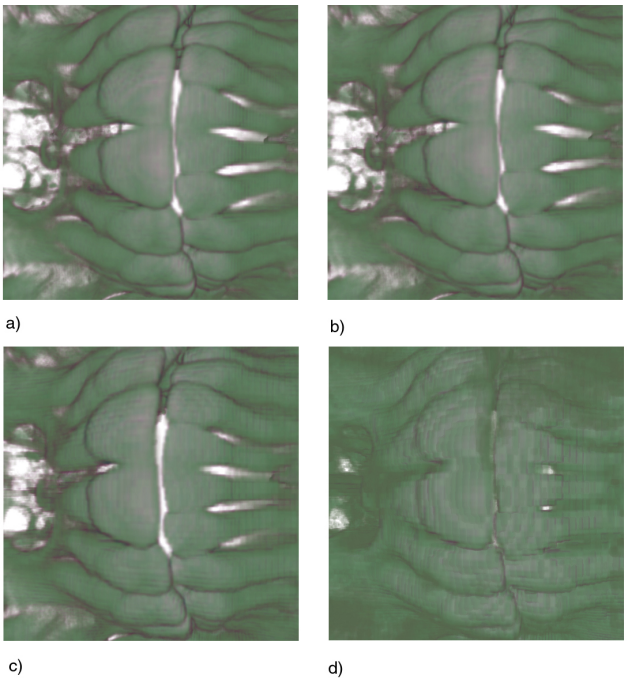

Figure 6.12: Skull data (1,1), a) $T = 0.0$, b) $T = 1.5$, c) $T = 5.5$, d) $T = 25.0$

6.2 Quantitative Results

The main problem with rendering huge data sets is their size. Some techniques have been used in this implementation to make the rendering more efficient. While the last section showed some visual results of the applied techniques, this section presents some quantitative results. Very important for compressing data sets is that the visual quality does not degrade too much. All compression techniques which were used can be set to lossless, i.e. one can construct the original data set from the compressed version. However, for most data sets some high frequencies can be discarded without

impairing the visual quality of the visualization. The last section showed some examples.
All rendering and compression has been performed on an Intel Xeon 2.4 GHz with 1 GB of main memory and a *n*vidia GeForce4Ti 4600 graphics accelerator. Table 6.1 gives an overview of the size of the data sets which were used, while Table 6.2 shows the time which was needed to convert and compress these data sets. As can be seen in Table 6.1 the size of the data

Data set	Size CC	Size BCC
Nucleon BCC	$41 \times 41 \times 41$	$29 \times 29 \times 58$
Lobster BCC	$301 \times 324 \times 56$	$213 \times 229 \times 79$
Statue Leg BCC	$341 \times 341 \times 90$	$241 \times 241 \times 127$
Engine BCC	$256 \times 256 \times 128$	$181 \times 181 \times 181$
UNC Brain BCC	$256 \times 256 \times 145$	$181 \times 181 \times 205$
Kidney D_4^*	$90 \times 90 \times 80 \times 64$	$63 \times 63 \times 56 \times 90$

Table 6.1: Data set sizes

sets decreased in two or three directions and increased in one direction. Table 6.2 shows the pre-processing time which was needed to convert the data sets from Cartesian grid to BCC or D_4^*. This pre-processing needs to be executed only once, unless some compression parameters need to be changed. All data sets which are seen in Table 6.1 were transformed to the BCC or the

Data set	Time (total)	Time (Hex)	Time (Wavelet)
Nucleon BCC	0.66	0.62	.02
Lobster BCC	58.0	54.9	8.5
Statue Leg BCC	112.6	106.4	5.7
Engine BCC	93.5	87.8	5.2
UNC Brain BCC	116.1	109.6	5.9
Kidney D_4^*	976.9	967.9	8.5

Table 6.2: Pre-processing time (in seconds)

D_4^* lattice and then refit to the next 2^n texture size for OpenGL. Currently all 3D textures have to be specified as 2^n in size. This will change soon with future implementations of OpenGL, which will allow the specification of arbitrary 3D texture sizes. However, until then some savings which are gained through the conversion to the hexagonal lattice can not be fully used because the data has to be embedded in the next bigger texture. Nevertheless, because the BCC data needs fewer samples, the remaining voxels for the next texture resolution are zero padded. This data can be encoded very efficiently and the resulting data size is usually smaller than the regular Cartesian. Additionally, because the BCC lattice has fewer samples, also fewer slices are needed to resample the volume which increases the overall performance. Table 6.3 shows the results for the hexagonal conversion in

Data set	MB	fps.
Nucleon CC	68 kb	73.2
Nucleon BCC	48 kb	82.6
Lobster CC	5.4 MB	13.1
Lobster BCC	3.8 MB	9.4
Statue Leg CC	10.4 MB	27.6
Statue Leg BCC	7.4 MB	15.06
Engine CC	8 MB	31.6
Engine BCC	5.9 MB	7.1
UNC Brain CC	9.5 MB	20.88
UNC Brain BCC	6.7 MB	7.6
Kidney CC	40 MB	25.26
Kidney D_4^*	20 MB	28.16

Table 6.3: Data sets CC vs. BCC

memory space and gives performance results in frames per second (fps). As expected, the savings which are gained through the conversion are 30 percent for the BCC lattice and 50 percent for the D_4^* lattice. Further reduction in the data size can be accomplished by using compression techniques. For this implementation, the integer wavelet transform is used and the data sets can be compressed using the following CDF-filters [CDF92]:

- (1,1), (1,3), (1,5),
- (2,2), (2,4), (2,6),
- (4,2), (4,4) and (4,6).

After the decomposition of the data set, the high frequencies are thresholded and encoded using RLE. Table 6.4 shows some results for two data sets with three selected wavelets. The wavelet decomposition is performed three levels deep and different thresholds were used, as can be seen in Table 6.4. Some of the high frequencies can be discarded in order to achieve a higher compression ratio. This results in less detail information, and the RLE algorithm can preform longer runs. However, due to the loss of information, the original signal can not be reconstructed entirely. Examples can be seen in Figures 6.11 and 6.12 in Section 6.1.2.
As can be seen from Tables 6.1 to 6.4 the original data sets can be stored very efficiently using the described techniques. The increase in rendering performance is not yet as visible for all data sets, due to the requirement of using textures with the size of 2^n.
For time varying data sets it is very important to be able to interactively change the current frame. Even without the additional optimizations which were discussed in Chapter 4.4.4, one can directly benefit from the D_4^* lattice. Table 6.5 shows the time which is needed to load another time frame and display it. The qualitative results from the previous section and also

Wavelet	Threshold	Frog CC	Skull CC
uncompressed		30.9 MB	16.0 MB
(1,1)	0.0	532.6 kb	701.1 kb
(1,1)	1.5	495.2 kb	419.5 kb
(1,1)	5.5	361.2 kb	184.5 kb
(2,2)	0.0	586.6 kb	705.6 kb
(2,2)	1.5	532.9 kb	534.7 kb
(2,2)	5.5	383.3 kb	233.4 kb
(4,2)	0.0	753.9 kb	696.6 kb
(4,2)	1.5	700.2 kb	696.4 kb
(4,2)	5.5	662.4 kb	696.5 kb

Table 6.4: Data sets (compression)

Data set	Frames	Time (seconds)
Kidney CC	64	0.12
Kidney D_4^*	90	0.018

Table 6.5: Time to change a frame

the quantitative results in this section have shown the usefulness and the applicability of the described approach. Using more efficient grids to store the data sets and adaptive compression techniques together with a pre-segmentation of the data set can decrease the size of the data by a large magnitude without degrading the visual quality of the visualization.

6.3 Multiparameter Visualization

Chapter 5 discussed possibilities for the multiparameter visualization of the fuel cell data set. While the focus in Chapter 4 was on fast visualization of huge time-varying data sets, the focus in Chapter 5 was on how to create expressive images from more than one data set. All techniques where explained and described using the fuel cell data, but are also applicable for the visualization of other multiparameter data sets.
Some selected techniques have been implemented, as can be seen in Figure 6.13. Figure 6.13 shows a little demo program which was used to experiment with different, commonly used visualization techniques. The visualized data set is only small in size, no special optimizations were necessary to increase the rendering performance. Possible visualization techniques are:

- direct volume rendering with different rendering functions,
- arbitrary data slicing (colour/contour),
- streamlines,
- 3D glyphs,

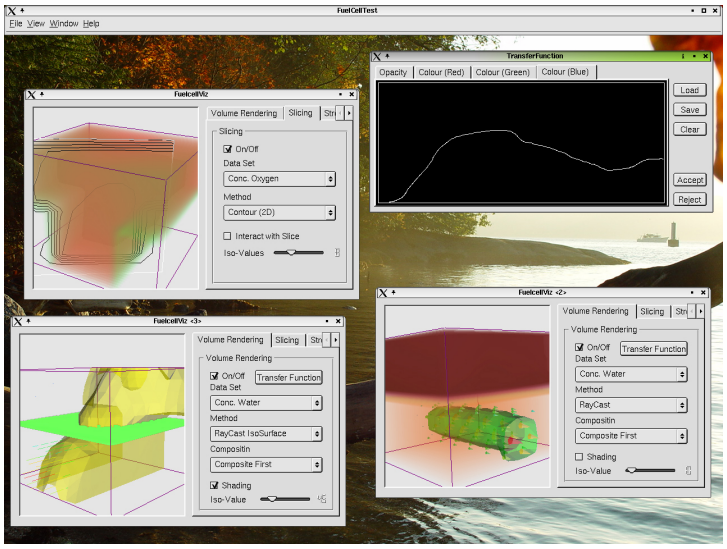

Figure 6.13: Simple fuel cell visualization

- hedgehogs, and
- iso-surfaces.

The implemented approach is the layer concept which was described in Chapters 5.2 and 5.3. Here one has the possibility to chose one of the six data sets which are available for the fuel cell data and visualize this one using an appropriate technique. One can use several different visualizations of one data set and blend them together, or different data sets and combine these to the final visualization. Chapter 7.1 shows some more screenshots and also some implementation details.
Other covered techniques are classic multiparameter visualization methods, like scatterplots, hierarchy, shadow projections, probing, special lenses and customized glyphs. Most of these techniques are described in theory and an possible application for the fuel cell data is outlined. Here, the special lenses are a very universal tool which allow a great degree of freedom and which can be combined with other techniques in several ways. But also higher-dimensional data visualization and non-photorealistic rendering techniques have some qualities which are useful for the visualization of the fuel cell data set.

Chapter 7

Design and Implementation

This Chapter is used to present some implementation details and to give more insight into the design of the implemented programs. The chapter is basically divided into two parts.
The first section shows a program which was used to visualize a small fuel cell data set using common visualization techniques. The program was designed to demonstrate the possibilities of these methods and how they could be adapted to visualize fuel cell simulations. One goal in the design process was also that this program can be used as a framework where additional visualization techniques can easily be implemented and evaluated.
Sections 7.2 and 7.3 provide further information for the implementation of the algorithms which were developed in Chapter 4. The method which is used to visualize huge time-varying data sets can be split into a preprocessing step and the final render part. Here, section 7.2 shows detailed information about the implementation of the different compression techniques and how the data is stored and organized on disk. Section 7.3 explains the implementation of the actual rendering algorithm which uses the previously generated compressed data sets. Here, special attention is payed to interactive volume rendering using OpenGL hardware and how the different classification and rendering techniques are actually implemented.

7.1 Simple Fuel Cell Visualization

The program which is presented in this section was implemented to test the applicability of some general visualization techniques for the fuel cell data set. The data set which was used for this program is very small in size ($13 \times 11 \times 100$), and hence no optimization techniques were necessary to improve the interactivity. The fuel cell data contains the following data sets:

- concentration of oxygen O_2,

- concentration of hydrogen H_2,
- pressure p,
- temperature T,
- velocity (vx, vy, vz), and
- concentration of water H_2O.

A more detailed description can be found in Chapter 2. The goal for this program was to build a demo application which can be used for discussions with mathematicians that generated the simulation data. Figure 7.1 shows a screenshot of the application displaying different visualization techniques. Another focus of the development of this program was to build a framework

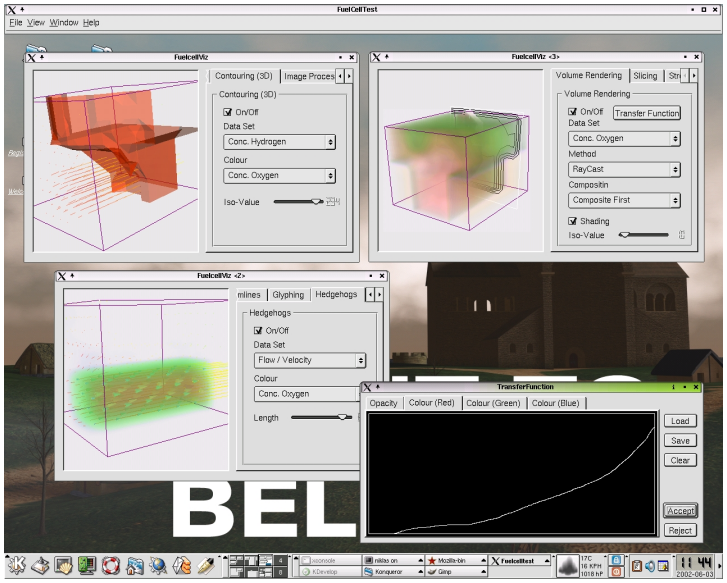

Figure 7.1: Simple fuel cell visualization

which could be used as a Rapid-Prototyping Environment to implement new visualization ideas for multiparameter data sets. The user interface was implemented using Qt [AS92] and build using the Qt Designer. For the graphic part, the Visualization Toolkit from Kitware Inc. [SMLS98] was chosen which allows one to easily implement all the necessary visualization techniques. It can also be used to write the program in a way that it can be easily extended with new visualization techniques.
To improve the user interface an extension to the Qt interface was used which allows a better MDI framework [Bre99]. The program enables one to either use one single method for one data set, or to combine different visualization techniques using the layer principle with one another (Chapter 5.2). Here, MDI can be used to compare several different combinations and to chose the one which presents the underlying information best.
The implemented techniques are:

- direct volume rendering with different blending modes,
- arbitrary slicing with colour and/or contouring,
- streamlines,
- 3D glyphs,
- hedgehogs, and
- iso-surfaces.

All techniques can be used with all data sets. Some methods also allow the combination of two different data sets. As an example, one can use the concentration of oxygen for colour mapping and the temperature data set to generate the geometric primitive which is used to map the information into visible form. Figures 7.2 and 7.3 show some more screenshots of the program. In Figure 7.2 one can see an example for volume rendering where the data set which describes the concentration of oxygen is displayed. Several different rendering parameters can be specified and transfer functions for colour and opacity can be defined using a dialog box which can be seen in Figure 7.1. In a pre-processing step, for all scalar data sets, the gradient

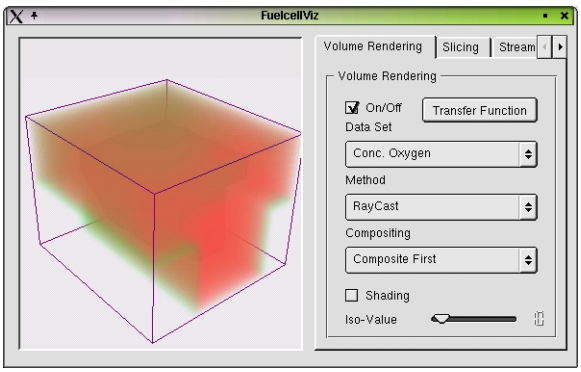

Figure 7.2: Direct volume rendering

is computed, so that these data sets can also be used for flow visualization, Figure 7.3. Also, from the vector data set, the magnitude is extracted and can be used for the visualization as well. Figure 7.3 shows the the flow data set rendered using hedgehogs. Hedgehogs are, as streamlines and 3D glyphs, used to depict flow information. Unlike 3D glyphs, hedgehogs are simple line segments which point in the direction of the flow and are scaled and coloured depending on the flow magnitude. Figure 7.3 shows the flow data, which is scaled and coloured using the oxygen concentration.
Other very common and supported visualization techniques are slicing, where an arbitrary slicing plane is used to extract information on a 2D image. This information can be presented using a colour scale and/or contour lines. Streamlines are also used to describe flow information. Here, particles are set into the data set which are traced and their path is visualized. Very im-

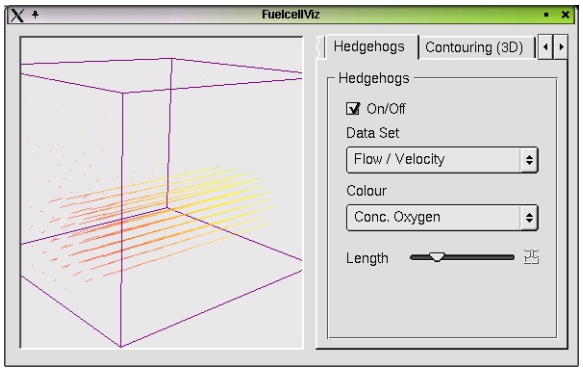

Figure 7.3: Hedgehogs

portant for the fuel cell data set is also the extraction of iso-surfaces. This contour information can be used to display the different layers of a data set and to highlight the changes in intensity. This is also interesting for the flow data. Some of these techniques are depicted in Figure 7.1 as well as described in more detail in Chapters 5.2 and 5.3.
The next section describes shortly some interesting implementation details.

7.1.1 Implementation Details

The user interface was designed that it can be easily extended with new visualization techniques. One would just have to add a new tab-box and implement the graphics related part using the VTK. The interface is also small in size which helps to compare several different renditions on one screen to evaluate their applicability. For the design of the user interface, the Qt designer was used which built a base class of the gui where one only had to derive a new class from to implement all gui related methods and to update the VTK rendering pipeline.
The VTK works in a way that one creates a rendering pipeline which is executed every time a new image has to be created. This can occur when the canvas was damaged or the data has changed. All visualizations are implemented as classes in C++ and can be added to and removed from the rendering pipeline as can be seen in Example 7.1:

```
viewer->GetRenderWidget()->GetRenderer()->AddActor(volume);
viewer->GetRenderWidget()->GetRenderer()->AddActor(glyph);
viewer->GetRenderWidget()->GetRenderer()->RemoveActor(contour);
```

Example 7.1: Enabling/Disabling visualizations

In the above example one can see how visualizations can be enabled or disabled using the VTK. This allows an easy implementation of the layer concept where several different visualizations can be blended together (see Chapter 5.2). In this example, iso-surfaces are disabled and volume rendering and 3D glyphs are turned on.

Example 7.2 shows a small part of the actual rendering pipeline. Here the part which is responsible for the volume rendering is set up:

```
opacityTransferFunction = vtkPiecewiseFunction::New();
  opacityTransferFunction->AddPoint(0.1, 0.0);
  ...
colorTransferFunction = vtkColorTransferFunction::New();
  colorTransferFunction->AddRGBPoint(0.1, 0.3, 0.3, 1.0);
  ...
volumeProperty = vtkVolumeProperty::New();
  volumeProperty->SetColor(colorTransferFunction);
  volumeProperty->SetScalarOpacity(opacityTransferFunction);
  ...
compFunction = vtkVolumeRayCastCompositeFunction::New();
  compFunction->SetCompositeMethodToClassifyFirst();
volumeRayCastMapper = vtkVolumeRayCastMapper::New();
  volumeRayCastMapper->SetInput(volumeData);
  volumeRayCastMapper->SetVolumeRayCastFunction(compFunction);
volume = vtkVolume::New();
  volume->SetMapper(volumeRayCastMapper);
  volume->SetProperty(volumeProperty);
```

Example 7.2: Volume rendering using the VTK

Example 7.2 only shows a small part of the pipeline to set up the volume rendering. First the two transfer functions for opacity and colour are created which are later assigned to the volume property that handles all the volume attributes. Here one can also define characteristics for shading and lighting. Next a composite function is created and assigned to the ray caster which specifies the used ray function (Chapter 3.1.4). Here, also the data set is assigned to the ray caster. This class performs the actual volume rendering. The volume which is defined next handles all the events and initiates the rerendering if necessary.
Some image processing is also performed and used in a pre-processing step to compute the gradient information. The data sets can also be low-pass filtered using a gaussian filter to achieve smoother images. Because all data sets have a different data range and type, they need to be resampled in the range of 0 to 255. Therefore, a simple VTK class was implemented to perform this task.

7.2 Volume Compression

The rendering pipeline which was developed in Chapter 4 can be divided in two parts. This section will be used to explain the pre-processing step in which the data set is resampled onto a more efficient grid to save memory space (Section 7.2.1). After the resampling, the resulting data is pre-segmented into regions of equal importance (Section 7.2.2). Unimportant

parts of the data sets are discarded. When this is finished, each brick is compressed using either wavelets (Section 7.2.3). The detail coefficients are thresholded depending on the importance of the brick and encoded using RLE and finally stored onto disk (Section 7.2.4).
Figure 7.4 shows a screenshot of the compression utility. It allows to import any kind of data set which will be first converted to unsigned char. On the left side some widgets can be seen which allow to specify information for the data import. Below this menu and on the right side are other widgets where one can define features for the conversion to hexagonal grids, compression and bricking. A progress bar on top of the application informs about the advance of the time-consuming process. The interface is designed and imple-

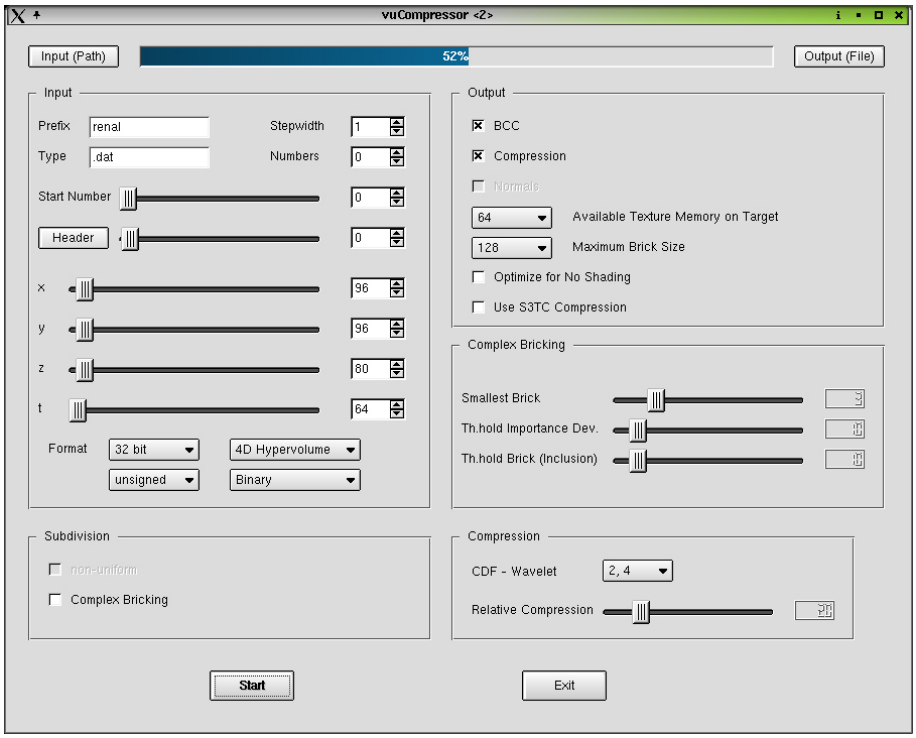

Figure 7.4: vuCompressor

mented using Qt and QtDesigner [AS92]. A first version was implemented using wxWindows [wxw92], but later discarded because of its inflexibility in comparison to the Qt library.

7.2.1 BCC Grids

After a data set is loaded and converted to byte data, it will be resampled onto the hexagonal lattice. Static data sets are resampled to BCC versus time-varying data sets which are resampled to D_4^*, see also Chapter 4.1 for more details. The conversion is simply done by using cubic interpolation

filters. Example 7.3 shows a code snippet which is used to resample a 4D Cartesian data set into the D_4^* lattice

```
// Compute resolution of hexagonal data set
m_xDim = int(double(xDim)/T);
m_yDim = int(double(yDim)/T);
m_zDim = int(double(zDim)/T);
m_tDim = int(double(tDim)/(T*0.5));
unsigned long vols = m_xDim*m_yDim*m_zDim;

for (int t=0; t < m_tDim; t++)
  for (int z=0; z < m_zDim; z++)
    for (int y=0; y < m_yDim; y++)
      for (int x=0; x < m_xDim; x++)
      {
        /*Find out the x, y, z and t coordinates of
        the data point in the rect grid*/
        double xRect = T * (x + (t%2)*0.5);
        double yRect = T * (y + (t%2)*0.5);
        double zRect = T * (z + (t%2)*0.5);
        double tRect = 0.5*T*t;
        //Interpolate the new data point
        m_Data[x+(y+z*m_yDim)*m_xDim+vols*t] =
              getCart(xRect,yRect,zRect,tRect);
      }
```

Example 7.3: Resampling to D_4^*

In this example first the new dimensions of the data set are computed. The x, y and z axes decrease by $T = \sqrt{2}$ and the time axes increases by $\sqrt{2}$. Then for each of the new data points the corresponding location in the Cartesian data set is computed and interpolated using the function *getCart(x, y, z, t)*. The resampling to BCC is similar, except that the algorithm only loops over x, y and z and that the sampling occurs on different positions.

7.2.2 Subdivision

After the data set is resampled onto a hexagonal lattice, it is eventually subdivided into smaller bricks if necessary. If the data exceeds the capacity of the available texture memory, then a pre-segmentation and bricking algorithm is used to break down the data set into smaller pieces. A more detailed description can be found in Chapter 4.2. For the bricking, two techniques are available. The first one subdivides the data by uniform splitting the volume into the biggest possible texture sizes. This technique can be seen in Example 7.4.

```
void VolumeRoot::SubdivideSimple()
{
// compute the biggest texture size ...
unsigned short bg_tex = 128;
```

```
unsigned int counter = 0;
unsigned short xPos, yPos, zPos;

// allocate memory for each brick...
sub_data = (SubVolume *)malloc(number_of_bricks * sizeof(SubVolume));

// subdivide the volume into biggest possible textures
for (int x=0 ; x<texBigX ; x++)
  for (int y=0 ; y<texBigY ; y++)
    for (int z=0 ; z<texBigZ ; z++)
    {
      xPos = x*(bg_tex-1);
      yPos = y*(bg_tex-1);
      zPos = z*(bg_tex-1);
      sub_data[counter] = SubVolume((GLint)counter, this,
                          bg_tex, bg_tex, bg_tex , 1.0);
      sub_data[counter].setData(xPos, yPos, zPos, bg_tex, bg_tex, bg_tex);
      counter++;
    }
```

Example 7.4: Simple bricking

In this example first the biggest possible texture is computed. For this simple example it is given, but usually computed by querying the available graphics hardware. The biggest texture size depends also on the size of the frame buffer and the rendering method used. If the gradient information is not needed, the texture can be four times larger. After this the number of bricks is computed and memory space to hold the required information for the *sub volumes* is allocated. Then every brick is initialized and the textures are loaded overlapping to avoid boundary artifacts.
The complex bricking is not shown here, as this would be too big and complex to fit here. In complex bricking, first a volume is computed which exhibits the temporal changes of a time-varying data set. After this, the importance for several small bricks is computed and evaluated. Bricks which are below a certain threshold are not included in the visualization. All other bricks are merged together depending on their importance to increase the efficiency. This importance is also used later for the wavelet decomposition and the determination of the Level-of-Detail.

7.2.3 Compression and Multiresolution

After the data has been processed and all sub volumes are created, each brick is further compressed and decomposed using wavelets. The implemented wavelet approach uses the lifting scheme which can be used to perform a real lossless integer wavelet decomposition. For the implementation, the Waili package [UVWJ+98] was used.
Examples 7.5 and 7.6 show how the wavelet compression is performed.

```
wavelet = IWT3d(1,1);
wavelet.setData(cData, xdim, ydim, zdim, compressionRatio);
wavelet.compress(level);
```

Example 7.5: Wavelet decomposition I

Example 7.5 shows simply how the wavelet is created and how the data is assigned prior to the compression. This is performed for each sub volume. The (1,1) wavelet in this example is the simple Haar wavelet for integer lifting.
Example 7.6 shows how the wavelet decomposition is actually performed.

```
void IWT3d::FastWaveletTransform()
{
  unsigned int id = wave->GetID();

  TransformDescriptor td2d[] = {{TT_ColsRows, id}};
  TransformDescriptor td1d[] = {{TT_Rows, id}};

  // transform xy slices ...
  for(unsigned short z=0 ; z<zMax ; z++)
  {
    NTChannel ch(xMax, yMax);
    copyXYSlice(ch, z);
    LChannel *lch = ch.Fwt(td2d, 1);
    IcopyXYSlice(*lch, z);
    delete lch;
  }

  // transform along z axis ...
  for(unsigned short z=0 ; z<zMax ; z++)
  {
    NTChannel ch(zMax, yMax);
    copyZSlice(ch, z);
    LChannel *lch = ch.Fwt(td1d, 1);
    IcopyZSlice(*lch, z);
    delete lch;
  }
}
```

Example 7.6: Wavelet decomposition II

In example 7.6 first the *xy* slices are compressed followed by the *z*-axis. The data is assigned into a non-transformed *NTChannel* and converted using the fast wavelet transform into a wavelet transformed *LChannel*. This wavelet decomposition is repeated on the low frequency volume depending on the assigned level depth, as can be seen in example 7.5. Every time frame of a time-varying data set is compressed in this manner, but here, the compression ratio can be adjusted, depending on the information contained in this brick at this point in time.

7.2.4 Encoding and Storage

After the data is decomposed into low and high frequencies the low resolution volume is stored uncompressed on the disk, and the remaining detail information is written successively into the same file. This way, first the low resolution volume can be accessed for fast preview and then successively the volume is reconstructed to the required resolution. Example 7.7 shows the RLE algorithm used.

```
int SubVolume::RLECompress(ofstream& fout, byte *input, int length, int flag)
{
  int index;
  byte pixel;
  int out = 0;
  int count = 0;
  byte *writeData = (byte *)malloc(length * 1 << flag * sizeof(byte));

  while (count < length)
  {
    index = count;
    pixel = input[index++];
    while (index < length && index - count < 127 && input[index] == pixel)
      index++;
    if (index - count == 1)
    {
      while ((index < length) && (index - count < 127) &&
             ((input[index] != input[index-1] || index > 1 &&
               input[index] != input[index-2])))
        index++;
      while (index < length && input[index] == input[index-1])
        index--;
      writeData[out++] = (byte)(count - index);
      for (int i=count ; i<index ; i++)
        writeData[out++] = input[i];
    }
    else
    {
      writeData[out++] = (byte)(index - count);
      writeData[out++] = pixel;
    }
    count=index;
  }

  // save file
  fout.write((byte *) writeData, out);
  writeData = 0;
  return(out);
}
```

Example 7.7: Run Length Encoding

In this example the input data set which is encoded and a pointer to the file are given as input. The implementation is the same as described earlier in Chapter 3.3.1, except this method only searches for long runs in one direction. After the encoded data is written into the file, the length is returned to the root volume and stored in a header file for later access when the volume is reconstructed during the rendering.

7.3 Volume Rendering

The last section showed in a few selected code examples how the pre-processing step is performed. This section describes how this data is used for the final visualization. Figure 7.5 shows a screenshot of the application visualizing the bonsai tree data set. The main widget of this application is

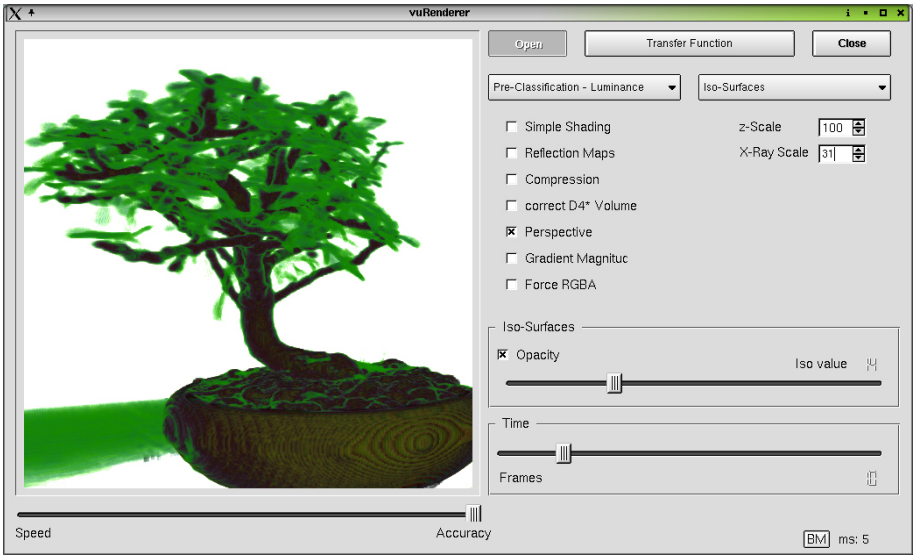

Figure 7.5: vuRenderer with bonsai tree

the render window on the left side. Below this one is a small slider which can be used to manually adjust between either high quality or fast interaction. Section 7.3.3 will explain more on this. On the right are some buttons, checkboxes and slider which are used to load a data set and to change some rendering parameters. Here a different transfer function can be applied as well as a different rendering style, like iso-surfaces or mip. The bottom right corner shows the rendering time in milliseconds and allows a benchmark which reports in frames per second.
As with the *vuCompressor*, this interface is also implemented using Qt and the QTDesigner [AS92]. In the following some small code examples are presented on volume rendering using 3D textures, simple classification and

shading and how Level-of-Detail can be included to increase the performance.

7.3.1 Rendering

This section is to demonstrate how 3D textures can be employed for volume rendering using texture mapping hardware. In this section it is shown how the data set is loaded and rendered, while the next section demonstrates how simple classification and shading can be performed. Example 7.8 shows how a 3D volume can be load a s 3D texture.

```
glActiveTextureARB(GL_Texture0_ARB);
glBindTexture( GL_Texture_3D_Ext, brick );
glTexParameteri(GL_Texture_3D_Ext, GL_Texture_Wrap_S, GL_Clamp);
glTexParameteri(GL_Texture_3D_Ext, GL_Texture_Wrap_T, GL_Clamp);
glTexParameteri(GL_Texture_3D_Ext, GL_Texture_Wrap_R_EXT, GL_Clamp);
glTexParameteri(GL_Texture_3D_Ext, GL_Texture_Min_Filter, GL_Linear);
glTexParameteri(GL_Texture_3D_Ext, GL_Texture_Mag_Filter, GL_Linear);
glTexEnvi(GL_Texture_Env, GL_Texture_Env_Mode, GL_Replace);

glTexImage3D(GL_Texture_3D_Ext, 0, GL_Color_Index_Ext, xdim, zdim, zdim,
             0, GL_Color_Index, GL_Unsigned_Byte, (GLubyte *)texture);
```

Example 7.8: 3D Texture Loading

Here the data is loaded as 3D texture and can be accessed using the texture number which is stored in *brick*. Some parameters have to be defined to correctly setup the texture. Example 7.9 demonstrates how the actual rendering is performed.

```
glActiveTextureARB(GL_Texture0_ARB);
glMatrixMode(GL_Texture);
glBindTexture( GL_Texture_3D_Ext, brick );
glPushMatrix();
glLoadIdentity();
glTranslatef( .5f, .5f, .5f);
glScalef( scale_x, scale_y, scale_z );
glRotatef(objangle[1], 1.f, 0.f, 0.f);
glRotatef(objangle[0], 0.f, 0.f, 1.f);
glTranslatef( -.5f, -.5f, -.5f);

// blending ...
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

for(int i=0; i<slices ; i++)
{
  glBegin(GL_QUADS);
    glVertex3f(-150.f,-150.f,-150.f+offR+i*(300.f-2*offR)/(slices-1));
    glVertex3f( 150.f,-150.f,-150.f+offR+i*(300.f-2*offR)/(slices-1));
```

```
    glVertex3f( 150. f , 150. f,−150.f+offR+i∗(300.f−2∗offR)/(slices−1));
    glVertex3f(−150.f, 150. f,−150.f+offR+i∗(300.f−2∗offR)/(slices−1));
  glEnd();
}
```

Example 7.9: Volume Rendering

In this example first the matrix mode is changed to the texture mode where the texture is rotated around the centre, depending on the current interaction. After this, blending is enabled and the texture is resampled by slicing planes through the volume. The texture coordinates are created automatically, and setup in a pre-processing step. Each slice is tri-linearly interpolated in hardware. To avoid ghosting artifacts, due to the clamped texture, clipping planes have to be enabled and also rotated as the texture. Other blending functions and the alpha test can be used to simulate different rendering techniques. See also Chapter 4.4 for more details.

7.3.2 Classification

Classification is a very important step in volume rendering and can be performed in hardware in pre- and post-classification (Chapter 4.4.6). Example 7.10 shows how simple pre-classification can be performed by loading the transfer function as OpenGL colour table.

```
colourtable  = (GLubyte ∗)malloc(1024 ∗ sizeof(GLubyte));
unsigned int counter = 0;
for (int i=0 ; i<256 ; i++)
{
  colourtable[counter] = (GLubyte ) i;
  colourtable[counter+1] = (GLubyte ) i;
  colourtable[counter+2] = (GLubyte ) i;
  colourtable[counter+3] = (GLubyte ) i;
  counter += 4;
}

// download colourtable
glEnable(GL_Color_Table);
glEnable(GL_Shared_Texture_Palette_Ext);
glColorTable(GL_Shared_Texture_Palette_Ext, GL_RGBA8, 256,
             GL_RGBA, GL_Unsigned_Byte, (GLubyte ∗)colourtable);
```

Example 7.10: Pre-classification

First a default transfer function is created which is simply a ramp function. This transfer function is downloaded to the texture memory as a colour table and accessed by the 3D texture via indices, see also the section before and Example 7.8. The advantage is that this colour table can be changed very fast and real time classification is possible also for huge data sets.
Shading can add a better perception of the 3 dimensionality of the visualized object. Here the data set is loaded as a RGBα texture with the pre-computed

and normalized gradient information stored as RGB values (see Chapter 4.4.6). Then simple shading can be performed using the texture environment dot3 function, as can be seen in Example 7.11.

```
glTexEnvi(GL_Texture_Env, GL_Texture_Env_MODE, GL_Combine_Ext);
glTexEnvi(GL_Texture_Env, GL_Combine_Alpha_Ext, GL_Replace);
glTexEnvi(GL_Texture_Env, GL_Combine_RGB_Ext, GL_DOT3_RGB_Ext);
glTexEnvi(GL_Texture_Env, GL_Source0_RGB_Ext, GL_Primary_Color_Ext);
glTexEnvi(GL_Texture_Env, GL_Operand0_RGB_Ext, GL_SRC_Color);
glTexEnvi(GL_Texture_Env, GL_Source1_RGB_Ext, GL_Texture);
glTexEnvi(GL_Texture_Env, GL_Operand1_RGB_Ext, GL_SRC_Color);
```

Example 7.11: Simple shading

7.3.3 Level-of-Detail

Level-of-Detail is an important feature when interactivity and quality are needed at the same time. LoD can be used such that the visual quality is neglected and set to a lower resolution when interactivity is needed. At the same time, the total performance increases and helps when navigating through the data set. If the goal of interaction is reached, the better resolution can be reloaded in background and used for further rendering.
LoD is used in the implementation in many ways. If a data set, or a new time frame is loaded, then first the low resolution volume which was generated using the wavelet decomposition, is loaded and displayed for fast feedback. Then two threads for each brick are started which compute the next higher resolution volume, as well as the gradient if needed. Example 7.12 shows a small code sample.

```
void reconstructNextLevel()
{
  // create threads
  pthread_t thread1, thread2;
  int iret1 , iret2 ;

  // Create independant threads
  iret1 = pthread_create( &thread1, NULL, (void*)&computeNextWavelet, (void*) level);
  iret2 = pthread_create( &thread2, NULL, (void*)&computeGradient, (void*) level);

  ...

}
```

Example 7.12: Multi threading

LoD is also used when the bricking technique has to be employed to visualize data sets which do not fit in the texture memory. Here two situations can be exploited for Level of Detail. First, bricks which are far away from the camera can be rendered view-dependent using a smaller resolution and less slices. Also, if the complex bricking, see Section 7.2.2 and Chapter 4.2, is

used then each brick can be rendered using a unique LoD which is computed by evaluating the importance of the brick.

Chapter 8

Summary and Future Work

The goal of this thesis was to examine the possibilities to visualize fuel cell simulations. As these data sets are huge in size and multiparametric, two areas of research had to be explored. Important for every visualization task is the quality of the display as well as the interactivity which enables the end user to explore the data to gather information. One is interested in detecting features in the data and separating them from other structures. This process can not be automated, but the software can provide tools to support these features.
As withe the data set, also the thesis is split into two parts. The first part was the development of a pipeline which allows one to interactively explore and visualize huge time-varying volumetric data sets by using compression techniques and exploiting commonly available graphics hardware. The second part was the analysis of possibilities for the visualization of multiparameter data sets which allows one to draw connections between several dependent data sets.
This Chapter is divided into two sections. The first one briefly summarizes the work and states the contribution of this thesis. The second part describes possibilities for future improvements and compares them with the existing solution.

8.1 Summary

The goal of this section is to summarize the work and to review the thesis shortly. In the end of this section the contributions of this thesis are presented and discussed. The intention behind the thesis was to improve the existing visualizations for the fuel cell data sets and to develop appropriate techniques for an expressive visualization. Even though the main focus was on the visualization of fuel cell data, all methods can be used with other data sets as well. Chapter 2 discussed some data sets used and explained

the fuel cell data set in more detail.
Chapter 3 was used as an introduction into the topic and explaind some pre-requisites in signal and wavelet theory as well as in volume rendering. These requirements were needed for the later discussions in Chapters 4 and 5.
Chapter 4 outlined a new rendering pipeline by combing existing technologies with new ideas into a volume rendering application. The presented technique is able to render huge data sets at interactive rates on standard commodity graphics workstations. The algorithm makes use of several compression techniques and a pre-segmentation of the data set based on spatial and temporal coherency which allows to decrease the size of the data set by a large magnitude. Hardware assisted volume rendering is used for the visualization of the data and adds to the interactivity as well as the quality of the display. State of the art classification and shading techniques have been implemented and extended.
In Chapter 5 several techniques suitable for the visualization of multiparameter data sets were examined and explored. Although the focus was on the visualization of fuel cell data sets, all techniques which were discussed are also applicable to most other multiparameter data sets. Here some general visualization goals were presented and adapted to the fuel cell data set. New ideas were shown for higher dimensional data visualization as well as for the use of non-photorealistic rendering techniques for volumetric data sets.
Qualitative and quantitative results were presented and discussed in Chapter 6. The first section of this chapter shows some screenshots of different data sets to evaluate the quality of the used rendering techniques. Here, also comparisons are shown and discussed for the compression techniques used with the BCC lattice and wavelets. The second part of Chapter 6 presented the achievable compression ratios and with how many frames per second the rendering could be performed.
Details for the implementation are shown and discussed in Chapter 7. Basically two applications have been implemented. The first one is used to demonstrate some simple multiparameter visualizations for a small fuel cell data set. This application was developed as a framework which allowed an easy extension with other visualization techniques for evaluation. The second application implemented most parts of Chapter 4. For better performance the algorithm was split into two programs, one for the pre-processing of the data set, and another one for the final rendering and visualization. Both programs are discussed and several important functions are shown in more detail.
Because the thesis covers several different fields of research, contributions are made in divers areas and shall be discussed in the following. The big advantage of the proposed approach from Chapter 4 is that it combines several techniques which allows one to visualize huge volumetric data sets at interactive rates. The method described allows the use of static as well

as time-varying data sets. Most of these techniques were never combined in similar way. The BCC lattice is very new to the scientific visualization community. It was first introduced in 2001 for splatting volumetric data sets [TMG01]. The D_4^* lattice was only described once [NM02]. Neither of these lattices has been used for hardware accelerated volume rendering using texture mapping hardware.
The bricking algorithm which helps to render data sets which are larger than the available texture memory is described in several publications. However, the drawback of the standard technique is that it is performed in a brute force way, by subdividing the entire volume. The bricking method which was described in Chapter 4 first segments the data set regarding to the inherent coherency and only uses those regions which are really interesting for the visualization. It also assigns a unique LoD to each brick which is used during the rendering to increase the performance. Wavelet compression for hardware based volume rendering is already described by three other groups. However, the current implementations are not able to losslessly reconstruct the original volume due to standard wavelet decompositions used. Wavelets have never been applied to 3- or higher dimensional hexagonal data sets, e.g. BCC and D_4^*.
The visualization of multiparameter data has mostly been discussed in theory. Only a few selected examples have been implemented: a simple fuel cell visualization which uses the layer principle and unimodal visualization techniques and a simple non-photorealistic volume rendering tool which uses the gradient magnitude and texture mapping hardware. Other contribution here are the evaluation of the different visualization techniques for the fuel cell data set, how they can be adapted and which combination would be most beneficial for the visualization. The areas of higher dimensional data visualization and the use of non-photorealistic rendering techniques for scientific visualization purposes have been discussed in more detail. Volume and iso-surface visualization using a hypercubic lattice is not thoroughly explored yet. Also the application of NPR techniques for data compression has not been described yet but would be very useful for the interactive rendering of huge volumetric data sets.

8.2 Future Work

As with most projects, there is always room for improvement. Even though the presented approach yields promising results, it is just a start for the interactive visualization of terascale volumetric data sets.
For the resampling of the data set onto a hexagonal lattice, cubic interpolation has been used. Although these are good filters for resampling, better results can be achieved by using interpolating higher polynomial filters or spline interpolation. One could also apply image processing techniques prior

or past the sampling to enhance the signal. Here edge enhancement could be used to antagonize the blurring which is introduced by the resampling.
The current used bricking method could be enhanced by using better filters to compute the information content of a brick. Here wavelets can be used to extract detail and structural information [GMR97]. The merging step which is used to combine regions of similar importance together into larger bricks is currently performed only in one predefined direction. Wavelets can be used to *suggest* areas with a high homogeneity and which are *suitable* for merging. One huge problem in multiresolution bricking is that neighbouring bricks may not correctly overlap at the texture border if they use different resolution levels. One solution is to include one slice of additional voxels, but because all textures have to be 2^n, the next texture size has to be used and one uses an inferior texture with the same memory requirements. This might change when the definition of arbitrary texture sizes also becomes available for 3D textures.
Non-separable filters can be used for the wavelet decomposition to avoid directional aliasing and to achieve a better image quality. This would be very advantageous for the BCC lattice, where one has nine axes of symmetry. A good starting point here would be the adaptation of the butterfly interpolation scheme for BCC grids. Additionally different encoding techniques could be used which might perform better than the used RLE algorithm. Candidates here are Huffman coding or LZW encoding [Sal98].
To improve the rendering, capabilities of today's and future graphics hardware could be used more extensively. The BCC lattice could eventually be sliced and resampled using vertex shaders. Also better classification could be performed with higher dimensional transfer functions. By including the first and/or the second derivative, one could use 3D dependent textures which are available within the Texture Shader3 option [*n*vi01], and use the approach from Kindlmann [KD98] to built good opacity transfer functions *automatically*. Better interpolation filters to resample the texture should be used as well. This would results in an increase in image quality by a large number. With future graphics hardware which would support more than four texture units and seven general combiners this might even be possible with post-classification using dependent texture lookups in one single pass.
The visualization of multiparameter data sets would directly benefit from all these improvements as they increase the performance and the display quality. The use of better graphical primitives might be helpful for the mapping of the information into the visualization which would allow an easier and faster recognition of the content. Here techniques from NPR can be used to highlight specific areas and to focus on interesting parts of the data only. As a side effect, these methods might be able to present the same information with less data. Additionally, new ideas of interacting could be used to faster explore the information within the data set. These methods would depend on the type and additional information about the data set. And

last, an improved user interface will help to easier find the information one is seeking.

List of Figures

List of Examples

Bibliography

[Abb94] Edwin Abbot. *Flatland: a romance of many dimensions*. 1894.

[AM76] Neil W. Ashcroft and N. David Mermin. *Solid State Physics*. Prentice-Hall, 1st edition, 1976.

[AS92] Trolltech AS. Qt, 1992. `http://www.trolltech.com/products/qt/index.html`.

[ATi01] ATi. Ati website, 2001. `http://www.ati.com/developer`.

[BBS94] Deborah Berman, Janson Bartell, and David Salesin. Multiresolution painting and compositing. In *Proceedings of Siggraph 94*, pages 85–90, 1994. New York, NY.

[BGK+99] David Blythe, Brad Grantham, Mark J. Kilgard, Tom McReynolds, Scott R. Nelson, Celeste Fowler, Simon Hui, Paula Womack, Linda Rae Sande, and Dany Galgani. Advanced graphics programming techniques using opengl, 1999. `http://www.opengl.org/developers/code/sig99/advanced99/notes/node297.ht%ml`.

[BMDF02] Steven Bergner, Torsten Möller, Mark S. Drew, and Graham D. Finlayson. Interactive spectral volume rendering. In *Proceedings of IEEE Visualization 02*, pages 101–108, October 2002.

[BNS] Imma Boada, Isabel Navazo, and Roberto Scopigno. Multiresolution volume visualization with a texture-based octree.

[BPRD98] C. Bajaj, V. Pascucci, G. Rabbiolo, and Schikore D. Hypervolume visualization: A challenge in simplicity. In *Proceedings Symposium on Volume Visualization 1998*, pages 95–102, 1998.

[BR98] Uwe Behrens and Ralf Ratering. Adding shadows to a texture-based volume renderer. In *Proceedings of IEEE Visualization 98*, pages 39–46, October 1998.

[Bre99] Falk Brettschneider. Qextmdi, 1999. `http://www.geocities.com/gigafalk/qextmdi.htm`.

[Bro00] Pat Brown. Ext_texture_compression_s3tc, 2000. `http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression%_s3tc.txt`.

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Symposium on Volume Visualization and Graphics 1994*, pages 91–98, 1994.

[CCF97] M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia. Extending distortion viewing techniques from 2d to 3d data. In *IEEE Computer Graphics and Applications, Special Issue on Information Visualization*, pages 42–51. IEEE Computer Society Press, July 1997.

[CDF92] A. Cohen, I. Daubechies, and J. Feauveau. Bi-orthogonal bases of compactly support wavelets. In *Communication on Pure Applied Math*, pages 45:485–560, 1992.

[CL93] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of ACM SIGGRAPH 93. Computer Graphics Proceedings and Annual Conference Series*, 1993.

[Cor] Able Software Corporation. 3d doctor screenshoot. `http://www.ablesw.com/3d-doctor/images.html`.

[CS88] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups*. Springer-Verlag, 1988.

[CTM02] Hamish Carr, Thomas Theußl, and Torsten Möller. Isosurfaces on optimal regular samples. To be published, 2002.

[Dau88] Ingrid Daubechies. Orthonormal bases of compactly supported wavelets. In *Communication on Pure Applied Mathematics*, pages 41(7):909–996, October 1988.

[DJL92] R. DeVore, B. Jawerth, and B. Lucier. Image compression through wavelet transform coding. In *IEEE Transactions on Information Theory*, pages 38(2):719–746, March 1992.

[DL90] Nira Dyn and David Levin. A butterfly subdivision scheme for surface interpolation with tension control. In *Transactions on Graphics*, volume 9(2), pages 160–169, April 1990.

[DS98] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. In *J. Fourier Anal. Appl*, pages 4(3):245–267, 1998.

[ECS00] David Ellswort, Ling-Jen Chiang, and Han-Wei Shen. Accelerating time-varying hardware volume rendering using tsp trees and color-based error metrics. In *Proceedings of IEEE Visualization 2000*, October 2000.

[EKE01] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Siggraph/Eurographics Workshop on Graph-*

ics Hardware 2001, 2001.

[ER01] David Ebert and Penny Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *IEEE Transactions on Visualization and Computer Graphics*, pages 253–264, July-Sept 2001. Vol.7, No.3.

[ESMM02] Alireza Entezari, Randy Scoggins, Torsten Möller, and Raghu Machiraju. Shading for fourier volume rendering. In *Symposium on Volume Visualization and Graphics 2002*, pages 131–138, October 2002.

[FB90] S. Feiner and C. Beshers. Visualizing n-dimensional virtual worlds with n-vision. In *Computer Graphics*, pages 37–38, 1990.

[FS94] Adam Finkelstein and David H. Salesin. Multiresolution curves. In *Proceedings of Siggraph 94*, pages 261–268, 1994. New York, NY.

[fSSSFU02] Centre for Systems Sience Simon Fraser University. Computer imaging, March 2002. `http://css.sfu.ca/update/vol14/14.2-computer-imaging.html`.

[GC95] Steven J. Gortler and Michael F. Cohen. Hierarchical and variational geometric modeling with wavelets. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 35–42, 1995. New York, NY.

[GDH97] M. H. Gross, L. Lippert Dittrich, and S. Häring. Two methods for wavelet-based volume rendering. In *Computer and Graphics*, pages 237–252, 1997. 21(2).

[GMR97] A. Gaddipati, R. Machiraju, and Yagel R. Steering image generation using wavelet based perceptual metric. In *Computer Graphics Forum (Proceedings of Eurographics '97*, pages 241–251, September 1997.

[Gro] NASA Data Analysis Group. Nas website. `http://www.nas.nasa.gov/Groups/VisTech/index.html`.

[GWGS02] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Straßer. Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization 02*, pages 53–60, October 2002.

[HH] A. J. Hanson and P. A. Heng. Illumination in 4d?

[HH92] Andrew Hanson and Pheng Heng. Four-dimensional views of 3d scalar field. In *Technical Report 358*, page 8, July 1992.

[HKERS02] Markus Hadwiger, Joe M. Kniss, Klaus Engel, and Christof Rezk-Salama. High-quality volume graphics on consumer pc hardware, 2002. Course Notes 42.

[Hoc97] Dawn M. Hoch. *Chilton's Ford Mustang/Cougar 1964-73 Repair Manual*. Haynes Publishing Group, 1997.

[Hou73] G.N. Houndsfieled. Computerized transverse axial scanning (tomography): Part 1 description of system. In *British Journal of Radiology*, pages 46:1016–1022, 1973.

[HTHG01] M. Hadwiger, T. Theßl, H. Hauser, and E. Gröller. Hardware-accelerated high-quality filtering on pc hardware. In *Prceedings of Vison, Modeling, and Visualization 2001*, pages 105–112, 2001.

[IHR96] Luis Ibáñez, Chafiaâ Hamitouche, and Christian Roux. Determination of discrete sampling grids with optimal topological and spectral properties. In *Proc. of the 6th International Workshop in Discrete Geometry for Computer Imagery DGCI96*, pages 181–192, 1996. Springer Verlag, Lyon, France.

[IHR97] Luis Ibáñez, Chafiaâ Hamitouche, and Christian Roux. Ray tracing and 3d object representation in the bcc and fcc grids. In *Proceedings of the 7th International Workshop in Discrete Geometry for Computer Imagery DGCI '97, Lecture Notes in Computing Science 1347*, pages 235–241. Springer-Verlag, 1997. Montpellier, France.

[IP02] Insung Ihm and Sanghun Park. Wavelet-based 3d compression scheme for very large volume data. In *Proceedings of IEEE Visualization 02*, pages 101–108, October 2002.

[Jac91] A. G. Jackson. *Handbook of Crystallography*. Springer-Verlag, 1991.

[JFS95] Charles E. Jacobs, Adam Finkelstein, and David H. Salesin. Fast multiresolution image querying. In *Proceedings of Siggraph 95*, pages 277–286, 1995. New York, NY.

[Kak84] Michio Kaku. *Hyperspace : a scientific odyssey through parallel universes, time warps, and the tenth dimension*. Oxford University Press, New York, 1984.

[KD98] Gordon Kindlmann and James Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *IEEE Symposium on Volume Visualization 1998*, pages 79–86, October 1998.

[KDG99] A. Köonig, H. Doleisch, and E. Gröoller. Multiple views and magic mirrors - fmri visualization of the human brain. Technical report, Institute of Computer Graphics and Algorithms, Vienna University of Technology, February 1999.

[Kil01] Mark J. Kilgard. Nv_texture_rectangle, 2001. `http://oss.sgi.com/projects/ogl-sample/registry/NV/`

`texture_rectangle.tx%t`.

[KM01] Steven Lee Kilthau and Torsten Möller. Splatting optimizations. In *Technical Report, School of Computing Science, Simon Fraser University*, pages 98–106, April 2001. SFU-CMPT-04/01-TR2001-02.

[KMH01] Robert Kosara, Sylvia Miksch, and Helwig Hauser. Semantic depth of field. In *Proceedings of IEEE Information Visualization 01*, pages 97–104, 2001.

[KOPR97] Thomas Kehmann, Walter Oberschelp, Erich Pelikan, and Rudolf Repges. *Bildverarbeitung für die Medizin*. Springer, 1st edition, 1997.

[KPHE02] Joe Kniss, Simon Premoze, Charles Hansen, and David Ebert. Interactive translucent volume rendering and precedural modeling. In *Proceedings of IEEE Visualization 02*, pages 109–116, October 2002.

[Lac95] Phil Lacroute. *Fast volume rendering using a shear-warp factorization of the viewing transformation*. PhD thesis, Stanford University, Sep 1995.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169, Jul 1987.

[Lev88] Marc Levoy. Display of surfaces from volume data. In *IEEE Computer Graphics and Applications*, volume 8(3), pages 29–37, May 1988.

[Lev92] M. Levoy. Volume rendering using the fourier projection-slice theorem. In *Proceedings of Graphics Interface '92, Canadian Information Processing Society*, pages 61–69, May 1992.

[LM02] Eric B. Lum and Kwan-Liu Ma. Hardware-accelerated parallel non-photorealistic volume rendering. In *International Symposium on Nonphotorealistic Rendering and Animation (NPAR02)*, June 2002.

[LME+02] Eidong Lu, Chrostopher Morris, David S. Ebert, Penny Rheingans, and Charles Hansen. Non-photorealistic volume rendering using stipling techniques. In *Proceedings of IEEE Visualization 02*, pages 101–108, October 2002.

[Loh98] Gabriele Lohmann. *Volumetric Image Analysis*. Wiley Teubner, 1st edition, 1998.

[Mei00] Michael Meissner. Volvis website, 2000. `http://www.volvis.org`.

[MGW02] Michael Meißner, Stefan Guthe, and Straßer Wolfgang. Interactive lighting models and pre-integration for volume rendering on pc graphics accelerators. In *Proceedings of Graphics Interface 2002*, 2002. to appear.

[MHB+00] Michael Meissner, Jian Huang, Dirk Bartz, Klaus Müller, and Crawfis Roger. A practical evaluation of popular volume rendering algorithms. In *Symposium on Volume Visualization and Graphics 2000*, pages 81–90, October 2000.

[MHW99] Michael Meißner, Ulrich Hoffmann, and Straßer Wolfgang. Enabling classification and shading for 3d texture mapping based volume rendering using opengl and extensions. In *Proceedings of IEEE Visualization 1999*, pages 207–214, October 1999.

[ML94] Stephen Marschner and Richard Lobb. An evaluation of reconstruction filters for volume rendering. In *Proceedings of IEEE Visualization 94*, pages 100–107, 1994.

[MSO] M. McGuigan, G. Smith, and S. Ohta. Visualization of four dimensional quantum chromodynamic data. In *Proceedings SPIE 1989*.

[MSS99] Maic Masuch, Stefan Schlechtweg, and Ronny Schulz. Speedlines: Depicting motion in motionless pictures. In *Siggraph 99 Conference Abstracts and Applications*, page 277, 1999.

[NM02] Neophytos Neophytou and Klaus Müller. Space-time points: 4d splatting on efficient grids. In *Symposium on Volume Visualization and Graphics 2002*, pages 97–106, October 2002.

[*n*vi01] *n*vidia. *n*vidia website, 2001. `http://developer.nvidia.com`.

[OS75] A. V. Oppenheim and W. Schafer. *Digital Signal Processing*. Prentice-Hall Signal Processing Series. Prentice-Hall, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 07632, 1975.

[Per96] Senthil Periaswamy. Detection of microcaclifications in mammograms using hexagonal wavelets. Master's thesis, University of South Carolina, Sep 1996.

[PGG01] Bimal Poddar, Dave Gosselin, and Dan Ginsburg. Arb_texture_env_dot3, 2001. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_env_dot3.tx%t`.

[PHK+99] Hanspeter Pfister, Jan Hardenbergh, Jim Knottel, Hugh Lauer, and Larry Seiler. The volumepro real-time ray-casting system. In *Proceedings of Siggraph 99*, pages 251–260, 1999.

[PSM02] Pujita Pnnamaneni, Sagar Saladi, and Joerg Meyer. 3-d haar wavelet transformation and texture-based 3-d reconstruction of biomedical data sets. In *Proceedings of IEEE Visualization*

02, pages 101–108, October 2002.

[PWC] Bhaniramka Praveen, Rephael Wenger, and Roger Crawfis. Isosurfacing in higher dimensions.

[RA] Antonia Aguilera Ramacuteírez and Ricardo Pérez Aguila. A method for obtaining the tesseract by unraveling the 4d hypercube.

[RMC+00] Niklas Röber, Torsten Möller, Anna Celler, Troy Farncombe, and Thomas Strothotte. Multidimensional analysis and visualization software for dynamic spect. In *Proceedings of the 47th annual meeting of the Society of Nuclear Medicine*, page 33 No.864, June 2000.

[RSEB+00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Siggraph/Eurographics Workshop on Graphics Hardware 2000*, 2000.

[Sal98] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, 1st edition, 1998.

[SCM99] Han-Wei Shen, Ling-Jen Chiang, and Kwan-Liu Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning tree(tsp). In *Proceedings of IEEE Visualization 1999*, pages 371–377, October 1999.

[SDS96] Eric J. Stollnitz, Tony D. Derose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann, 1st edition, 1996.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.

[Ser82] J. Serra. *Image Analysis and Mathematical Morpholg*. Academic Press Inc., 1982.

[Set] Visible Human Data Set. Visible human data set. `http://css.sfu.ca/update/vol14/14.2-computer-imaging.html`.

[Sha49] C.F. Shannon. Communication in the presence of noise. In *Proceedings of the IRE 1949*, January 1949.

[SHER99] Christopher D. Shaw, James A. Hall, David S. Ebert, and D. Aaron Roberts. Interactive lens visualization techniques. In *Proceedings of IEEE Visualization 99*, pages 155–160, October 1999.

[SL98] S. Schuler and A. Laine. *Hexagonal QMF Banks and Wavelets in Time Frequency and Wavelets in Biomedical Signal Processing*. IEEE Press and John Wiley, 1st edition, 1998.

[Slo98] Neil J. A. Sloane. The sphere packing problem. In *Proceedings of the International Congress of Mathematicians*, pages 387–396, Berlin, 1998. Doc.Math.J.DMV Extra Volume ICM III.

[SM00] Heidrun Schumann and Wolfgang Müller. *Visualisierung: Grundlagen und allgemeine Methoden*. Springer, 1st edition, 2000.

[SM02] Jon Sweeney and Klaus Mueller. Shear-warp deluxe: The shear-warp algorithm revisited. In *Joint Eurographics, IEEE TCVC Symposium on Visualization 2002*, pages 95–104, May 2002. Barcelona, Spain.

[SMLS98] William Schroeder, Ken Martin, Bill Lorenson, and Will Schroeder. *The Visualization Toolkit: An Object-Oriented Approach to 3-D Graphics(2nd Edition)*. Prentice Hall Computer Books, 2nd edition, 1998.

[SR98] S. Schlechtweg and A. Raab. *Rendering Line Drawings for Illustrative Purposes, In.: Th. Strothotte, H. Wagener (eds.): Abstraction in Interactive Computational Visualization: Exploring Complex Information Space*. Springer, 1st edition, 1998.

[Srd] Marko Srdanovics. Jumping jack icon/glyph. `http://www.cs.uml.edu/~msrdanov/viz/`.

[SS89] Richard C. Staunton and Neil Storey. A comparison between square and hexagonal sampling methods for pipeline image processing. pages 142–151, 1989. Vol. 1194.

[SS95] P. Schröder and W. Sweldens. Spherical wavelets: Efficiently representing functions on the sphere. In *Proceedings of Siggraph 95*, pages 161–172, 1995.

[Swe95] W. Sweldens. The lifting scheme: A new philosphy in biorthogonal wavelet constructions. In *Wavelet Applications in Signal and Image Processing III*, pages 68–79. Laine, A. F. and Unser, M., 1995.

[Sys01] Ballard Power Systems. Ballard power systems website, 2001. `http://www.ballard.com`.

[TL93] Takashi Totsuka and Marc Levoy. Frequency domain volume rendering. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27(4), pages 271–278, Aug 1993.

[TMG01] Thomas Theußl, Torsten Möller, and Eduard Gröller. Optimal regular volume sampling. In *Proceedings IEEE Visualization 2001*, pages 91–98, Oct 2001.

[Tor01] Melanie Karla Tory. Non-photorealistic visualization of flow data sets, 2001. `http://www.cs.sfu.ca/~mktory/personal/`

cmpt888/index.htm.

[TRM+01] Melanie Tory, Niklas Röber, Torsten Möller, Anna Celler, and Stella M. Atkins. 4d space-time techniques: A medical imaging case study. In *Proceedings of IEEE Visualization 01*, pages 473–476, October 2001.

[TS02] Melanie Tory and Colin Swindells. Exovis: An overview and detail technique for volumes. Technical Report SFU-CMPT-TR2002-05, Computing Science Dept., Simon Fraser University, 2002.

[TUW] Abteilung für Computergraphik TECHNISCHE UNIVERSITÄT WIEN, Institut für Computergraphik und Algorithmen. Vienna christmas tree. http://www.cg.tuwien.ac.at/gallery/xmas/2001/.

[UVWJ+98] G. Uytterhoeven, F. Van Wulpen, M. Jansen, D. Roose, and A. Bultheel. Waili: A software library for image processing using integer wavelet transforms. In *SPIE Proceedings, The International Society for Optical Engineering*, volume 3338, pages 1490–1501, February 1998.

[vGK96] A. van Gelder and K. Kim. Direct volume rendering with shading via three-dimensional textures. In *Symposium on Volume Visualization 96*, pages 23–30, 1996.

[vvL93] J. J. vanWijk and Robert van Liere. Hyperslice, visualization of scalar functions of many variables. 1993.

[WB] C. Weigle and D. C. Banks. Extracting iso-valued features in 4-dimensional scalar fields.

[WEE02] Daniel Weiskopf, Klaus Engel, and Thomas Ertl. Volume clipping via per-fragment operations in texture-based volume visualization. In *Proceedings of IEEE Visualization 02*, pages 93–100, October 2002.

[Wel84] Alexander Frank Wells. *Structural Inorganic Chemistry*. Oxford University Press, 5th edition, 1984.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24(4), Sep 1990.

[WWH+] Manfred Weiler, Rüdiger Westermann, Chuck Hansen, Kurt Zimmermann, and Thomas Ertl. Level-of-detail volume rendering via 3d textures.

[wxw92] wxwindows.org. wxwindows, 1992. http://www.wxwindows.org/.