Technische Universität Wien

Diplomarbeit

Practical Reconstruction Schemes and Hardware-Accelerated Direct Volume Rendering on Body-Centered Cubic Grids

unter der Leitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller,
Institut 186 für Computergraphik und Algorithmen,
und
Dipl.-Ing. Thomas Theußl
als verantwortlich mitwirkendem Universitätsassistenten,

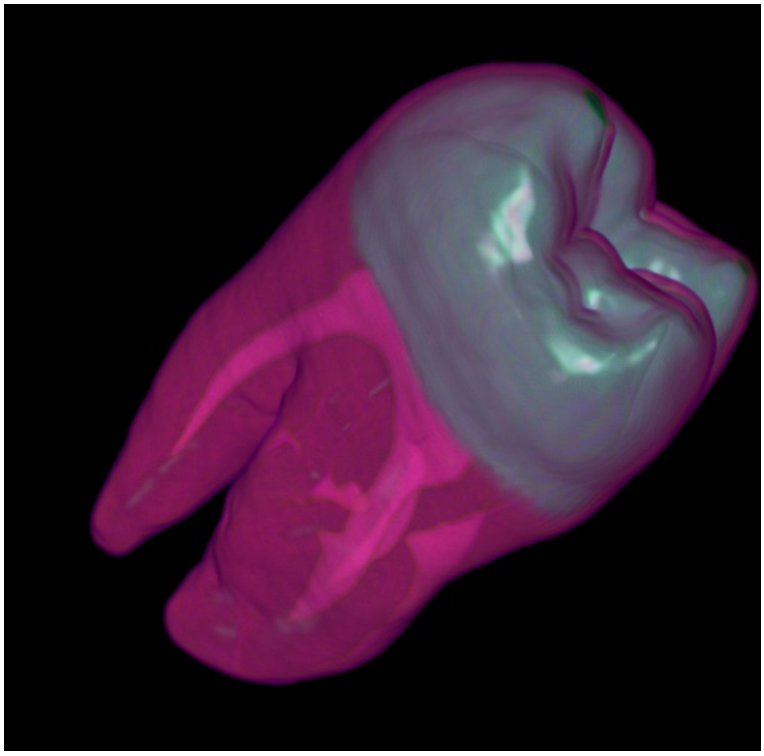
eingereicht an
an der Technischen Universität Wien,
Fakultät für Technische Naturwissenschaften und Informatik,

von
Oliver Mattausch,
Matrikelnummer 9506124,
Neuklostergasse 3,
2700 Wiener Neustadt, Österreich,
geboren am 2. Dezember 1976 in Wiener Neustadt

Wien, im Dezember 2003

Oliver Mattausch

Practical Reconstruction Schemes and Hardware-Accelerated Direct Volume Rendering on Body-Centered Cubic Grids



mailto:
matt@cg.tuwien.ac.at

Abstract

It is well known in the signal-processing community that the Body-Centered Cubic grid is the optimal sampling grid in 3D. In volume visualization, the Cartesian grid is by far the most popular type of grid because it is convenient to handle. But it requires 29.3% more samples than the Body-Centered Cubic grid. In order to convince people used to Cartesian grids for years of the advantages of Body-Centered Cubic grids, we must prove their usability in many different volume rendering algorithms. Further we have to show that we get a performance gain without or with only slight loss of image quality compared to Cartesian grids. Therefore we introduce several practical reconstruction schemes on Body-Centered Cubic grids, which are very general and can be used in a number of applications and tasks.

Together with the development of powerful and flexible consumer graphics hardware, interactive hardware-accelerated volume rendering algorithms gain popularity. Rendering performance becomes a big issue, which can be a strong argument in favour of Body-Centered Cubic grids. We adapted some of the most popular volume rendering approaches exploiting hardware-acceleration to Body-Centered Cubic grids: both 2D and 3D texture-based volume rendering and the projected tetrahedra algorithm. At least partly we succeeded in achieving a performance gain on our new grid and further produced some impressive rendering results comparable to the Cartesian grid version.

Kurzfassung

Es ist lange bekannt in der Signalverarbeitungsgemeinde, daß das Body-Centered Cubic Gitter das optimale Sampling Gitter in 3D ist. In der Volumsvisualisierung ist das Kartesische Gitter der bei weitem populärste Gittertyp, da es einfach zu behandeln ist. Allerdings braucht es 29.3% mehr Samples als das Body-Centered Cubic Gitter. Um Leute, die über Jahre hinweg an Kartesische Gitter gewöhnt sind, von den Vorteilen der Body-Centered Cubic Gitter zu überzeugen, müssen wir ihre Anwendbarkeit in vielen verschiedenen Volumsdarstellungsverfahrung beweisen. Weiters müssen wir zeigen, daß wir an Performance gewinnen ohne oder nur mit wenig Verlust von Bildqualität. Darum stellen wir mehrere praktische Rekonstruktionsmethoden auf Body-Centered Cubic Gittern vor, die sehr allgemein sind und in einer Reihe von Anwendungen und Aufgaben verwendet werden können.

Gleichzeitig mit der Entwicklung von mächtiger und flexibler Consumer-Graphics Hardware gewinnen interaktive Hardware-beschleunigte Algorithmen an Beliebtheit. Rendering Performance wird zu einer wichtigen Größe, was ein starkes Argument für Body-Centered Cubic Gitter sein kann. Wir haben einige der populärsten Volumsdarstellungsmethoden, die Hardware-Beschleunigung ausnutzen, auf das Body-Centered Cubic Gitter umgesetzt: sowohl 2D als auch 3D Textur basierte Volumsdarstellungsverfahren und Verfahren, die auf projizierte Tetraeder beruhen. Zumindest teilweise waren wir erfolgreich einen Gewinn an Performance auf unserem neuen Gitter zu erreichen und erzielten außerdem einige beeindruckende Rendering-Ergebnisse vergleichbar zu der Version auf dem Kartesischen Gitter.

Contents

Acknowledgments

I thank my parents and grandparents for their constant support, Jan Schödl, Fritz Dörfl, Markus Reisenbauer, Stefan Haslinger, Markus Hecher, Ulli Schackl, Bernd Klauninger and Andi Beer for being my closest friends over many years and trinking my ice tea (well, Andi Beer never did), Michi B., Jürgen Lorenz and Gernot "The Body" Langer for being my fellow colleagues and making the old "Fachschaft" a fun place to be, Meguro Rie, Yamada Fumikazu and the other members of the Tokyo Gakugei Univerity Kendo Club, Kamemoto "Kame" Ryutaro and all my other Kendo friends from Vienna, who prevented me from getting mad from only thinking of my studies, Eduard Gröller, Thomas Theußl, Markus Hadwiger for support in my efforts to write this thesis and their valueable advice, the VRVis center for allowing me to use their hardware rendering framework, Christoph Berger and Markus Hadwiger, for being the authors of this cool piece of software (and thanks for lending me your private ATI card, Markus!), Stephan Guthe for giving me useful advice during his visit in Vienna, the makers of Thunderforce and Musha Alestse, the games that changed my life, and finally Günter Schödl for being my teacher, choach, neighbour and the father of my oldest friend.

Chapter 1

Introduction

In this chapter we sketch the basics of volume rendering and the volume rendering pipeline. Afterwards important issues of modern consumer graphics hardware are discussed. Then we introduce optimal regular sampling with emphasis on the Body-Centered Cubic grid and it's properties. At last we give an overview of this thesis.

1.1 Volume Rendering

Volume rendering refers to the visualization of volumetric data given as a set of scalar or vectorial samples. The data samples are arranged in a so called sampling grid, which assigns a position in space to each sample. The sources for volume data can be classified into three main groups:

- Data sampled from analytical functions
- Data aquired from scanning real world objects
- Computational simulation data

The nature of the data also influences the type of the employed sample grid. For example, volume data produced in computed tomography (CT) or magnetic resonance imaging (MRI) are usually sampled on a rectilinear grid, whereas curvilinear grids are often used in the simulation of fluid dynamics.

Direct versus Indirect Volume Rendering

Deriving an intermediate representation of the volume data, usually in form of a polygonal approximation of contour surfaces, is called indirect volume rendering. Direct volume rendering refers to computing an image directly from the volume data. In indirect volume rendering, we have the advantage that we can use surface rendering techniques which have been extensively studied and refined for quite some time in computer graphics. Moreover, consumer graphics

hardware is highly specialized to fulfill this task. Unfortunately much of the information contained in the data is thrown away in this approach. Furthermore, the complexity of the intermediate representation is hard to predict and can go out of bounds in terms of memory requirements.

On the contrary, direct volume rendering produces semi-transparent cloud or smog like images. This is a representation of the whole volume, because all samples contribute to the final image to a certain extent, yielding a better insight into the data. Unlike indirect volume rendering, the algorithm complexity is independent from the geometry of objects. The main drawback of direct volume rendering is the amount of data that have to be processed. Therefore direct volume rendering methods are traditionally considered to be relatively slow.

Reconstruction

We assume that the volume data is sampled from a continuous scalar density function $f(x,y,z)$ defined for all points of the volume. To reconstruct the original scalar field, we have to interpolate density values between samples. Theoretically, ideal reconstruction is possible with the sinc filter, but it cannot be used in practice because of it's infinite extend. On rectilinear grids, the trilinear interpolation is commonly applied, because it is a good tradeoff between reconstruction quality and performance.

Classification

Next color and opacity values are assigned as a function of interpolated density $f(x,y,z)$, usually referred as the transfer function. Using an emmision-absorption model [71, 36], we get attenuated colors (i.e., emission) $c(f(x,y,z))$ and extinction factors (i.e., absorption) $\tau(f(x,y,z))$ as output. This process is called post-classification, as opposed to pre-classification, where color and opacity are assigned to the vertices only in a preprocessing step and are interpolated afterwards. Both approaches have some advantages, but we consider post-classification to be superior to pre-classification. In the latter approach colors not represented in the transfer function may appear in the final image, and further small details in the scalar field can be lost.

Ray integration

When rendering an image we cast viewing rays through the volume. Along the ray we integrate the color and opacities in order to compute the final image. We refer to a ray parameterized by t as $r(t)$. The output color C of a ray with maximum length l is given by the well known volume rendering integral:

$$C = \int_0^l c(f(r(t)) \exp\left(- \int_0^t \tau(f(r(t'))) \mathrm{d}t' \right) \mathrm{d}t \tag{1.1}$$

Solving this integral analytically is generally not possible. Instead it is discretizised using the Riemann sums over a number of n samples:

$$C \approx \sum_{i=0}^{n} C_i \prod_{j=0}^{i+1} (1 - A_j) \tag{1.2}$$

Compositing is the process of iteratively evaluating this formula in either front-to-back or back-to-front order. This is the well known formula of back-to-front compositing, for instance:

$$C'_i = C_i + (1 - A_i)C'_{i+1} \tag{1.3}$$

1.2 Graphics Hardware

Consumer graphics hardware undergoes rapid developments. Not only in the direction of higher throughput rates and more texture memory, but also regarding flexibility and programmability. Formerly hard-wired stages of the rendering pipeline can be now manipulated with vertex and pixel shaders. This makes consumer graphics hardware, usually optimized for computer games, suitable for other applications like volume rendering. Volume rendering with its enormous data rates still poses some problems even for hardware of the newest generation.

A few years ago pixel shading abilities were quite limited. An important improvement to standard texture mapping was the multitexture extension, which allow us to combine the output of multiple textures. It can be used for light maps, for example, but we will see that it is also of importance for texture-based volume rendering. Another feature exploited for volume rendering are dependent textures, where the output texels of one or more textures are assigned as texture coordinates for another one. Early pixel shaders, like NVIDIA's register combiners, only have a limited range of available instructions. Today's fragment programs are much more powerful as they allow assembler style programming, and are supported by both big vendors ATI and NVIDIA. Recent developments, like NVIDIA's CG compilers, are going in the direction to make platform independent code written in a high-level programming language possible.

In this work we used OpenGL and extensions as programming interface to the hardware. It is a well designed, well documented graphics library, and therefore perfectly suited for scientific research.

1.3 Regular Optimal Sampling

In volume rendering applications, usually Cartesian grids are used. They are convenient to handle because the indices of a sample are equivalent to the position in space. But it is well known that they are not optimal in terms of sampling efficency. If the number of samples needed for lossless reconstruction of a signal is minimal, we speak about optimal sampling. This is the case if the replicated spectra in frequency domain are packed as closely as possible without overlapping.

We assume the sampled function to be band-limited and isotropic, Such a function has a spherical frequency support. The task of the closest possible packing of (hyper-) spheres is known as the famous sphere packing problem [55], which has attracted the attention of mathematicians over centuries, and is still not solved for the general case. Fortunately, among regular sampling grids several optimal solutions have been found. In the 2D case the Hexagonal Close Packing (HCP) was proven to be the optimal packing scheme (refer figure 1.1). For every packing scheme in frequency domain, there exists a dual sampling grid in spatial domain, and the dual of a HCP grid is also a HCP grid.

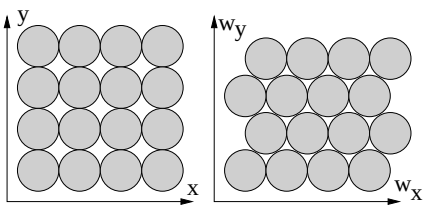


Figure 1.1: 2D Cartesian (left image) and Hexagonal packing (right image) of replicated spherical spectra in the frequency domain.

In the 3D case, again the Hexagonal Close Packing grid and the Face-Centered Cubic (FCC) grid are both optimal packing schemes. The FCC grid looks like a Cartesian (CC) grid with an additional sample point in the center of each cubic cell face. Like in the 2D case, the HCP grid is it's own dual, whereas the FCC grid corresponds to the Body-Centered Cubic (BCC) grid. A BCC grid is a Cartesian grid with an additional sample point in the center of each cubic cell, as shown in figure 1.2. Hence the HCP and the BCC grid are optimal grids for sampling. It has been proven [60] that they need 29.3% less samples than a Cartesian grid for equal resampling quality. We concentrate our work on the Body-Centered Cubic grid because it is much easier to index than the Hexagonal Close Packing grid and has some convenient properties. We will exploit these properties for our methods in the following chapters. The BCC grid can be seen as

- a stack of 2D CC grids were odd-numbered planes are translated by half a unit in both dimensions with respect to even-numbered planes.
- two interleaved 3D CC grids, where one grid (usually denoted as secondary grid) is translated by half a cell spacing in all three axes relatively to the other grid (denoted as primary grid). Two penetrating cells from each grid are shown in figure 1.3.
- a sheared and scaled CC grid, where the transformation matrix is given by the sampling matrix (explained in section 1.4).
- a tetrahedral mesh which is simple and uniquely defined by the Delauney complex [6].

1.4 The Body-Centered Cubic Grid: A formal definition

We formally describe sampling as a mapping from indices to actual sampling positions [17]. In the following we restrict our explanations to grids where this mapping can be defined by three basis vectors and an origin, which is the case for the Cartesian and the Body-Centered Cubic grid. Such grids are also referred as lattices in literature. The basic vectors can be written as a matrix V, called the sampling matrix:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = V \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} \tag{1.4}$$

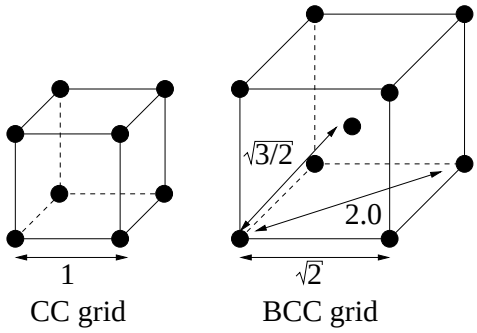


Figure 1.2: CC and BCC cells in relative proportions image redrawn from [59]) A BCC grid requires less samples to represent the same volume, hence the sample points are wider apart.

For a 3D Cartesian grid with cell spacing $T = 1$, V is the identity matrix:

$$V_{CC} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1.5}$$

The locations of the replicated spectra in the frequency domain are described by the dual matrix U of V, called the periodicity matrix. They have the following relationship, where I refers to the identity, and ^{T} to the transpose:

$$U^T V = 2\pi I \tag{1.6}$$

The Hexagonal Close Packing grid is an optimal grid for sphere packing in 2D:

$$V_{hex2D} = \begin{pmatrix} T & \frac{1}{2}\frac{\sqrt{3}}{2}T \\ 0 & \frac{\sqrt{3}}{2}T \end{pmatrix} \tag{1.7}$$

It is easy to verify that the corresponding sampling matrix in spatial domain also describes a Hexagonal Close Packing grid. Among the optimal sampling schemes in 3D, we again find the HCP grid. But it is difficult to use in practice, because it cannot be described by a sampling matrix. On the contrary, we can define the equally optimal Body-Centered Cubic grid using the following matrix:

$$V_{BCC} = \begin{pmatrix} T & 0 & \frac{1}{2}T \\ 0 & T & \frac{1}{2}T \\ 0 & 0 & \frac{1}{2}T \end{pmatrix} \tag{1.8}$$

This sampling matrix is a shear matrix, where the samples are sheared in the direction of the positive x and y axis. Other shear axes and directions would also result a sampling matrix describing a BCC grid. This scheme is still not very practical, and thus we want to fit the sample points into a rectangular scheme. By introducing a modulo operation we get a more compact memory layout. k denotes the third index component:

$$V_{BCC} = \begin{pmatrix} T & 0 & \frac{1}{2k}T(k \bmod 2) \\ 0 & T & \frac{1}{2k}T(k \bmod 2) \\ 0 & 0 & \frac{1}{2}T \end{pmatrix} \tag{1.9}$$

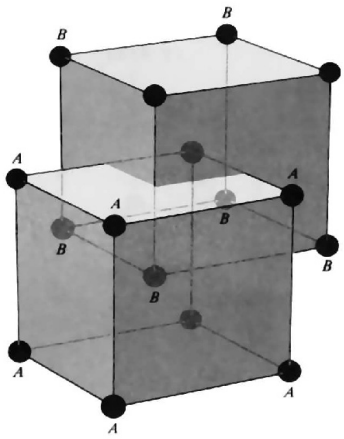


Figure 1.3: Two interleaved cubic cells, one from the primary (denoted as A) and one from the secondary (denoted as B) CC grid in a BCC grid. Image taken from [16].

In figure 1.4, we show both storage schemes in 2D. As the Body-Centered Cubic grid is only defined in the spatial, such 2D visualizations can be seen as an orthogonal projection along the *y* axis.

1.5 Thesis Outline

The Cartesian Grid is the standard grid in volume rendering, although it is rather inefficient. From signal theory we know that the Body-Centered Cubic grid is optimal for the purpose of sampling, saving 29.3% sample points. One way to to show the superiority of BCC grids in practice is to adapt existing rendering methods. A performance gain should be achieved compared to the original CC grid algorithm, while providing similar rendering quality. By exploiting powerful graphics hardware, interactive or near interactive volume rendering is possible even for larger volumes. This makes the potential performance gain with BCC grids even more desirable. Volume rendering methods may rise in popularity and find use in time critical applications as well, like in the gaming industry.

We give an overview of the most popular volume rendering algorithms in chapter 2. Special emphasis is on hardware-accelerated methods and the developments in the field of texture-based rendering and tetrahedral cell projection. In the same chapter we discuss the approaches which have been proposed on the BCC grid so far.

Chapter 3 is about general and practical reconstruction schemes. We used a raycasting system to test them. But they are applicable in other BCC grid rendering methods as well. Furthermore we present several gradient estimation schemes.

Afterwards, we discuss how we can exploit our knowledge of BCC grids for the adaption of concrete hardware-accelerated approaches. In chapter 4, we sketch the basics of 2D and 3D texture-based volume rendering, then describe how every particular rendering mode can be used on a BCC grid. Another popular hardware-accelerated technique is the projected tetrahedra

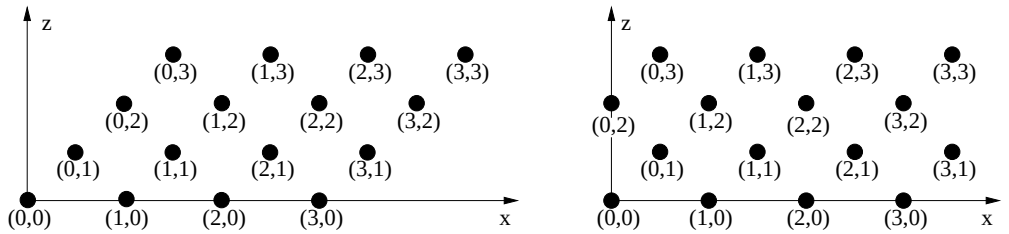


Figure 1.4: BCC storage schemes corresponding to equation 1.8 (left) and equation 1.9 (right) shown in 2D.

approach. We give an overview of the ideas behind the original projected tetrahedra algorithm in chapter 5. Furthermore we discuss the adaption of the algorithm to a tetrahedral mesh defined by a BCC grid. Several performance optimizations and some improvements regarding image quality and shading are introduced. We propose an adjacency structure for storing the tetrahedra information, which allows a fast traversal of the visible tetrahedra.

Chapter 6 deals with implementation issues. We will describe our general design concepts and discuss the most important code sections. In chapter 7 we present the results we achieved in our experiments. We tested rendering quality, performance and, if necessary, memory usage.

In chapter 8 we give a summary of the important points made in this thesis. Conclusions are drawn in chapter 9. We show directions about future work on BCC grids in chapter 10.

Chapter 2

Volume Rendering: State of the Art

First we discuss the most important volume rendering methods which have been proposed so far. Furthermore we give an overview of hardware-accelerated volume rendering. Literature about texture-based algorithms and tetrahedral cell projection is discussed in more details, because it directly concerns our work. We also present some data structures introduced to speed up splatting, as we can use some of them to accelerate the projected tetrahedra algorithm on BCC grids as well. At the end we shortly describe several volume rendering algorithms already proposed on BCC grids.

2.1 Volume Rendering Algorithms

Volume rendering algorithms are classified into image- and object-order algorithms, also referred as backward and forward mapping algorithms in literature. Image-order algorithms traverse through all image pixels and compute the color of the actual pixel by shooting rays into the volume, whereas object-order algorithms loop through the data samples and project their contribution to the output image. Typical image-order algorithms are raycasting [32] and raytracing [27]. Object-order algorithms are splatting [67], cell projection [54, 38, 68, 69, 53], and texture-based volume rendering [4]. The speed-optimized shear-warp algorithm proposed by Lacroute and Levoy [30] is considered to be a hybrid method (see figure 2.1). Fourier Domain Volume Rendering (FDVR) by Malzbender [34] follows a completely different concept. A comprehensive study of popular rendering algorithms was proposed by Meißner et al. [40].

2.2 Hardware-Accelerated Volume Rendering

In volume rendering enormous amount of data that must be processed. Therefore direct volume rendering algorithms have been traditionally either rather slow, or there is a tradeoff between rendering speed and quality, like for the fast shear-warp algorithm (refer figure 2.1). This drawback has limited the popularity of direct volume rendering over the years. Interactive volume rendering with high-quality required special hardware, like the VolumePro board [45]. Although consumer graphics hardware is not specialized for this task, a number of approaches have been proposed

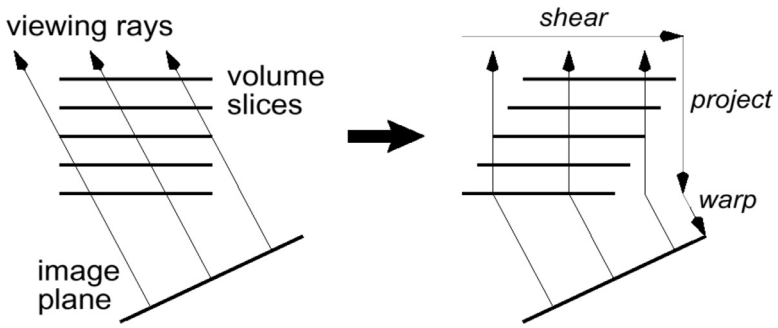


Figure 2.1: The shear-warp transformation allows fast reconstruction in the planes. It is the software equivalent to 2D texture-based volume rendering using object-aligned slices. Image taken from [30].

that exploit hardware-acceleration for volume rendering. Until recently, interactive rendering times were reported, but the quality was no match for the best software algorithms. The increasing power, accuracy and flexibility of consumer graphics hardware leads to a fast quality improvement of hardware-based volume rendering. Among the most popular hardware-accelerated approaches are texture-based volume rendering and the projected tetrahedra algorithm.

2.2.1 Texture-Based Volume Rendering

After the SGI reality engine was introduced [1], exploiting the power of 3D texture mapping hardware for interactive volume rendering was proposed by Cabral et al. [4]. They rendered 512 $\times$ 512 $\times$ 64 volumes in 0.1 seconds on a four Raster Manager SGI Reality Engine Onyx with one 150 MHZ CPU. Similar rendering rates were achieved by Cullip and Neumann [13]. Although impressive frame rates were reached, shading was still not supported. Gelder and Kim [21] introduced directional lighting using a lookup of 3 or 4 parameters. Unfortunately, hardware support for the lookup was not possible. Therefore a new 3D texture had to be generated for each classification or viewpoint change. Westermann and Ertl [66] proposed shading with a fast multi-pass algorithm, which was restricted to iso-surface rendering. Meißner et al. [40] extended this approach for diffusely shaded semi-transparent volume rendering. High-quality shading was achieved by Dachille et al. [14] at the expense of losing interactivity. They used a combination of hardware trilinear interpolation and blending, and software classification and lighting calculation.

Another approach is 2D texture-based volume rendering using object-aligned slices, which is closely related to the shear-warp algorithm(figure 2.1). Rezk-Salama et al. [48] significantly improved quality and performance of 2D texture-based volume rendering by the use of multi-texture blending. The image quality can be enhanced with the generation of intermediate slices. Alternatively, the slice number can be reduced by blending several textures to a single polygon. Engel et al. [19] rendered pre-integrated slabs instead of slices, achieving high accuracy without super-sampling for arbitrary transfer functions. Because of the very flexible and accurate per vertex and per pixel shading abilities of modern consumer graphics hardware, the gaps in rendering quality and accuracy between software and hardware volume rendering are vanish-

ing. Shaded pre-classification, post-classification, pre-integration as well as non photo-realistic rendering (NPR) can be done entirely on the GPU in interactive frame rates with impressively high quality [18, 3]. With the introduction of floating point precision textures [28], a rendering accuracy equal to high-quality software algorithms can be achieved.

2.2.2 Projected Tetrahedra Algorithm

The projected tetrahedra algorithm (PT) was introduced by Shirley and Tuchman in 1990 [54]. It became very popular because of it's flexibility and ability to exploit graphics hardware to speed up rendering. Therefore it is still considered as one of the main approaches for the rendering of unstructured grids [65], among sweep plane algorithms [20] and slicing [47].

Improving accuracy of the original algorithm by use of exponential transparency textures was proposed by Stein et al. [56] and Max et al. [37]. Other improvements were achieved in making depth sorting of polyhedral cells more efficient, for example by Stein et al. [56] and Comba et al. [10]. Cignoni et al. [9] introduced means to enhance the performance of the original algorithm with both simplification of the data and the rendering process. Data simplification using multi-resolution tetrahedral meshes was also proposed [8, 74].

In recent years the algorithm had a revival as subject of intensive research, because new features of modern consumer graphics hardware can be exploited to improve both rendering quality and performance of the original approach. Wittenbrink [72] suggested some general enhancements like rendering triangle fans instead of single triangles. Accurate renderings for arbitrary transfer functions can be achieved using pre-integration, which was introduced by Röttger et al. [51] and is still a topic of research [50, 23, 18]. Wylie et al. [73] and Weiler et al. [64] used vertex and fragment shaders to load the entire cell projection step on the hardware, making it possible to use OpenGL optimizations like vertex arrays. Wylie et al. [73] also suggested their work to be used for a new OpenGL extension GL_TETRA_EXT. King et al. [29] introduced a new architecture which allows order-independent transparency and discussed new primitives: tetrahedral fan and tetrahedral strip. The order-independent transparency is achieved with a re-circulating frame buffer (R-buffer). Unfortunately, this architecture does not yet exist physically.

2.3 Data Structures for Splatting

There are similarities between the projected tetrahedra algorithm and splatting to a certain extent. Both are forward projection algorithms. In case of splatting the algorithm is centered around voxels projected to the screen in the form of spherical Gaussian kernels, as opposed to an algorithm centered around tetrahedral cells. Similar to zero opacity splats, tetrahedra with an irrelevant contribution to the final image can be skipped completely during the traversal process. Thus we scanned literature about data structures used for accelerating splatting on rectangular grids, in order to make data traversal in projected tetrahedra techniques more efficient.

Fast splatting with texture mapping hardware was proposed by Crawfis et al. [12]. Using a special data structure, Crawfis [11] further accelerated splatting by rendering only certain splats belonging to an iso-surface. Laur et al. [31] introduced hierarchical splatting with a progressive

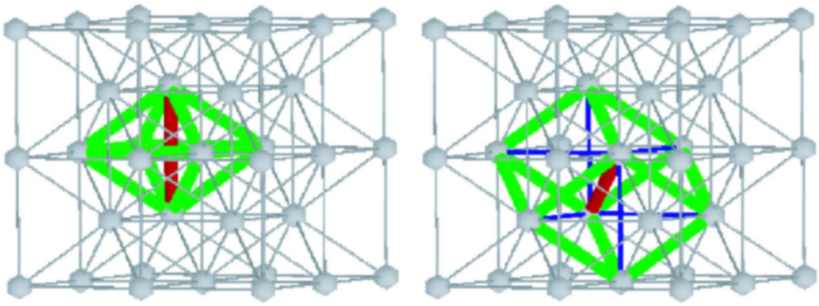


Figure 2.2: The cell structures used for the marching octahedra (left image) and marching hexahedra (right image) algorithms in a BCC grid. Image taken from [6].

refinement of the grid using an octree partition of the space. Another approach is a 3D adjacency structure suggested by Orchard et al. [44] for splatting on rectangular grids.

2.4 Volume Rendering on the BCC Grid

Several popular rendering algorithms have already been adapted to the BCC grid, among them splatting, the shear-warp approach, and raycasting. Most recently, Röber et al. [49] proposed 3D texture-based volume rendering on the BCC grid (see chapter 4.5 for further details), and Dornhofer [16] introduced an adaption of Fourier Domain Volume Rendering in his diploma thesis. Interestingly, most BCC grid rendering algorithms achieve an image quality comparable to the CC grid, but the BCC grid version generally appears blurrier than it's CC grid counterpart.

2.4.1 Splatting

Theußl et al. [60] presented the BCC version of splatting [67]. The implementation is straightforward, because just the calculation of the sample positions is slightly more complex. Westover style splatting uses Gaussian spherical reconstruction kernels, which have spherical support in frequency domain. Therefore splatting seems to be perfectly suited for BCC grid rendering. As pure object-order algorithm, the complexity is proportional to the volume size. Hence the lower number of samples fully contribute to a performance gain of up to 47%. The reason why this ratio even exceeds 29.3% may be due to better memory cache behavior. Neophytou and Mueller [43] extended splatting for BCC to the 4^{th} dimension, stating that they get savings from approximately 50% to 80% for time varying data sets.

2.4.2 Shear-Warp Algorithm

Lacroute and Levoy's [30] speed-optimized shear-warp algorithm was extended by Sweeney and Mueller [57] to support BCC grid rendering as part of their shear-warp deluxe project. They exploited the fact that a BCC grid can be divided into two separate CC grids and run-length-encoded both volumes separately. During traversal of the object-aligned slices the algorithm

Figure 2.3: "Girders" are artifacts that occur on iso-surfaces in a BCC grid. Image taken from [5].

switches between both CC grids, while taking the offset of half a unit for the secondary grid into account.

2.4.3 Raycasting

Ibáñez et al. [26] used a generalization of the Bresenham algorithm for raycasting in a BCC grid. They suggested linear interpolation inside a tetrahedral cell, but they give no further details nor do they present any experimental results about this interpolation in their work. Strategies for resampling in a BCC grid were proposed by Theußl et al. [59]. They investigated several reconstruction scheme for use in a high-quality raycasting system and compared them to corresponding CC grid interpolators. In detail we will discuss their work in chapter 3.

2.4.4 Iso-surface Reconstruction

Iso-surface reconstruction on CC is usually based on cubic cells, like in the popular marching cubes algorithm by Lorensen and Cline [33]. Chan and Purisma [7] suggested a tessellation of space into tetrahedra based on a BCC grid for iso-surface reconstruction. Besides the better sampling efficency of BCC there are no ambiguities in the mesh, unlike tetrahedralizations based on Cartesian grids. Treece et al. [61] extended this approach. The main problem of tetrahedral grids is the large number of created triangles. To reduce the number of triangles required in BCC grids, Carr et al. [6] investigated not only marching tetrahedra, but also marching octahedra and marching hexahedra for BCC grid iso-surface reconstruction. In figure 2.2, we show where these geometric structures are located in a BCC grid.

However, in most cases the iso-surfaces were significantly rougher on BCC grids than on CC grids using marching cubes. Carr et al. [6] suggested that cubes could be better suited for the reconstruction of a function with spherical support than tetrahedra. Marching hexahedra produces less triangles than marching tetrahedra and marching octahedra, and achieves better rendering results. But it still performs worse than marching cubes in terms of rendered triangles and image quality. In a different paper, Carr et al. [5] identified a special kind of artifact occurring when BCC grids are used as subdivision of cubic grids, which they referred to as "girders" (see figure 2.3). Such artifacts can also be observed in their BCC iso-surface renderings, and they always show up if certain conditions are met.

Iso-surfacing on BCC and FCC grids was also proposed by Ibáñez et al. [25]. They use BCC grids because of their spectral properties and operate directly on the Voronoi regions.

Chapter 3

Practical Reconstruction Schemes

In this chapter we present several practical resampling strategies on BCC grids first proposed in [59], and extended with some new methods. We implemented and tested them in a raycasting system for getting the best possible rendering quality without the unwanted side effects introduced by other rendering methods. They can find application in many other rendering algorithms which need an interpolation in a BCC grid. We exploited all of the BCC properties listed in chapter 1. The reconstruction schemes range from fast methods with reasonable quality (e.g., bilinear interpolation) to more complex methods comparable to trilinear interpolation in terms of rendering quality and performance. In the last part of this chapter we introduce several schemes for gradient reconstruction in BCC grids.

Some of the reconstruction methods on the BCC grid have parameters which are depending on the current view direction. Interpolation is used in many applications where we initially do not have a view direction (e.g., segmentation). For such applications we must define a virtual view direction. Nevertheless we consider this as a drawback which limits the usability of some of the reconstruction schemes.

3.1 Bilinear Interpolation

The bilinear interpolation scheme exploits the fact that a 3D CC grid can be seen as stack of 2D CC grids. Thus a faster bilinear interpolation in the planes suffices, which was extensively used to speed up calculation for the shear-warp algorithm [30] and for raycasting on Cartesian grids [62]. To use this scheme for raycasting, we must assure that the first resampling location of the ray entering a volume is located on such a plane. The planes are chosen so that they are most perpendicular to the current view direction.

We already know that a BCC grid can be seen as stack of CC grid planes as well, and thus the scheme can be easily extended to BCC grids. The step size used for resampling is determined by the distance between the planes (see figure 3.1). Depending on the current view direction, it varies between 1 and $\sqrt{3}$ in the CC grid. In the BCC grid, we generally have a smaller distance between the planes for all possible view directions. It has a variation between $\sqrt{2}/2$ and $\sqrt{3/2}$.

In a common assumption, step size should be less than 1 for accurate resampling [62]. This

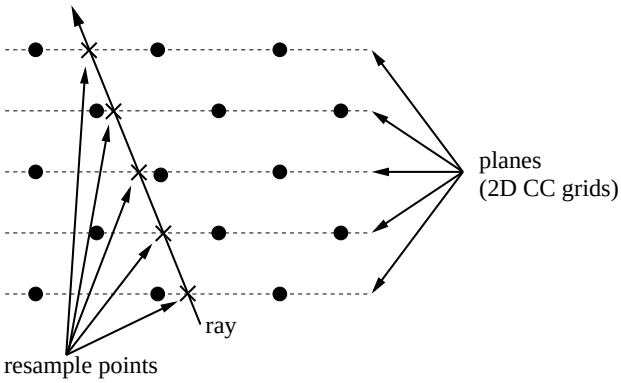


Figure 3.1: Bilinear interpolation in the planes most perpendicular to the viewing direction. The planes are comprised of 2D CC grids.

is never the case in the CC grid. In the BCC grid the step size is sufficient to meet this constraint, if the planes are relatively perpendicular to the actual view direction. On the other hand, the 2D CC planes consist of half of the sample points in the BCC grid, and the points are $\sqrt{2}$ times wider apart. Hence we lose details in the planes.

In order to halve the step size, Wan et al. [62] suggested an intermediate sample step exactly in between two planes. The required trilinear interpolation is still less complex than standard trilinear interpolation. This is similar to the insertion of intermediate slices proposed by Sweeney et al. [57] for the shear-warp algorithm. Simplified trilinear interpolation on the BCC grid can be achieved with a specialized version of the sheared trilinear interpolator from chapter 3.5.

Wan et al. [62] also suggested that the scheme is perfectly suited for adaptive resampling, halving the step size as long as the density difference between two adjacent sample locations exceeds a certain threshold.

3.2 Bilinear plus Spatial Interpolation

The bilinear interpolation scheme can be extended into the spatial dimension. As a consequence, the resample points are not restricted to lie on a plane like in the bilinear reconstruction scheme. A resample point can be located anywhere between two adjacent planes. We first use bilinear interpolation on both planes. Then the bilinearly interpolated density values are linearly interpolated with an interpolation factor corresponding to the spatial position between the planes. Hence we denote this reconstruction method as bilinear plus spatial interpolation. In the 2D visualization from figure 3.2, we denote the interpolated density values from the planes as S_i and S_{i+1}, and α refers to the spatial position between the planes. Then the resulting density value S_α denotes to:

$$S_\alpha = (\frac{\sqrt{2}}{2} - \alpha)S_i + \alpha S_{i+1} \qquad \text{for } 0 < \alpha < \frac{\sqrt{2}}{2} \tag{3.1}$$

Similar to the bilinear approach, we employ the stack of planes which is most perpendicular

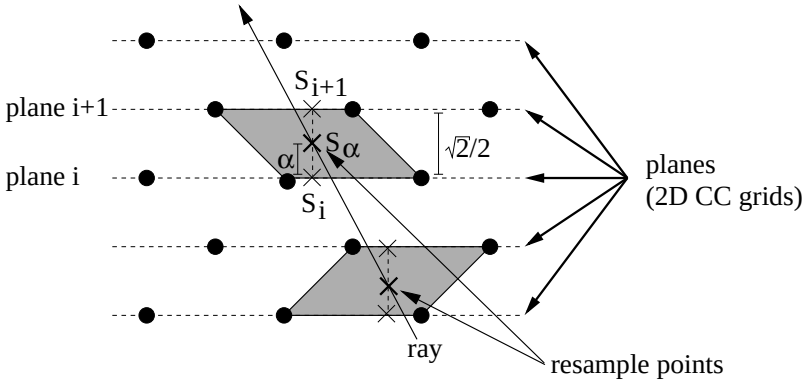


Figure 3.2: Bilinear plus spatial interpolation shown in 2D. After we bilinearly interpolate the values from the planes (S_i and S_{i+1}), a linear interpolation using the spatial position α as weight yields the final scalar value S_α (according to equation 3.1).

to the current viewing ray direction. Our experiments showed that the best results are achieved if the planes are chosen in that way. This reconstruction is a trilinear interpolation in a sheared cubic cell. The shear direction of the current cell (e.g., towards the positive or negative axes) depends on the actual location of the resampling point. In figure 3.2, we can see that the first cell containing a resampling point is sheared into the positive direction. The second resampling point is located slightly more to the left, and accordingly the cell is sheared into the negative direction.

3.3 Barycentric Interpolation

The barycentric interpolation exploits the fact that a BCC grid can be seen as uniquely defined tetrahedral mesh. We use the barycentric coordinates of the resampling point inside the current tetrahedron as weights and interpolate between the vertex values. This results in a piece-wise linear interpolation, and is not expected to be equal in quality to a trilinear interpolation. On the other hand, we expected it to perform better than any kind of trilinear reconstruction. Only the four vertices of the tetrahedron must be processed, as opposed to eight samples for trilinear interpolation in a hexahedral cell. However, it is not trivial to find the actual tetrahedral cell and the barycentric coordinates. Thus we did not achieve a performance gain over trilinear methods on our architecture (there is still the possibility of a performance gain on other architectures). In addition to the linear interpolation, the following computational steps are required:

1. Find the tetrahedron where the resampling point is located in. First we determine the corresponding octant of the cell in the primary (secondary) grid using three comparisons. We need some more comparisons (x greater or smaller y, x greater or smaller z, and y greater or smaller z) to find the current tetrahedron. The tetrahedralization is visualized in figure 3.3.

2. Compute the barycentric coordinates. This is done by transforming the resampling point into a coordinate system where one vertex of the tetrahedron is in the origin and the other

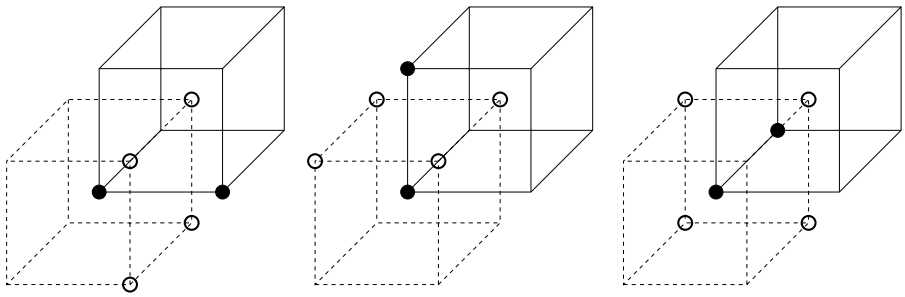


Figure 3.3: The Delaunay tetrahedralization of the BCC grid. Two adjacent points (white dots) together with the two points of the spine (black dots) in *x*, *y*, and *z* direction (from left to right) make up a tetrahedron.

vertices lie in unit distance on the *x*, *y* and *z* axis. The barycentric coordinates are equivalent to the new location of the resampling point. In figure 3.3, we can see that there are twelve sets of different tetrahedra in the BCC grid after translation to the origin. The transformation matrices can therefore be precomputed and stored in a table.

3.4 Trilinear Interpolation

In order to apply standard trilinear interpolation in a BCC grid, we could resample the grid into a larger CC grid in a preprocessing step. This step would completely destroy the storage advantage of the BCC grid. Hence we rather find and resample a Cartesian grid cell on the fly. The largest cubic cell that has no other sample points inside is given by the intersection of the primary and secondary grid cell that both contain the resampling point (the grey cube in right part of figure 3.4). This is the same octant of a cubic cell we used to find the tetrahedron in barycentric interpolation (section 3.3). A 2D visualization (left image in figure 3.4) will help in understanding the more complicated spatial case (right image in figure 3.4). Two corners of this cubic cell are given by the BCC grid. We have two choices to interpolate the density values from each of the other six cubic cell corners (also shown in figure 3.5), considering either 2 or 6 samples in the calculation:

- A linear interpolation between the 2 sample points *a* and *b* as shown in the left image of figure 3.5. *a* and *b* are from the same (either primary or secondary) CC grid. They are the sample points closest to the corner of the cubic cell which we want to resample. This method is prone to quite visible artifacts.

- A linear combination of *a* and *b* from one (primary or secondary) CC grid and the samples *c*, *d*, *e*, and *f* from the other CC grid (see right image of figure 3.5). The weights are proportional to the distance to the corner of the cubic cell. This is a linear interpolation in the octahedron defined by the 6 samples. We get better results with this approach, with the expense of much higher complexity.

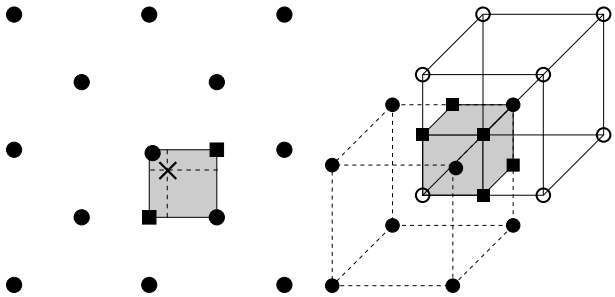


Figure 3.4: Trilinear interpolation in the BCC grid (right image) and the situation in 2D (left image). To interpolate at a position in the shaded box, first the missing corners of the box (square dots) are interpolated.

As a third choice we could consider just the four samples *c*,*d*,*e*, and f for resampling of the cubic cell corner. However, with this approach we do not take the spatially closest samples *a* and *b* into account. Accordingly, the results are not very good. It is easy to see that trilinear interpolation on the BCC grid is a continuous, piecewise cubic reconstruction of the resampling point. But it requires more computations than trilinear interpolation on the CC grid. In the following chapters we will refer to this interpolator as trilinear with 2 samples or trilinear with 6 samples, depending on the number of samples used for resampling of the cubic cell corners.

3.5 Sheared Trilinear Interpolation

We already know that a BCC grid can be seen as sheared and scaled CC grid [60]. The cubic cells are transformed into sheared cubic cells in the BCC grid. The cells consists of four short edges and eight long edges. The long edges are aligned to the 2D CC grid planes. In the sheared cubic cells we use a trilinear interpolation, which we denote as sheared trilinear interpolation. A 2D visualization of sheared trilinear interpolation is shown in figure 3.6.

First we apply linear interpolations along the four short edges of the sheared cubic cell. The resulting interpolated values are referred as *a*, *b*, *c*, and *d* in figure 3.6. In the figure we can also see that the weights used in this interpolation are determined by the distances α and $\frac{\sqrt{2}}{2} - \alpha$ from the neighboring 2D CC grid planes. This is similar to the spatial interpolation in the bilinear plus spatial reconstruction scheme. The scalar values *a*, *b*, *c*, and *d* can then be used for a bilinear interpolation of the final density value.

We can choose from three stacks of 2D CC grid planes, like in the bilinear interpolation and bilinear plus spatial interpolation. The cells are sheared along the two axes that span the planes, hence we denote them as shear planes. There is an additional degree of freedom in sheared trilinear interpolation, because the shear directions of the cells must also be specified. We made the following considerations about the proper selection of the shear planes and the shear directions of the cells:

- We choose the shear planes so that they are most perpendicular to the viewing ray direction,

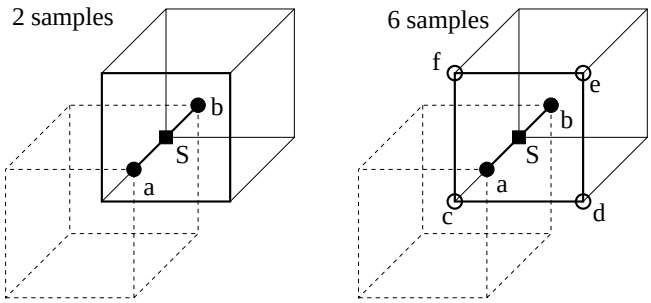


Figure 3.5: Trilinear interpolation in the BCC grid. To interpolate the corner value *S* of the cubic cell, we can either consider only two samples (*a* and *b* in the left image) or six samples (*a*-*f* in the right image).

similar to choosing the proper resampling planes in the bilinear interpolation approach from section 3.1. Figure 3.6 illustrates this approach in a 2D visualization.

- We made the observation that choosing the shear directions is less important than choosing the shear planes properly, but nevertheless slightly influences image quality. We choose the shear directions (either positive or negative) of the cells so that the cell borders are as parallel as possible to the viewing ray. This assures that the ray will pass through the sheared cell as similar as possible to how it would pass through the original cubic cells. The shear directions are equal for all cells and only change if there is a change of the view direction. If the viewing ray pointed slightly to in the right (right image from figure 3.6), a shear towards the positive axis would be chosen.

3.6 Alternative Sheared Trilinear Interpolation

For sheared trilinear interpolation we used the fact that the BCC grid can be seen as sheared and scaled CC grid. The idea behind the alternative calculation is to apply the inverse shear and a scale in the *z* axis to the volume and to each resample location (for the storage schemes from equation 1.8 and equation 1.9). Then we can apply standard trilinear interpolation in the cubic cells of a CC grid. This alternative calculation of the sheared trilinear interpolation in a BCC grid can be done with the following steps:

1. Interpret the BCC grid as CC grid, where the index values are used as sample positions.
2. Apply the inverse transformation (a shear and a scale in *z* axis) to each resample location.
3. Use trilinear interpolation in a CC cell for the reconstruction of the new resample location.

The inverse transformation matrix for the first BCC storage scheme (equation 1.8) is given by:

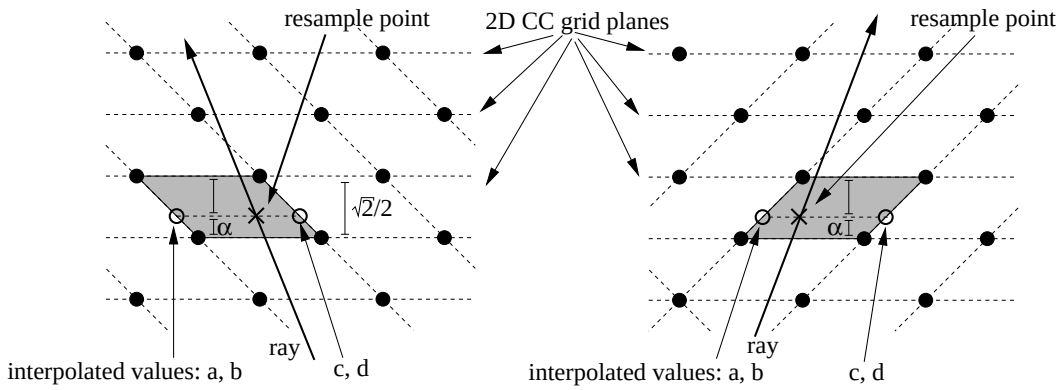


Figure 3.6: 2D visualization of sheared trilinear interpolation in the BCC grid. The interpolated values *a*, *b*, *c*, *d* are used in a bilinear interpolation. The planes and shear direction of the cells are chosen so that they are as closely as possible aligned to the ray direction.

$$V_{INV} = \begin{pmatrix} \frac{1}{T} & 0 & -\frac{1}{T} \\ 0 & \frac{1}{T} & -\frac{1}{T} \\ 0 & 0 & \frac{2}{T} \end{pmatrix} \tag{3.2}$$

There are some restrictions to the alternative sheared trilinear approach. The shear planes and shear directions are implicitly determined by the sample indices and cannot be chosen freely. For the first storage scheme, the shear planes must be the planes spanned by the *x* and the *y* axis, and the shear directions of the cells are towards the positive *x* and *y* axis. For a BCC grid stored in memory with the second scheme (equation 1.9), there is a different inverse transformation matrix:

$$V_{INV} = \begin{pmatrix} \frac{1}{T} & 0 & -\frac{1}{k}\frac{1}{T}(k \bmod 2) \\ 0 & \frac{1}{T} & -\frac{1}{k}\frac{1}{T}(k \bmod 2) \\ 0 & 0 & \frac{2}{T} \end{pmatrix} \tag{3.3}$$

However, this matrix is only suited for transforming index values, but is not applicable for floating point resampling locations. In the following calculations we use a cell spacing unit $T = 1$. The indices *i* and *j* are equal to their spatial positions *x* and *y* in the BCC grid for all indices where *k* is even. Whereas each resampling point that is located on a plane with odd *k* must be shifted by 0.5 in negative direction of the *x* and *y* axis. Between slice planes, we must linearly interpolate between 0.0 and 0.5. As we have spacing 0.5 between the planes in the *z* direction, the actual *z* position at a plane with index *k* is $\frac{k}{2}$. The offset $o(z)$ that is subtracted from the *x* and *y* component of each resample location must therefore be a piecewise linear function of the *z* position, that is, a tent function:

$$o(z) = \min(z - \lfloor z \rfloor, 1 - (z - \lfloor z \rfloor)) \tag{3.4}$$

A 2D visualization of the alternative sheared trilinear interpolation is shown in figure 3.7. The inverse shear transforms the BCC grid in the leftmost image into the rectilinear grid from the

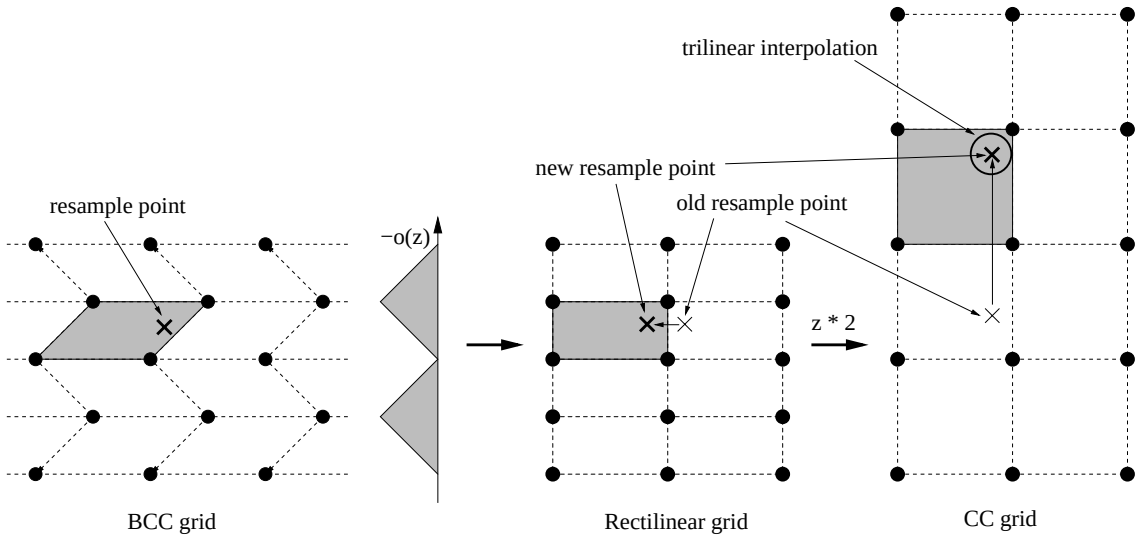


Figure 3.7: Alternative sheared trilinear interpolation shown in 2D. The inverse shear according to $o(z)$ is applied to the resample point and the BCC grid. After a scale in z direction is applied to the new resample point and the rectilinear grid, we can use trilinear interpolation in a CC grid.

middle image. $o(z)$ only produces positive values, which are subtracted from the x and y positions of the resample point. It can be verified that $o(z)$ defines an alternating positive-negative shear of the cells. $o(z)$ is zero at the first plane. Between the first and the second plane of the volume, we (inverse) shear the resample location and the volume towards the negative x and y axes. This means that the shear direction of the cells in the BCC grid is towards the positive axes. At the second plane, the offset $o(z)$ reaches the maximum. From the second plane to the third plane, we (inverse) shear the resample location and the volume towards the positive x and y axes, until the offset $o(z)$ is zero. Hence the shear direction of the cells in the BCC grid is towards the negative axes. From the third plane to the fourth plane, the resample location is again (inverse) sheared towards the negative x and y axis, and so forth.

As a final step we must scale the z position of the resample location and the volume by 2.0. In figure 3.7, this step transforms the rectilinear grid from the middle image into the CC grid from the right image. The new resample point can be trilinearly interpolated in a CC grid where the sample positions are equal to the indices. The matrix V_r for transforming a resample location into a new one is given by:

$$V_r = \begin{pmatrix} 1 & 0 & -o(z) \\ 0 & 1 & -o(z) \\ 0 & 0 & 2 \end{pmatrix} \tag{3.5}$$

It is not possible to define the shear directions of the cells in any other way, because the samples that span a cell are determined by the index values. The indices $\chi_{c(i,j,k)}$ of a Cartesian grid cell c with a smallest index (i, j, k) of a corner point are given by

$$\chi_{c(i,j,k)} = (i+u, j+v, k+w) \qquad \text{for } u, v, w \in 0, 1. \tag{3.6}$$

It can be verified that this equation must also be true for the indices of our sheared cubic cells. Assume a positive shear direction for all cells. Then we have following indices:

$$\chi_{c(i,j,k)} = \begin{cases} (i+u, j+v, k+w) & \text{if odd}(k) \\ (i+u+w, j+v+w, k+w) & \text{if even}(k) \end{cases} \quad \text{for } u,v,w \in 0,1 \tag{3.7}$$

But this is a contradiction to equation 3.6. The same can be stated for a negative only shear direction. Fortunately, our results show that choosing the shear planes appropriately is much more important for the visual appearance than the shear directions. To change the shear planes when the principal viewing axis changes to x (y), we must rearrange the indices of the volume so that the indices are aligned with the volume planes along the x (y) axis instead of the z axis. The computational overhead of the reordering can be avoided by keeping keep the volume in memory three times, which results in a large memory overhead (i.e., three BCC grid volumes need 2.12 times more storage than one equivalent CC grid volume). Nevertheless this method has a big advantage. We are able to compute a sheared trilinear interpolation in environments that are otherwise restricted to standard trilinear interpolation in a CC grid, like 3D texture mapping hardware.

3.7 Gradient Reconstruction

Gradient calculation is an important issue in computer graphics, as they give valuable information regarding the shape of surfaces. In volume rendering, they are used for classification and directional shading. Two gradient estimation schemes on the BCC grid were proposed by Theußl et al. [60].

The most commonly used gradient estimator on Cartesian grids is the central differences operator. BCC grids are comprised of two CC grids (the primary and secondary grid). Hence the implementation of central differences on the BCC grids is straightforward. As a consequence of the used indexing scheme, just the z direction must to be handled differently. We must take the next sample from the same CC grid (either the primary or the secondary grid) for calculation, i.e., the second sample in the BCC grid:

$$\begin{aligned} g_{1_x}[x,y,z] &= \frac{1}{2T}(f[x+1,y,z] - f[x-1,y,z]) \\ g_{1_y}[x,y,z] &= \frac{1}{2T}(f[x,y+1,z] - f[x,y-1,z]) \\ g_{1_z}[x,y,z] &= \frac{1}{2T}(f[x,y,z+2] - f[x,y,z-2]) \end{aligned} \tag{3.8}$$

In the following chapters we will refer to this approach as central differences 1. This scheme does not consider the closest samples for gradient calculation. Theußl et al. [60] developed a second scheme. This approach uses the fact that a BCC grid can be seen as CC grid with a sample point in the center of each cubic cell (see figure 1.2). They compute the average of the central differences at each edge of the cubic cell the sample point is located in. The eight samples on the corners of the cubic cell are used in the computation, which are the actual closest samples:

$$
\begin{aligned}
g_{2_x}[x,y,z] &= \frac{1}{4T} \sum_{\substack{i,j\in 0,1 \\ k\in -1,1}} h(i)\, f[\overline{x}-i, \overline{y}-j, z-k] \\
g_{2_y}[x,y,z] &= \frac{1}{4T} \sum_{\substack{i,j\in 0,1 \\ k\in -1,1}} h(j)\, f[\overline{x}-i, \overline{y}-j, z-k] \\
g_{2_z}[x,y,z] &= \frac{1}{4T} \sum_{\substack{i,j\in 0,1 \\ k\in -1,1}} h(k)\, f[\overline{x}-i, \overline{y}-j, z-k]
\end{aligned}
\tag{3.9}
$$

where $\overline{x}$ and $\overline{y}$ refer to

$$
\begin{aligned}
\overline{x} &= x + (z \bmod 2) \\
\overline{y} &= y + (z \bmod 2)
\end{aligned}
$$

and $h(x)$ denotes

$$
h(x) = \begin{cases} -1 & \text{if } x > 0 \\ 1 & \text{if } x \leq 0 \end{cases} \tag{3.10}
$$

We will refer to this method as central differences 2 in the following chapters. The calculation is more complex in this approach, but gradient calculation is usually done in a preprocessing step, and thus not time-critical. Even computational demanding methods can be used in practice.

We suggest to take the average of the results of both central differences 1 (g_1 from equation 3.8) and 2 (g_2 from equation 3.9) as the gradient. In the vicinity of the current sample are points from both the primary and the secondary grid. The central differences 1 method is computed only with samples from the same CC grid, the second only with samples from the other CC grid. Instead we assume that a linear combination of both methods yields a more balanced gradient calculation:

$$
\begin{aligned}
g_{3_x}[x,y,z] &= \frac{g_{1_x}[x,y,z] + g_{2_x}[x,y,z]}{2} \\
g_{3_y}[x,y,z] &= \frac{g_{1_y}[x,y,z] + g_{2_y}[x,y,z]}{2} \\
g_{3_z}[x,y,z] &= \frac{g_{1_z}[x,y,z] + g_{2_z}[x,y,z]}{2}
\end{aligned}
\tag{3.11}
$$

Note that the spatial distances are already taken into account in the weight factors used for calculating central differences 1 and central differences 2, hence simply averaging the results of the first two methods suffices. We will refer to this method as central differences 3 in the following chapters. A comparison of the central differences methods is shown in the figure 3.8.

The same approach that we used to adapt central differences 1, i.e., operating on the primary (secondary) grid, can be employed for a simple adaption of other CC grid gradient estimations

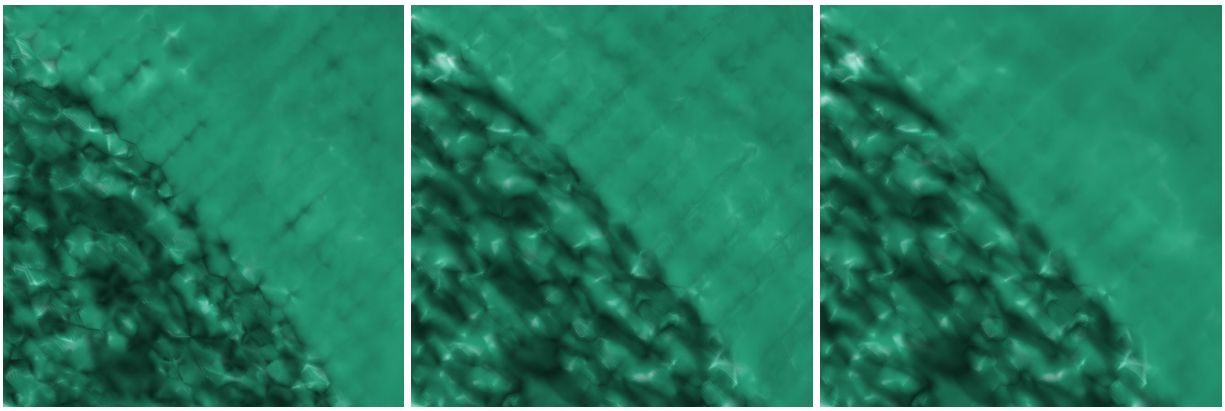


Figure 3.8: Raycasting on Melon dataset (closeup) using different gradient estimators: Barycentric interpolator with central differences 1, 2 and 3 (left to right). Artifacts on the skin are reduced in the rightmost image.

Gradient Estimation scheme	Operates on (CC grid)	# samples	Closest samples
Central differences 1	same (primary or secondary)	6	no
Central differences 2	other (primary or secondary)	8	yes
Central differences 3	both (primary and secondary)	14	yes
Sobel filter	same (primary or secondary)	27	no
Adaptive grey level	same (primary or secondary)	3 - 6	no

Table 3.1: Gradient estimation schemes on the BCC grid. The terms "same", "other", and "both" indicate if we consider samples from the the same, the other or both (primary / secondary) CC grids when calculating the gradient of a sample position. "# samples" refers to the number of samples considered in the calculation.

as well, like the more sophisticated 3×3 sobel filter or the adaptive grey level estimation scheme proposed by Yagel et al. [46], for instance.

We will shortly sketch the idea of the latter approach. In the primary (secondary) CC grid there are six samples which are adjacent to the current voxel in the x, y, and z axis. The adaptive gray level estimation scheme considers either three or six from these samples, depending on the object thickness. To estimate the object thickness, we compare the scalar values of both adjacent samples in each of the three axes to the current voxel value. If the voxel value is higher (or lower) than both adjacent samples, we consider only one of the two samples in the gradient calculation, i.e., the sample with the higher data difference to the current voxel value. We use this sample in the calculation of either forward or backward differences. If the voxel value is between the scalar values of the adjacent samples, both samples are taken for the calculation of central differences.

In all of our gradient estimation methods on the BCC grid we either do not take the closest sample points (e.g., central differences 1) or average over many samples (e.g., central differences 2 and 3) grid. This is shown in table 3.1, where we summarized all gradient estimation schemes that we implemented and tested for use on BCC grids. Hence the danger of unwanted smoothing

effects is given [2].

We still do not take the closest samples in the adaptive grey level estimation approach on the BCC grid. However, this approach is known to preserve high frequencies better than central differences. Our experiments showed that the adaptive grey level estimation scheme slightly compensates the tendencies towards smoothing on the BCC grid. This can be observed in the rendering results from section 7.3.1.

Chapter 4

Texture-Based Volume Rendering

In this chapter we explain the idea behind texture-based volume rendering. Then we discuss 2D texture-based rendering, two of the most important extensions, (i.e. multitexture blending, pre-integration), and 3D texture-based volume rendering. For all approaches, we describe how they can be modified to support BCC rendering.

4.1 Basics

The idea behind texture-based volume rendering is to represent the volume as a set of textured semi-transparent quadrilaterals. The advantage of this approach is that the volume rendering pipeline [32] can be entirely loaded on the hardware, achieving interactive frame rates for medium sized volumes. Bilinear or trilinear interpolation is efficiently computed by specialized texture hardware, and the compositing step is done by the alpha blending functionality of the hardware. There are two main approaches to use texture-mapping for volume rendering, 2D texture-based volume rendering using object aligned slices and 3D texture-based volume rendering using view-aligned slices (refer to figure 4.1).

4.2 2D Texture-Based Volume Rendering

For 2D texture-based volume rendering, the volume is downloaded to 2D textures as stack of 2D CC slices. The result is a bilinear reconstruction in the planes. In principle, this approach is the hardware-accelerated equivalent to the shear-warp algorithm. The image quality is best when the slices are as parallel as possible to the view plane (rotated by 90°, they would be completely invisible). Therefore three stacks of texture slices are stored, which represent the principal view axes. The slice stack that is most perpendicular to the actual viewing direction is processed. The main advantage of 2D textures over 3D textures is that they are not only available, but also performance-optimized on virtually every graphics hardware. The drawbacks of the approach are reduced image quality and slicing artifacts, mostly visible on the sides of the volume. Because of the missing spatial interpolation, super-sampling is not possible. Furthermore, popping artifacts

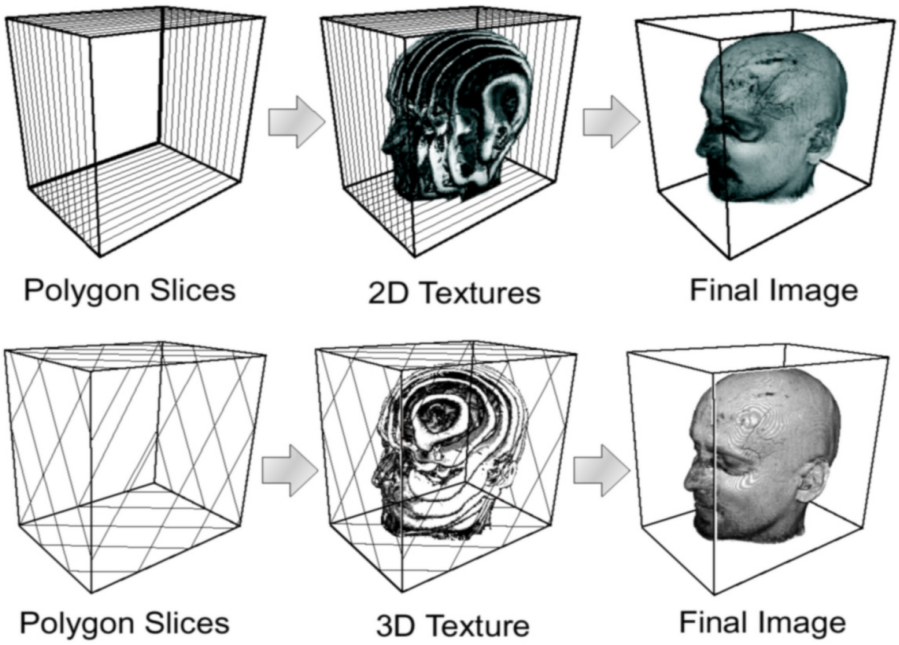


Figure 4.1: 2D texture-based rendering using object-aligned slices and 3D texture-based rendering using view-aligned slices. Image taken from [18].

occur when the slice stack is changed. Because of a varying slice distance depending on the current view axis, we also must correct the resulting opacity values.

2D Texture-Based Volume Rendering on BCC grids

Following the bilinear approach from chapter 3.1, we are using the fact that the BCC grid is also comprised of 2D CC grid slices. If the BCC grid is stored with the second scheme from Theußl et al. [60] (described in chapter 1.4), a reordering of the samples is necessary before downloading the slices to the 2D textures in the principal view axis x (y). This is because in the principal view axis x (y), the sample points with an equal i (j) index do not belong to the same slice of the volume (refer to figure 1.4). We must consider the offset of half a unit for the position of every second slice. There are two choices to implement this offset. We can

1. set the proxy polygon positions accordingly.
2. assign texture coordinates which are translated by minus half a unit.

In our implementation we used the second approach, where we have to consider that texture coordinates are usually normalized in the range $[0..1]$. In the following we denote the spatial extent of our volume in x, y and z direction as s_x, s_y, and s_z (using cell spacing $T = 1$). In texture space, the offset of half a unit is given by $(\frac{1}{2s_x}, \frac{1}{2s_y})$ for the principal view axis z. For principal view axis x it is given by $(\frac{1}{2s_y}, \frac{1}{2s_z})$, and for the principal view axis y it is given by $(\frac{1}{2s_x}, \frac{1}{2s_z})$.

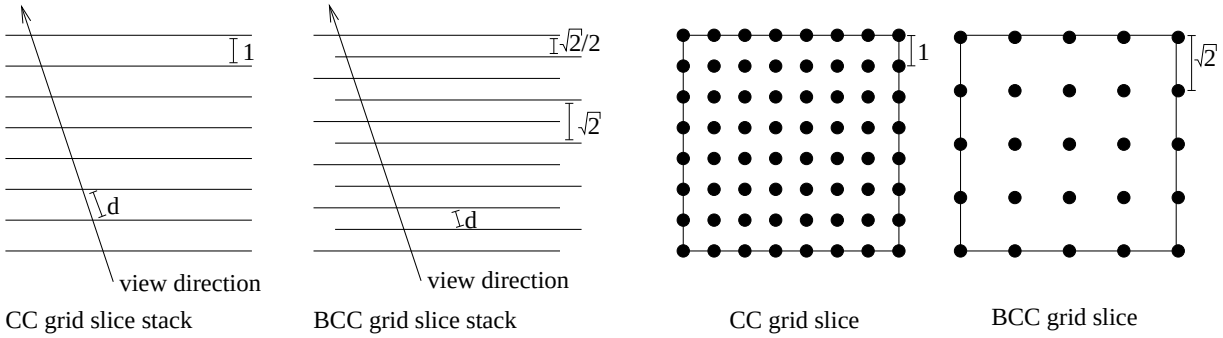


Figure 4.2: The corresponding slice stacks and slices used for 2D texture-based rendering in the CC and BCC grid. The sampling distance d is smaller on BCC grids, whereas the slices contain more information on CC grids.

We have $\sqrt{2}$ times more planes in the BCC grid. Therefore we also need $\sqrt{2}$ more textured slices in the BCC grid to match all planes in the volume. This also means that we constantly have a higher sample frequency in the BCC grid. On the other hand, the size of a single texture is two times smaller than a texture for the CC grid. As a consequence, less information is stored in a single BCC grid slice. The situation is shown in figure 4.2.

4.3 Multi-Texture Blending

We can cope with the major shortcomings of 2D texture-based rendering with the multi-texture blending approach proposed by Rezk-Salama et al. [48]. Intermediate slices are rendered by blending the textures of the two neighboring slices on the intermediate proxy polygon with a factor that matches it's spatial position between the slices. This substitutes the bilinear interpolation with a trilinear one, making super-sampling possible. If α is the distance from the front slice S_i, then the blending equation for intermediate slice $S_{i+\alpha}$ is given by:

$$S_{i+\alpha} = (1-\alpha)S_i + \alpha S_{i+1} \tag{4.1}$$

Multi-Texture Blending on BCC grids

In order to adapt multi-texture blending to BCC grids, we only have to consider that either the front or the back slice in the blending step belongs to the secondary grid. Like we explained in section 4.2, texture coordinates moved by an offset of minus half a unit with respect to the primary grid must be assigned to secondary grid slices. The resulting interpolation is equivalent to the bilinear plus spatial interpolation in a sheared cubic cell, which is introduced in chapter 3.2.

4.4 Pre-Integration

To use pre-integration for texture-based volume rendering, we render pre-integrated slabs instead of slices. A slab consists of the volume between two adjacent slices. The transfer function is integrated in a pre-processing step for all possible combinations of scalar values s_f and s_b on the front and back slice of a slab. The slab thickness is usually assumed to be constant (neglecting the varying slice distance). The resulting pre-integration table can be downloaded to a (dependent) 2D texture. The scalar value s_f and s_b on front and back slice of a slab are employed as texture coordinates for a dependent texture lookup of the pre-integrated color and opacity values.

With the use of pre-integration, accurate renderings for arbitrary transfer functions are possible without super-sampling. Hence this is another approach to overcome some of the limitations of 2D texture-based volume rendering. Although we discuss only pre-integration for rendering object-aligned slabs in this work, it must be mentioned that pre-integration can also be exploited for rendering view-aligned slabs using 3D textures. Furthermore, we introduce pre-integration for tetrahedral cells in chapter 5.8.

It is difficult to use correct directional shading together with pre-integration, because we are restricted to maximal three different parameters in order to exploit hardware-acceleration (i.e., to use 2D or 3D textures).

Pre-Integration on BCC grids

Adapting the pre-integration approach for object-aligned slabs to BCC grids is similar to the adaption of multi-texture blending to BCC grids. We must assign texture coordinates translated by minus half a unit to either the front or the back slice, in order to get the correct density values of the back and front slice for the dependent texture lookup. The thickness of a single slab is $\sqrt{2}/2$. Accordingly, we have $\sqrt{2}$ times more slabs in the BCC grid than in the CC grid.

4.5 3D Texture-Based Volume Rendering

If the GPU supports 3D textures, we can download the volume to a single 3D texture and render the slices parallel to the view plane. 3D texture-based volume rendering has several advantages over 2D texture-based approaches, first of all the "natural" trilinear interpolation, but also constant slice distance for all view angles, no popping artifacts and possible super-sampling.

3D Texture-Based Volume Rendering on BCC grids

Adapting 3D texture-based rendering to the BCC lattice seems to be difficult at first sight because of the hard-wired CC grid trilinear interpolation. As a brute force approach, we could resample a Cartesian grid from the BCC grid. This would yield a texture six times as large as the original one, so this is not practicable.

Röber et al. [49] downloaded the primary and secondary grid to two separate 3D textures, translating the secondary grid by half a unit and blending the resulting fragments with a factor

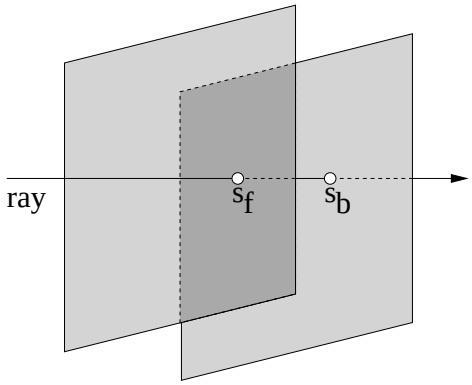


Figure 4.3: A slab consists of the volume between two slices. Back and front scalar value of the slab (s_f, s_b) are used for a lookup into a dependent texture storing the pre-integration table. Image redrawn from [18].

of 0.5 in the register combiners. This method has the drawback that the interpolations in the primary and secondary grid do not take the spatially closest samples into account.

Fortunately, we already introduced a sheared trilinear interpolation approach in chapter 3.5. The BCC grid can be seen as sheared and scaled CC grid. This interpolation can therefore be alternatively calculated by applying the inverse transformation to a resample point and to the volume. Then we can employ trilinear interpolation in a cubic cell of a CC grid (see chapter 3.6). Using the second storage scheme from Theußl et al. [60], this inverse shear must be an alternating positive-negative one. The idea becomes intuitively clear when looking at figure 4.4. The inverse shear can be calculated by subtracting an offset from the current resampling point. The offset is a (tent) function of z position (refer to figure 3.7). Because we are using view-aligned slices, the offset cannot be applied per vertex. Instead we must compute the offset and subtract it from the actual texture coordinates per fragment, which is the equivalent to a resampling point in hardware rendering. Fortunately today's fragment programs are very powerful, and provide all means necessary for this calculations. After the transformations in the pixel shaders, the hardware does standard trilinear interpolation in a rectangular volume during the rasterization step. This reconstruction is equivalent to a sheared trilinear interpolation in a sheared cubic cell.

When the texture coordinates have been updated, it makes no difference whether the data was originally given on a BCC or CC grid, and we can use any rendering mode that was originally developed for CC grid rendering. Examples are pre-classification, post-classification, shaded post-classification, to mention just a few. This is another advantage of our method over the approach from Röber et al. [49], which is quite inflexible in that regard.

Given the texture coordinates (x_t, y_t, z_t), we will sketch the procedure for the principal view axis z. For convenience, the offset function from equation 3.4 was given by

$$o(z) = \min(z - \lfloor z \rfloor, 1 - (z - \lfloor z \rfloor)).$$

Note that the texture coordinates (x_t, y_t, z_t) are given in the range $[0..1]$. To use the offset function $o(z)$ for 3D texture-based volume rendering, we must do the following:

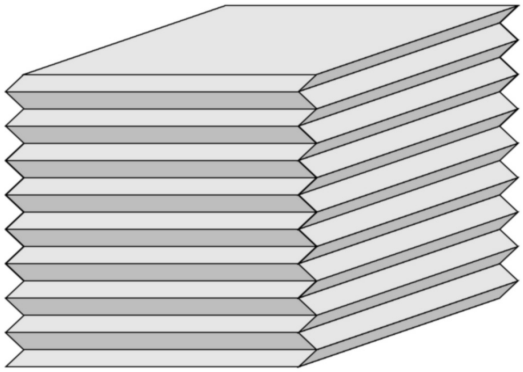


Figure 4.4: The BCC grid as rectangular volume where an alternating shear is applied per volume slab. We do the reverse process: Apply the inverse shear and interpolate in the rectangular box. Image taken from [16].

- Transform z_t from texture space to world space.
- Compute the offset function $o(z)$
- Transform the offset back to texture space.
- Subtract the offset in texture space from texture coordinates x_t and y_t.

Like in section 4.2, we denote the spatial extent of our volume in x, y and z direction as s_x, s_y, and s_z (using cell spacing $T = 1$). The mapping from texture space to world space is given by:

$$z = z_t s_z - \frac{1}{4} \tag{4.2}$$

Because of the way how the hardware maps indices to texture coordinates, the subtraction of $\frac{1}{4}$ is necessary in above equation for the correct calculation of the current z value. Now the actual texture coordinates x_t and y_t are updated with the proper offset value:

$$\begin{aligned} x_t &= x_t - \frac{o(z)}{s_x} \\ y_t &= y_t - \frac{o(z)}{s_y} \end{aligned} \tag{4.3}$$

In figure 4.5 we show the Cube dataset that is rendered with (left image) and without (right image) applying the inverse shear to the fragments. The offset values $o(z)$ are color coded in red and green, indicating the positive-negative shear of the cells. Note that the pike-shaped artifacts on the sides of the cube in the left image disappear in the right image.

In chapter 3.5 we stated that we get the best visual quality when the shear planes are chosen most perpendicular to the current view direction. To achieve this in hardware, we have to

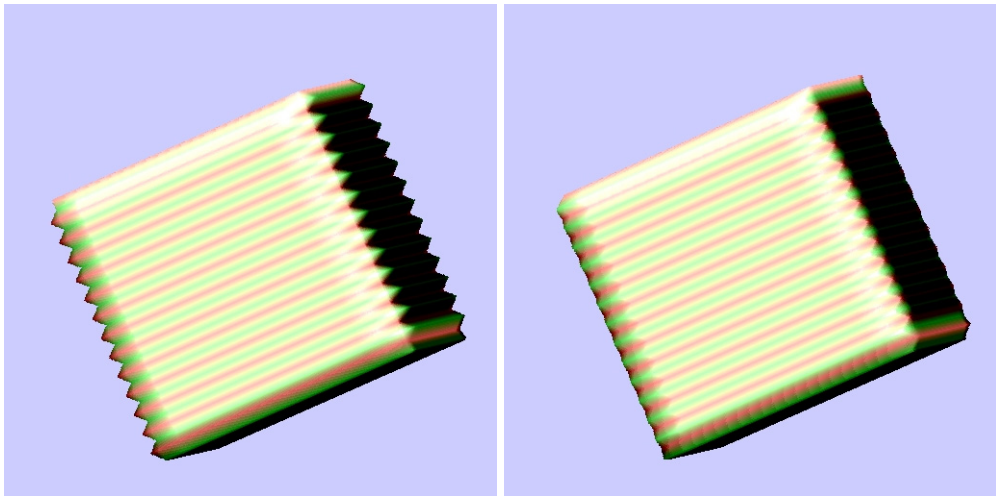


Figure 4.5: 3D texture-based (iso-surface) rendering of a cube without (left image) and with (right image) application of the inverse shear.

rearrange the sample points for the remaining two principal view axes x and y. The resulting volumes for all principal view axes must be stored in three separate 3D textures. This is also the main drawback of the method, as memory consumption is much higher, and popping artifacts are introduced if the principal view axis changes. The offset is always a function of the current principal view axis, respectively.

Chapter 5

Projected Tetrahedra Algorithm

In this chapter we will first outline the original algorithm by Shirley and Tuchman [54]. In more details we discuss the steps of the algorithm that are implemented differently depending on the type of grid, i.e., the tetrahedralization and the depth sort (which effectively is a back-to-front traversal in both CC and BCC grids). Then we introduce a couple of improvements to enhance rendering performance. Further we describe an adjacency structure which allows fast traversal and show how it can be used on a tetrahedral mesh. Afterwards we present powerful techniques to increase rendering quality. At last we suggest some approaches to enable directional shading.

5.1 Algorithm Overview

We approximate the volume rendering integral by projecting semi-transparent tetrahedral cells to the screen in back-to-front manner. The algorithm consists of following basic steps:

1. Decompose the volume into a tetrahedral mesh. Density values are stored at each vertex. The scalar function is assumed to be a linear combination of the vertex values.
2. Depth sort the tetrahedra.
3. Classify tetrahedra and decomposite into triangles according to the projected profile. The 4 different cases are shown in figure 5.1.
4. Determine color and opacity values at the triangle vertices using ray integration at the "thick" vertex.
5. Rasterize the triangles.

The idea behind this algorithm is that explicit ray integration must be done only once per tetrahedron because of the properties of a tetrahedral cell. The rasterization step can completely be left to the graphics hardware. Although the algorithm works on any kind of tetrahedral mesh, our intention was to find a version specialized for BCC grids. For comparison reasons, we implemented an equally optimized CC version. The algorithm differs only in step 1 and 2 between

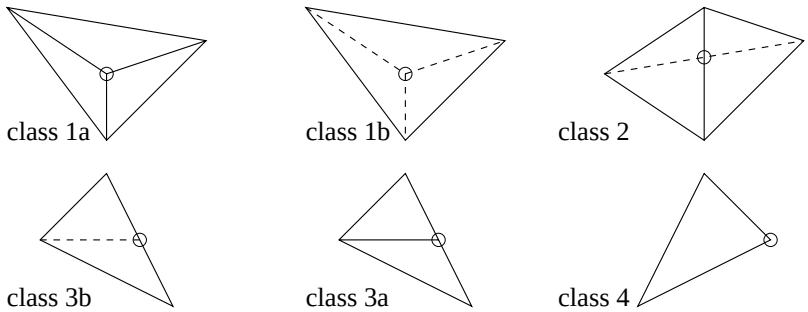


Figure 5.1: In the top row we see the two basic decomposition classes, in the bottom row the degenerate cases. The circle refers to the "thick" vertex that has non zero length. Ray integration is only done along this vertex.

BCC grids and CC grids. We restrict the algorithm to orthogonal projection in order to exploit the regular grid structures for performance optimizations. The algorithm is very well suited to render data given on a BCC grid for two reasons:

1. According to Carr et al [6], tetrahedra defined with the Delauney complex are the natural cells on the BCC grid, like cubic cells are on the CC grid.

2. We have less tetrahedra on a BCC grid than on an equivalent CC grid, hence there is a potential performance gain.

5.2 Tetrahedralization

On the CC grid several possible tetrahedralizations exist, yielding different numbers of tetrahedra per cell. In the original projected tetrahedra algorithm the cell is decomposed into 5 tetrahedra [54] (see figure 5.2). We also used this decomposition in our CC version, because it yields the smallest possible number of tetrahedra. This tetrahedralization has two different rotational states, and we must alternate between the states in a checkerboard manner, in order to avoid ambiguities in the mesh [54] which leads to Mach band effects.

We already know that we have a uniquely defined Delauney tetrahedralization on a BCC grid, where we have six times more tetrahedra than sample points [6]. Given that we have 29.3% less samples in the BCC grid, it can easily be verified that the BCC grid still consists of about 15% less tetrahedra overall.

5.3 Back-to-Front Traversal

It is not necessary to use one of the more general depth-sorting algorithms [70, 38, 10] for CC or BCC grid rendering. Instead, we break up the tetrahedral mesh into cells where depth-sorting can be done implicitly by traversing them in back-to-front manner. The cells are given by cubes

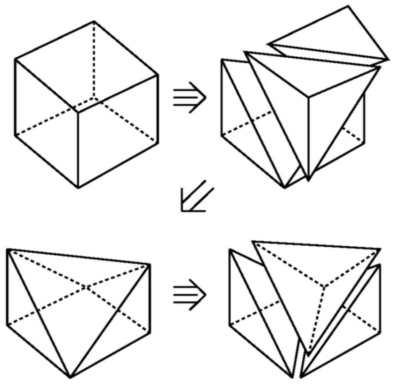


Figure 5.2: Decomposition of a cubic cell into 5 tetrahedra. Image taken from [54].

on CC grids and octahedra on BCC grids. Inside these cells, depth sorting is trivial. Further, there is no need to store the tetrahedra explicitly as a mesh, instead we generate them on the fly during traversal.

Whereas we simply visit the cubic cells one by one on the CC grid, the situation is slightly more complicated on the BCC grid. A sample point shares one octahedron with each of the six adjacent samples from the same grid (either primary or secondary) and 4 samples from the other grid. These octahedra further consist of 4 tetrahedra. The situation is shown for barycentric interpolation in figure 3.3. Each sample point is a vertex in 6 different octahedra and further 24 tetrahedra. Hence we cover all octahedra twice if we traverse only the sample points from the primary (secondary) grid and generate all possible octahedra with the current sample. Thus we process just the three octahedra sharing the currently traversed sample and the adjacent sample in positive (negative) *x*, *y* and *z* direction. Now all octahedra and at the same time all tetrahedra are processed exactly once. A 2D visualization of a traversal step is shown in figure 5.3. Correct BCC rendering is sketched by the following pseudo-code fragment:

```
traverse all samples of primary (secondary) grid front-to-back {
   for each sample do {
      compute octahedra with adjacent samples in
      positive (negative) x,y,z direction
      depth sort octahedra
      split up the octahedra into tetrahedra
      depth sort tetrahedra
      render tehahedra
  }
}
```

The back-to-front traversal is done with nested loops. Following Orchard et al. [44], we call the axis that is most collinear to the view vector the "slow" axis, because it corresponds to the outmost loop. Similar, we have a "medium" axis and a "fast" axis that corresponds to the inner

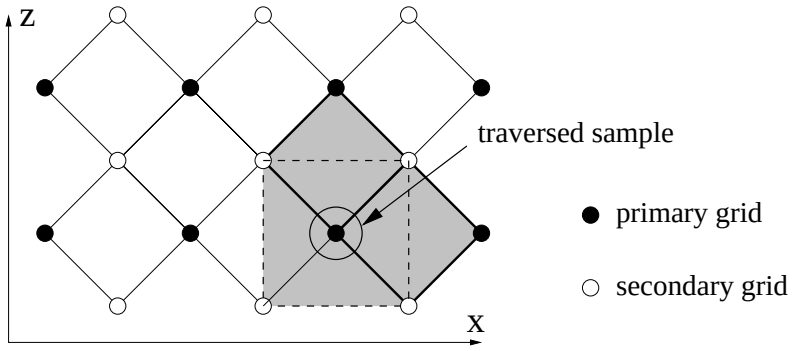


Figure 5.3: Traversal step of the projected tetrahedra algorithm on the BCC grid shown in 2D. In each step three octahedra (shown in grey, the one in the *y* axis outlined with dotted lines) are created with adjacent samples in the directions of the positive *x*, *y*, and *z* axes, then tetrahedralized.

loop. The algorithm yields a the correct depth order for orthogonal projection. For perspective projection, one can verify that our method introduces minor errors because of the interleaved octahedra.

5.4 Speeding up the Basic Algorithm

We describe some simple, but effective optimizations in our system in order to accelerate the Projected Tetrahedra algorithms:

- Wittenbrink [72] suggested to use triangle fans instead of single triangles as OpenGL primitives. This approach effectively reduces rasterization complexity, because it avoids unnecessary multiple renderings of the same vertices.
- The restriction to orthogonal projection, together with a regular grid structure, allows us to exploit coherency among the cells. Except for a translation, there are only 10 different types of tetrahedra in the CC grid (five tetrahedra per cell for two rotational states) and 12 in the BCC grid. Hence depth order, classification and the interpolation factors used for thick vertex computation can be preprocessed and reused during traversal.
- For the triangle decomposition we must apply a view transformation and it's inverse on the tetrahedra vertices. A vertex (i.e., a sample point) is shared by up to 10 tetrahedra on the CC and up to 24 tetrahedra on the BCC grid. To avoid a huge amount of unnecessary computations, we have to assure that an already calculated view transformation is reused when revisiting a sample point as vertex of another tetrahedron. For this reason we use a binary flag for each sample point that indicates whether the sample point has been already visited and transformed. We must keep in mind that all flags are switched after one traversal, and likewise the meaning of the flag values are reversed in the next traversal.

5.5 A 3D Adjacency Structure

To improve the efficency of volume data traversal, we concentrated on modifying the 3D adjacency structure by Orchard et al. [44] for splatting on rectangular grids. It allows fast traversal of non zero voxels in depth order with minimal overhead, skipping all transparent voxels. According to the authors, it performs better than some comparable acceleration data structures like the octree approach from Laur et al. [31] on realistic datasets.

The visible voxels (i.e., opacity exceeds a certain threshold) are organized in linked lists, where together with each voxel six pointers to the adjacent voxels in all directions are stored. Two voxels are called adjacent if and only if they are visible and there is no other visible voxel between them. In addition to voxels representing an actual data sample, the structure is encapsulated by a box of virtual voxels. There are different types of virtual voxels, referred as box face voxels, box edge voxels and box corner voxels. The introduction of virtual voxels allows us to skip a number of empty voxel scan lines or even voxel slices. A scanline of visible voxels is capped by a box face voxel on both ends, scanlines of box face voxels are again encapsulated by two box edge voxels, and 8 box corner voxels are placed in the volume corners. It can be stated that a box face voxel represents a non transparent voxel scanline, a box edge voxel represents a volume slice and a box corner voxel represents the whole volume.

To build up the structure first of all the 8 box corner voxels are installed (because there is at least one visible voxel, they must always exist). The rest of the structure is created in a bottom-up approach, virtual voxels are allocated only on demand, i.e., if the represented area contains a visible voxel. If a new voxel is inserted into the structure, we recursively insert new virtual voxels (box face, box edge) if they do not exist yet (see figure 5.4). Traversal of the data structure begins with the corner farthest away and proceeds like the following:

1. Go through the list of edge voxels back-to-front (the outmost loop of the slow axis).
2. For each edge voxel, traverse the box face list (the middle loop of the medium axis).
3. For each edge face, traverse the visible voxel list (the innermost loop of the fast axis).

5.6 The Adjacency Structure on a Tetrahedral Mesh

The adjacency structure was originally proposed for a rectangular grid structure, but it can be adapted to the tetrahedral mesh defined by a BCC grid as well, because we already know from section 5.3 that we operate on the rectangular primary (secondary) grid during traversal.

The place of a voxel as lowest element in the hierarchy is taken by the geometric complex we are processing in a traversal step. This is a cubic cell in the CC grid and three octahedral cells perpendicular to each other in the BCC grid. Let us denote such a complex as "traversal complex". Likewise, we refer to a virtual voxel as "virtual complex". Such a traversal complex is further decomposed into tetrahedra, which are the new lowest elements in the hierarchy. The adjacency structure is shown in figure 5.5.

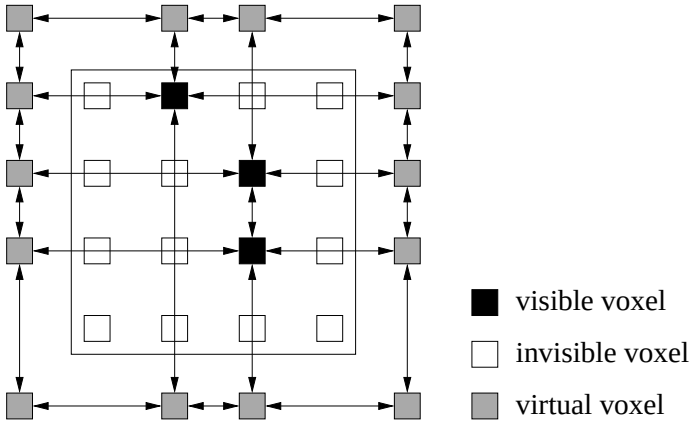


Figure 5.4: A 2D image of the adjacency structure (image redrawn from [44]).

Following the implicit logic of the structure, we define a traversal complex to be visible if at least one of it's tetrahedra is visible. As an underlying rectangular grid structure is not given inside a traversal complex, we must explicitly test each tetrahedron for zero opacity in order to be able to discard transparent tetrahedra. But this is not much of a drawback, considering that we can skip large volume regions without any opacity testing.

Zero opacity testing inside a traversal complex is most efficiently done by storing an additional checksum per complex. As we already know, using orthogonal projection enables us to preprocess and store the tetrahedra information as entry in a table, ordered by tetrahedral depth. A unique ID number is given to each tetrahedron in the table, which is a power of two. The checksum is now calculated as the sum of the ID numbers of all tetrahedra identified as non transparent during creation of the structure. During traversal the checksum is masked with the ID of the actual tetrahedron. If the result is zero, the tetrahedron is transparent and discarded, else rendered.

5.7 Correct Transparency Calculation

The advantage of the original projected algorithm is the great simplicity. It works on almost all available consumer graphics boards. But it is prone to artifacts, most visible as Mach band effects for view directions nearly parallel to a coordinate axis. This is caused by errors introduced by linearly interpolating the color and transparency between the vertices, which should rather be an exponential variation. Instead it can be verified that, assuming a linear transfer function, the extinction factor τ varies linearly inside a tetrahedron. Stein et al. [56] (also in [37]) compute the correct transparency $1-\exp(-\tau l)$ without giving up hardware-acceleration by storing the exponential transparency values in a 2D textures. Then τ and the segment length l are used as texture coordinates. Color values still vary linearly inside a tetrahedron. In order to achieve accurate color and opacity for arbitrary transfer functions, it is inevitable to use pre-integration.

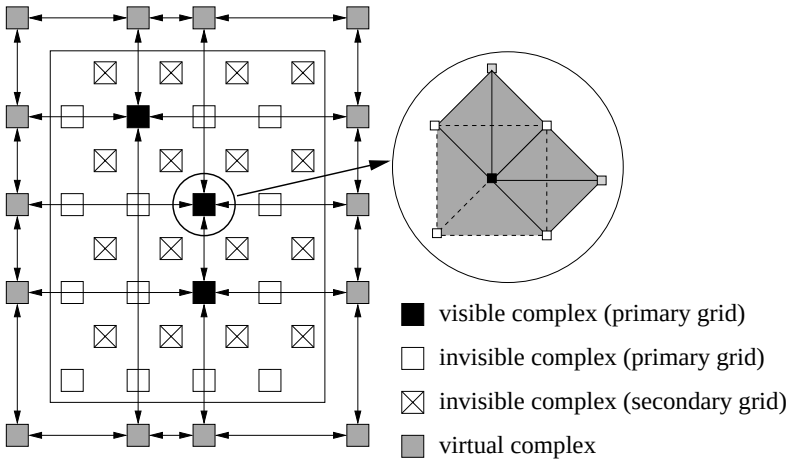


Figure 5.5: 2D visualization of the adjacency structure on a tetrahedral mesh defined by a BCC grid. A traversal complex consists of three octahedra that are split up into four tetrahedra each. Compare this to the original structure in figure 5.4.

5.8 Pre-Integration

Pre-integration is another approach to further increase rendering accuracy without sacrificing hardware-acceleration. Because of high frequencies in the transfer functions, generally a sample frequency above the Nyquist rate of the scalar field must be used. In this approach the integration of the scalar field is done separately from the integration of the transfer function, which is precomputed and stored in a lookup table. As a consequence, accurate renderings can be done with the projected tetrahedra algorithm regardless of high frequencies in the transfer function. The lookup table stores pre-integrated color and opacity for each combination of front and back face scalar values s_f and s_b of a viewing ray entering and exiting a tetrahedron, and the segment length l (see figure 5.6). In order to exploit hardware-acceleration, the table is downloaded to a 3D texture. Because we use orthogonal projection, all parameters have linear variation and it is correct to use s_b, s_f and l as texture coordinates. We introduce minor errors using perspective projection, where we have a non linear variation of segment length l. Following Röttger et al. [51], we will now show the compositing formula with respect to pre-integration. In this formula $\mathrm{RGB}_{\mathrm{t3D}}$ (i.e., the pre-integrated color entry of the 3D texture) and $1-\alpha_{\mathrm{t3D}}$ (i.e., the pre-integrated opacity entry of the 3D texture) denote the part of the equation which is calculated in a preprocessing step. An existing color I is updated to a new color I' according to following equation:

$$I' = \underbrace{\int_0^l \exp\left(-\int_0^t \tau(s_l(u))\mathrm{d}u\right) c(s_l(t))\tau(s_l(t))\mathrm{d}t}_{\mathrm{RGB}_{\mathrm{t3D}}} + \underbrace{\exp\left(-\int_0^l \tau(s_l(t))\mathrm{d}t\right)}_{1-\alpha_{\mathrm{t3D}}} I \qquad (5.1)$$

where $s_l(x)$ is linearly interpolated between s_f to s_b:

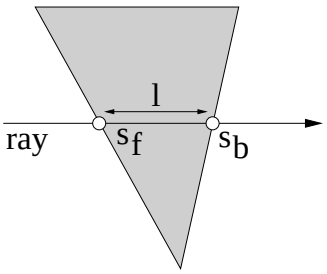


Figure 5.6: A ray piercing through a tetrahedron. The parameters s_f, s_b and l are used for pre-integration. Image redrawn from [51].

$$s_l(x) = s_f + \frac{x}{l}(s_b - s_f) \qquad \text{for } 0 \leq x \leq l$$

We have to consider that we always calculate associated colors with pre-integration, (i.e., colors are already multiplied with the opacity) and set the alpha blend function accordingly in order to compute correct compositing. The maximal segment length l of a tetrahedron given by our tetrahedralizations of CC and BCC grids is $\sqrt{3}$ on CC grids and $\sqrt{2}$ on BCC grids.

5.9 Shading Issues

Directional shading is a natural way to give important clues about the shape of objects and the spatial relationships inside a volume. Shading can be easily inserted into the original projected tetrahedra algorithm by Shirley et al. [54] by storing the gradient data in the normals' portion of the tetrahedra vertices and let the hardware compute standard Gouraud shading.

As stated in section 5.8, pre-integration is a very powerful technique and produces accurate renderings. Hence we would like to use shading in combination with pre-integration. Unfortunately we are restricted to three different parameters because we have only three texture coordinates, thus additional shading information cannot simply be inserted into the pre-integration process.

We developed three different shading schemes for pre-integrated volumes. In two of the three methods, we apply shading separate from the color and opacity pre-integration. The drawback of this approach is that the accuracy of the shading calculation is not equivalent to the pre-integrated color and opacity accuracy.

Pre-integration and Gouraud Shading

In this approach, we set the vertex color to white. Then we let the hardware compute the standard Gouraud shading, resulting in a shaded grey value for each fragment of the tetrahedra. The grey value is modulated with the texel resulting from the pre-integration lookup. As the highlight would be destroyed by this modulation, we must ensure that it is added afterwards, either by applying the appropriate OpenGL extension or in the pixel shaders.

Pre-integration and Phong Shading

In the second approach we store the gradients in the color portion of the vertices. This results in an interpolation of the gradients instead of the colors. In the pixel shaders we can operate with the interpolated gradient vector and the actual pre-integrated texel output. These parameters are used as input for the Phong shading equation.

For correct shading we must normalize the gradient vectors in the pixel shaders. This can be done by either employing a cube map or by normalizing the gradient directly in the pixel shaders, which is considered to be faster. If we are using NVIDIA's register combiners, there is no square root operation available, forcing us to employ an approximation formula. Fortunately it was proven that the formula is reasonable accurate for the conditions given in a normal mesh. In the fragment- and vertex shaders of newer graphics hardware the gradients can be calculated using the analytic normalization formula.

Pre-Integrated Gouraud Shading

An alternative approach is the implementation of real pre-integrated Gouraud shading, sacrificing a correct opacity calculation for the diffuse and specular term. For Gouraud shading, we calculate the lighting on each vertex and linearly interpolate the shaded colors in between. The per vertex lighting is calculated with the approximate Phong equation

$$I = \underbrace{k_a}_{\text{ambient}} + \underbrace{k_d(\vec{n}\cdot\vec{l})}_{\text{diffuse}} + \underbrace{k_s(\vec{n}\cdot\vec{h})^n}_{\text{specular}} \tag{5.2}$$

where $\vec{n}$ denotes the vertex normal, $\vec{l}$ the light vector, $\vec{h}$ the half way vector (calculated between view vector $\vec{v}$ and light vector $\vec{l}$) and n the shininess coefficient. $(\vec{n}\cdot\vec{h})$ serves as approximation for $(\vec{v}\cdot\vec{r})$, the dot product between view vector $\vec{v}$ and reflection vector $\vec{r}$. The idea is to split up the pre-integration for the ambient, diffuse and specular term [22]. A separate lookup table is calculated for the pre-integration of the transfer function (this table is used for the direction-independent ambient term), for diffuse lighting and for specular lighting. The tables are then downloaded to three different 3D textures. The parameters used for pre-integration of the three terms in the Gouraud shading equation differ as follows:

Ambient term: This term does not contain directional information, hence we do standard pre-integration and take the usual values s_f, s_b and l as parameters.

Diffuse term: For this term we take $(\vec{n}\cdot\vec{l})_f$, $(\vec{n}\cdot\vec{l})_b$, the dot products on front and back face and the segment length l as parameters.

Specular term: We use the parameters $(\vec{n}\cdot\vec{h})_f^n$ on the front and $(\vec{n}\cdot\vec{h})_b^n$ on the back face, and again the segment length l.

The situation is also shown in figure 5.7. In the lookup tables we store the pre-integrated output color and opacity values for every combination of these values, where $(\vec{n}\cdot\vec{l})$ and $(\vec{n}\cdot\vec{h})^n$

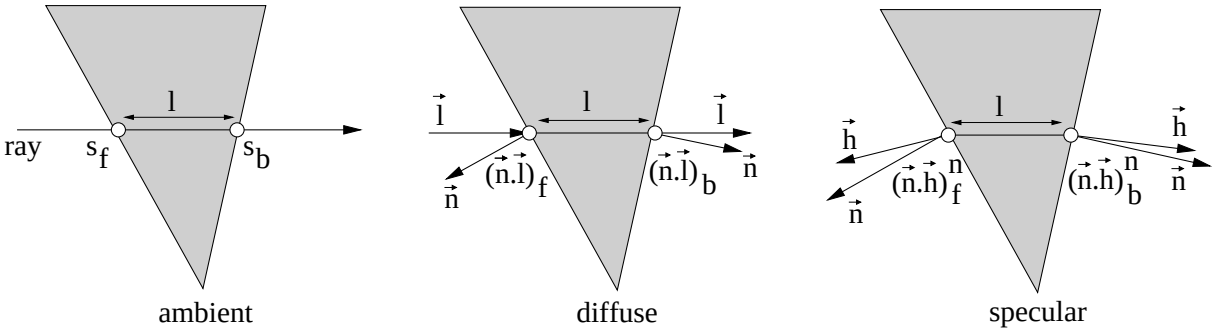


Figure 5.7: The three parameters for the ambient, diffuse and specular term of Gouraud shaded pre-integration.

are in the interval $[0..1]$, and l lies in $[0..l_{max}]$. These parameters are linearly interpolated over the polygon, thus correct Gouraud shading is calculated. At last we must sum up the terms in the pixel shaders. As we compute diffuse and specular term independently from the actual transfer function, we have to set k_d, k_s, extinction factor τ and shininess n to a constant value manually.

Chapter 6

Implementation

In this chapter, we will first give an overview of the framework that we used for two of the three major parts of this work (i.e., the reconstruction schemes using a raycasting system and the projected tetrahedra algorithm), then we discuss implementation issues of our approaches. We give an overview of the rendering systems, present important features, and provide some implementation details that are of particular interest.

6.1 The vuVolume Framework

The raycasting system and the projected tetrahedra algorithm for rendering data given on CC and BCC grids were implemented as part of the vuVolume framework [15]. The framework is implemented in C++ and exploits a deep tree structure of inheritance for providing flexibility in adding new rendering methods. The tree structure is represented in the directory tree, i.e., the descendants of a class are written into subdirectories. The root of the tree is the abstract class `Volume`. The next tree levels further specify the volume, e.g., the branch `Regular` refers to a CC volume and the branch `BCC` to a volume given on a BCC grid. New volume rendering methods are added as a leaf of the tree. The different abstraction levels are listed in table 6.1. For example, we added the the projected tetrahedra algorithm as a leaf `ProjTetr` to the branch `Volume` - `BCC` - `Unimodal` - `3d` - `1B` - `Intensity`. The algorithms are strictly separated

Level	Description
Volume	The base class
Geometry	The type of the sampling grid
Modality	single or multiple datasets over geometry
Dimension	1d, 2d or 3d
Data implementation	Refers to number of bytes per sample
Transfer function	Intensity (1d), gradient (2d), curvature (3d)
Algorithm	The implementation of the rendering algorithm

Table 6.1: Different levels of abstraction in the vuVolume tree.

from their user interfaces, which are realized with the platform independent wxWindows toolkit.

6.2 Practical Reconstruction Schemes

The emphasis in the implementation of the reconstruction schemes was on providing an environment that is equally fair for all interpolators regardless of the used grid (i.e., a CC or BCC grid). The reconstruction schemes were integrated into a raycasting system. As absolute rendering speed is not the main issue, we exploit full float precision for optimal rendering accuracy in this system. From all possible optimizations we only implemented early ray termination in combination with front-to-back compositing. The system supports post classification and two types of transfer functions, a one dimensional function of intensity only, and a two dimensional function of intensity and gradient magnitude (i.e., gradient weighted opacity following Levoy et al. [32], also see figure 6.1). We provide an interface that makes it very easy to extend the system and add new interpolation schemes. To be more specific, all reconstruction schemes are inherited from the abstract superclass `interpolator`:

```
class interpolator
{
 public:
    interpolator();
    interpolator(volData *voldata);
    virtual ~interpolator();
    virtual void interp(float *rgba, float x,
                        float y, float z) = 0;
    virtual float interpData(float *norm, float x,
                             float y, float z) = 0;
    void setVolData(volData *voldata) {
        m_VolData = voldata;
    }
  protected:
    volData *m_VolData;
};
```

The `volData` object encapsulates the access to the volume data and volume gradients. To create a new interpolation scheme, only the two methods `interp` and `interpData` must be implemented. The first method `interp` interpolates color and opacity in the array `rgba` for a spatial position given by float values `x,y,z`. Hence it is used for pre-classification. The method `interpData` serves the purpose of post-classification, as it returns an interpolated data value, while the interpolated normal and it's length is returned in the the array `norm`.

For the reconstruction methods bilinear, bilinear plus spatial interpolation and sheared trilinear which depend on the current principal view axis, there is an additional level of inheritage. A child class is derived to implement each of the three principal view axes, in order to avoid an

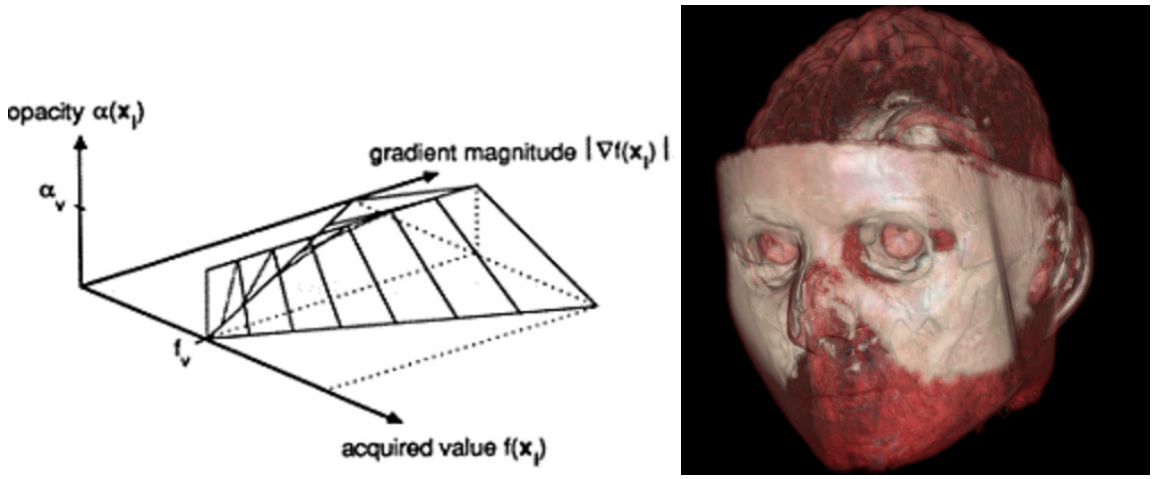


Figure 6.1: Gradient weighted opacity function (left image, taken from [32]) used in our raycasting system to emphasize interesting parts of the surfaces (right image).

additional `if` statement and thus preserve better readability of the code. The implementation scheme of our interpolators is sketched in figure 6.3.1 as a tree view. The class `bilinear`, for instance, is the superclass for classes `bilinearXY` (where `XY` refers to the axes that span up the resampling planes), `bilinearXZ` and `bilinearYZ`. For trilinear interpolation on BCC grids, there exists an abstract class `bccTrilinear` and child classes `bccTrilinear6S` and `bccTrilinear2S`, which refer to the type of intermediate cubic cell construction with either 2 or 6 samples, respectively.

6.3 Texture-Based Volume Rendering

We integrated the BCC grid rendering extensions into the hardware-accelerated volume rendering framework proposed by Berger [3] and written by Hadwiger and Berger for the VRVis Research Center [63]. First we will give an overview of the framework, then present our fragment program for 3D texture-based BCC grid rendering. Afterwards we describe how the fragment program can be inserted into the framework in order to provide BCC grid support for almost all rendering modes.

6.3.1 A Flexible Hardware-Rendering Framework

In addition to texture-based volume rendering, hardware-accelerated ray casting was recently included into the system. There is also support for higher-order filtering proposed by Hadwiger et al. [24]. Rendering algorithms and GUI are both implemented in C++, using OpenGL and extensions as graphics library. The framework supports most of the currently available consumer graphic chips from the vendors NVIDIA and ATI. On startup, the underlying graphics hardware and the supported extensions are detected.

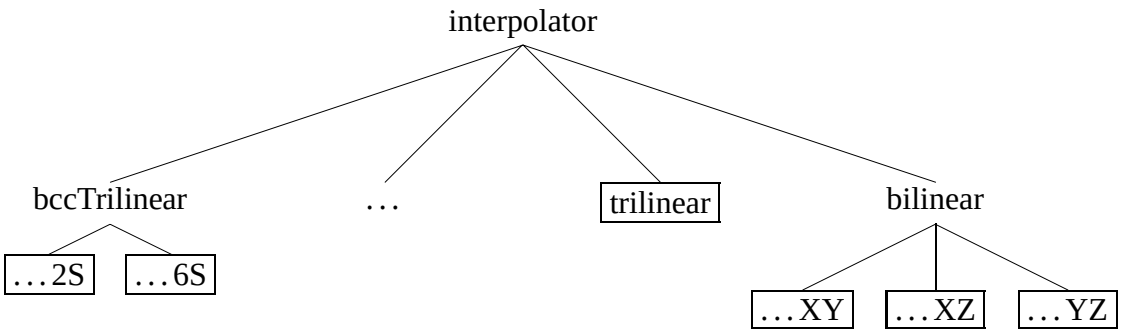


Figure 6.2: All reconstruction schemes are derived from the abstract class interpolator. View dependent interpolators are declared abstract and have non-abstract child classes that refer to the main view planes.

It is one major design goal to encapsulate the calls to the graphics hardware into a very general API, in a way that the actual rendering algorithms are kept as independent as possible from the underlying graphics processor. This is not an easy task, as different types of hardware must be handled very differently in order to provide the same rendering results. Not all of the rendering modes can be used on every graphics board. For example, pre-classification is not available on ATI cards because the necessary extension GL_EXT_shared_texture_palette is not supported. With the powerful fragment- and vertex-program functionality of new generation hardware, the interface to the hardware is becoming more uniform for graphics chips from both vendors. This is exploited by the framework for an increasingly flexible handling of the different rendering modes.

Several rendering modes are included into the system. 2D texture-based rendering and 3D texture-based volume rendering are both supported. Multi-texture blending can be turned on and off on demand for any 2D texture rendering technique. 2D texture-based rendering and 3D texture-based rendering can be combined with pre- and post-classification, as well as pre-integrated slab rendering. Additional rendering modes include iso-surface rendering (using the alpha test), maximum intensity projection (MIP) and nonphotorealistic (NPR) rendering modes like tone shading, for example.

6.3.2 Fragment Program for 3D Texture-Based Rendering on a BCC grid

Any available rendering mode running on the current hardware can also be used for BCC grid rendering without additional requirements. The only exception is 3D texture-based rendering on a BCC grid, which requires a special fragment program. The fragment program that provides correct 3D BCC grid rendering for the principle view axis z is given by:

```
TEMP    tmp,    offs;
MAD     offs,   program.env[ 6 ], tex, {0.25,0.25,0.25,0};
FRC     tmp,    offs;
ADD     offs,   {1.0,1.0,1.0,0.0}, -tmp;
MIN     offs,   tmp, offs;
```

```
MOV     tmp,     offs.zzww;
MAD     tex.xyz, tmp, program.env[ 7 ], tex;
```

Let us shortly analyze the program with respect to the formulas from chapter 4.5:

- The environment vector `program.env[6]` is loaded with the spatial extent of the volume in *x*, *y* and *z* axis (using cell spacing $T = 1$), `program.env[7]` with the reciprocals of the these values. In the first line, temporary variables `offs` and `tmp` are created.
- In line 2, we apply a multiplication and addition (`MAD`), which is a mapping from texture to index space similar to equation 4.2. Adding a vector $(0.25, 0.25, 0.25, 0)$ in line 2 and applying a positive offset in the last line is equivalent to subtracting the same vector like in equation 4.2 and adding a negative offset.
- The `FRC` operation in line 3 does a truncation of variable `offs`, and together with the `ADD` and `MIN` operations they compute equation 3.4, storing the result back into variable `offs`.
- The `MOV` and `MAD` operations in the last two lines update the texture coordinates according to equation 4.3.

If the principal view direction is the *x* axis, only line 7 changes to:

```
MOV     tmp,     offs.wxxw;
```

Likewise, for principal view direction *y*:

```
MOV     tmp,     offs.ywyw;
```

6.3.3 Rendering Support for 3D Texture-Based Rendering on a BCC grid

The authors of the framework implemented a flexible system of fragment program strings that handles most rendering modes in a single uniform interface. Each fragment program is given an unique ID, which is determined by the features that are turned on (off). For example, the fragment program responsible for post shaded slices is given a different ID whether multi-texture blending is turned on or off. If turned on, just the piece of fragment program string that enables the spatial interpolation is inserted into the proper position in the main fragment program string. A table is used to keep track of the currently active fragment programs, which are created and bound on demand. In OpenGL, fragment programs must be bound to the hardware before usage similar to texture objects.

This system makes it possible for us to implement BCC grid support quite elegantly. In case that 3D texture mapping and on the same time BCC grid rendering is enabled, all that has to be done is to insert the fragment program string responsible for 3D BCC grid rendering into the main fragment program string, To be more specific, one out of three slightly different fragment program strings is constructed and used, i.e. the one that corresponds to the current principal view axis.

For any of the main rendering modes (for example post-classified shaded slices), a function exists that creates the corresponding fragment program string `ps`. If the current rendering mode uses 3D texture-based rendering on BCC grids, a function is invoked in it's body which inserts the fragment program from section 6.3.2 into `ps`. Given that the principal view direction is stored in `axis`, this function is written as:

```
void PSFragmentProgramBCCOffs(GLProgramString &ps, int axis)
{
    ps+=" TEMP tmp,  offs;"
    ps+=" MAD  offs  program.env[ 6 ],tex,{0.25,0.25,0.25,0};";
    ps+=" FRC  tmp,  offs;";
    ps+=" ADD  offs, {1.0,1.0,1.0,0.0},-tmp;";
    ps+=" MIN  offs, tmp, offs;";

    switch ( axis ){
    case 0:
    ps+=" MOV  tmp, offs.wxxw;";
    break;
    case 1:
    ps+=" MOV  tmp, offs.ywyw;";
    break;
    case 2:
    ps+=" MOV  tmp, offs.zzww;";
    brak;
    }
    ps+=" MAD  tex.xyz, tmp, program.env[ 7 ], tex;";
}
```

6.4 Projected Tetrahedra Algorithm

The projected tetrahedra algorithm is also implemented as part of vuVolume. As graphics library OpenGL and extensions are used. In this section, we will sketch the general concept and also list some code fragments that provide important details of the implementation.

6.4.1 Decomposition

The decomposition and traversal step is the only difference in the implementations of the CC grid and BCC grid versions of the Projected Tetrahedra algorithm, encapsulated in the very general abstract superclass `decompositor`. The equally abstract child class `structDecompositor` operates on structured grids (which includes the CC grid and the BCC grid). It provides an interface for the specialized classes `ccDecompositor` and `bccDecompositor`. All classes inherited from `decompositor` implement the two abstract methods

```
virtual void init();
virtual bool next(tetrahedron &tetr);
```

where function `init()` handles the initialization process, i.e., it encapsulates the creation of the data structure storing the tetrahedral mesh and does some pre-processing (regarding the tetrahedral ordering, for example). Function `next()` returns the next tetrahedron in view order and sets the pointer to the next element. If there is no more element, it returns false, else true. Class `structDecompositor` implements both the `init()` and the `next()` function. It is highly specialized for orthogonal projection, in order to exploit coherence and repeating patterns among the tetrahedra. The central method `init()` has the following principal structure:

```
void structDecompositor::init()
{
    orderTetr();
    ...
    if(m_IsUpdTetr)
    {
        ...
        buildTetrStruct();
        ...
    }
    preclassify();
    ...
}
```

In this code fragment, we can see that first function `orderTetr()` is invoked, which determines the depth order of the tetrahedra inside a traversal complex according to the current view direction. The depth ordering information is stored in a table and used during traversal. For storing the tetrahedra, method `buildTetrStruct()` creates the adjacency structure from chapter 5.6. After a transfer function change this adjacency structure is rebuild from scratch, although this structure would be flexible enough to be modified by adding and removing elements afterwards. In function `preclassify()`, we classify the tetrahedra of a traversal complex according to the projected outline and compute the interpolation factors used for decomposition into triangles. Again this information is stored in a table for usage during traversal. The functions `orderTetr()` and `buildTetrStruct()` are abstract, because they differ in their implementation depending on the type of grid.

6.4.2 3D Adjacency Data Structure

The functionality of the 3D data structure is encapsulated in the class `extData`. It builds up a linked list structure of traversal complexes. For indices (`x`, `y`, `z`), a new traversal complex is inserted into the structure using the method

```
void insert(byte info, int t_id, int x, int y, int z, int di);
```

where the byte value `info` refers to the rotational state of the tetrahedralization on the CC grid (refer to chapter 5.2), `di` to the actual index into the volume data array, and `t_id` to the unique index of a tetrahedron inside a traversal complex. After insertion, the virtual complexes are updated recursively. Class `tComplex` implements a traversal complex:

```
class tComplex
{
 public:
    tComplex();
    tComplex(int info);

    int m_Next[6];
    byte m_Info;
    int m_Checksum;
    int m_Index;
};
```

The attributes have the following purposes:

- An array of integers `m_Next` stores 6 pointers to the adjacent elements in all directions.
- `m_Info` is a container for some type of additional information, e.g., it indicates the traversal complex type (box face, box edge, or box corner). If the traversal complex is non virtual, it refers to the rotational state of the decomposition on the CC grid (see chapter 5.2).
- The integer `m_Checksum` stores the checksum responsible for tetrahedron visibility inside a traversal complex (refer to chapter 5.6).
- `m_Index` represents the original data index of one of the data samples that span the complex. This attribute could actually be derived from other values, but it is stored to provide fast access to the data indices of the tetrahedra vertices.

6.4.3 Data Traversal

We traverse through the adjacency structure by using three nested loops:

```
tetrahedron tetr;
vertex *v[7];
extData *ext_d = m_Decomp.getExtData();

while(ext_d->nextSlice())
    while(ext_d->nextScan())
        while(ext_d->nextTComplex())
        {
            m_Decomp.initTComplex();
```

```
            while(m_Decomp.next(tetr))
            {
                vertex_num = tetr.triangulate(v,invview,
                                    m_Decomp.getDecInfo());
                ...
            }
        }
```

In the outmost loop we traverse the box edges (function `nextSlice()`), then the box faces (function `nextScan()`) and at last the traversal complexes (function `nextTComplex()`). In the innermost loop, function `next()` returns all visible tetrahedra inside a traversal complex for rendering, where the tetrahedra are instances of class `tetrahedron`. The decompositor `m_Decomp` is either an instance of `bccDecompositor` or `ccDecompositor`.

6.4.4 Triangle Decomposition

The method `triangulate` of class `tetrahedron` is responsible for the decomposition into triangles. It returns an array of triangle vertices `v`, which are ordered so that the first 2 vertices are start and end point of the thick vertex and can otherwise by directly inserted into an OpenGL triangle fan.

```
int tetrahedron::triangulate(vertex **v, matrix3x3 &invm,
                             DECOMP_INFO &info)
{
    v1_index = m_ClassTbl[info.tblIdx].m_VInfo[0];
    v2_index = m_ClassTbl[info.tblIdx].m_VInfo[1];
    v3_index = m_ClassTbl[info.tblIdx].m_VInfo[2];
    v4_index = m_ClassTbl[info.tblIdx].m_VInfo[3];

    switch(m_ClassTbl[info.tblIdx].m_Class)
    {
    case 1:
        return case1Dec(v, invm, 0, info.slope,info.slope2);
    case 2:
        return case1Dec(v, invm, 1, info.slope,info.slope2);
    case 3:
        return case2Dec(v, invm, info.slope,info.slope2);
    case 4:
        return case3Dec(v, invm, 0, info.slope);
    case 5:
        return case3Dec(v, invm, 1, info.slope);
    case 6:
        return case4Dec(v);
```

```
    default:
        return 0;
    }
}
```

Dependent of the current tetrahedron classification, the corresponding decomposition function is invoked. A pointer into the decomposition class table derived from the tetrahedra face normals is given by `info.tblIdx`. The interpolation factors (`info.slope`, `info.slope2`), tetrahedron thickness at the thick vertex and `info.tblIdx` are precomputed and stored in a table as variables of the type `DECOMP_INFO`. The variable `invm` is the inverse view matrix used to transform the tetrahedron vertices back to world space.

6.4.5 Avoiding Redundant Calculations

In order to avoid a massive amount of redundant calculations, the following structure is allocated to each sample point that belongs to a traversal complex:

```
struct VTX_INFO
{
    bool  visited;
    float pos[3];
    float transfPos[3];
};
```

This structure stores the original position in world space `pos`, the transformed position in view space `transfPos` and a boolean flag `visited`, which is switched after the first view transformation has been applied to this vertex. This structure requires 25 byte. As up to 24 tetrahedra share a vertex, we avoid up to 23 unnecessary transformations of the same vertex.

6.4.6 Pre-Integration

The OpenGL parameters used for back-to-front compositing in the original projected tetrahedra algorithm are `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA`. Care must be taken that pre-integration always produces associated colors, that is, colors premultiplied with the opacity. We do not want to multiple associated colors with the opacity again. Instead we must use `GL_ONE` together with `GL_ONE_MINUS_SRC_ALPHA` during alpha blending for the pre-integration approach.

6.4.7 Shading

We presented three schemes for shading pre-integrated volumes in chapter 5.9. The Phong shading approach (chapter 5.9) and the pre-integrated Gouraud shading (chapter 5.9) need shading calculations in the pixel shaders. For the latter approach, the output texels from the ambient, diffuse and specular pre-integration textures must be combined per pixel. We used NVIDIA's

register combiners (i.e., extension GL_NV_register_combiners [28]) for our implementation. Register combiners are quite inflexible as they have a fixed number of general combiner stages, but have the advantage that they also work for many of the slightly older graphic chips. We also used register combiners to add the highlight after modulating the texture with the shaded grey value in the Gouraud shading approach (chapter 5.9).

6.4.8 Normal Approximation

The main drawback in using register combiners for our approach is the missing square root operation. Fortunately, there exists an approximation formula for normalization of a gradient that is reasonable accurate if certain conditions are given and can be calculated in two general combiner stages. With a Taylor series expansion it can be shown [52] that the normalized vector $\vec{v}_n$ of a vector $\vec{v}$ can be approximated with formula

$$\vec{v}_n \cong \frac{\vec{v}}{2}(3 - \vec{v} \cdot \vec{v}) = \vec{v} + 0.5\vec{v}(1 - \vec{v} \cdot \vec{v}) \tag{6.1}$$

if $\vec{v}$ is derived from the interpolation of unit-length vectors and the angle between all pairs of the original per-vertex vectors is not very big. This method is considered to be faster than the usage of a cube map.

Chapter 7

Results and Comparisons

In this chapter we will first make some general comments about the used comparison strategy and the used datasets. Then we will present the experimental results we achieved from testing the different approaches on BCC grids and CC grids regarding rendering quality and performance. We first show the results we achieved from testing the practical reconstruction schemes, then the results from texture-based rendering and at the end the results from the projected tetrahedra algorithm.

7.1 Strategy for Comparing the Rendering Quality

Let us first make some principal comments about the way how we tested and compared the rendering quality. Measuring the quality in a quantitative manner is not an easy task. Several error metrics exists, but sometimes they do not reflect the human perception. The human visual system is less sensitive to errors like stochastic noise, whereas it is very sensitive to regular patterns. Following the well known comparison study for volume rendering algorithms from Meißner et al. [41], we employ visual quality assessment only to test the different methods on BCC grids and compare them to equivalent schemes on CC grids. This thesis is about the application of BCC grids for practical usage in various volume rendering systems, and for the end user the visual quality is the most important issue.

For all figures showing comparison images of different methods on CC and BCC grids we used the same transfer function and view direction. For raycasting and texture-based volume rendering, we applied the same sample (slice) distance. Except for those methods where this parameter is fixed, because (bilinear) interpolation in the planes is used.

7.2 Datasets

The datasets that we used in our experiments include real world datasets (Tooth, Skull, Uncbrain), simulation results (Hipiph, Fuel), and artificial datasets which are sampled from a mathematical formula (e.g., the Marschner-Lobb and the Cube dataset). Most of the real world and simulation datasets do not allow a fair comparison between CC and BCC grids, because they are resampled

Dataset	Type	Resampled	CC Dimension	BCC Dimension
Cube	Artificial	no	$40 \times 40 \times 40$	$28 \times 28 \times 56$
Fuel	Simulation	yes	$64 \times 64 \times 64$	$45 \times 45 \times 90$
Hipiph	Simulation	yes	$64 \times 64 \times 64$	$45 \times 45 \times 90$
Marschner-Lobb	Artificial	no	-	$49 \times 49 \times 98$
Lobster	Real world	no	$120 \times 120 \times 34$	-
Device	Real world	no	$128 \times 128 \times 64$	$91 \times 91 \times 91$
Melon	Real world	no	$128 \times 128 \times 64$	$91 \times 91 \times 91$
Engine	Real world	yes	-	$129 \times 129 \times 129$
Uncbrain	Real world	yes	-	$129 \times 129 \times 146$
Tooth	Real world	yes	-	$181 \times 181 \times 227$
Skull	Real world	yes	-	$181 \times 181 \times 362$

Table 7.1: Description of all datasets used in our experiments. The "Resampled" column indicates whether the BCC grid volume was resampled from the CC grid volume.

versions of the original CC grid dataset. A possible solution would be the use of artificial datasets that can be easily sampled on both grids. Artifical datasets have the benefit that their regular and predictable structure give a good hint of the artifacts produced by a specific reconstruction method. Unfortunately, most of our artificial datasets like the cube are too simple to be a real test. Usually, the Marschner-Lobb dataset [35] dataset is taken for quality evaluation of reconstruction schemes, because it is a challenge for any interpolator. It contains very high frequencies and is and therefore it is very sensitive to any resampling error.

Unfortunately the Marschner-Lobb is not suited for comparing CC grid schemes to BCC grids schemes. It has a cubic frequency support (as was shown by Neophytou et al. [43]), and we implicitly assumed a spherical frequency support for the BCC theory to make sense. Therefore we use the Marschner-Lobb dataset only to compare BCC grid methods with other BCC grid methods. Equally important are real world datasets that allow a fair comparison. We aquired two datasets of this type, the Device and Melon dataset. These volumes are MR scanned and sampled separately into the BCC grid and CC grid format and therefore should be free of errors that are caused by an additional resampling step. All datasets we used in our experiments are listed in table 7.1.

7.3 Practical Reconstruction Schemes

We used a raycasting system to implement and compare the different reconstruction schemes in terms of quality and speed. Raycasting is the volume rendering algorithm with the least side effects which could negatively influence rendering accuracy. The main focus was to provide an environment for comparison of all reconstruction schemes on both CC grids and BCC grids, and to achieve the best possible rendering accuracy. We employ only post-classification for rendering. It produces much sharper results than pre-classification, which tends to produce blurry and fog like images. By using post-classification, it is easier to detect artifacts and differences between

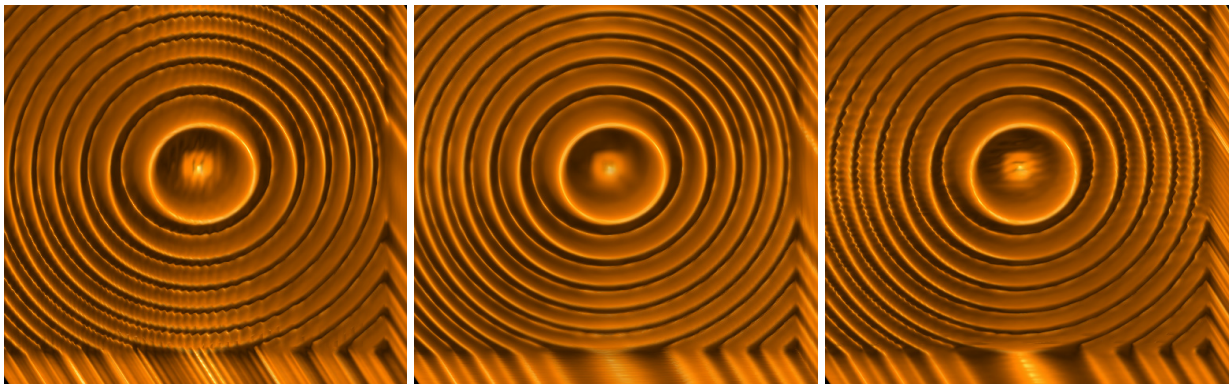


Figure 7.1: Raycasting of the Marschner-Lobb dataset. If the shear planes for the sheared trilinear interpolator are chosen dependent on the viewing direction (middle image), artifacts are removed which can occur otherwise (left and right image).

the reconstruction schemes. As stated before, one and two dimensional transfer functions are supported. We used an Athlon Pentium 2.2 GHZ with 512 MB and a GeForce4 TI graphics chip on a Suse Linux 8.0 platform. Image size is 512×512 pixel.

7.3.1 Rendering Results

The effects of a properly chosen shear plane for the sheared trilinear interpolation are shown in figure 7.1. The sheared cells are visible as saw tooth shaped artifacts in the leftmost and rightmost image, whereas they disappear if the shear parameters are chosen depending on the current view direction (middle image).

Figure 7.2 shows renderings of different reconstruction schemes of the Fuel dataset on BCC grids and CC grids. The bilinear interpolator on the CC grid suffers from heavy under-sampling artifacts, even in combination with a simplified trilinear interpolation (left image in top row). The bilinear interpolation on the BCC grid (right image in top row) reveals some slicing plane artifacts. These artifacts vanish nearly completely when halving the sample distance by employing a simplified trilinear interpolation (left image in middle row). Despite the limited quality of barycentric interpolation (right image in middle row), no major artifacts can be detected in this image. It even seems to be the sharpest image among the BCC grid interpolators. The alternative sheared trilinear interpolation (left image in bottom row) looks slightly blurrier than the barycentric interpolation. Except for a slightly blurrier appearance on BCC grids no major quality differences to trilinear interpolation on a CC grid (right image in bottom row) are visible.

Zooming into the teeth region of the Skull dataset (figure 7.3) shows that the BCC renderings with the bilinear interpolator (middle image) and sheared trilinear interpolator (right image) are smoother that the trilinear interpolation on the CC grid (left image). In this case the smooth curves of a tooth seems to be better approximated by the BCC grid renderings.

Results of the different reconstruction schemes for the Device dataset on the BCC grid are shown in figure 7.4. Central differences 1 was used as gradient estimation scheme in all BCC grid renderings, and central differences was applied for the CC grid renderings. The second and

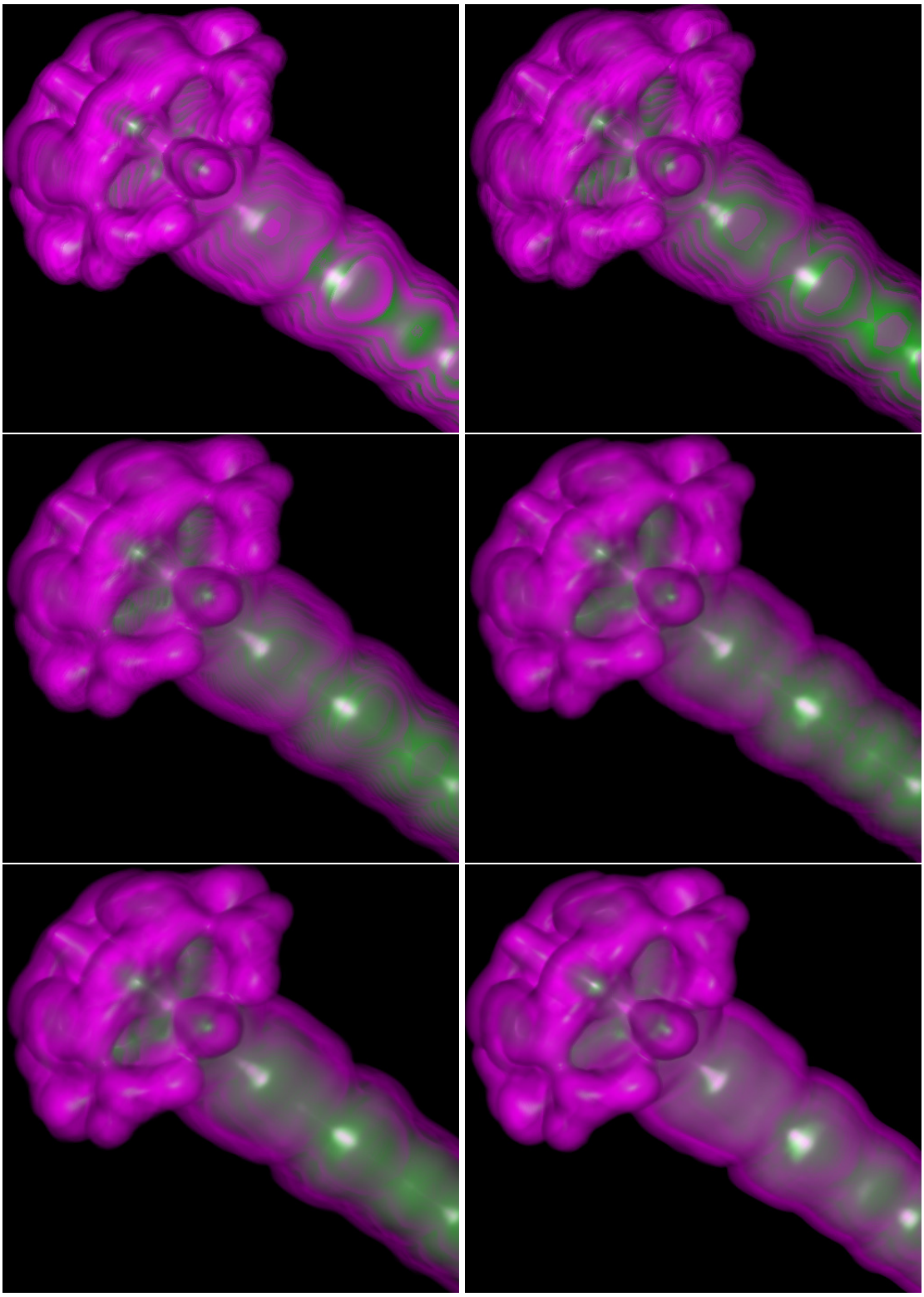


Figure 7.2: Raycasting on the Fuel dataset. Top row: bilinear plus simplified trilinear on the CC grid (left), bilinear on the BCC grid (right). Middle row: bilinear plus simplified trilinear on the BCC grid (left), barycentric (right). Bottom row: alternative sheared trilinear (left), trilinear on the CC grid (right).

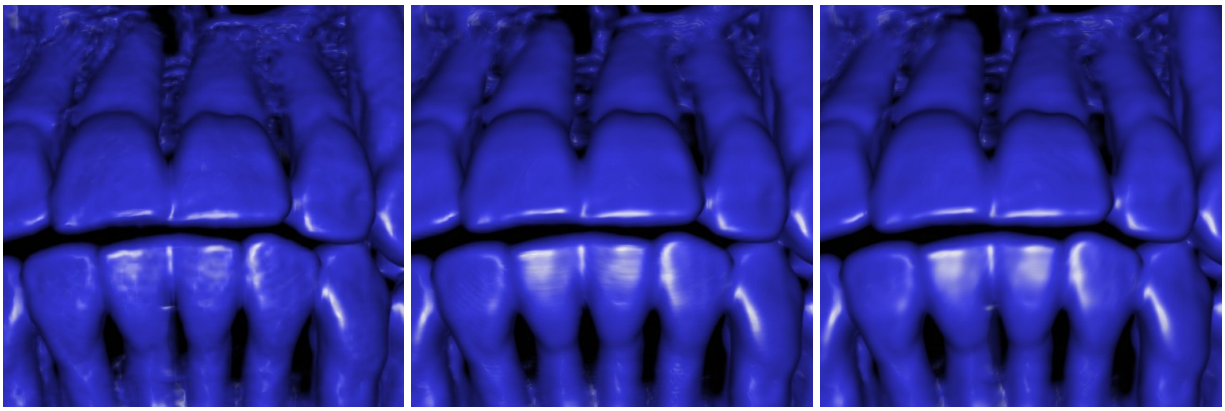


Figure 7.3: Raycasting on the Skull dataset. From left to right: trilinear interpolation on the CC grid, bilinear interpolation on the BCC grid, and sheared trilinear interpolation on the BCC grid.

the last row shows magnifications of a region in the above image. The barycentric interpolation (left image in the top row) produces artifacts that are clearly visible in the magnification image below (left image in the second row). The trilinear interpolation on the BCC grid with 2 samples (middle image in the top row) shows very similar artifacts in the closeup (middle image in the second row). These artifacts are partly, but not fully removed by employing trilinear interpolation on the BCC grid with 6 samples (rightmost image in the top row). The artifacts are fully smoothed out using the bilinear plus spatial interpolation (left image in the third row) and sheared trilinear interpolation (middle image in the third row). This can be seen more clearly in the zoomed regions for the bilinear plus spatial (left image in the bottom row) and sheared trilinear interpolation (middle image in bottom row). The trilinear interpolation on the CC grid (right image in the third row) also reveals slight artifacts which can be observed in the magnification (right image in the bottom row). On the same time much more details of the surface material are visible.

Unfortunately, the Device datasets are sampled slightly differently on CC grids and BCC grids, e.g., visible as noise in the lower right corner of the CC grid dataset that cannot be seen for any settings on the BCC grid. Therefore we must handle the achieved results with some care. The same can be stated for the Melon dataset.

In figure 7.5, we show renderings of the Melon dataset. Central differences 1 was used for gradient reconstruction on the BCC grid, and central differences on the CC grid. Bilinear interpolation on the BCC grid (left image in the top row) washes out some of the details, whereas the image produced with the barycentric interpolation (right image in the top row) preserves many details but looks very rough. The trilinear interpolation on BCC grids with 6 samples (left image in the middle row) appears to be much smoother. This trend is further continued for bilinear plus spatial interpolation (right image in the middle row) and sheared trilinear interpolation (left image in the bottom row). Sheared trilinear interpolation looks quite smooth, but also slightly flat and blurry. The rendering produced with trilinear interpolation on the CC grid (right image in the bottom row) is the most detailed one.

A comparison of the gradient calculation schemes on the BCC grid are shown in figure 7.6.

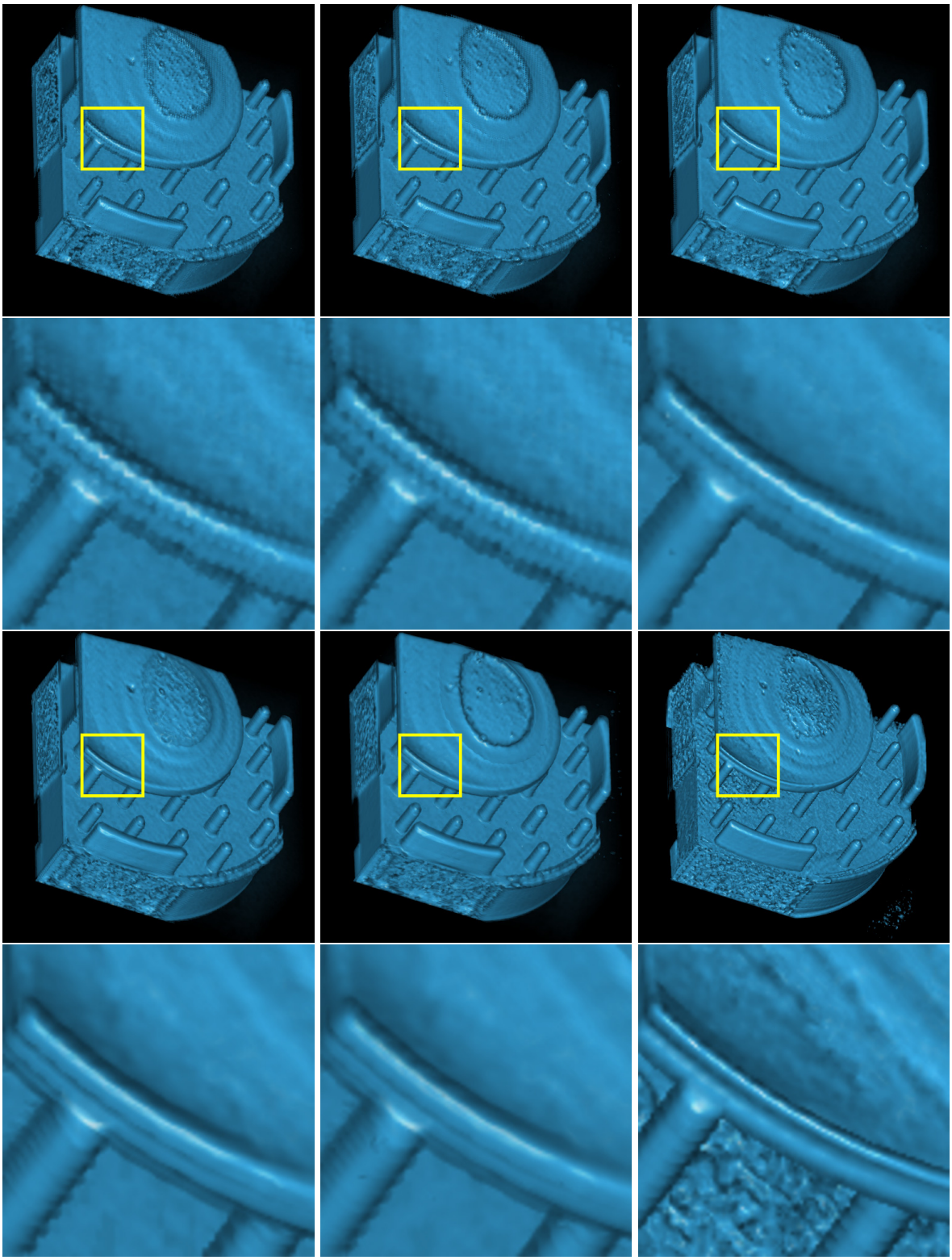


Figure 7.4: Raycasting on the Device dataset. A zoomed region is shown below each image. Top two rows (left to right): barycentric, trilinear on the BCC grid with 2 and with 6 samples. Bottom two rows (left to right): bilinear plus spatial, sheared trilinear and trilinear on the CC grid.

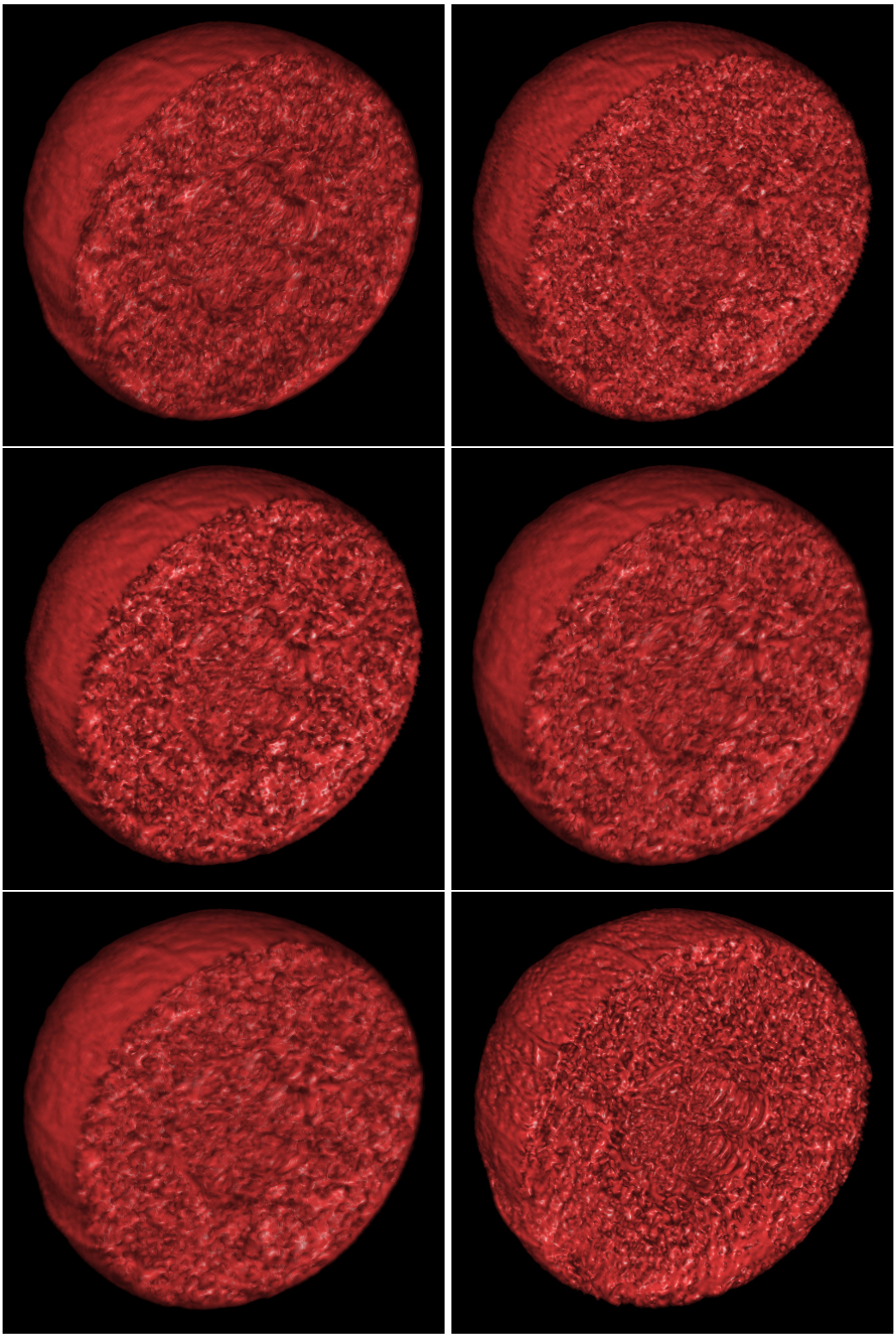


Figure 7.5: Raycasting on the Melon dataset using different interpolators. Top row: bilinear on the BCC grid (left) and barycentric (right). Middle row: trilinear on the BCC grid with 6 samples (left), bilinear plus spatial (right). Bottom row: Sheared trilinear (left), trilinear on the CC grid (right).

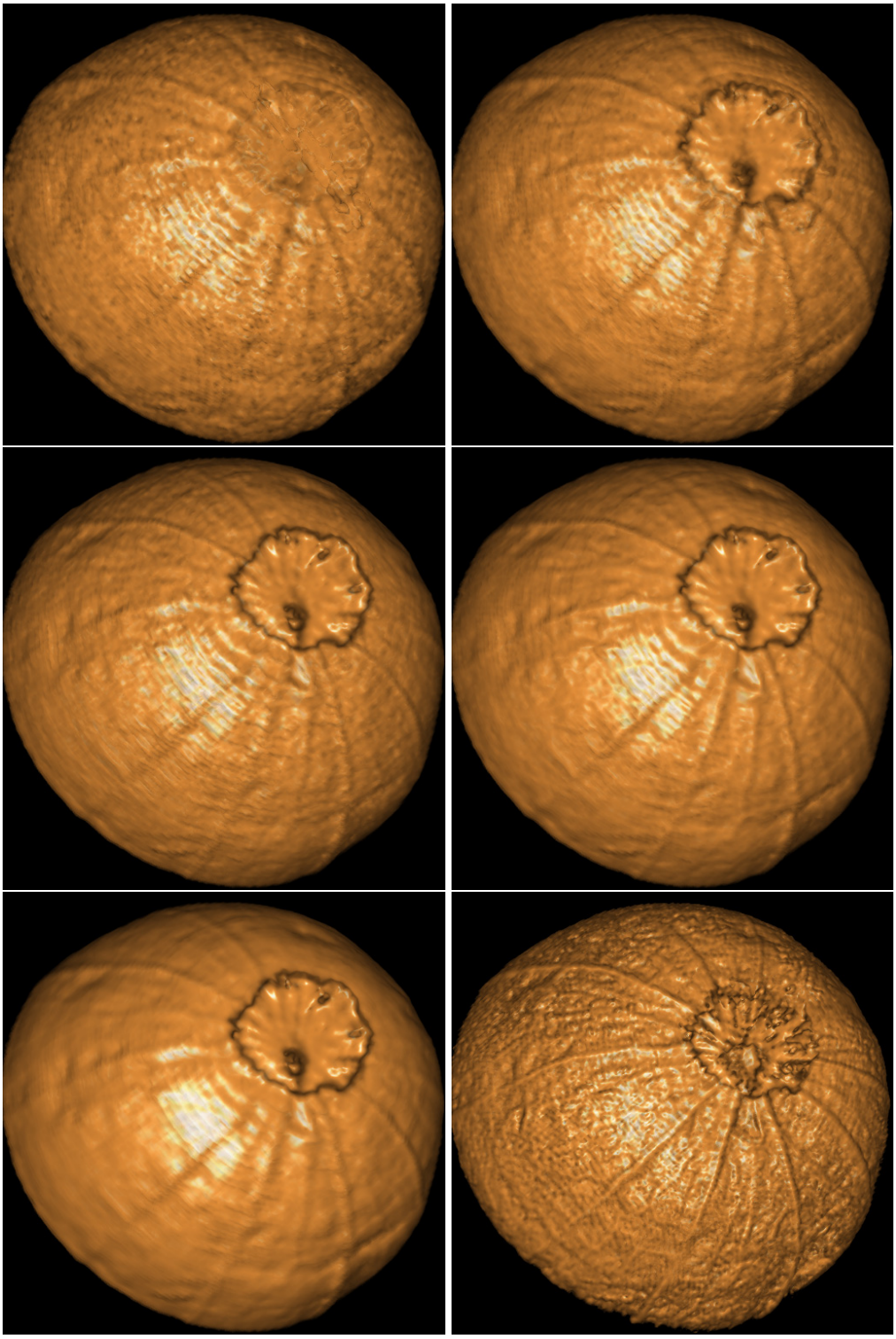


Figure 7.6: Raycasting on the Melon dataset raycasting using different gradient reconstruction schemes. Tow row: adaptive grey level estimation on the BCC grid (left) and central differences 1 on the BCC grid (right). Middle row: central differences 2 (left) and 3 (right) on the BCC grid. Bottom row: sobel filter on the BCC grid (left) and central differences on the CC grid (right).

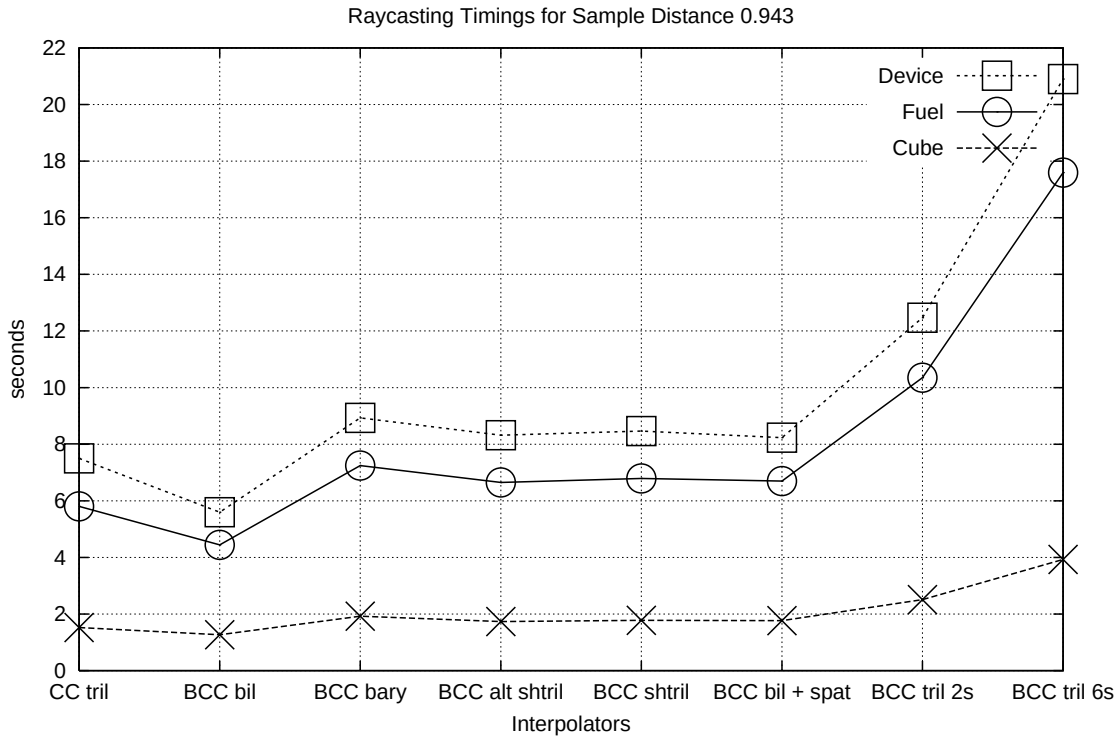
Interpolator	Abbreviation	Grid	Cube	Fuel	Device
trilinear	tril	CC	1.520 sec	5.803 sec	7.501 sec
bilinear	bil	BCC	1.269 sec	4.446 sec	5.596 sec
barycentric	bary	BCC	1.924 sec	7.246 sec	8.930 sec
alternative sheared trilinear	alt shtril	BCC	1.735 sec	6.648 sec	8.320 sec
sheared trilinear	shtril	BCC	1.776 sec	6.787 sec	8.462 sec
bilinear plus spatial	bil + spat	BCC	1.766 sec	6.696 sec	8.233 sec
trilinear 2 samples	tril 2s	BCC	2.507 sec	10.351 sec	12.469 sec
trilinear 6 samples	tril 6s	BCC	3.926 sec	17.592 sec	20.900 sec
trilinear	tril	CC	2.650 sec	10.670 sec	12.723 sec
simplified trilinear	simp tril	BCC	2.339 sec	9.675 sec	12.176 sec
barycentric	bary	BCC	3.586 sec	13.721 sec	17.129 sec
alternative sheared trilinear	alt stril	BCC	3.089 sec	12.318 sec	15.491 sec
sheared trilinear	shtril	BCC	3.146 sec	12.575 sec	15.877 sec
bilinear plus spatial	bil + spat	BCC	3.141 sec	12.418 sec	15.597 sec
trilinear 2 samples	tril 2s	BCC	4.726 sec	20.100 sec	24.235 sec
trilinear 6 samples	tril 6s	BCC	7.694 sec	34.662 sec	40.951 sec

Table 7.2: Raycasting timings for different interpolators and the datasets Cube, Fuel and Device. The sample distance in the upper (lower) half of the table is determined by the bilinear (simplified trilinear) interpolator. The abbreviations are used in the graphical charts from figure 7.7.

From the renderings we can state that the adaptive grey level estimation scheme (left image in the top row) seems to preserve most details among the BCC grid schemes, but does not filter out much of the noise. Central differences 1 (right image in the top row) and central differences 2 (left image in the middle row) appear to smooth out some of the higher frequencies. Central differences 3 (right image in the middle row) is obviously smoother, but also less blurry than central differences 1 and central differences 2. Furthermore, it seems to pronounce the highlights. The sobel filter on the BCC grid (left image in the bottom row) produces the smoothest results, but the skin of the melon is given a quite unnatural appearance. Trilinear interpolation on the CC grid using central differences (right image in the bottom row) produces by far the most detailed rendering.

7.3.2 Performance

We tested the performance of the interpolators on three datasets with different characteristics (refer to table 7.2). We used an artificial dataset (Cube), a computational simulation dataset (Fuel), and a real world dataset (Device). The results are shown more comprehensively in the graphical charts of figure 7.7. Because of their inflexible sampling rate, we do not directly compare the performance of the bilinear interpolation on the CC grid and the BCC grid. The sampling distances are determined by the bilinear interpolator and the simplified trilinear interpolation on the BCC grid. For this interpolators, we must match the planes of the volume in each resam-



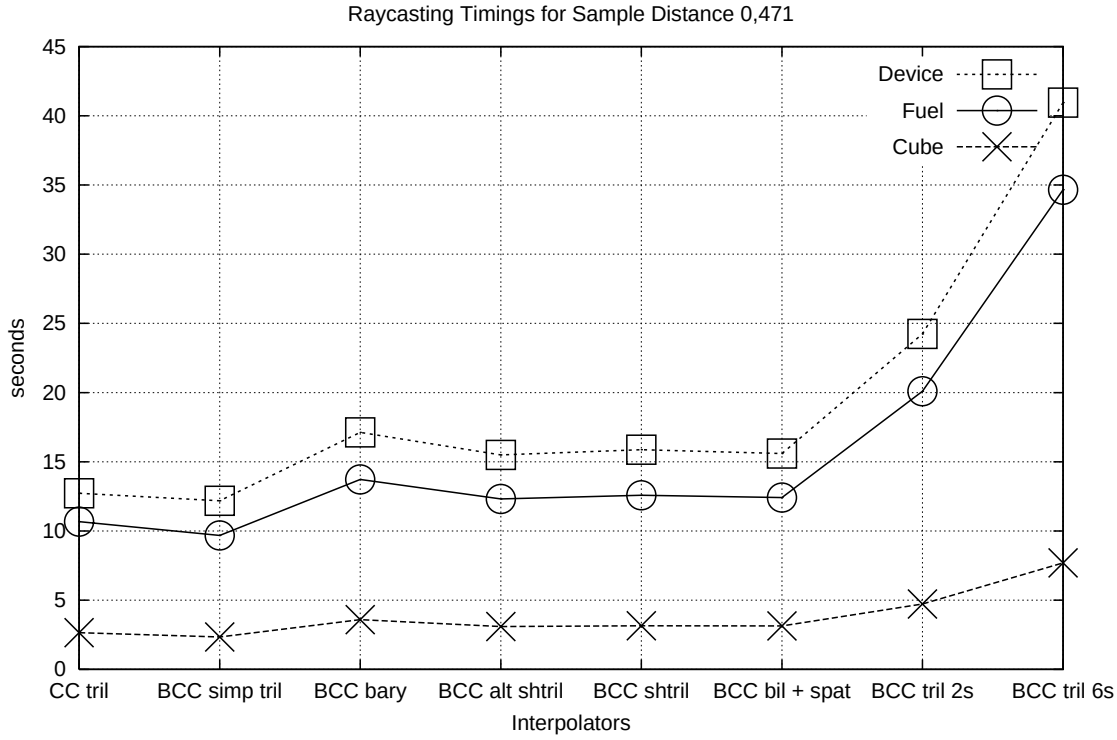


Figure 7.7: Reconstruction scheme timings from table 7.2 for sample distances 0,943 (0,471) in order to align the resampling points with the planes for the bilinear (simplified trilinear) interpolation. The used abbreviations are also explained in the table.

pling step at a given view angle. The timings from all datasets show the same characteristics. The bilinear interpolation is clearly the fastest method. Even the bilinear interpolation combined with the simplified trilinear interpolation is still well ahead of other methods. The three types of interpolators that operate on sheared cubic cells, i.e., bilinear plus spatial, sheared trilinear and the alternative sheared trilinear interpolation have nearly equal rendering times. The alternative sheared trilinear interpolation performs slightly better than the sheared trilinear interpolation, because the task of finding the actual sheared cubic cell is less complex. These methods are not much slower than trilinear interpolation on the CC grid and even faster than the lower quality barycentric interpolation. The reason is the additional overhead that is needed for finding the actual tetrahedron and the calculation of the barycentric coordinates. Trilinear interpolation on the BCC grid is by far the slowest method, the more so if using 6 samples for the resampling of the cubic cell. Trilinear interpolation on the CC grid is only beaten by the bilinear interpolation on the BCC grid. This had to be expected, because raycasting is an image-order algorithm, hence the smaller size of the volume on the BCC grid does not result in any performance gain.

7.4 Texture-Based Volume Rendering

The hardware rendering system was tested on an Athlon Pentium 2.2 GHZ with 512 MB RAM on a Windows XP platform. We used an ATI 9700 Pro graphics chip. The image size is 512×512 pixels.

7.4.1 Rendering Results

In the following comparisons, 2D texture-based rendering refers to the standard approach as explained in section 4.2. The benefits of pre-integration can be observed in figure 7.8, which shows images of semi-transparent unshaded volume renderings. The 2D texture-based renderings (using post-classification) are prone to slicing artifacts on the CC grid (left image in the top row) and BCC grid (right image in the top row). The artifacts are less visible on the BCC grid, as an effect of the higher number of slices. By applying pre-integration, the slicing artifacts disappear in the renderings of both the CC grid (left image in the bottom row) and the BCC grid (right image in the bottom row).

We employ iso-surface rendering to identify a unique kind of artifact on the BCC grid in figure 7.9. The artifacts are clearly visible in the highlights on the cube edges that are most perpendicular to the image plane. For 2D texture-based rendering (leftmost image), they appear as slicing artifacts. With the additional spatial interpolation of multi-texture blending (middle image) and 3D texture-based rendering (rightmost image), the artifacts are smoothed out to some extent. But they still show up as bumps.

In figure 7.10 we can observe semi-transparent images of the Hipiph dataset rendered with the use of post-classified shading and gradient weighted opacity. Slicing artifacts occur for 2D texture-based rendering (left image in the top row), clearly visible in the magnification (right image in the top row). These artifacts disappear for multi-texture blending (left and right image in the middle row) and 3D texture-based rendering (left and right image in the the bottom row).

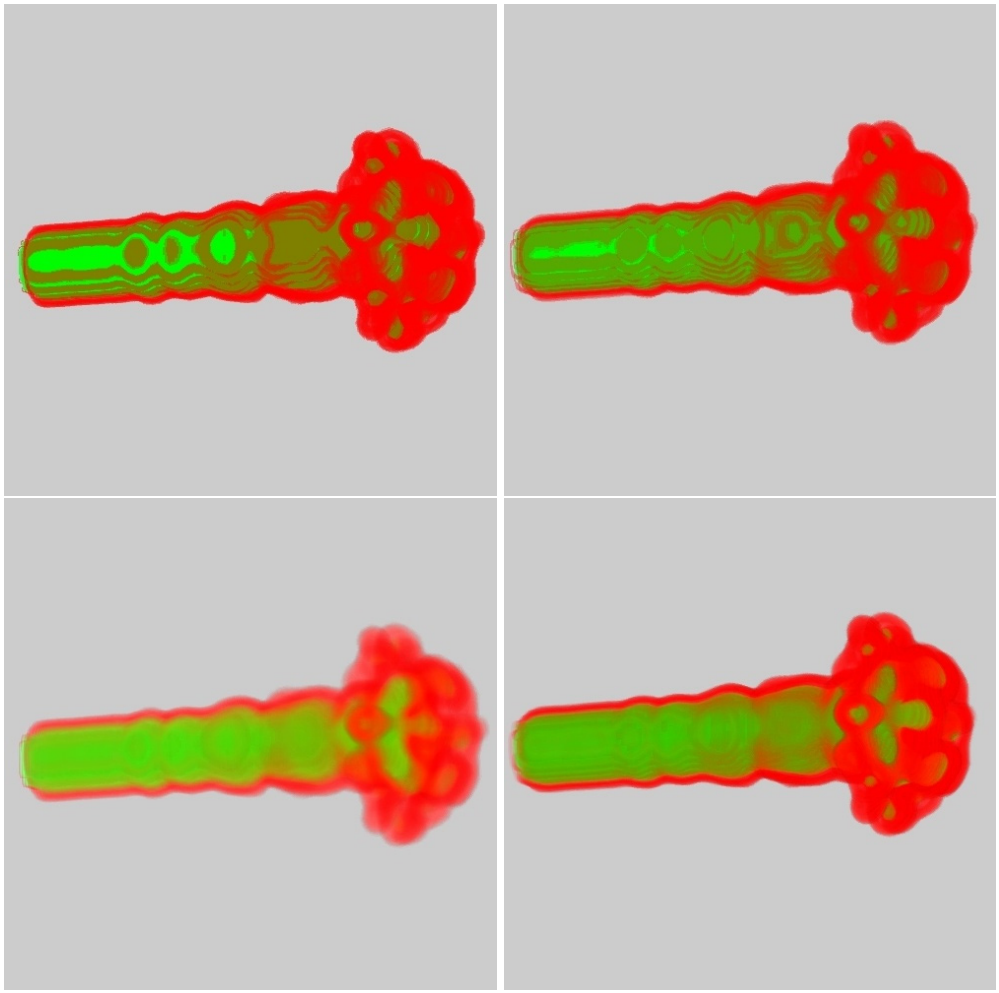


Figure 7.8: Texture-based volume rendering on the Fuel dataset. Top row: 2D texture-based rendering on the CC grid (left) and the BCC grid (right) using post-classification. Bottom row: pre-integration on the CC grid (left) and the BCC grid (right).

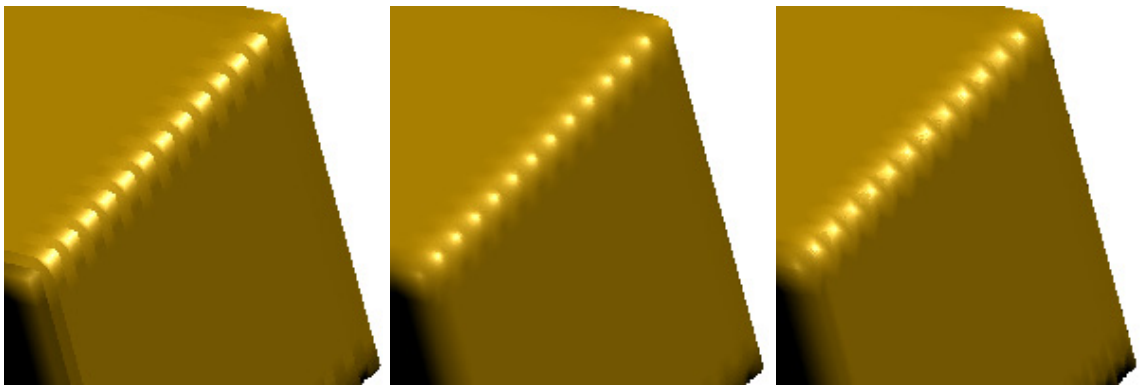


Figure 7.9: Texture-based (iso-surface) rendering of the Cube dataset on the BCC grid. From left to right: 2D texture-based rendering, multi-texture blending, and 3D texture-based rendering.

Figure 7.11 shows a closeup of a highlighted region in renderings of the Fuel dataset. Highlights are extremely sensitive to shading artifacts, especially in combination with the used isosurface rendering. We can clearly see the limitations of 2D texture-based rendering on the CC grid (left image in the top row) and the BCC grid (right image in the top row). The middle row shows multi-texture blending on the CC grid (left image) and on the BCC grid (right image). Very soft highlights are produced by multi-texture blending on the BCC grid that surpasses multi-texture blending on the CC grid in terms of smoothness. Virtually no differences can be found between multi-texture blending on the CC grid and 3D texture-based rendering on the CC grid (left image in the bottom row). Sharp edges are revealed in the highlights of the 3D texture-based rendering on the BCC grid (right image in the bottom row). They are caused by the alternating positive-negative shear direction of the cells, which is not optimal for the rendering quality (refer to section 3.5).

BCC grid rendering support is implemented for most of the rendering modes in the framework, even for NPR rendering modes like tone shading. Tone shading produces results that are similar to technical drawings. The technique is shown in figure 7.12 for the engine block rendered with multi-texture blending (left image) and the Uncbrain rendered with 3D texture-based rendering (right image).

A comparison of several gradient estimation schemes for texture-based rendering is shown in figure 7.13. Interestingly, the differences between the estimators are even more visible than in the raycasting approach (refer to figure 7.6). Similar to raycasting, the CC grid rendering with central differences (left image in the top row) provides the most detailed renderings, but is not free of artifacts. Noise is reduced by using the sobel filter on the CC grid (right image in the top row). Central differences 1 (left image in the middle row) is prone to strong artifacts on the BCC grid. These artifacts are slightly reduced with central differences 2 on the BCC grid (right image in the middle row). A major improvement to image quality is achieved with central differences 3 (left image in bottom row). Additional smoothing is introduced when applying the sobel filter on the BCC grid (right image in bottom row).

7.4.2 Performance

Although a corresponding OpenGL extension is already listed in the extension specification [28], non power of two textures are still not supported in our environment. Therefore a performance comparison of CC and BCC grid renderings turns out to be very difficult. We must either blow up the BCC or CC grid volume to the next power of two, always treating one of the two grids in an unfair way. For 2D texture-based techniques, a fair comparison would be possible with the texture rectangle extension from NVIDIA. But this could still not solve the problem in case of 3D texture-based volume rendering, because a similar extension does not exist for 3D textures.

In table 7.3 (refer to figure 7.14 for a graphical chart of the timings), we are are able to present a fair comparison of the rendering methods by using the following approach: The CC grid timings are measured on the original data size, e.g., 64^3 (128^3). The corresponding BCC data size is $45 \times 45 \times 91$ ($91 \times 91 \times 181$). As stated above, direct measurement is not possible, but we are able to estimate the performance by finding a lower and an upper boundary of the actual rendering time. This is done by employing the next smaller and the next greater power of two data

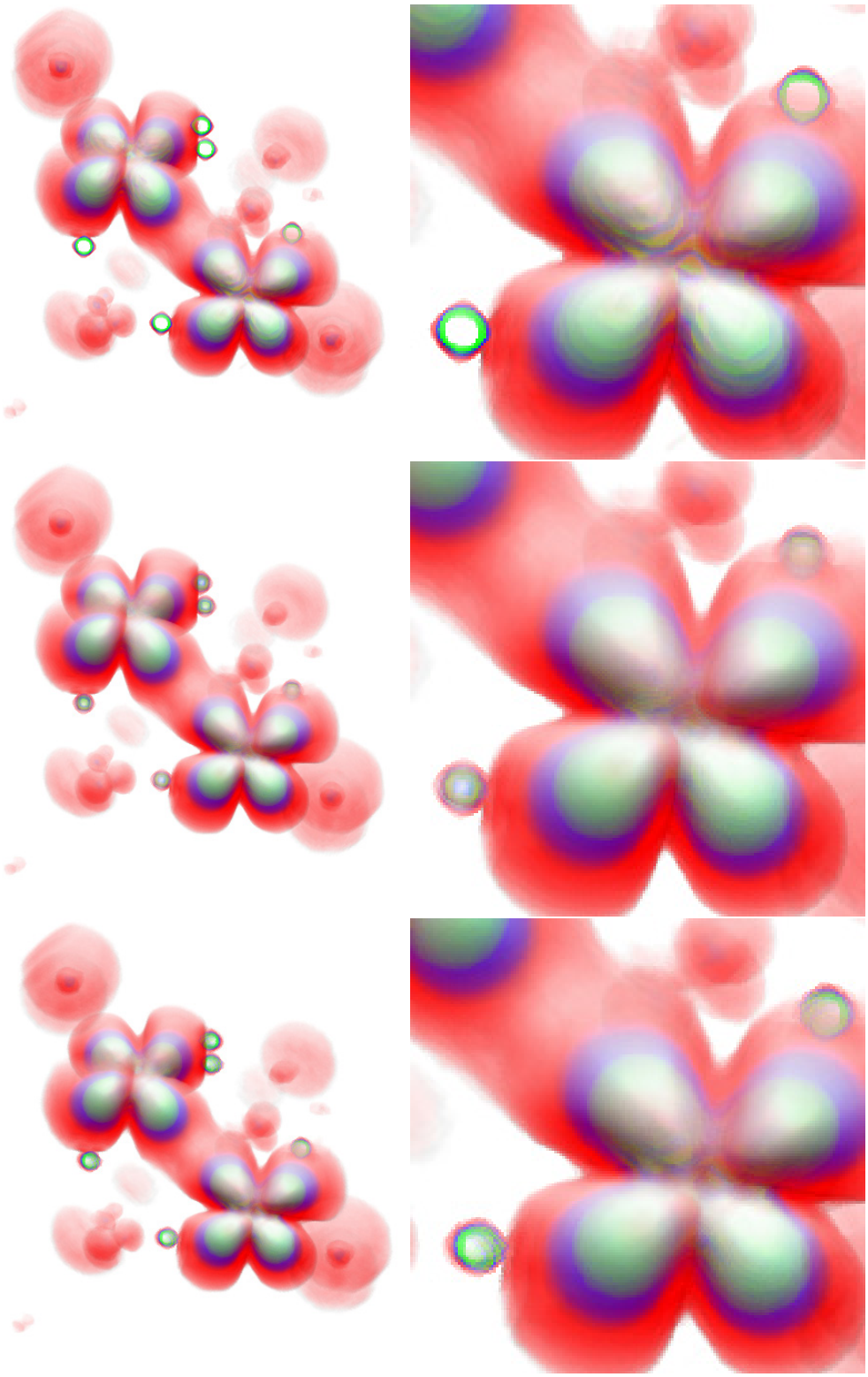


Figure 7.10: Texture-based volume rendering of the Hipiph dataset on the BCC grid using post-classification and gradient weighted opacity. Magnifications of a region are shown in the right column. From top to bottom row: 2D texture-based rendering, multi-texture blending, and 3D texture-based rendering.

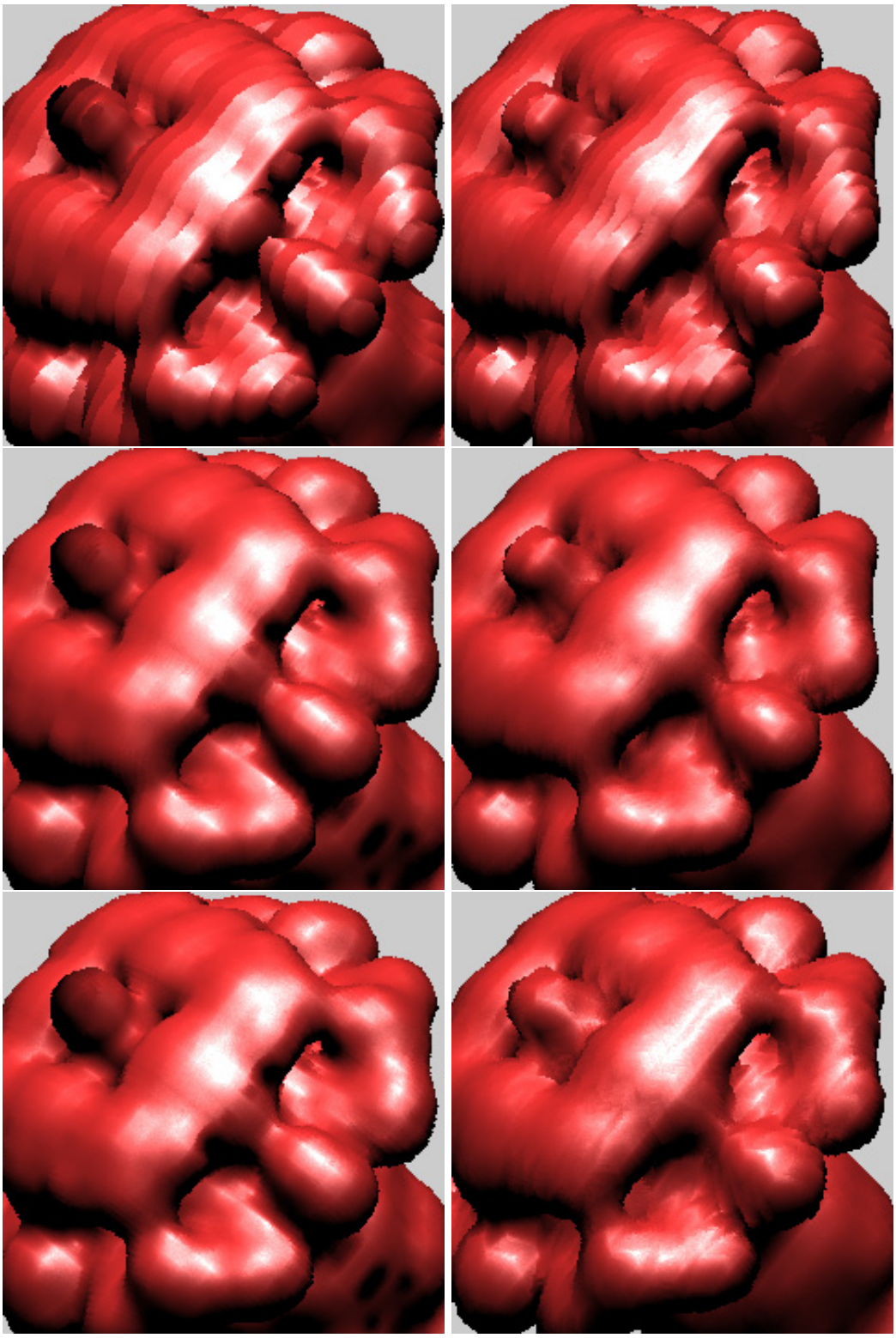


Figure 7.11: Texture-based (iso-surface) rendering of the Fuel dataset on the CC grid (left column) and the BCC grid (right column). From top to bottom row: 2D texture-based rendering, multi-texture blending, and 3D texture-based rendering.

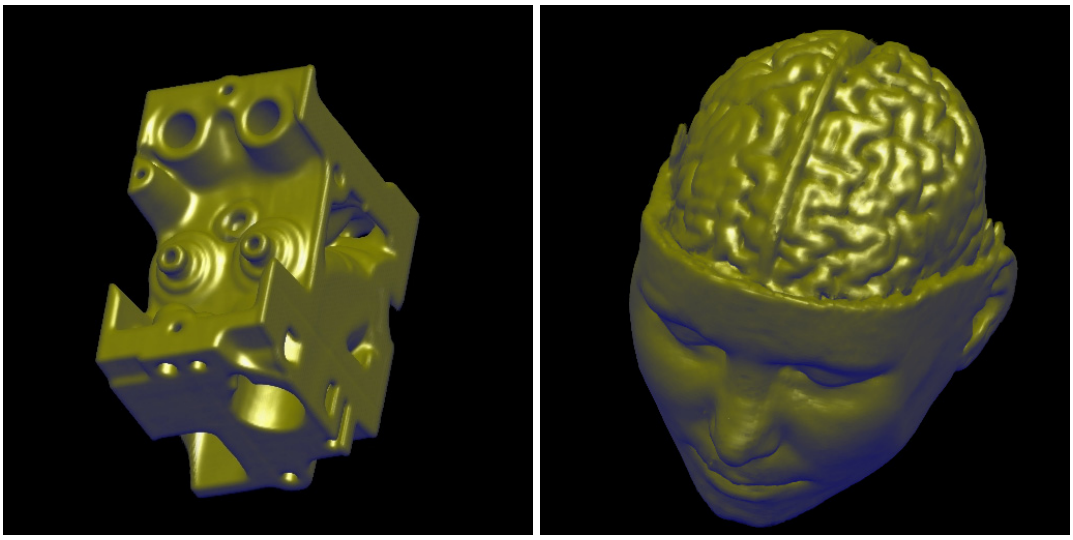


Figure 7.12: Texture-based rendering (tone shading) on the BCC grid. Multi-texture blending (left) on the Engine dataset and 3D texture-based rendering (right) on the Uncbrain dataset using.

CC data size	$64 \times 64 \times 64$			$128 \times 128 \times 128$		
Type	BCC 32	BCC 64	CC	BCC 64	BCC 128	CC
2D tex	50 ms	51 ms	36 ms	76 ms	80 ms	57 ms
pre-integration	66 ms	66 ms	58 ms	115 ms	119 ms	93 ms
multi-tex	61 ms	63 ms	64 ms	108 ms	113 ms	109 ms
3D tex	218 ms	220 ms	86 ms	418 ms	422 ms	175 ms
shaded 2D tex	197 ms	213 ms	144 ms	389 ms	388 ms	269 ms
shaded multi-tex	214 ms	213 ms	207 ms	383 ms	441 ms	410 ms
shaded 3D tex	415 ms	412 ms	262 ms	826 ms	840 ms	498 ms

Table 7.3: Texture-based rendering timings. Post-classification was used for all methods except for pre-integration. "BCC 32" ("BCC 64", "BCC 128") correspond to a BCC grid power of two dataset (e.g. "BCC 64" means data size $64 \times 64 \times 128$). "2D tex" refers to 2D texture-based and "3D tex" to 3D texture-based rendering. "multi-tex" refers to multi-texture blending.

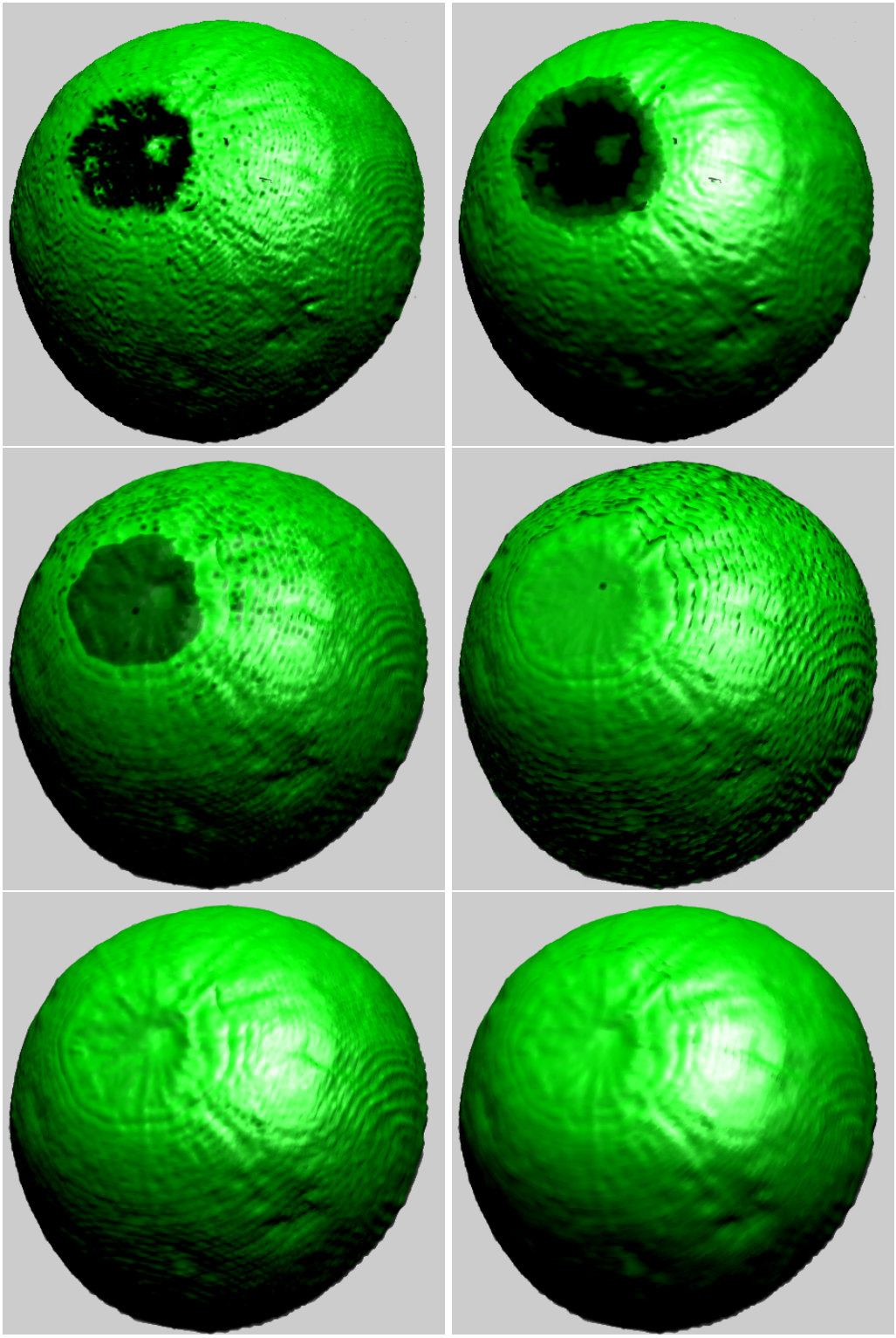


Figure 7.13: Texture-based rendering of the Melon dataset using different gradient estimators. Top row: central differences (left) and the sobel filter (right) on the CC grid. Middle row: central differences 1 (left) and 2 (right) on the BCC grid. Bottom row: central differences 3 (left) and the sobel filter (right) on the BCC grid.

size, e.g. $32 \times 32 \times 64$ and $64 \times 64 \times 128$ for the BCC grid datasets corresponding to the 64^3 CC grid dataset, respectively $64 \times 64 \times 128$ and $128 \times 128 \times 256$ for the datasets corresponding to the 128^3 CC grid dataset. We provide timings for 2D texture-based volume rendering, 3D texture-based volume rendering, and multi-texture blending. Post-classification is used for all these methods. Furthermore, we present timings for the pre-integration approach. Considering the restrictions regarding the number of slices for 2D texture-based volume rendering and the pre-integration method, we applied 64 (128) slices for CC and 91 (181) slices for the BCC grid. We can freely choose the slice number in the other rendering modes, and we decided to use the realistic number of 91 (181) slices for a reasonable resampling quality on both grids. The situation in an environment that supports non power of two textures can be estimated with the help of this comparison scheme.

The smaller texture size on BCC was expected to result in a smaller rasterization time per texture, especially for 3D textures. Unfortunately, the difference turned out to be hardly noticeable. Studying the table and the graphical charts in figure 7.14, we identified three different characteristics for the main approaches of texture-based volume rendering:

2D texture-based rendering, pre-integration: Although we have a slight performance gain from smaller texture sized on the BCC grid, the required $\sqrt{2}$ times larger number of texture slices result in a significantly higher rendering time. On the other hand, this pays off in form of a higher sample frequency on the BCC grid.

Multi-texture blending: In principle, there is no additional overhead on BCC grids. The only difference to the CC grid is the assignment of other texture coordinates. Considering the smaller texture sizes, computational load is equal or smaller. This is also reflected in the rendering times, which are nearly equal on both grids.

3D texture-based rendering: The texture size had little to no influence on the rendering time in our experiments. On the contrary, the seven per fragment instructions necessary on the BCC grid heavily affect rendering time, especially for unshaded post-classification. Shaded post-classification also uses quite complex fragment programs. Hence the performance loss caused by the instructions for BCC grid rendering is less noticeable for this rendering mode.

7.4.3 Memory Usage

We sketch the situation for a scenario where non power of two textures already exist. Then we can state that we save 29.3% texture memory for 2D texture-based rendering and pre-integration on the BCC grid. This is because the slices must be aligned to the planes of the volume and cover all volume samples exactly once. For multi-texture blending, we need half the memory per slice stack on the BCC grid, because the textures are half the size and we are not required to align the slices with the volume planes. In 3D texture-based volume rendering on the BCC grid, the 3D textures are 29.3% smaller than on the CC grid. However, we already know that three 3D textures (one for each principal view axis) are needed to meet the requirements regarding the shear planes of the cells (see section 3.5). But we can state that it is only required to have one 3D

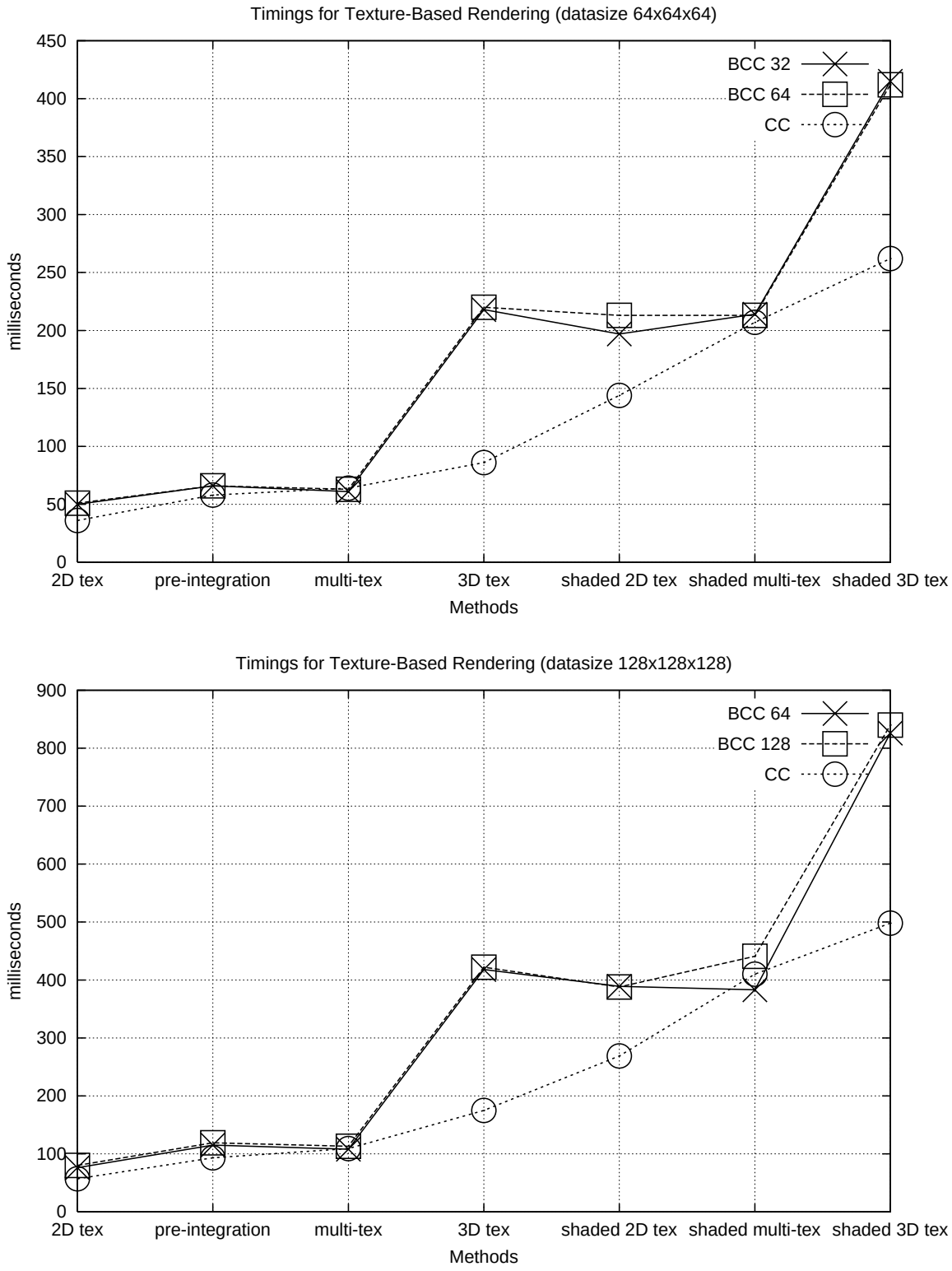


Figure 7.14: Texture-based volume rendering timings corresponding to table 7.3. "2D tex" refers to 2D texture-based and "3D tex" to 3D texture-based rendering. "multi-tex" refers to multi-texture blending.

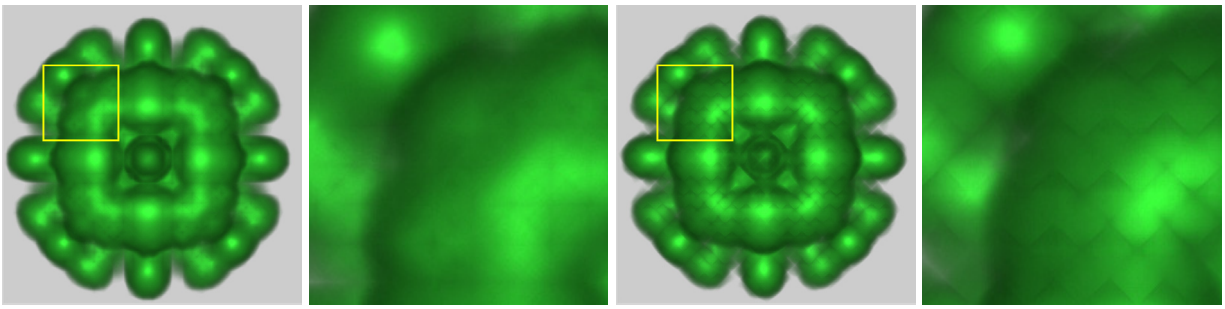


Figure 7.15: The projected tetrahedra algorithm on the Fuel dataset using pre-integration and Phong Shading. From left to right: CC grid rendering and a zoom, BCC grid rendering and a zoom.

texture in texture memory at once, and a new texture must be loaded into texture memory only when the principal view direction changes.

7.5 Projected Tetrahedra Algorithm

The projected tetrahedra system exploits orthogonal projection on regular grid structures in order to maximize rendering speed. At the same time we wanted provide a rendering quality that is comparable to high quality rendering methods like splatting. We want to achieve this through powerful extensions like pre-integration. We compared image quality and performance of the projected tetrahedra algorithm and the implemented extensions on the CC and BCC grid. A 3D adjacency structure from chapter 5.5 is used to speed up traversal time. We used an Athlon Pentium 1.4 GHZ with 512 MB RAM in combination with a GeForce4 TI graphics chip on a Suse Linux 8.0 platform. Image size is 800×800 pixels for all images.

7.5.1 Rendering Results

Figure 7.15 reveals Mach band artifacts which can occur in renderings of the BCC grid version of the Fuel dataset (leftmost image), clearly noticeable in the closeup (second image from the left). Such artifacts are most visible at special view angles (e.g. 90°, 270°) and reveal the underlying grid structure. The Mach band artifacts are reduced on the CC grid rendering (second image from the right) and closeup (rightmost image) by applying a checkerboard style decomposition of the cubic cells as stated in chapter 5.2.

Images of the Hipiph dataset are shown in figure 7.16. The right column shows a closeup of the renderings from the left column. The images are rendered with the original projected tetrahedra algorithm (top row), with the use of exponential transparency textures (middle row), and with the use of pre-integration (bottom row). We can observe that there are only small differences between the renderings produced with the original projected tetrahedra algorithm to the renderings using exponential transparency textures. On the contrary, the application of pre-integration provides a major improvement of rendering quality.

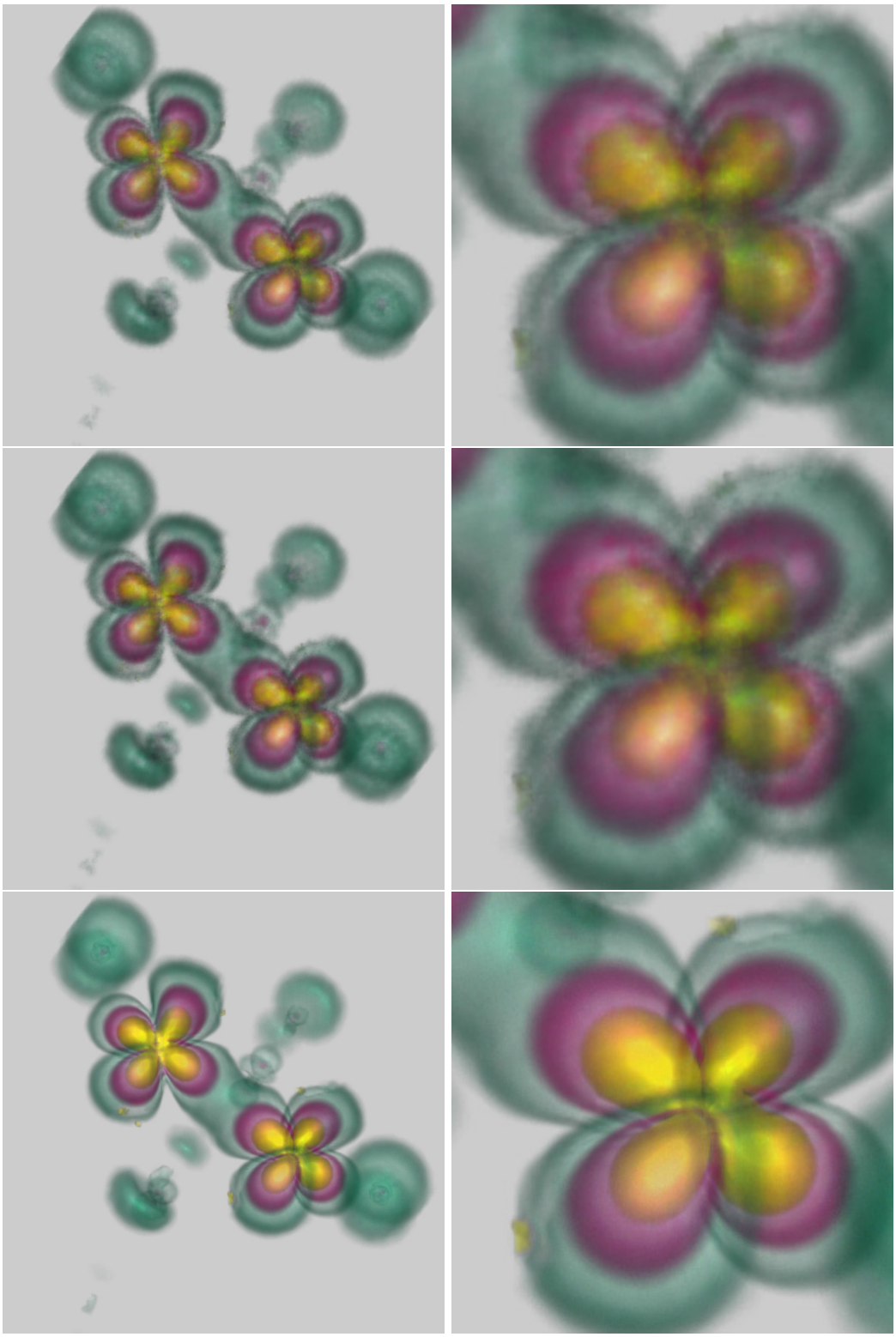


Figure 7.16: The projected tetrahedra algorithm for the Hipiph dataset on the BCC grid. Top row: Original projected tetrahedra algorithm (left) and closeup (right). Middle row: exponential transparency textures (left) and closeup (right). Bottom row: pre-integration (left) and closeup (right).

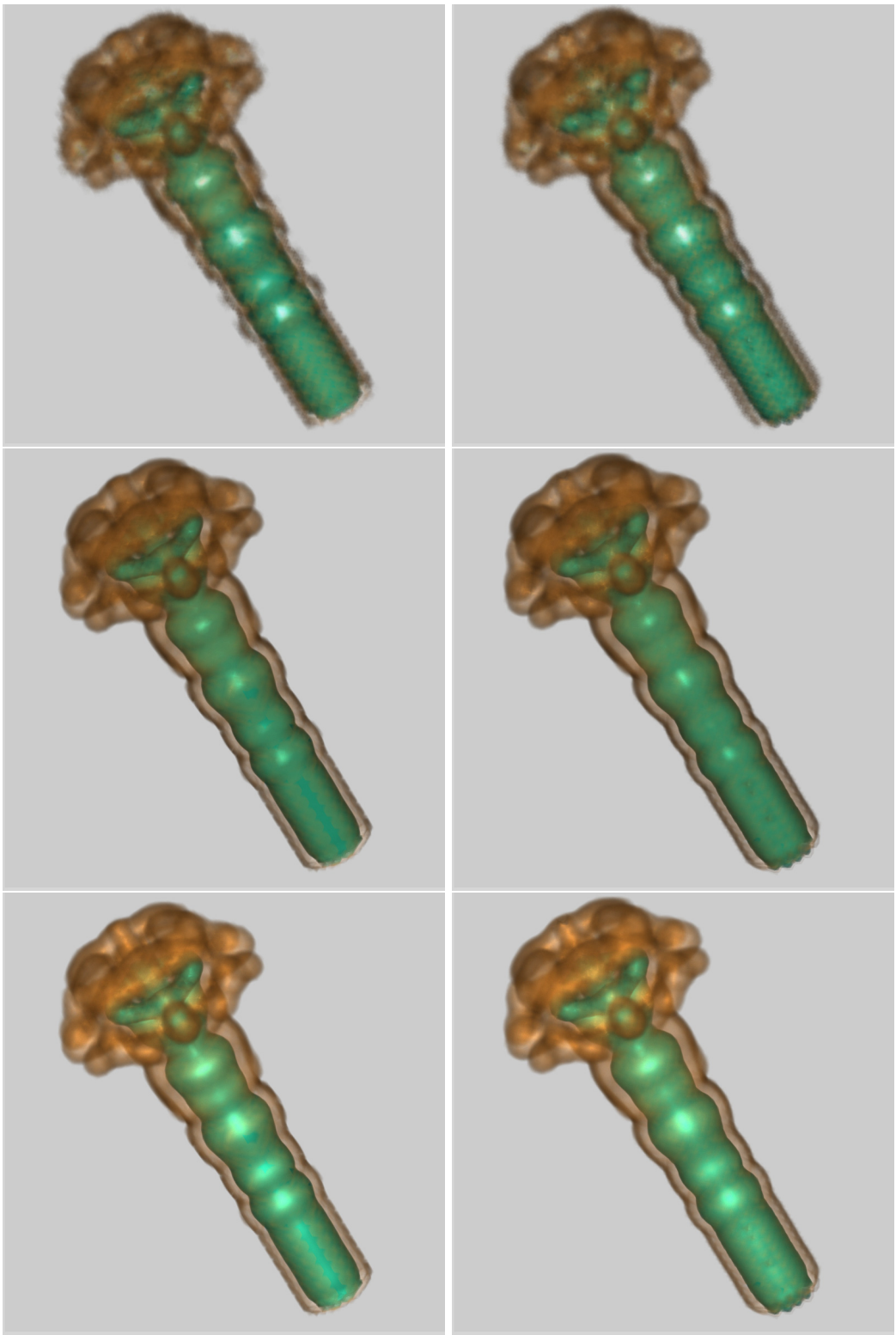


Figure 7.17: The projected tetrahedra algorithm on the Fuel dataset. The CC grid is shown in left column, the BCC grid in the right column. From top to bottom row: exponential transparency textures, pre-integration and Gouraud shading, pre-integration and Phong shading.

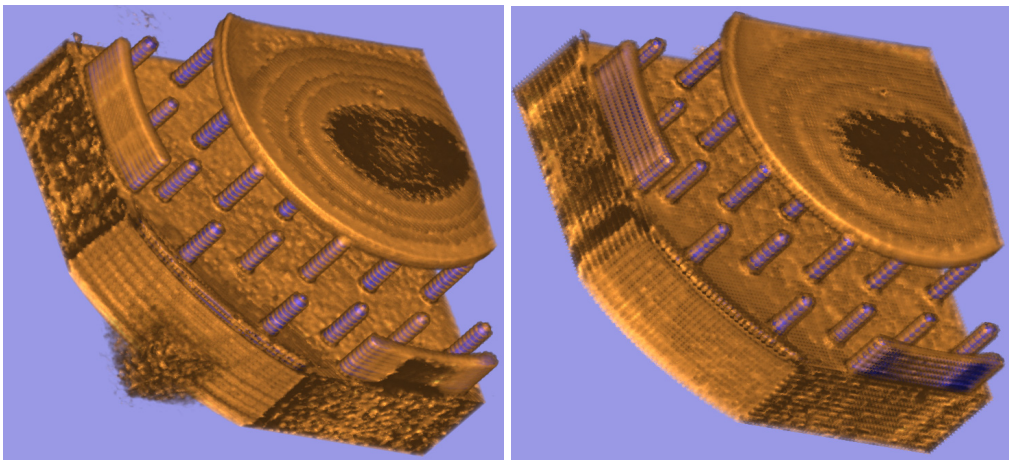


Figure 7.18: The projected tetrahedra algorithm on the Device dataset using pre-integration and Phong shading. The CC grid is shown in the left image and the BCC grid in the right image.

We present a comparison of projected tetrahedra renderings of the Fuel dataset on the BCC grid and the CC grid in figure 7.17. The images in the top row were rendered using exponential transparency textures on the CC grid (left image in the top row) and the BCC grid (right image in the top row). Pre-integration and Gouraud shading on the CC grid (left image in the middle row) and the BCC grid (right image in the middle row) provides much better rendering quality than the exponential transparency textures. The images rendered with pre-integration and Phong shading on the CC grid (left image in the bottom row) and on the BCC grid (right image in the bottom row) look brighter than the Gouraud shaded images. For all images, no major quality difference can be observed between CC grid and BCC grid renderings. Pre-integration contributes much to the rendering quality. Interestingly, the highlights look shinier in the renderings using exponential transparency textures than in the renderings exploiting pre-integration. This is because pure white highlights are applied for the exponential transparency textures. On the other hand, we found out that highlights derived from the colors of the transfer function look nicer for shading in combination with pre-integration.

A comparison of the CC grid (left image) and BCC grid (right image) renderings on the Device dataset is shown in figure 7.18. There are some differences in the volume data represented by the CC and BCC grid. Noise can be observed in the CC grid dataset which is missing in the BCC grid dataset. Both renderings appear to be almost equally detailed, although shading quality is better in the CC grid rendering.

Rather severe shading artifacts can occur for high frequencies datasets even with the use of pre-integration. This is caused by the fact that the shading quality does not match the quality of color and opacity calculation. Figure 7.19 shows images of the Lobster dataset on the CC grid, where the unshaded rendering (image in the top row) looks clearly better than the shaded one (left image in the bottom row). In such a case, the pre-integrated Gouraud shading approach (refer to chapter 5.9) can yield nicer looking results (right image in the bottom row). Of course this is just a subjective comparison. The method has the drawback that color and transparency of the diffuse

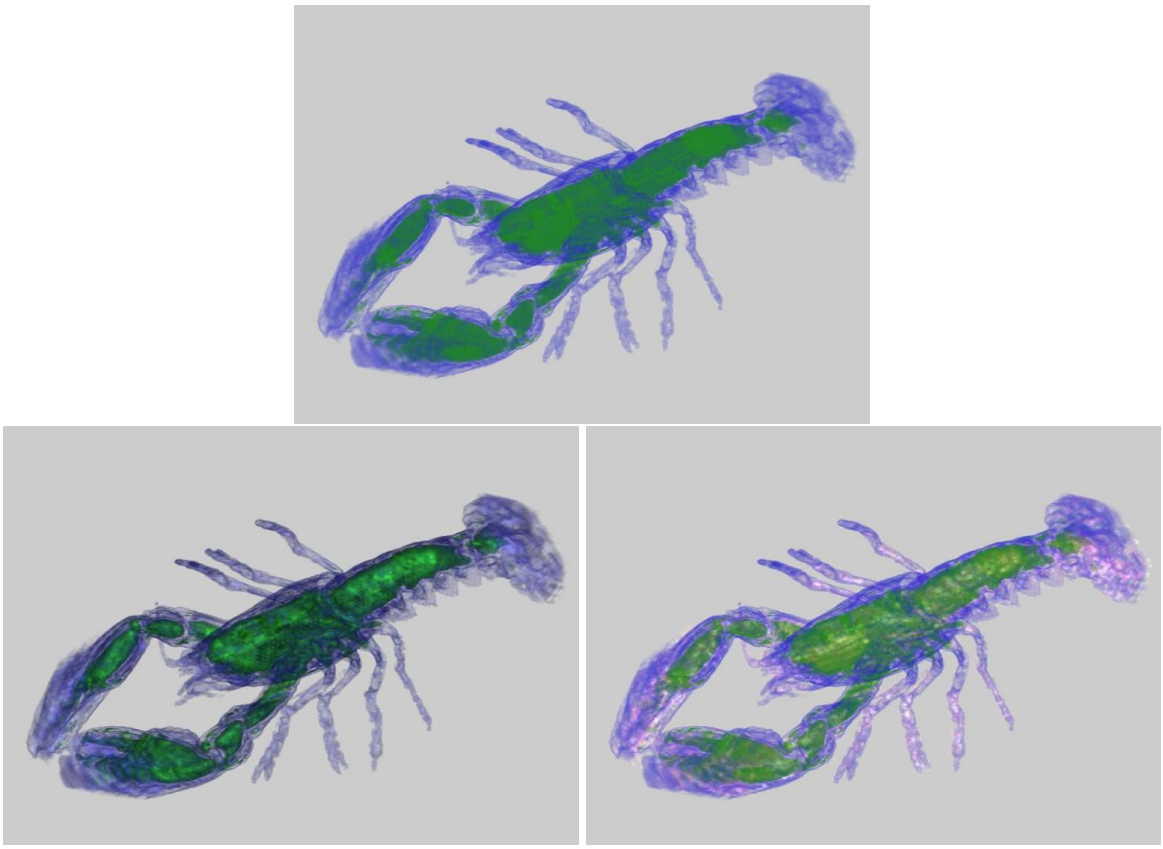


Figure 7.19: The projected tetrahedra algorithm for the lobster dataset on the CC grid using pre-integration. Unshaded (top row), Phong shading (bottom row left), and pre-integrated Gouraud shading (bottom row right).

and specular terms are constant during the pre-integration process and must be set to appropriate values beforehand. Especially the transparency values need to be adjusted carefully in order to produce good shading effects. However, the images rendered with this method generally have a rather flat shaded appearance.

In figure 7.20 we can see rendering of the Uncbrain dataset on the BCC grid. By using the simple central differences 1 method (leftmost image) we introduce artifacts, which are clearly visible in the closeup (second image from the left). Replacing the central differences with a more sophisticated 3×3 sobel filter (second image from the right) reduces shading artifacts, which can be clearly seen in the rightmost image.

7.5.2 Performance

Some statistics about the projected tetrahedra algorithm on different datasets (Fuel dataset on the CC and the BCC grid, Hipiph dataset on the CC and the BCC grid) can be observed in table 7.4. The rendering times on these datasets are also visualized in the graphical chart of figure 7.21 for better comprehension. We can see from the tables that only 5% of all tetrahedra in the grid

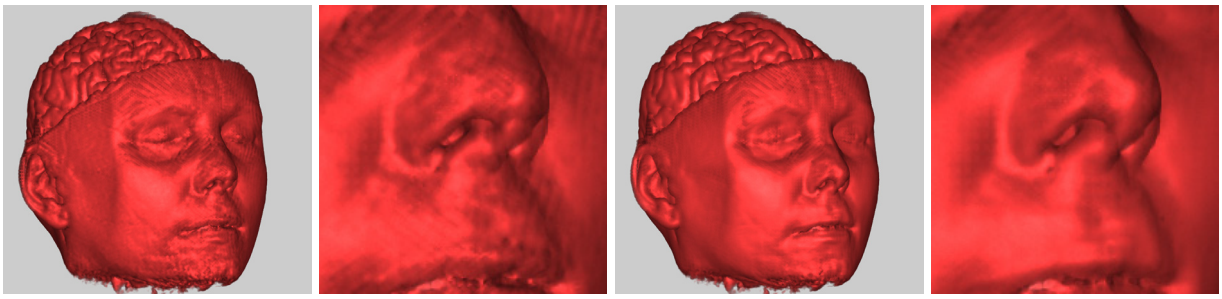


Figure 7.20: The projected tetrahedra algorithm for the Uncbrain dataset on the BCC grid using pre-integration. From left to right: central differences 1 and a closeup, sobel filter and a closeup.

are actually rendered for the Fuel and Hipiph dataset. In a traversal complex that is marked as visible, statistically almost all tetrahedra are also of non-zero opacity: between 9 and 10 from maximal 12 tetrahedra on the BCC grid and more than 4 from maximal 5 tetrahedra on the CC grid. Considerably less tetrahedra must be rendered on the BCC grid. This results in a better performance of the algorithm, as the overall tetrahedra number is of course the most significant figure for rendering time. Most noticeable, when using pre-integration with Phong or Gouraud shading we achieve better a performance than with the original algorithm. This despite the costly 3D texture mapping (using a 128^3 texture) and shading calculation in the register combiners. The reason is the relatively expensive software computations per tetrahedron. They are more complex for the original projected tetrahedra algorithm and the exponential transparency textures. Obviously the computations use up more rendering time than the 3D texture mapping. Much slower is the pre-integrated Gouraud shading method, because processing three 3D textures in the register combiners is quite demanding even for powerful graphics chips.

7.5.3 Memory Usage

As the system is a leaf in the `1B` (1 byte) branch of the vuVolume framework (see section 6.1), the data is stored as an array of bytes. The normals are stored as 3 float values (4 byte each), hence we need 13 bytes per basic data sample. Every visible traversal complex structure stores 17 byte (refer to the implementation section 6.4.2). The information about several tetrahedra is stored in a single complex. Still the memory consumption is less than the 28 byte per visible voxel reported for the original data structure [44]. Let us denote the size of the volume as s, and let t be the number of visible traversal complexes, plus the virtual traversal complexes. Then we have a memory consumption m of $13s + 17t$.

For maximum rendering performance, we create a separate structure described in section 6.4.5 which stores all vertices belonging to a traversal complex. This structure needs 25 byte. If q refers to the number of samples that are part of this additional structure, we have a new formula for the extended memory consumption m_e:

$$m_e = 13s + 17t + 25q \tag{7.1}$$

The number t of visible traversal complexes is small compared to the number of data samples,

Dataset	Fuel BCC	Fuel CC	Hipiph BCC	Hipiph CC
Statistics				
# tetrahedra	1093500	1310720	1093500	1310720
# data points	182250	262144	182250	262144
# nz tetrahedra	51821	61469	40463	49804
# nz traversal complex	5147	13015	4373	10966
nz tetrahedra / traversal complex	10.068	4.722	9.253	4.542
tetrahedra / nz tetrahedra	21.101	21.323	27.025	26.318
Timings				
linear	62 ms	75 ms	53 ms	65 ms
exponential	64 ms	76 ms	52 ms	65 ms
pre-int and Gouraud	47 ms	57 ms	39 ms	51 ms
pre-int and Phong	55 ms	65 ms	44 ms	58 ms
pre-integrated Gouraud	80 ms	95 ms	64 ms	79 ms

Table 7.4: Statistics and timings of Projected Tetrahedra on different datasets. Some important numbers and ratios are shown. "nz" refers to non-zero opacity. "Linear" refers to the linear transparency variation in the original projected tetrahedra algorithm. "exponential" refers to the exponential transparency textures, and "pre-int" to the pre-integration.

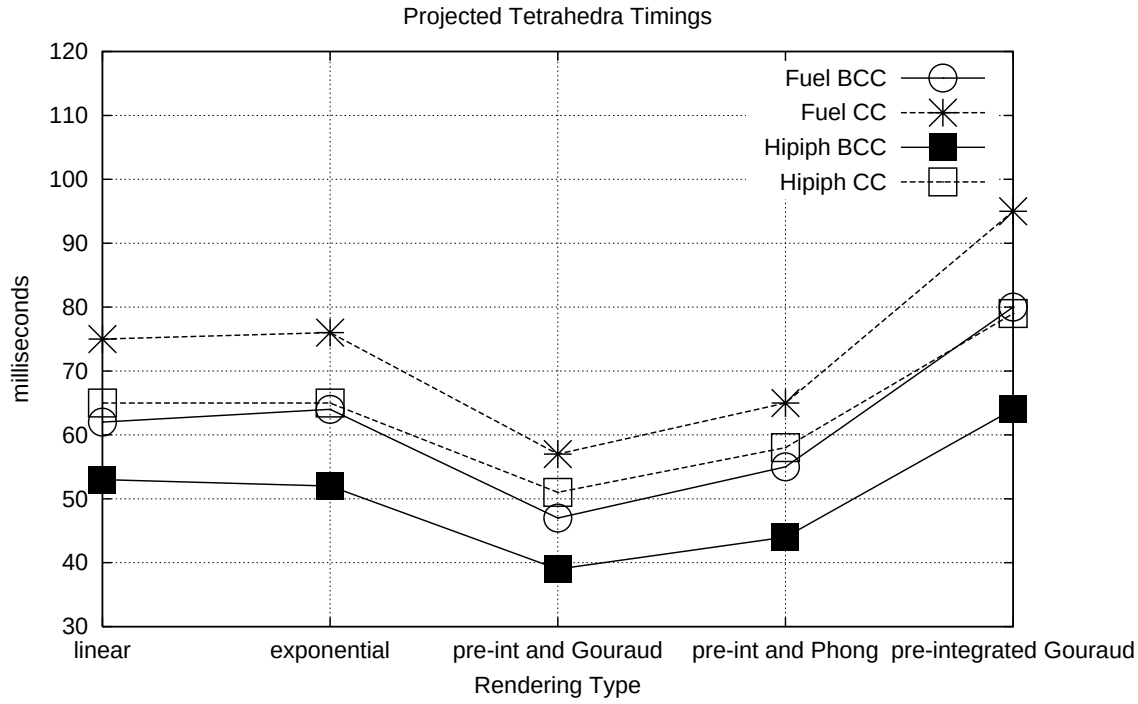


Figure 7.21: Projected Tetrahedra Timings from table 7.2.

especially on the BCC grid. According to the Fuel dataset statistics in table 7.4, the ratio of traversal complexes per data voxel is less that 3%. In our experiments we found out that the number of sample points belonging to a traversal complex is usually about 2 times the overall number of traversal complexes on the BCC grid. On the CC grid, the number of sample points belonging to a traversal complex is slightly higher than the number of traversal complexes. Hence t and q are small numbers compared to the datasize s. We can state that the memory consumption of our adjacency structure is quite reasonable for most datasets that are used in practice. In principle it would be possible to delete the original data array and store only the normals of the tetrahedron vertices. As a consequence, the memory consumption could even drop under the overall consumption of the basic storage scheme (i.e., 13 byte per sample).

Chapter 8

Summary

In this chapter we give a summary of the important points of this thesis, our BCC grid reconstruction schemes and the proposed hardware-accelerated algorithms on the BCC grid.

8.1 Introduction

The Cartesian (CC) grid is the dominant type of sampling grid in volume visualization. Although we known from signal theory [17] that it is not optimal in terms of storage efficiency. To find the optimal sampling grid, we have to solve the dual problem of packing the replicated spectra in frequency domain as closely as possible so that they do not overlap. In volume rendering, it is assumed that we are dealing with isotropic, band-limited scalar functions which have spherical spectra. Our problem is equivalent to the famous sphere packing problem [55]. There is no general solution to this problem yet, but fortunately for our purpose some optimal packing schemes among regular grids in 2D and 3D are known.

In 3D the Hexagonal Close Packing (HCP) grid and the Face-Centered Cubic (FCC) grid are optimal sphere packings. The dual of the HCP grid in the spatial domain is again a HCP grid, the dual of the FCC grid is the Body-Centered Cubic (BCC) grid. Both schemes require about 29.3% less samples than on the CC grid to store the same amount of information [60]. The BCC grid is easier to handle than the HCP grid, because it can be described by a sampling matrix as opposed to the HCP grid and has some convenient properties that can be exploited for volume rendering. The BCC grid can be seen as

- stack of 2D CC grids, where the odd-numbered planes are translated by half a unit in both dimensions with respect to the even-numbered planes.
- two interleaved 3D CC grids, where one grid (denoted as secondary grid) is translated by half a cell spacing in all three axis relatively to the other grid (denoted as primary grid).
- sheared and scaled CC grid.
- tetrahedral mesh which is simple and uniquely defined by the Delauney complex [6].

Our goal is to show the the volume visualization community that the BCC grid is an alternative to the CC grid in the practice. Therefore we must first proof it's usability in different types of volume rendering algorithms. Furthermore, we have to achieve a performance gain over most comparable rendering approaches on the CC grid due to the reduced memory requirements, with equal or only slightly reduced image quality.

An evolving field of volume visualization deals with exploiting the power of flexible consumer graphics hardware for interactive or near-interactive volume rendering. Hardware acceleration could increase the popularity of traditionally slow direct volume rendering for usage in time-critical applications as well. The potential performance gain of BCC grids is therefore even more desirable in such hardware-based approaches. We adapted two of the most popular hardware-accelerated volume rendering methods to the BCC grid, texture-based volume rendering and the projected tetrahedra method, which we expected to be especially suited for BCC grids.

8.2 Previous work

Volume rendering can be classified into image-order algorithms like raytracing [27] and raycasting [32], and object-order algorithms like splatting [67], cell projection [54, 38, 68, 69, 53] and texture-based volume rendering [4]. A hybrid method is the performance optimized shear-warp algorithm proposed by Lacroute et al. [30], which can be considered as pure software predecessor of 2D texture-based volume rendering.

Texture-based volume rendering was proposed by Cabral et al. [4] after introduction of the first SGI Reality Engine [1]. Support for directional shading was introduced by Gelder et al. [21] and Dachille et al. [14] at the expense of losing interactivity. Later shading was proposed for fast iso-surface rendering by Westermann et al. [66] and semi-transparent diffuse shading by Meißner et al. [40]. Limitations of 2D texture-based rendering were successfully reduced with multi-texture blending proposed by Rezk-Salama et al. [48]. Accurate integration without supersampling using pre-integration was introduced by Engel et al. [19].

The projected tetrahedra algorithm was proposed by Shirley et al. [54]. Accuracy of the opacity integration was improved by Stein et al. [56] and Max et al. [37]. Accurate integration of arbitrary transfer functions is possible with the use of 3D pre-integration textures proposed by Röttger et al. [51]. This technique was further refined in [50, 23, 18].

Theußl et al. [60] first proposed BCC grids for direct volume rendering. They implemented Westover style splatting on the BCC grid, which was extended to the 4^{th} dimension for time-varying data by Neophytou and Mueller [43]. The shear-warp algorithm was extended to support BCC rendering by Sweeney et al. [57]. Ibáñez et al. [26] used a generalization of the Bresenham algorithm for raycasting on the BCC grid, but they did not present any details about the used interpolation. Strategies for reconstruction in a BCC grid for high-quality raycasting were proposed by Theußl et al. [59]. In this work we present their methods in detail. Dornhofer modified Fourier Domain Volume Rendering (FDVR) from Malzbender [34] for use on a BCC grid [16]. 3D texture-based rendering on BCC grids was first introduced by Röber et al. [49]. Iso-surface reconstruction on the tetrahedral mesh defined by a BCC grid was proposed by Chan

and Purisma [7] and Treece et al. [61]. To cope with the large number of created triangles on such a mesh, Carr al. [6] investigated and compared the marching cubes [33] variants marching tetrahedra, octahedra and hexahedra for BCC grid iso-surface reconstruction.

8.3 Practical Reconstruction Schemes

We developed some general strategies for practical reconstruction on the BCC grid. We inserted them into a raycasting system for high-quality rendering, but they can be used in several BCC rendering algorithm needing an interpolation between sample positions. Some of them are optimized for speed, others for rendering quality, while all preserve reasonable complexity.

Some methods are depending on the current view direction. This is a major drawback, because applications exist which initially do not have a view direction (segmentation, for example). For such methods we must define a virtual shear direction.

8.3.1 Bilinear Interpolation

Bilinear interpolation operates directly on the 2D CC grid planes. Every second plane is translated by half a unit. In a raycaster we must ensure that the entry point of the ray is also on a plane. The resampling plane most perpendicular to the current view direction is chosen. On the BCC grid we get a higher sample frequency than on the CC grid, because the planes are closer together by a factor of $\sqrt{2}$. However, we lose information in the planes, because they consist of half the number of samples than on the CC grid. To further halve the sample distance, we can apply simplified trilinear interpolation directly in between two planes. This interpolation is a specialized version of the sheared trilinear interpolation (see section 8.3.5).

8.3.2 Bilinear plus Spatial Interpolation

Bilinear interpolation can be extended to a real trilinear interpolation on arbitrary resample positions. This can be achieved by using an additional spatial interpolation of two bilinearly interpolation density values on the adjacent 2D CC grid planes. Therefore we refer to this reconstruction scheme as bilinear plus spatial interpolation. We can see the situation in figure 8.1, where the interpolated values from upper and lower plane are denoted as S_i and S_{i+1}, and α refers to the spatial distance from the planes. We calculate the final density value S_α like the following:

$$S_\alpha = (\frac{\sqrt{2}}{2} - \alpha)S_i + \alpha S_{i+1} \qquad \text{for } 0 < \alpha < \frac{\sqrt{2}}{2} \tag{8.1}$$

Like for bilinear interpolation, we choose the stack of CC grid planes that is most perpendicular to the actual view direction. This method results in an interpolation in a sheared cubic cell, where shear directions of the cells are determined by the resample location (shown in the right image of figure 8.1).

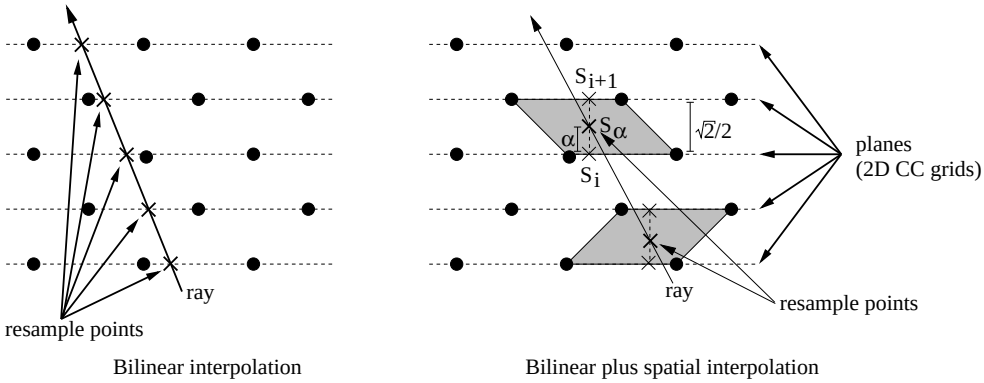


Figure 8.1: Bilinear interpolation operates in the planes (left image). Using a spatial interpolation of the interpolated values S_i and S_{i+1}, we can interpolate on arbitrary positions between the planes (right image).

8.3.3 Barycentric Interpolation

Barycentric interpolation operates on the tetrahedral mesh that is defined by the BCC grid. The barycentric coordinates are calculated and used for an interpolation between the tetrahedron vertices. The interpolation between the vertices is piecewise linear and therefore fast but of limited quality. Additional computations must be done to find the current tetrahedron that contains the resampling point and to calculate the barycentric coordinates:

1. Find the tetrahedron where the resampling point is located in. First we determine the corresponding octant of the cell in the primary (secondary) grid using three comparisons. We need some more comparisons (x greater or smaller y, x greater or smaller z, and y greater or smaller z) to find the current tetrahedron. The situation is visualized in figure 8.2.

2. Compute the barycentric coordinates. This is done by transforming the resampling point into a coordinate system where one vertex of the tetrahedron is in the origin and the others in unit distance on the x, y and z axis. The barycentric coordinates are equivalent to the new location of the resampling point. After translation to the origin, only 12 different types of tetrahedra exist, thus the transformation matrices can be precomputed and stored in a table.

8.3.4 Trilinear Interpolation

In order to use CC grid trilinear interpolation for BCC grid reconstruction, we could resample a large CC grid. Instead, we resample the largest cubic cell that contains no other sample point on the fly and trilinearly interpolate in this cell. The cell is defined by the octant of a primary (secondary) grid CC cell. This octant is the intersection of the two cells from the primary and the secondary grid which contain the current resampling point. Two sample points of this small cubic cell already exist in the grid, the other six must be resampled. The resampling of a cubic cell corner can be done by interpolating between either two or six adjacent sample points (taking four samples is also possible, but yields bad results). Refer to figure 8.3.

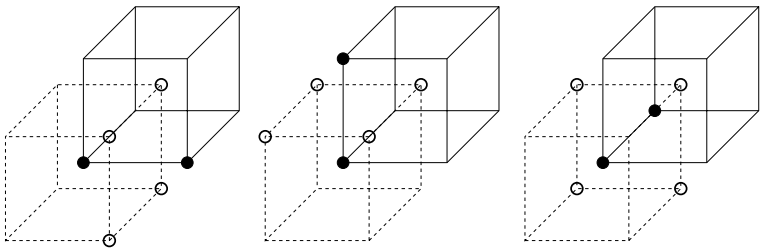


Figure 8.2: The Delaunay tetrahedralization of the BCC grid. Two adjacent points (white dots) together with the two points of the spine (black dots) in *x*, *y*, and *z* direction (from left to right) make up a tetrahedron.

8.3.5 Sheared Trilinear Interpolation

The BCC grid can be seen as sheared and scaled CC grid, where we operate on sheared cubic cells. We can use a special kind of trilinear interpolation in these cells, which we denote as sheared trilinear interpolation. The shear is applied along the two axes that span the 2D CC grid planes, which we refer as shear planes. First we apply linear interpolations along the four short edges of the sheared cubic cell. The resulting interpolated values are referred as *a*, *b*, *c*, and *d* in figure 8.4. In the figure we can also see that the weights used in this interpolation are determined by the distances α and $\frac{\sqrt{2}}{2} - \alpha$ from the adjacent 2D CC grid planes. The scalar values *a*, *b*, *c*, and *d* can then be used for a bilinear interpolation of the final density value. The shear planes and the shear directions of the cells can be chosen freely in this approach. We made following considerations about these parameters:

Shear planes: First we choose the shear planes which are most perpendicular to the actual view direction as shear planes. This is closely connected to the bilinear reconstruction scheme from section 8.3.1. Figure 8.4 illustrates this approach in 2D.

Shear directions: Next, we also choose the shear directions depending on the viewing ray direction. The cells are sheared either into positive or negative directions so that the sheared cell borders are as parallel as possible to the ray. This assures that the ray will pass through the sheared cell as similar as possible as it would pass through the original cubic cells. If the viewing ray pointed slightly towards the right, the shear direction towards the the positive axis would be chosen, as can be seen in figure 8.4.

8.4 Texture-Based Volume Rendering

Texture-based volume rendering is a pure hardware approach, as the entire volume rendering pipeline is delegated to the graphics chip. The based idea is to download the volume to textures. Textured proxy polygons are then blended into the frame buffer back-to-front for semi-transparent volume rendering. There are two main approaches in texture-based rendering, 2D

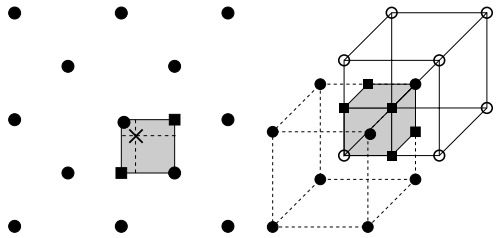


Figure 8.3: Trilinear interpolation in the BCC grid. To interpolate at a position in the shaded box, first the missing corners of the box (square dots) are interpolated.

texture-based rendering using object-aligned slices and 3D texture-based rendering using view-aligned slices (refer figure 8.5). We sketch the adaption of 2D texture-based rendering and 3D texture-based rendering to the BCC grid. Likewise we show the adaption of two important extensions to overcome the limitations of standard 2D texture-based rendering, i.e., multi-texture blending and pre-integration. For the BCC grid approaches we exploit the introduced reconstruction schemes (refer section 8.3).

2D Texture-Based Volume Rendering

2D texture-based rendering exploits 2D texture mapping for a bilinear reconstruction in the planes. The adaption of the standard 2D texture-based rendering is straightforward. We can exploit the fact that a BCC grid can be seen as stack of 2D CC slices. Texture coordinates which are translated by half a negative unit (in texture space) must be assigned to every second slice. On the BCC grid we need a higher number of textured slices than on the CC grid. Hence we get a higher sample frequency in exchange for additional rasterization costs. The textures are only half the size on the BCC grid, which means that details are lost in the slices.

Multi-Texture Blending

The use of multi-texture blending allows intermediate slices on arbitrary positions between the volume planes. Two adjacent 2D textures are blended on a single intermediate slice. The blend factors are equal to the spatial position between the back and front slice. In order to adapt the multi-texture blending approach to BCC grids, we must consider that either front or back slice is from the secondary grid. Therefore we assign texture coordinates which are translated by minus half a unit (in texture space) to either front or back slice. After blending front and back slice texture on the intermediate slice, we achieve a trilinear interpolation. This interpolation is equivalent to the bilinear plus spatial interpolation in a sheared cubic cell (refer section 8.3.2).

Pre-Integration

Pre-integration allows accurate integration of the transfer function without super-sampling. Pre-integrated slabs (i.e., the volume between two slices) are rendered instead of slices. The pre-

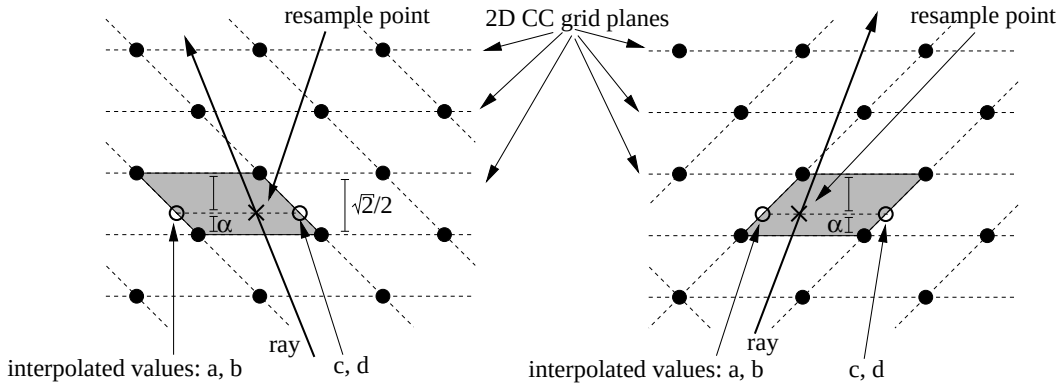


Figure 8.4: 2D visualization of sheared trilinear interpolation in the BCC grid. The interpolated values *a*, *b*, *c*, *d* are used in a bilinear interpolation. The planes and shear direction of the cells are chosen so that they are as closely as possible aligned to the ray direction.

integration table is downloaded to a 2D texture. The scalar values s_f and s_b on the front and back slice are used for a dependent texture lookup. To employ this appoach on a BCC grid, we assign texture-coordinates translated by minus half a unit to either the front or back slice. We have more but thinner slabs on the BCC grid. This results in a higher sample frequency for the integration of the scalar field.

3D Texture-Based Volume Rendering

Röber et al. [49] download the samples from the primary and secondary grid to separate textures for 3D texture-based rendering. Then they trilinearly interpolate twice in each texture and blend the resulting values. However, the interpolations in the primary and secondary grid do not take into account the spatially closest samples.

The hardware uses a hard-wired CC trilinear interpolation inside a 3D texture. To exploit this interpolation for reconstruction in a BCC grid, we can use an alternative calculation of the sheared trilinear interpolation from section 8.3.5. We use the fact that a BCC grid can be seen as sheared and scaled CC grid. The idea is to apply the inverse transformation to the volume and to each resampling location. Then we can use trilinear interpolation in a CC grid. In this approach the shear directions of the sheared cubic cells are determined by the applied indexing scheme. Using the second storage scheme from Theußl et al. [60], it can be verified that we have an alternating positive-negative shear of the cells. The shear direction is changing at each plane of the volume. The inverse shear can be calculated by subtracting the following offset from the *x* and *y* components of a resample location on a given *z* position:

$$o(z) = \min(z - \lfloor z \rfloor, 1 - (z - \lfloor z \rfloor)).$$

The alternative sheared trilinear interpolation is shown in figure 8.6. In order to use this approach for texture-based volume rendering, we update the given texture coordinates (x_t, y_t, z_t) per fragment. We denote the spatial extent of our volume in *x*, *y* and *z* direction as s_x, s_y, and

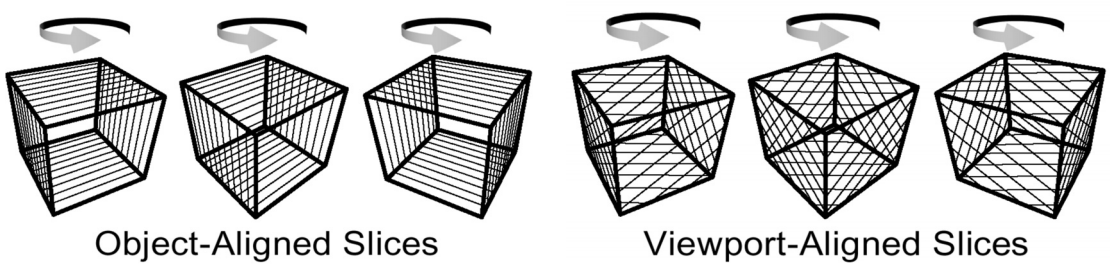


Figure 8.5: Object-aligned slices and view-aligned slices. Image taken from [48].

s_z (using cell spacing $T = 1$). The mapping from texture space (range $[0..1]$) to world space is given by:

$$z = z_t s_z - \frac{1}{4} \tag{8.2}$$

A subtraction of $\frac{1}{4}$ is necessary in above equation, because of the way how the hardware maps indices to texture coordinates. Then the texture coordinates x_t and y_t are updated:

$$x_t = x_t - \frac{o(z)}{s_x} \qquad y_t = y_t - \frac{o(z)}{s_y}$$

The shear directions of the cells are implicitly determined by the used storage scheme. We can manage to set the shear planes so that they are most perpendicular to the current view direction. For this purpose we must rearrange the sample points and store them in three different 3D textures, one for each principal view axis. This is also the main drawback of our approach, as popping artifacts are introduced when the view direction changes.

8.5 Projected Tetrahedra Algorithm

The projected tetrahedra algorithm [54] is a simple and flexible algorithm which uses the properties of a tetrahedral cell. The graphics hardware is exploited to interpolate the scalar function between the vertices. The method consists of the following steps:

1. Decompose the volume into a tetrahedral mesh. Density values are stored at each vertex. The scalar function is assumed to be a linear combination of the vertex values.

2. Depth sort the tetrahedra.

3. Classify tetrahedra and decomposite into triangles according to the projected profile. The two different cases are shown in figure 8.7.

4. Determine color and opacity values at the triangle vertices using ray integration at the "thick" vertex.

5. Rasterize the triangles.

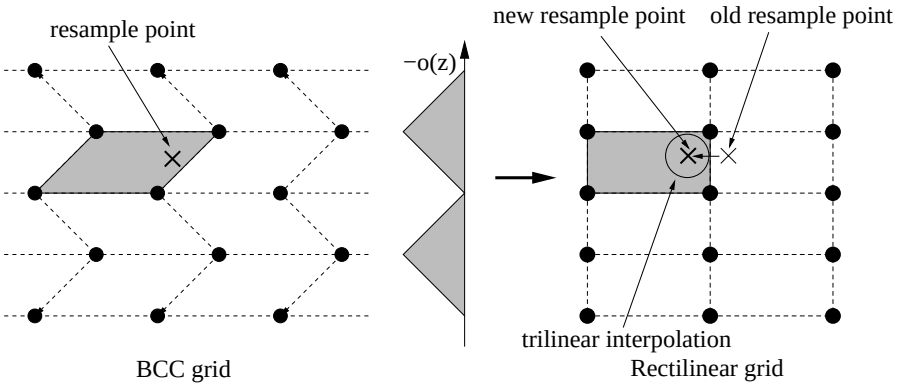


Figure 8.6: Alternative sheared trilinear interpolation in a BCC grid. The inverse shear according to $o(z)$ is applied to the resample point and the BCC grid. Then trilinear interpolation in a rectilinear volume can be used (e.g., for reconstruction in a 3D texture).

Implementation on the BCC grid is straightforward. The tetrahedral mesh is defined by the Delauney tetrahedralization on the BCC grid. We concentrated on the exploitation of the regular grid structure to speed-optimize the projected tetrahedra method on the BCC grids.

Back-to-Front Traversal

On regular grid structures like the CC and BCC grid, we can do the depth-sort step implicitly with a back-to-front traversal of the sample points. The tetrahedra are created on the fly. For this purpose, we define a cell structure that contains all tetrahedra processed in one traversal step, which we denote as traversal complex. This traversal complex is trivially given by a cube on the CC grid. On the BCC grid, it can be verified that we cover all tetrahedra of the grid by traversing only the primary (secondary) grid and tetrahedralizing the three octahedra spanned between the current sample, the adjacent samples of the primary (secondary) grid in positive x, y, and z axis and four samples from the secondary (primary) grid (see figure 8.2 for the location of the tetrahedra). A traversal step is shown in figure 8.8). For depth-ordering octahedra, back-to-front traversal suffices. Thus we process 12 tetrahedra per traversal step (4 per octahedron). Depth-ordering the tetrahedra inside a traversal complex is trivial. It can be done in a pre-processing step for orthogonal projection, because we have only 12 different types of tetrahedra on the BCC grid.

Adjacency Structure for Tetrahedral Grids

To further speed up traversal, we adapted the adjacency data structure from Orchard et al. [44] to store our tetrahedral mesh. This structure was originally proposed to speed up splatting in rectangular grids. It stores only visible voxels (i.e. opacity exceeds a certain threshold). Together with a voxel six pointers to the adjacent visible voxels in all axes are stored, allowing to skip transparent voxels completely. To enable skipping of larger volume regions, a hierarchy of three

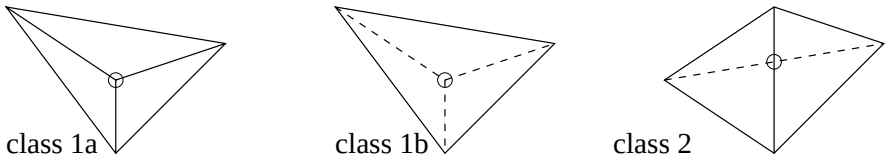


Figure 8.7: Basic tetrahedra decomposition classes. The circle refers to the "thick" vertex.

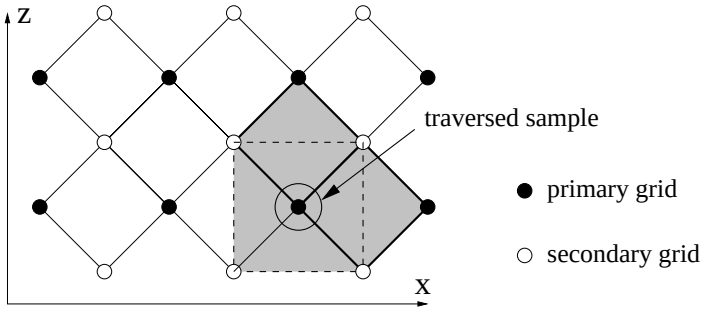


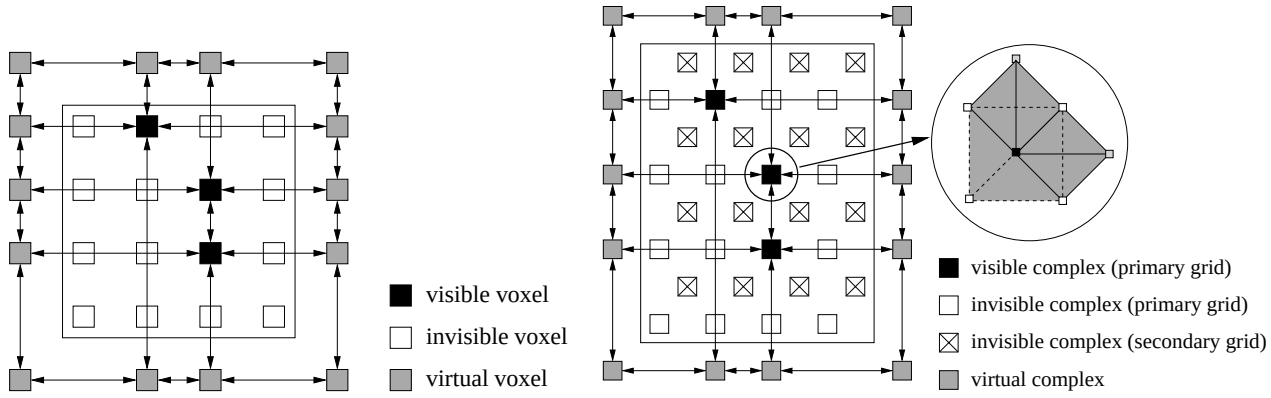
Figure 8.8: Traversal step of the projected tetrahedra algorithm on the BCC grid shown in 2D. In each step three octahedra (shown in grey, the one in the *y* axis outlined with dotted lines) are created with adjacent samples in the directions of the positive *x*, *y*, and *z* axes, then tetrahedralized.

types of virtual voxels is introduced (box corner, box edge and box face voxels), which encapsulate the structure like a box. Box face voxels are stored on each end of visible voxel scanlines. Box edge voxels are capping both ends of non-zero box face scanlines, and box edge scanlines are capped with two box corner voxel (refer to the left image in figure 8.9).

To use the data structure for a tetrahedral mesh, we store traversal complexes instead of voxels (refer to the right image in figure 8.9). A traversal complexes is marked as visible if one of the 12 tetrahedra inside is non-zero. For each tetrahedra of a visible traversal complex, we must explicitely check for visibility. Fortunately, the visibility information can be elegantly stored together with a traversal complex as a checksum, which is the sum of the ID numbers of all visible tetrahedra (i.e., each of the 12 tetrahedra is given an unique ID number). Visibility testing is easily done during traversal by masking this checksum with the unique tetrahedron ID.

Improving the Rendering Quality

For linear transfer functions, we can calculate the correct transparency $1-\exp(-\tau l)$ by applying 2D exponential transparency textures [56]. The extinction coefficient τ and the segment length l are taken as texture coordinates. The most important approach to increase rendering accuracy without sacrificing hardware-acceleration is pre-integration [51]. A 3D texture is used to store pre-integrated ray segments, taking the entry point s_f, the exit point s_b and the segment length l of a ray as parameters. It can be seen as an integration of the transfer function separate from the integration of the scalar field.



Original Structure

Structure for a BCC tetrahedral mesh

Figure 8.9: A 2D image of the adjacency structure. The structure is encapsulated by a box of virtual voxels (traversal complexes) which are only created on demand.

Shading Issues

Directional shading is a very important issue in volume rendering, because it provides important spatial information about the rendered object. For introducing directional shading into the original projected tetrahedra algorithm, we simply store the normals with a vertex and apply standard OpenGL Gouraud shading. It is difficult to use directional shading in combination with pre-integration, because we only have 3 parameters left as indices into a 3D texture. We developed three different techniques with individual strengths and drawbacks:

Pre-integration and Gouraud Shading: To achieve Gouraud shading, we set the vertex color to white and apply the standard OpenGL Gouraud shading. The resulting grey value is then modulated with the 3D pre-integration texture. To preserve the highlights, we must assure that the specular output is added after the modulation.

Pre-integration and Phong Shading: For Phong shading we store the normals in the color portion, in order to get the interpolated gradients in the pixel shaders. The gradient and the output texel of the pre-integration texture can be used for the calculation of the Phong shading equation. If there is no square root operation available in the pixel shaders, we can either use a slow cube map for normalization, or use an approximation formula that is reasonable accurate under the given conditions [52].

Pre-integrated Gouraud Shading [22]: An alternative approach is the implementation of real pre-integrated Gouraud shading, sacrificing a correct opacity calculation for the diffuse and specular term. The per vertex lighting in Gouraud shading is calculated with the approximate Phong equation $I = k_a + k_d(\vec{n} \cdot \vec{l}) + k_s(\vec{n} \cdot \vec{h})^n$. The idea is to split up the pre-integration for the ambient, the diffuse and the specular term [22]. The tables are downloaded to three different 3D textures. The three parameters used for pre-integration of the Gouraud shading terms differ as follows:

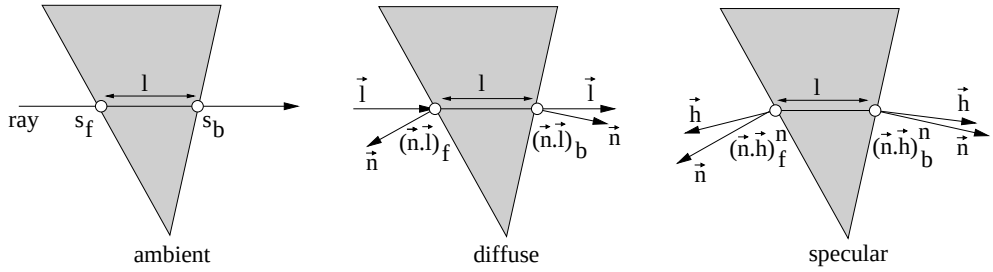


Figure 8.10: The parameters of the terms of Gouraud shaded pre-integration.

Ambient term: s_f, s_b and l.

Diffuse term: $(\vec{n} \cdot \vec{l})_f$, $(\vec{n} \cdot \vec{l})_b$ (i.e dot products on front face f and back face b) and l.

Specular term: $(\vec{n} \cdot \vec{h})_f^n$,$(\vec{n} \cdot \vec{h})_b^n$ and l.

The situation is shown in figure 8.10. These parameters are linearly interpolated over the polygon, thus correct Gouraud shading is calculated. At the end we must sum up the terms in the pixel shaders. As we compute diffuse and specular term independent from the actual transfer function, we manually have to set k_d, k_s, extinction factor τ, and shininess n to a constant value.

8.6 Conclusions

We presented several practical reconstruction strategies on the BCC grid and tested them in a raycasting system. Among them are fast methods with reasonable quality (e.g., bilinear interpolation), and higher quality methods comparable to the CC grid trilinear interpolation (e.g., bilinear plus spatial, sheared trilinear interpolation).

We adapted the major approaches for texture-based volume rendering to the BCC grid, i.e., standard 2D texture-based rendering, 3D texture-texture based rendering, multi-texture blending, and pre-integration.

Further we presented a speed-optimized version of the projected tetrahedra algorithm on the BCC grid by exploiting the regular grid structure and orthogonal projection. We adapted a 3D adjacency data structure originally used for splatting to store a tetrahedral mesh on the CC and BCC grid, which allows fast traversal by skipping large regions of transparent tetrahedra.

We achieved some impressive rendering results. However, we observed a reduced rendering quality on some of our datasets like the Melon dataset (regardless of the used method). The renderings are either not equally sharp and detailed, or otherwise have a rather rough appearance.

The reconstruction schemes are slightly more complex on the BCC grid. We get different measures regarding the performance of texture-based volume rendering, depending on the used method. But we achieved no significant performance gain. The smaller data size contributes to noticeable better rendering rates for the projected tetrahedra algorithm on the BCC grid. This is caused by the fact that the algorithm is a pure object-order technique.

Chapter 9

Conclusions in general

In this chapter we first draw conclusions about the introduced practical reconstruction schemes on the Body-Centered Cubic grid, then about the hardware-assisted methods modified for BCC grid rendering.

9.1 Practical Reconstruction Schemes

We investigated several reconstruction schemes on BCC grids first proposed in [60] for raycasting and extended this work with some new ideas. The implementation of the bilinear interpolation approach on BCC grids is straightforward. This scheme directly interpolates on the slices, and the sample rate has to be chosen accordingly. It is simple and fast, but sometimes prone to artifacts. Caused by under-sampling, the slice planes are clearly visible especially on the side of the volume. However, the sample frequency is still higher on BCC grids than on CC grids, because the planes are closer together by a factor of $\sqrt{2}$. On the contrary, there is a loss of details in the planes compared to bilinear interpolation on the CC grid. Much better results are achieved by substituting the bilinear interpolation with a trilinear one using the bilinear plus spatial interpolation scheme. Barycentric interpolation operates on the natural cell structure of the BCC grids, the tetrahedral cell. As the barycentric coordinates have linear variation inside a tetrahedron, the resulting interpolation is only a linear one, resulting in a bad approximation of the ideal reconstruction filter. Sheared trilinear interpolation provides high quality reconstruction with reasonable speed if the shear planes are chosen properly. Trilinear interpolation on the BCC grid is slow because of the additional computations needed to resample a cubic cell on the fly.

As expected, the bilinear interpolation is by far the fastest reconstruction scheme. Although the barycentric interpolation itself is fast, there is additional overhead of finding the actual tetrahedron and the barycentric coordinates. Both schemes suffer from artifacts. Bilinear plus spatial and sheared trilinear interpolation both yield very similar results. We consider these two schemes to provide the best quality renderings among BCC interpolation schemes. They are comparable to CC grid trilinear interpolation in terms of image quality and rendering time is just a little slower.

There is one additional drawback of all interpolators that operate on or between the 2D CC

grid planes, e.g., bilinear, bilinear plus spatial, and sheared trilinear interpolation. The planes are not uniquely defined and must be chosen depending on the current principal view direction. This leads to popping artifacts when the principal view axis changes. Furthermore, there are applications where we initially do not have a direction (e.g., segmentation). For such methods, we would have to define a virtual view direction. We even have an additional degree of freedom in sheared trilinear interpolation, where we have to determine the shear direction of the cells.

Generally, most images rendered on BCC grids look slightly blurrier, and sometimes reveal less details than the corresponding CC grid rendering. We found out in our experiments that for each reconstruction scheme, we can make out a clear tendency regarding the amount of blurriness it introduces. Barycentric interpolation produces very rough, but on the same time the sharpest results. Next comes trilinear interpolation with 2 samples for resampling of the cubic cell, then trilinear interpolation with 6 samples. Even smoother results are produced by the bilinear plus spatial interpolation scheme. The surface details are least visible with sheared trilinear and bilinear interpolation.

Further we investigated several gradient estimation schemes on BCC grids. Among them are three different adaptions of the central differences approach, the adaptive grey level estimation scheme proposed by Yagel et al. [46] for CC grids, and a version of the 3×3 sobel filter. The central differences 1 and 2 were proposed by Theußl et al. [60]. Central differences 3 is a linear combination of the first two approaches. All presented methods except central differences 3 operate on either the primary or secondary grid. The latter approach also yields the best results among the central differences variations, because the artifacts produced by the first two techniques are smoothed out. As expected, the sobel filter is smoothest. The adaptive grey level estimation produces the sharpest images and preserves most details.

9.2 Texture-Based Volume Rendering

We adapted both major paradigms for texture-based volume rendering to BCC grids: 3D texture-based volume rendering using view-aligned slices and 2D texture-based volume rendering using object-aligned slices textures. Further we modified multi-texture blending and pre-integration for rendering on a BCC grid. Both methods are important extensions to overcome the limitations of 2D texture-based rendering. In the pre-integration approach we render volume slabs instead of slices. The transfer function is pre-integrated in a preprocessing step separately from the integration of the scalar field. This allows to improve the rendering quality without super-sampling. Because of the flexibility of the rendering framework, practically all of the implemented rendering modes can be easily extended to support BCC grids, for example shaded post-classification, using either standard 2D texture-based rendering, multi-texture blending, or 3D texture-based rendering. Also some of the NPR rendering modes of the framework (e.g., tone shading) fully support BCC grid rendering.

2D texture-based volume rendering uses an interpolation which is equivalent to the bilinear reconstruction on BCC grids (explained in section 3.1). 2D texture-based rendering is slower on BCC grids, because more slices must be rendered. We achieve a higher sample frequency on BCC grids, whereas the textures are half the size of CC grid textures.

Similar considerations can be made for the pre-integration approach. On the BCC grid we have more but thinner slabs. This can be considered as an advantage in terms of the rendering accuracy, because the integration of the scalar field can be done with a higher sample frequency than on CC grids.

Multi-texture blending on BCC grids uses a reconstruction which is equivalent to the introduced bilinear plus spatial interpolation scheme from section 3.2. This is a trilinear interpolation in a sheared cubic cell. The multi-texture blending approach produces the best results among all of our BCC grid methods. For some datasets, the results seem to be smoother than the corresponding CC grid rendering. However, for some volumes like the Melon dataset we again observed that details are lost. Although textures are only half of the size on BCC grids, we achieved no significant performance gain.

In order to adapt 3D texture-based rendering using view-aligned slices we used an alternative sheared trilinear interpolation, which applies an inverse shear to the resample point followed by a CC trilinear interpolation. We achieved very good results, despite of the hard-wired trilinear interpolation of the 3D texture hardware. Our method preserves many of the advantages of the original 3D texture-based rendering on CC grids, like a valid trilinear interpolation in a (sheared) cubic cell, a constant slice distance (because of the view-aligned slices) and the ability for super-sampling. Furthermore, it can be implemented quite elegantly into the used framework (section 6.3.1), and almost all rendering modes implemented for the CC grid version can also be used for BCC grid rendering.

On the other hand, our approach has some drawbacks which have to be discussed. Because of the used indexing scheme, an alternating shear direction of the cells is necessary. This can cause some artifacts in form of a visible shear. We know that the shear planes most perpendicular to the actual view direction must be chosen. To achieve this with the alternative sheared trilinear interpolation scheme, three 3D textures must be stored at once (similar to the three slice stacks in the 2D texture approach). This also introduces popping artifacts that otherwise do not happing for view-aligned slices.

Unfortunately, the smaller 3D texture size of approximately 29.3% on BCC grids does not speed up rendering rate as we expected, whereas the required per fragment instructions noticeable slows down the rendering time. This slowdown is far less noticeable if rendering modes are used which need equally or even more expensive fragment programs, like shaded post-classification. As fragment programs will get faster in the next generations of graphic chips, we assume that 3D texture-based rendering on BCC grids will perform much better in the future.

Texture-based volume rendering does not achieve the desired performance gain on BCC grids despite being an object-order algorithm. The reason is that the most significant factor for the performance is the rasterization time. For rasterization, the hardware has to evaluate the contribution to an output image pixel similar to raycasting, which is an image-order algorithm.

9.3 Projected Tetrahedra Algorithm

We did not restrict our work to a simple reimplementation of the original projected tetrahedra algorithm on the BCC grid, which would have been straightforward. Instead, we proposed a

version of the algorithm that is highly specialized to our new grid for providing fast depth-order rendering. To speed up the processing of the visible tetrahedra, we adapted a adjacency structure that was originally proposed to improve performance of splatting on regular grids. The structure contains only visible voxels that are connected by a system of linked list, allowing us to skip large regions in the volume. To use this structure in the tetrahedral mesh derived from the CC and the BCC grid, we defined a structure which we denoted as traversal complex. This is the geometric structure that contains all tetrahedra processed in one traversal step, i.e., a cubic cell on CC grids and three perpendicular octahedra on BCC grids.

Further we implemented and presented some important improvements to the original algorithm regarding performance and rendering speed, like exponential transparency textures and pre-integration. Although directional shading is a very important feature, not much has been published before about shading in tetrahedral cell projection. Therefore we proposed several schemes for introducing shading and combining it with pre-integration. We suggested three different approaches: pre-integration with Gouraud shading, pre-integration with Phong shading and alternatively pre-integrated Gouraud shading. In the latter approach three textures corresponding to the ambient, diffuse, and the specular terms in the shading equation are pre-integrated.

Using the adjacency structure for fast back-to-front traversal, the algorithm yields interactive frame rates for small to medium datasets. Similar to splatting, the algorithm has it's strength when many tetrahedra are invisible and can be discarded. The performance may sometimes exceed that of texture-based rendering if this requirement is met, like for the Fuel or Hipiph dataset. For larger datasets where a tetrahedron covers only some pixels in the output image, the projected tetrahedra algorithm loses much of it's efficiency.

Rendering accuracy is brought to a higher level with the use of pre-integration, although some shading artifacts are still visible. We achieved the best shading quality with the Phong shading approach, which is superior to the slightly duller Gouraud shading method. Pre-integrated Gouraud shading needs careful adjustment of additional parameters, i.e., opacity and color for the integration of the specular and the diffuse term. The shading quality can be greatly enhanced by the application of more sophisticated gradient estimation schemes, like the 3×3 sobel filter. The rendering quality on the BCC grid is very similar to the quality on CC grid. Only slight differences are visible, like stronger Mach band effects on BCC grids. On the other hand we always have less tetrahedra to process on BCC grids, which makes the algorithm noticeable faster.

The one remaining question is: Is this algorithm, which is widely known as an algorithm for the rendering of unstructured grids, an alternative for well established rendering algorithms on regular grids, like splatting? Our answer is: Yes, if pre-integration and the other extensions are implemented.

Chapter 10

Future Perspectives

In this chapter we give an overview of future perspectives concerning the BCC grid. This includes a discussion of still open questions and a list of possible explanations. Afterwards we suggest some future research about the acquisition of more datasets, analysis of the frequency support and high-quality reconstruction. Finally we present some ideas about other methods that could be used to exploit hardware for performance acceleration on BCC grids.

10.1 Open Questions

We already achieved some impressive results on the BCC grid. However, there is still the problem of noticeable reduced rendering quality for some datasets. According to the BCC grid theory, which is profound and well investigated, this should not happen. Rather we would expect equal quality on both grids. For non analytical datasets, the reason could be errors introduced by resampling of the CC volume. However, we made similar observations for datasets generated and sampled directly on both grids, e.g., the Melon or the Device dataset. In fact, the results from these datasets even emphasize the described tendencies, as the gap between BCC grids and CC grids is higher than for most of the resampled volumes. We could not fully find the reasons for this facts in the scope of this thesis , but we are able to give a list of possible explanations that have to be verified in future works:

- We know not much about the process how the melon and device dataset were generated. Some kind of resampling could have happened during the scanning process, causing resampling errors as if the BCC grid datasets were resampled from CC grid datasets.
- Equal resampling quality can only be achieved with ideal reconstruction. We have to investigate higher-order reconstruction schemes for a better approximation of the ideal sinc filter to find out about this matter.
- For non analytic datasets, we implicitly assumed the scalar function to have spherical support, which may not be the case in practice.

- All gradient estimation schemes use either not the actual closest samples or otherwise an average of the closest samples, introducing smoothing.
- Simple schemes like central differences are not able to exploit optimal resampling properties of the BCC grid.
- Interpolation in the natural cell (i.e., the tetrahedron) on the BCC grid has the potential to yield equivalent quality to interpolation in the natural cell on the CC grid (i.e., the cube), but barycentric (= linear) interpolation is not sufficient.
- Following Carr et al. [6], the cell primitives used for reconstruction on the BCC grid (sheared cell, tetrahedron) do not approximate the shape of a sphere as good as a cube. Hence they might not be suited for reconstruction of functions with spherical support.
- If ideal reconstruction is not used, we have errors on both grids. In this case, the BCC grid has it's strength when approximating round structures (Fuel, Hipiph dataset), whereas straight or rough surfaces (Cube, Device dataset) are better approximated on the CC grid.

10.2 Datasets

Much more datasets of different types are required, which allow a fair comparison of rendering methods on both grids without unwanted side effects:

- We need real world datasets directly sampled on the BCC grid, which show a variety of different characteristics, for example regarding physical density.
- There should be an analytic dataset that is the equivalent to the Marschner-Lobb dataset in the field of functions that have spherical support. More specific, the function should be simple but challenging for any reconstruction scheme. The analytic shape of the function must be well known and of a type so that resampling errors are intuitively recognized.
- It should impose no problems to sample simulation data directly on a BCC grid.

10.3 Analysis of the Frequency Domain

We should know more about the spectral properties of the proposed interpolation methods. An evaluation may solve some unanswered questions about reconstruction on BCC grids. Further, we do not have much knowledge about our datasets in frequency domain. We assumed a spherical support, but this has yet to be proven. In case that the spectrum is different from a sphere, it would be very interesting to investigate how the shape of the frequency support affects rendering quality. Theußl [58] suggested the construction of a frequency support renderer for a graphical display of the spectrum. Interestingly, this results in another volume rendering problem.

10.4 High-Quality Reconstruction

We have done much research on the topic of practical reconstruction schemes on BCC grids. However, there is still much more work to do in the direction of higher-order reconstruction (i.e., truncated sinc, cubic filters). It lies in the nature of such schemes that they provide high quality renderings but are impracticable slow. Nevertheless they are of interest in order to learn more about BCC grids.

As we assume a spherical support in the BCC grid, the application of a rotational symmetric filter is the logical choice. This filter takes the Cartesian distance function $\sqrt{x^2+y^2+z^2}$ as input weights for the samples (refer left image of figure 10.1). Unfortunately, this kind of filter is not separable and therefore quite difficult to handle. Another issue is the filter design, as it is challenging to make them interpolating due to the spatial position of the points in the BCC grid. There is one further problem to solve: The spherical filter extend makes it computationally inefficient to process all samples that fall into a rectangular cubic window, like it is usually done for separable filters, because many tested samples would be outside of the filter range.

In a completely different approach, we can achieve high-quality reconstruction by raising the quality of the interpolation in a tetrahedral cell, which seems to be better suited for implementation on BCC grids. It is well known that interpolation in a 4-node tetrahedron is only piece-wise linear. Therefore Williams et al. [69] proposed a 10-node tetrahedral cell (see right image of figure 10.1) in their high-accuracy renderer. This is also called quadratic tetrahedron, because inside of the cell there is a quadratic variation of the scalar field. They further extended the scheme to a cubic tetrahedron. With the price of quickly growing computational expense, tetrahedra of even higher order can be constructed.

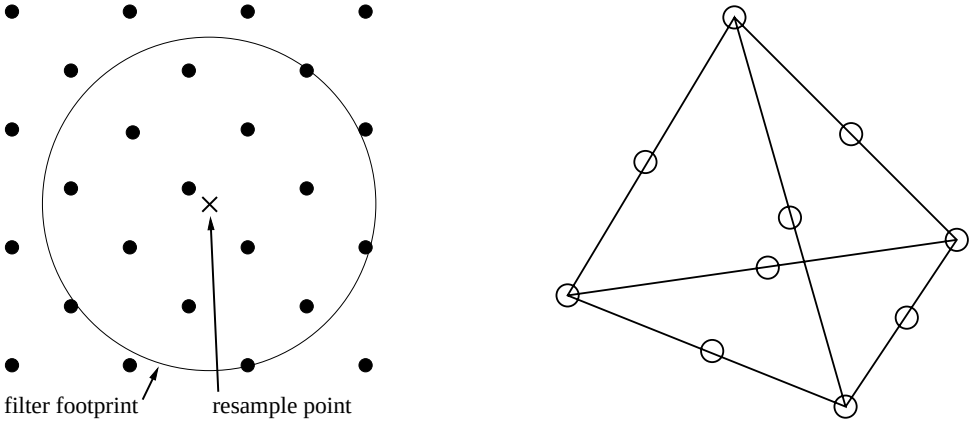
More research must also be done in the field of gradient reconstruction. Higher-order derivate reconstruction schemes like the cubic spline based gradient filter proposed by Bentum et al. [2] have potential to enhance image quality in terms of visible details and sharpness on BCC grids.

10.5 Texture-Based Volume Rendering

After widening our basic knowledge about the application of higher-order filters on the BCC grid, we can think of implementing these filters in hardware following the approach of Hadwiger et al. [24].

We can observe from the texture-based timing tables from section 7.4.2 that we get no real performance gain on BCC grids, and we have a significant slowdown for 2D texture-based volume rendering.

In order to cope with this problem, the principal view axes could be defined to be parallel to the diagonals of the volume faces (instead of principal view axes parallel the volume edges). The situation is sketched in figure 10.2. After the samples are downloaded to the textures accordingly, we can use all techniques for object-aligned slices (and slabs), like 2D texture-based volume rendering, multi-texture blending, or pre-integration. The new approach has several possible advantages regarding performance and even rendering quality over the basic alignment of the principal view axes:



Rotational symmetric filter Quadratic tetrahedron

Figure 10.1: High-quality reconstruction: Convolving the samples inside the range of a rotational symmetric filter on the left (image taken from [59]) and a quadratic (10-node) tetrahedron on the right.

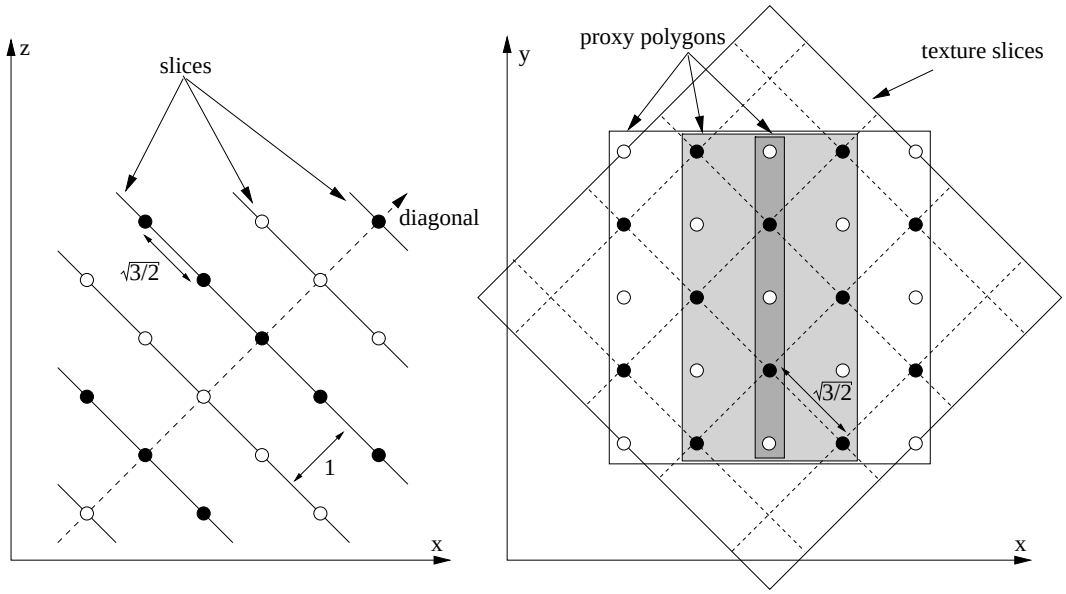


Figure 10.2: Slice planes aligned to the diagonal of a volume face in a BCC grid. Viewed down the y axis (left), and down the diagonal (right). Sample distance is reduced to $\sqrt{3/2}$ and slice distance is increased to 1. The proxy polygon size is varying.

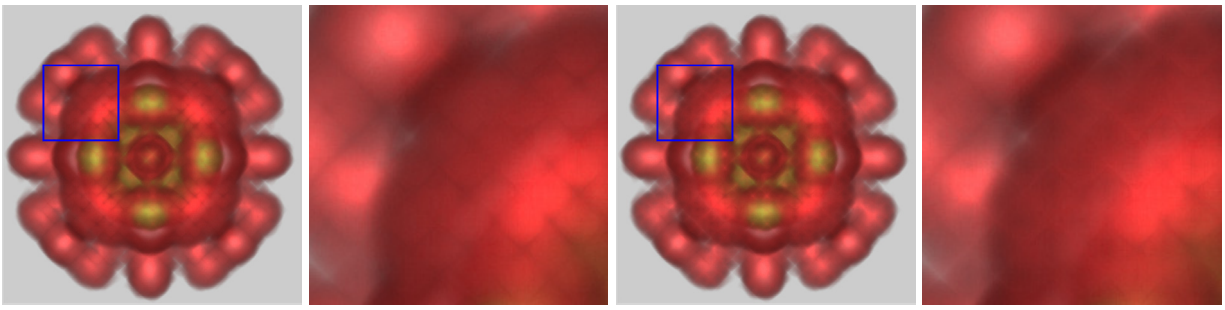


Figure 10.3: The projected tetrahedra algorithm for the the Fuel dataset on the BCC grid. From left to right: Pre-integration without normal weight and a regional zoom, pre-integration with pre-integrated normal weight and a zoom.

- The number of slices in the new principal view directions (defined by the diagonals) is the same as in the equivalent CC volume, which is a reduction of factor $\sqrt{2}$ to the basic approach. On the other hand, the slice distance is increased from $1/\sqrt{2}$ to 1.
- The required 2D texture size is equal to the corresponding CC grid size (see right part of figure 10.2). But if we set the size of each proxy polygon individually in order to tightly fit the sample points into the quadrilaterals (figure 10.2), we could benefit from a much reduced rasterization load for the hardware.
- Inside the slice planes, samples are $\sqrt{3/2}$ apart, as opposed to $\sqrt{2}$ in the original implementation, probably yielding a better resampling quality and an improvement in terms of visible details.
- For four of the six samples needed for the calculation of the central differences, we are able to use the spatially closest samples. This fact can result in a possible enhanced shading quality.

10.6 Cell Projection Algorithms

There is still be potential to enhance rendering quality and shading of projected tetrahedra algorithms without sacrificing performance, by exploiting new capabilities of the hardware. Meißner et al. improved shading in combination with pre-integration for hardware-accelerated raycasting [42] and texture-based volume rendering [39] by pre-integrating a normal weight. The weight is used to interpolate between the back and front normal of a slab. The resulting normal is then applied in the shading equation. This normal much better represents the normal of the pre-integrated slab than just taking the average between front and back normal, for instance. This approach is suitable for the projected tetrahedra algorithm as well [22], using one 3D pre-integration texture for the transfer function and one for the interpolation weight. In figure 10.3, we see that artifacts can appear for a view direction relatively parallel to one of the principal

view axes (first and second image from the left). These artifacts are partly reduced by using a pre-integrated normal weight (first and second image from the right).

As stated in section 2.4.4, Carr et al. [6] gets better performance and rendering quality for the marching octahedra and marching hexahedra than for the marching tetrahedra algorithm. Likewise, we could think of a cell projection algorithm that is centered around an octahedral (hexahedral) cell.

Bibliography

[1] Kurt Akeley. Reality engine graphics. In *Proceedings of SIGGRAPH '93*, pages 109–116, 1993.

[2] Mark J. Bentum, Barthold B. A. Lichtenbelt, and Tom Malzbender. Frequency analysis of gradient estimators in volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):242–254, September 1996.

[3] Christoph Berger. A framework for flexible, hardware-accelerated, and high-quality volume rendering. In *Proceedings of the Central European Seminar on Computer Graphics*, pages 205–214, 2003.

[4] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In Arie Kaufman and Wolfgang Krueger, editors, *Proceedings of the 1994 Symposium on Volume Visualization*, pages 91–98. ACM SIGGRAPH, October 1994.

[5] Hamish Carr, Torsten Möller, and Jack Snoeyink. Simplicial subdivisions and sampling artifacts. In *Proceedings of the IEEE Visualization '01*, pages 99–106. IEEE Computer Society, 2001.

[6] Hamish Carr, Thomas Theussl, and Thorsten Möller. Isosurfaces on optimal regular samples. In *Proceedings of the 2003 Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym 2003)*, pages 39–48, 2003.

[7] S. L. Chan and E. O. Purisima. A new tetrahedral tesselation scheme for isosurface generation. In *Computers & Graphics*, volume 22(1), pages 83–90, February 1998.

[8] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Multiresolution representation and visualization of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):352–369, October/December 1997.

[9] P. Cignoni, C. Montani, D. Sarti, and R. Scopigno. On the optimization of projective volume rendering. In *Visualization in Scientific Computing '95*, pages 55–71. Sixth Eurographics Workshop, Springer Computer Science, May 1995.

[10] João Comba, James T. Klosowski, Nelson Max, Joseph S. B. Mitchell, Claudio T. Silva, and Peter L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured-grids. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pages 369–376. The Eurographics Association and Blackwell Publishers, 1999.

[11] Roger Crawfis. Real-time slicing of data space. In *Proceedings of the IEEE Visualization '96 (Vis '96)*, pages 271–277. IEEE Computer Society, 1996.

[12] Roger Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *Proceedings of the IEEE Visualization '93 (Vis '93)*, pages 261–266. IEEE Computer Society, 1993.

[13] T.J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical Report TR93-027, University of North Carolina, 1993.

[14] Frank Dachille, Kevin Kreeger, Baoquan Chen, Ingmar Bitter, and Arie Kaufman. High-quality volume rendering using texture mapping hardware. In *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH workshop on Graphics hardware*, pages 69–76. ACM Press, 1998.

[15] vuVolume documentation. http://www.cs.sfu.ca/ torsten/volclass/documentation/, 2003.

[16] Alois Dornhofer. A discrete fourier transform pair for arbitrary sampling geometries with applications to frequency domain volume rendering on the body-centered cubic lattice. Master's thesis, Vienna University of Technology, 2003.

[17] D. E. Dudgeon and R. M. Mersereau. *Multidimensional Digital Signal Processing*. Prentice-Hall, Inc., Englewood-Cliffs, NJ, 1st edition, 1984.

[18] Klaus Engel and Thomas Ertl. Interactive high-quality volume rendering with flexible consumer graphics hardware. Technical report, University of Stuttgart, February 2002. STAR.

[19] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, 2001.

[20] Ricardo Farias, Joseph S. B. Mitchell, and Claudio T. Silva. Zsweep: an efficient and exact projection algorithm for unstructured volume rendering. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, pages 91–99. ACM Press, 2000.

[21] Allen Van Gelder and Kwansik Kim. Direct volume rendering with shading via three-dimensional textures. In *Proceedings of the 1996 IEEE Symposium on Volume Visualization*, pages 23–30, 1996.

[22] Stefan Guthe, 2003. Personal Communication.

[23] Stefan Guthe, Stefan Röttger, Andreas Schieber, Wolfgang Strasser, and Thomas Ertl. High-quality unstructured volume rendering on the pc platform. In *Proceedings of the Conference on Graphics Hardware 2002*, pages 119–125. Eurographics Association, 2002.

[24] Markus Hadwiger, Ivan Viola, Thomas Theußl, and Helwig Hauser. Fast and flexible high-quality texture filtering with tiled high-resolution filters. In *Proceedings of Vision, Modeling, and Visualization 2002*, pages 155–162, 2002.

[25] L. Ibáñez, C.Hamitouche, and C.Roux. Determination of discrete sampling grids with optimal topological and spectral properties. In *Procceedings of the 6th International Workshop in Discrete Geometry for Computer Imagery DGCI'96*, pages 181–192, 1996.

[26] L. Ibáñez, C.Hamitouche, and C.Roux. Ray casting in the BCC grid applied to 3D medical image visualization. *Proceedings of the 20th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 20(2):548–551, 1998.

[27] James T. Kajiya and Brian P. Von Herzen. Ray tracing volume densities. In *Proceedings of SIGGRAPH '84*, pages 165–174, 1984.

[28] Mark J. Kilgard. NVIDIA OpenGL extension specifications. Technical report, NVIDIA Corporation, June 2003.

[29] Davis King, Craig M. Wittenbrink, and Hans J. Wolters. An architecture for interactive tetrahedral volume rendering. In *Proceedings of the International Workshop on Volume Graphics 2001*, pages 163–180. Springer-Verlag, 2001.

[30] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94*, Computer Graphics Proceedings, Annual Conference Series, pages 451–458. ACM SIGGRAPH, July 1994.

[31] David Laur and Pat Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. In *Proceedings of SIGGRAPH '91*, pages 285–288. ACM Press, 1991.

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

[33] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of SIGGRAPH '87*, pages 163–169. ACM Press, 1987.

[34] Tom Malzbender. Fourier volume rendering. *ACM Transactions on Graphics (TOG)*, 12(3):233–250, 1993.

[35] Stephen R. Marschner and Richard J. Lobb. An evaluation of reconstruction filters for volume rendering. In R. Daniel Bergeron and Arie E. Kaufman, editors, *Proceedings of the IEEE Visualization '94 (Vis '94)*, pages 100–107, 1994.

[36] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[37] Nelson Max, B. Becker, and R. Crawfis. Flow volumes for interactive vector field visualization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the IEEE Visualization '93 (Vis '93)*, pages 19–24, San Jose, CA, October 1993. IEEE Computer Society Press.

[38] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. In *Proceedings of the 1990 Workshop on Volume Visualization*, pages 27–33. ACM Press, 1990.

[39] Michael Meißner, Stefan Guthe, and Wolfgang Straßer. Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. In *Proceedings of the Graphics Interface 2002*, pages 209–218, May 2002.

[40] Michael Meißner, Ulrich Hoffmann, and Wolfgang Straßer. Enabling classification and shading for 3D texture mapping based volume rendering using openGL and extensions. In David Ebert, Markus Gross, and Bernd Hamann, editors, *Proceedings of the IEEE Visualization '99 (Vis '99)*, pages 207–214, San Francisco, 1999. IEEE.

[41] Michael Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of four popular volume rendering algorithms. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, pages 81–90, October 2000.

[42] Michael Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. Vizard ii: a reconfigurable interactive volume rendering system. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 137–146. Eurographics Association, 2002.

[43] Neophytos Neophytou and Klaus Mueller. Space-time points: 4d splatting on efficient grids. In *Proceedings of the 2002 IEEE Symposium on Volume Visualization*, pages 97–106. IEEE Press, 2002.

[44] Jeff Orchard and Torsten Möller. Accelerated splatting using a 3d adjacency data structure. In *GI 2001*, pages 191–200, June 2001.

[45] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The volumepro real-time ray-casting system. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH '99*, pages 251–260, 1999.

[46] D. Cohen R. Yagel and A. Kaufman. Normal estimation in 3D discrete space. *The Visual Computer*, 8(5-6):278–291, 1992.

[47] David M. Reed, Roni Yagel, Asish Law, Po-Wen Shin, and Naeem Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *Proceedings of the 1996 Symposium on Volume Visualization*, pages 55–62. IEEE Press, 1996.

[48] 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 Stephan N. Spencer, editor, *Proceedings of the 2000 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 109–118. ACM Press, August 2000.

[49] Niklas Röber, Markus Hadwiger, Alireza Entezari, and Torsten Möller. Texture based volume rendering of hexagonal data sets. Technical report, Graphics, Usability, and Visualization (GrUVi) Lab Simon-Fraser University, 2003.

[50] Stefan Röttger and Thomas Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 23–28. IEEE Press, 2002.

[51] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of the IEEE Visualization '00 (Vis '00)*, pages 109–116. IEEE Computer Society Press, 2000.

[52] Gerald Schröcker. Hardware accelerated per-pixel shading. In *Proceedings of the Central European Seminar on Computer Graphics*, pages 233 – 246, 2002.

[53] Greg Schussman and Nelson Max. Hierarchichal perspective volume rendering using triangle fans. In Arie Kaufman, Bill Lorensen, and Klaus Mueller, editors, *Proceedings of the International Workshop on Volume Graphics*, pages 309–320. Springer-Verlag, 2001.

[54] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 63–70, November 1990.

[55] N. J. A. Sloane. The sphere packing problem. In *ICM: Proceedings of the International Congress of Mathematicians*, pages 387–396, 1998.

[56] Clifford M. Stein, Barry G. Becker, and Nelson L. Max. Sorting and hardware assisted rendering for volume visualization. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 83–89. ACM Press, 1994.

[57] J. Sweeney and K. Mueller. Shear-warp deluxe: The shear-warp algorithm revisited. In *Proceedings of the 2002 Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym 2002)*, pages 95–104, Barcelona, Spain, May 2002.

[58] Thomas Theußl. Personal communication.

[59] Thomas Theussl, Oliver Mattausch, Torsten Möller, and Eduard Gröller. Reconstruction schemes for high quality raycasting of the body-centered cubic grid. Technical Report TR-186-2-02-11, Vienna University of Technology, Institute for Computer Graphics and Algorithms, December 2002.

[60] Thomas Theußl, Torsten Möller, and Eduard Gröller. Optimal regular volume sampling. In Thomas Ertl, Ken Joy, and Amitabh Varshney, editors, *Proceedings of the IEEE Visualization 2001 (Vis '01)*, pages 91–98, 2001.

[61] G. M. Treece, R. W. Prager, and A. H. Gee. Regularised marching tetrahedra: improved iso-surface extraction. *Computers & Graphics*, 23(4):583–598, 1999.

[62] Ming Wan, Arie Kaufman, and Steve Bryson. Optimized interpolation for volume ray casting. *Journal of Graphics Tools: JGT*, 4(1):11–24, 1999.

[63] VRVis webpage. http://www.vrvis.at/, 2003.

[64] Manfred Weiler, Martin Kraus, and Thomas Ertl. Hardware-based view-independent cell projection. In *Proceedings of the 2002 IEEE Symposium on Volume Visualization*, pages 13–22. IEEE Press, 2002.

[65] Rüdiger Westermann. The rendering of unstructured grids revisited. In D. Ebert, J. M. Favre, and R. Peikert, editors, *Proceedings of the 2001 Joint Eurographics - IEEE TCVG Symposium on Visualizatation (VisSym 2001)*, pages 65–74. Springer-Verlag, May 2001.

[66] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In Michael Cohen, editor, *Proceedings of SIGGRAPH '98*, pages 169–178. ACM SIGGRAPH, July 1998.

[67] Lee Westover. Footprint evaluation for volume rendering. In *Proceedings of SIGGRAPH '90*, pages 367–376. ACM SIGGRAPH, August 1990.

[68] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. In *Proceedings of SIGGRAPH '91*, pages 275–284. ACM SIGGRAPH, 1991.

[69] Peter Williams, Nelson Max, and Cliff Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, 1998.

[70] Peter L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics (TOG)*, 11(2):103–126, April 1992.

[71] Peter L. Williams and Nelson Max. A volume density optical model. *1992 Workshop on Volume Visualization*, pages 61–68, 1992.

[72] Craig M. Wittenbrink. Cellfast: Interactive unstructured volume rendering. Technical Report HPL-1999-81R1, Hewlett Packard Laboratories, September 1999.

[73] Brian Wylie, Kenneth Moreland, Lee Ann Fisk, and Patricia Crossno. Tetrahedral projection using vertex shaders. In *Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics*, pages 7–12. IEEE Press, 2002.

[74] Yong Zhou, Baoquan Chen, and A. Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. In *Proceedings of the IEEE Visualization '97 (Vis 97)*, pages 135–142. IEEE Computer Society, 1997.

APPENDIX A
List of Abbreviations

API: application programming interface.

BCC: Body-Centered Cubic grid.

CC: Cartesian Cubic grid.

CD: central difference gradient estimation method.

CG: high-Level C-like graphics programming language from NVIDIA.

CT: computed tomography.

FCC: Face-Centered Cubic grid.

FDVR: fourier domain volume rendering.

GPU: graphics processing unit.

GUI: graphical user interface.

HCP: Hexagonal Close Packing grid.

MIP: maximum intesity projection.

MRI: magnetic resonance imaging.

NPR: nonphotorealistic rendering.

PT: projected tetrahedra algorithm.