Fast Visualization by Shear-Warp using Spline Models for Data Reconstruction

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Gregor Schlosser
aus Cosel

Mannheim, 2009

Dekan: Professor Dr. Felix Freiling, Universität Mannheim
Referent: Professor Dr. Jürgen Hesser, Universität Mannheim
Korreferent: Professor Dr. Reinhard Männer, Universität Mannheim

Tag der mündlichen Prüfung: 11.02.2009

Acknowledgements

First of all I am very thankful to Prof. Reinhard Männer the head of the Chair of Computer Science V at the University of Mannheim and the director of the Institute of Computational Medicine (ICM). I am especially grateful to Prof. Jürgen Hesser who gave me the chance to work and to write my dissertation at the ICM. He introduced me into the attractive field of computer graphics and proposed to efficiently and accurately visualize three-dimensional medical data sets by using the high performance shear-warp method and trivariate cubic and quadratic spline models, respectively. During my work at the ICM I always found a sympathetic and open ear with him, he gave me advice and support and in innervating discussions he helped me to master my apparently desperate problems. I am also obliged because of his precious comments regarding the initial versions of this thesis.
I am also very thankful to Dr. Frank Zeilfelder who introduced me into the difficult field of multivariate spline theory. He made the complex theory understandable from a practical point of view. His accuracy and carefulness made me favorably impressed. In this context I also feel obliged to Dr. Christian Rössl for his helpful hints on implementing the Super-Splines.
I would like to acknowledge the work of Sarah Mang and Florian Münz who examined in their thesis a hierarchical volume visualization algorithm and a curvature-based visualization method based on the trivariate spline models, respectively.
The entrepreneurial spirit of all the members in the institute made the work within the group multifarious, interesting, exciting, and pleasant. I would like to thank all members for the inspiring ambience as well the patience. I also would like to apologize for my complex nature. My special thanks go to Amel Guetat, Dennis Maier, Dmitry Maksimov, and Lei Zheng for many enjoyable discussions and the enjoyable time off the job. Many thanks also to Christiane Glasbrenner and Andrea Seeger for the help regarding the administration effort. Furthermore, in this instant I appreciate to Christof Poliwoda and Thomas Günther from Volume Graphics for supplying me with their library which I have used during my work at the ICM. And I apologize all other people I may have forgotten to mention in this context.
Last but not least I would like to thank my uncle, my brother and especially my wife who encouraged me during a period of my life been difficult. Finally, I would like to dedicate this work to my lovingly mother deceased 1994.

Abstract

This work concerns oneself with the rendering of huge three-dimensional data sets. The target thereby is the development of fast algorithms by also applying recent and accurate volume reconstruction models to obtain at most artifact-free data visualizations.

In part I a comprehensive overview on the state of the art in volume rendering is given. Even though a discussion of the whole extent on actual techniques, methods, and algorithms is beyond of the scope of this work, from each to the next chapter techniques necessary to setup a volume rendering framework are discussed. These are, among other topics, data reconstruction models, rendering requirements (as e.g. modeling, illumination, and shading techniques), rendering methods (e.g. ray-casting and splatting), acceleration practices, and non-photorealistic visualization techniques (e.g. using curvature and silhouette enhancement). After the overview part of this thesis every other part includes the related work and a short introduction section into the respective subject. However, because of the comprehensive character of computer graphics this thesis is written in a form that should be understandable for researches with a very different scientific background and should circumvent the reader from consulting the references too much. Therefore the aim is to keep it as self-explanatory as possible and to require only basic knowledge in computer graphics and volume processing. The author would like to apologize some of the readers for reiterating topics in several sections that are already well known in the computer graphics community.

Part II is devoted to the recently developed trivariate (linear,) quadratic and cubic spline models defined on symmetric tetrahedral partitions $\triangle$ directly obtained by slicing volumetric partitions $\diamondsuit$ of a three-dimensional domain. This spline models define piecewise polynomials of total degree (one,) two and three with respect to a tetrahedron, i.e. the local splines have the lowest possible total degree and are adequate for efficient and accurate volume visualization. The spline coefficients are computed by repeated averaging of the given local data samples, whereby appropriate smoothness properties necessary for visualization are fulfilled automatically, i.e. the spline derivatives deliver optimal approximation order for smooth data. The piecewise spline representation admits to exploit Bernstein-Bézier techniques from the field of Computer Aided Geometric Design. In this part, implementation details for linear, quadratic and cubic spline models in Bernstein-Bézier form are given, their assets and drawbacks are discussed, and their application to volume rendering in form of a ray-casting algorithm is presented. Additionally, the counterpart linear and quadratic tensor product spline models and the well known and often applied trilinear models specified on partitions $\diamondsuit$ are discussed for comparison reasons.

The following part III depicts in a step by step manner a fast software-based rendering algorithm, called *shear-warp*. This algorithm is prominent for its ability to generate

projections of volume data at real time. It attains the high rendering speed by using elaborate data structures and extensive pre-computation, but at the expense of data redundancy and visual quality of the finally obtained rendering results. However, to circumvent these disadvantages a further development is specified, where new techniques and sophisticated data structures allow combining the fast shear-warp with the accurate ray-casting approach. This strategy and the new data structures not only grant a unification of the benefits of both methods, they even easily admit for adjustments to trade-off between rendering speed and precision. With this further development also the 3-fold data redundancy known from the original shear-warp approach is removed, allowing the rendering of even larger three-dimensional data sets more quickly. Additionally, real trivariate data reconstruction models, as discussed in part II, are applied together with the new ideas to onward the precision of the new volume rendering method, which also lead to a one order of magnitude faster algorithm compared to traditional approaches using similar reconstruction models.

In part IV, a hierarchy-based rendering method is developed which utilizes a wavelet decomposition of the volume data, an octree structure to represent the sparse data set, the splines from part II and a new shear-warp visualization algorithm similar to that presented in part III. Additionally, the main contribution in this part is another shear-warp like algorithm for hierarchical data structures using algorithms well known in graph theory. This method should show benefits compared to traditional algorithms (cf. part I) where expansive non object-order traversals of the hierarchical data structures have to be performed.

This thesis is concluded by the results centralized in part V. Here the different spline models are discussed with respect to their numerical accuracy, visual quality, and computation performance. It is obvious that the lower the polynomial degree of the considered spline model, the better the performance, but the worse the visual quality of the images obtained by the volume rendering algorithm. However, with the often used trilinear (tensor product) model satisfying visual results are possible, even though they generate small jag-like artifacts. With the trivariate quadratic splines defined on tetrahedral partitions $\triangle$ smooth, more accurate and more natural looking images can be generated. Where the corresponding cubic model should be applied only when zooming into a volume data set, in this case it reveals its potential of generating artifact-free visualizations of the data when only first derivatives are considered for shading. Additionally, the performance of the new shear-warp algorithm with the newly developed data structures is given, where it is shown, that, this method is able to generate qualitative images using the above splines at still interactive frame rates. Finally, the analysis of the hierarchically based algorithm is shown in opposition to the method developed in part III. Further details concerning the splines are given in the appendix.

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Visualisierung von großen drei-dimensionalen Datensätzen. Das Ziel dabei ist es, zum einen schnelle Algorithmen zu entwickeln, und zum anderen auch kürzlich - wie auch neu entwickelte - präzisere Datenrekonstruktionsmodelle für die artefaktfreie Volumenvisualisierung zu verwenden. Damit wird dem Anwender die Möglichkeit geboten zwischen einer hoch akkuraten oder einer schnellen Darstellung der Daten zu wählen.

Das erste Kapitel der Arbeit gibt in komprimierter Form den aktuellen Stand der Technik zum Thema Volumenvisualisierung wieder. Da eine vollständige Diskussion der heute gebräuchlichen bzw. untersuchten Techniken und Algorithmen zu diesem Thema die Größe dieser Arbeit sprengen würde, wird stattdessen dem Leser versucht eine Art "Roten Faden" zu präsentieren, wo zum einen ein Volumen-Rendering System aufgebaut wird, und zum anderen aktuelle Methoden diskutiert werden. Im speziellen werden Themen wie z.B. Datenrekonstruktionsmodelle, Vorgaben zum Rendering (z.B. geometrische Modelle, Belichtungs- und Schattierungstechniken), Rendering Methoden (z.B. Ray-Casting und Splatting), Beschleunigungstechniken, und nicht-photorealistische Visualisierung (z.B. unter Verwendung von Kanten- und Krümmungsverstärkung) behandelt. Nach dem einleitenden Kapitel widmet sich jedes weitere Kapitel einem speziellen Thema, wobei auch dort eine kurze Einleitung in die jeweilige Thematik, wie auch der Stand der Technik, wiedergegeben werden. Diese Herangehensweise wurde gewählt, um auch dem Leser mit einem etwas anderen technischen Hintergrund die Arbeit einfach zugänglich zu machen. Mit anderen Worten, der Text sollte so selbsterklärend wie möglich gehalten werden, und nur Grundlagenwissen im Bereich Computer Graphik und Volumenvisualisierung voraussetzten. Hierfür möchte ich mich bei all denjenigen Lesern entschuldigen, für die einige der Abschnitte lediglich lästige Wiederholungen konstituieren, bzw. dass so manche angesprochene Themen bei Gruppen, die sich mit der Volumenvisualisierung beschäftigen, wohlbekannt sind, und somit lediglich eine Form der Nachbildung darstellen.

Der zweite Teil dieser Arbeit beschreibt die Implementierung der erst kürzlich entwickelten trivariaten (linearen,) quadratischen, und kubischen Bernstein-Bézier Spline Modelle definiert auf symmetrischen Tetraeder Partitionen eines drei-dimensionalen Definitionsbereichs. Diese Art von Splines beschreiben hinsichtlich eines Tetraeders stückweise Polynome von totalen Grad (eins,) zwei, und drei, d.h. die lokalen Splines haben kleinstmöglichen totalen Grad und sind somit angemessen für die effiziente und die artefaktfreie Volumenvisualisierung. Die Bernstein-Bézier Koeffizienten der Splines werden aus den gegebenen Daten mittels entsprechender Mittelungsoperatoren berechnet, so dass die benötigten Stetigkeitsbedingungen der Splines automatisch erfüllt werden. Weiterhin ist die Approximationsordnung der Splines optimal für glatte Daten und es lassen sich Standard Bernstein-Bézier Techniken aus dem Bereich "Computer Aided Geometric Design" anwenden. Dieser Teil der Arbeit diskutiert insbesondere die Implementierungsdetails für diese neuartigen (linearen,) quadratischen, und kubischen Spline Modelle in

Bernstein-Bézier Form, deren Vor- und Nachteile, und deren Anwendung im Rahmen einer Volumen-Rendering Methode, namentlich Ray-Casting. Zum Vergleich werden nicht nur die wohlbekannten linearen und quadratischen Tensor-Produkt Splines definiert auf Einheitswürfeln im drei-dimensionalen Definitionsbereichs gegenübergestellt, sondern auch das oft benutzte Trilineare Standardmodel.

Im dritten Teil der Arbeit wird sowohl schritt für schritt der software-basierte, hocheffiziente Volumen-Rendering Algorithmus namens "Shear-Warp" diskutiert als auch dessen neuartige Erweiterungen, die im Rahmen dieser Arbeit entstanden sind. Bekannt ist der Algorithmus für seine hocheffiziente Realisierbarkeit. Zum einen basierend auf einer Zerlegung der Projektionsabbildung in eine Shear- und eine Warp-Abbildung und zum anderen basierend auf der Laufl̈angenkomprimierung der Daten. Damit ist man in der Lage interaktiv Projektionen aus Volumendaten zu erzeugen. Die Effizienz der Methode wird also unter anderem durch die Vorverarbeitung der Volumendaten in eine visualisierungskonforme Form erzielt, die auch gleichzeitig in ausgearbeiteten und der Methode entsprechenden Datenstrukturen angeordnet werden. Wesentliche Nachteile der ursprünglichen Technik sind deren Visualisierungsqualität und Datenredundanz. Genau hier setzen unsere Neuentwicklungen an, die in diesem Kapitel präsentiert werden. Zum einen wird eine Methode vorgestellt, welche die Kombination vom effizienten Shear-Warp und dem präziseren Ray-Casting erlaubt. Man kann damit zwischen einer effizienteren oder aber einer genaueren Visualisierung der Daten wählen. Die Projektionsqualität wird durch den Einsatz von trivariaten quadratischen und kubischen Datenrekonstruktionsmodellen erhöht, wie sie im zweiten Kapitel diskutiert werden. Zum weiteren kann die 3-fache Datenredundanz der ursprünglichen Implementierung komplett durch den Einsatz unserer neuen Datenstrukturen und Algorithmen vermieden werden. Dies wiederum erlaubt es sehr hoch aufgelöste Volumendaten in immer noch annehmbarer Zeit zu visualisieren.

Im vorletzten Teil der Arbeit wird ein hierarchie-basierter Ansatz zur Volumenvisualisierung diskutiert und implementiert. Hierzu wird als erstes eine hierarchische Wavelet-Transformation zur Datenreduktion durchgeführt, dann werden die reduzierten Daten mittels einer Octree-Datenstruktur repräsentiert, und zuletzt mittels der im Kapitel II entwickelten Splines hierarchisch kodiert und über einen neuen Shear-Warp Ansatz, sehr ähnlich zu dem im Kapitel III besprochenen, dargestellt. Der Hauptbeitrag dieses Kapitels ist jedoch die Diskussion einer weiteren Methode. Auch diese basiert auf der Shear-Warp Zerlegung der Projektionsmatrix und auf hierarchisch angeordneten Volumendaten. Hier könnten Algorithmen, bekannt in der Graphentheorie, zur Anwendung kommen, um die hierarchisch kodierten Daten in einer effizienten, objektorientierten Weise zu durchlaufen und entsprechend der Strahlengleichung zu visualisieren.

Das abschließende Kapitel diskutiert zentral und zusammenfassend die Resultate dieser Arbeit. Zuerst werden insbesondere die verschiedenen approximativen Spline Modelle hinsichtlich ihrer Genauigkeit, der visuellen Qualität, und Berechnungseffizienz untersucht und diskutiert. Hier ist es zwar offensichtlich, dass der Polynomgrad eines Spline Modells sowohl die Performance als auch die visuelle Qualität der Rendering Methode bestimmt, d.h. je geringer der Grad der stückweise definierten Polynome, desto schneller ist zwar der Visualisierungsalgorithmus, aber alsgleich auch schlechter die Approximationsordnung der Splines und damit ungenauer die Visualisierung. Hier sollen die Resultate zum einen helfen die richtige Wahl von Splines bzg. der angestrebten

Visualisierungsqualität zu treffen, und zum anderen auch als Referenz für mögliche weitere Nachfolgeimplementierungen von ähnlichen Methoden dienen, wie z.B. interpolierenden Spline Modellen derselben Ordnung. Zusammenfassend kann man sagen, dass das trilineare Modell (Tensor-Produkt Splines) zufrieden stellende visuelle Resultate mit sehr guter Performance liefern. Beim genaueren Hinsehen sind jedoch kleine Zacken-Artefakte in den Bildern zu erkennen, die insbesondere aufgrund des linearen Charakters des Models entstehen und dort auftreten, wo die korrespondierenden Volumendaten hochfrequente Anteile besitzen. Trivariate quadratische Splines definiert auf Tetraederpartitionen hingegen führen zu glatteren, genaueren, und auch natürlich aussehenden Resultaten. Auch dieses Modell erzeugt Diskontinuitäten im Bild, die aber erst beim hineinzoomen sichtbar werden, und aufgrund der Unstetigkeit der Gradienten dieses Spline Modells, die zwischen einzelnen Tetraedern innerhalb eines Volumenelements auftreten können, entstehen. Hier spielt das gleichartige kubische Spline Modell eine entscheidende Rolle, welches stetig differenzierbar ist, und wenn nur erste Ableitungen für das "Shading" in Betracht gezogen werden, auch bei sehr starker Vergrößerung des Volumens artefaktfreie Visualisierungen der Daten ermöglicht. Schließlich werden noch die Resultate der neu entwickelten Shear-Warp Methode und der neuen Datenstrukturen dargelegt. Insbesondere wird auch gezeigt, dass der neue Algorithmus - basierend auf der Shear-Warp Zerlegung und der Anwendung von unseren neuartigen Splines - immer noch interaktive Bildwiederholraten für Parallelprojektionen ermöglicht und sogar eine Ordnung effizienter ist, als herkömmliche Standardimplementierungen basierend auf der Ray-Casting Methode. Für seine Generalisierung auf Perspektivprojektionen jedoch, konnten die ursprünglichen Probleme, wie Visuelle Qualität und Datenredundanz, beseitigt werden, aber auf Kosten einer sehr langsamen Implementierung. Hier besteht allerdings noch einiges an Potential für eine effizientere Realisierung, welche auch im Text diskutiert wird. Zum Schluss des Kapitels wird noch eine Analyse der hierarchiebasierten Methode in Bezug auf die zuvor diskutierten Techniken gegeben.

Contents

Part I

Physically Based Volume Rendering

1 Introduction

Physically based volume rendering [KvH84] [DCH88] and [Kau91] [WS01] became more and more popular in the late eighties. The purpose was to create a two-dimensional image directly from three-dimensional grids. This technique makes it possible to visualize the whole volume instead of showing only partial views of it, i.e iso-surfaces or streamline extraction. There are different techniques which can be used for volume rendering. They can be split into image-order, object-order, hybrid or domain-based methods. On one side, object-order algorithms use a forward mapping scheme – i.e. volume data is projected onto the screen – where space leaping is applied easily to accelerate the rendering. On the other side, image-order algorithms use a backward mapping scheme, i.e. rays are cast from the eye location through each pixel in the image plane into the grid to determine the final value of the pixel. This technique allows to easily apply early-ray termination for accelerated rendering. Hybrid algorithms often try to combine the advantages of both methods. Finally, domain-based algorithms first transform the volume data from the spatial domain into another domain, i.e. compression, frequency, or wavelet domain and then project the data directly from that domain onto the screen. Further, there are different volume rendering modes, also called *shaders*, e.g. x-ray rendering, maximum intensity projection, iso-surface rendering and full volume rendering. All modes can be combined with any volume rendering technique, e.g. like ray-casting or shear-warp. The difference between the modes is, how the sampled (interpolated) values taken along a ray from the grid (volume data) are combined into the corresponding pixel of the final image. In x-ray rendering the interpolated values (samples) are simply composited, where in maximum intensity projection only the largest interpolated value is written to the image pixel. In iso-surface rendering the first intersection position of the ray and the grid has to be found, where the interpolated value from the grid has a user-defined threshold, then shading is performed at this position and the result is written to the image pixel. In full volume rendering, all interpolated values along a ray taken at specified intervals are further processed to simulate light transport within a volumetric object according to many possible models. In this way, a three-dimensional grid (volume data) is considered as a semi-transparent media and is rendered according to the physics of light propagation [Kaj86] [Sab88] [Kru90] [HHS93]. The desired volume features are often extracted using transfer functions for color and opacity depending on the data. The interpretation of such data fields is considerably difficult because of their intrinsic complexity. Further, the manner in which the discrete pixel values are computed from the three-dimensional continuous function can considerably affect the quality of the final image generated by the volume rendering algorithm. Therefore, the process has to be performed carefully, otherwise artifacs will be present. Fortunately, with more or less amount of additional computation, one can substantially improve the quality of the rendered images. The challenge in volume rendering is due to the size of the data, the high-quality images one would like to obtain and the fact that it has to be visualized in real-time. Various approaches have been developed in the past, where often they try

to take advantage of the underlying computer architecture, deploy efficient compression and caching strategies or declare some oversimplified assumptions on the underlying physical or data model, which mostly lead to low-quality images. This chapter tries to give an overall view (cf. [MPWW00] [MHIL02] [EHKRS02] [KM05] [EHK+05]) of the significant theory, concepts, techniques, and ideas applied to physically based volume rendering.

Figure 1.1: *Volume rendering – e.g. streamline extractions (left), iso-surface rendering (middle), and full volume rendering (right) – reveal different information about the volume data set.*

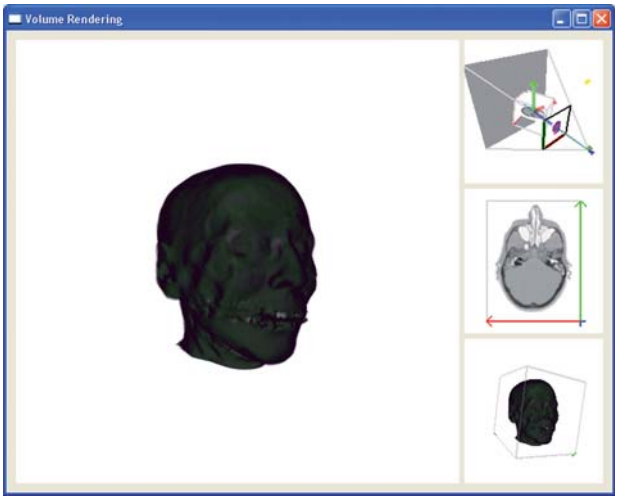


Figure 1.2: *Simple graphical user interface for volume rendering with a model view (top right), a slice view (middle right), object view (bottom right), and the final image (left).*

2 Sampling Theory

Sampling theory is the theory of taking sample values from functions defined over continuous domains, $f(x) \in \mathbb{R}, \quad x \in \mathbb{R}$ and then using those samples $f(nT) \in \mathbb{R}, \quad n \in \mathbb{Z}$ at intervals T to reconstruct a similar continuous function as the original. This theory provides an elegant mathematical framework to describe the relationship between a continuous function and its digital representation (also denoted as signal).
First, *Fourier analysis* provides a well studied theory which can be used to evaluate the quality of the match between the reconstructed and the original function. One of the foundations of the Fourier analysis is the *Fourier transform*, which represents a function in the frequency domain (functions are usually expressed in the spatial domain). Many functions can be decomposed into a weighted sum of phase-shifted sinusoids (sometimes called *Eigenfunctions*) using the Fourier transform. This representation of a function gives insight into some of its characteristics, for example, the distribution of frequencies. The Fourier transform (or *Fourier analysis equation*) of a one-dimensional function $f(x)$ is given by

$$F_f(\omega) = \int_{-\infty}^{+\infty} f(x) e^{-i\omega x} dx \tag{2.1}$$

where $e^{-ix} = \cos x + i \sin x$, $i = \sqrt{-1}$ and the new function F is a function of the frequency ω. The transform from the frequency domain back to the spatial domain is defined by the *Fourier synthesis equation* (or the *inverse Fourier transform*) as

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} F_f(\omega) e^{i\omega x} d\omega. \tag{2.2}$$

The Fourier transform operator $\mathcal{F}\{f(x)\} = F_f(\omega)$ has some useful properties, like e.g. scaling $\mathcal{F}\{af(x)\} = a\mathcal{F}\{f(x)\}, \quad a \in \mathbb{R}$, linearity $\mathcal{F}\{af(x) + bg(x)\} = a\mathcal{F}\{f(x)\} + b\mathcal{F}\{g(x)\}$, multiplication $\mathcal{F}\{f_1(x) f_2(x)\} = \mathcal{F}\{f_1(x)\} \star \mathcal{F}\{f_2(x)\}$, etc. . More about the Fourier transform properties and the relation to signals can be found in every good signal processing textbook (e.g. [FvDFH97] [Mal99] [GW02]).
However, the standard Fourier transform gives a representation of the frequency of a function f, but this frequency information can not be localized in space x. For this reason, a windowed Fourier transform has been introduced which cuts off only a well-localized interval of f first and then takes the Fourier transform of the signal by

$$F_f^{win}(\omega, t) = \int_{-\infty}^{+\infty} f(x) g(x - t) e^{-i\omega x} dx. \tag{2.3}$$

This is a standard technique for time-frequency localization where its discrete version, i.e. by setting $t = nt_0, \omega = m\omega_0$ with $m, n \in \mathbb{Z}$ and $t_0, \omega_0 > 0$ fixed, is even more interesting for signal analysis. However, for compactly supported window functions g with $\|g\| = 1$ and appropriate ω_0, t_0 the Fourier coefficients F_f^{win} are sufficient to reconstruct the

original function f. Many different window functions g have been proposed, where the appropriate ones have mostly compact support and a reasonable smoothness, as, for example, the Gaussian. Hence, in many applications g and its Fourier transform are supposed to be concentrated in time and frequency.
However, it is well known that the sampling theorem [Sha49] gives a sufficient condition on the support of the Fourier transform $F_f(\omega)$ to reconstruct $f(x)$ from its samples $f(nT)$ exactly. Representing a discrete signal by a sum of Diracs, i.e to any sample $f(nT)$ a Dirac $f(nT)\delta(t-nT)$ located at $x = nT$ is associated, a uniform sampling of the continuous function $f(x)$ corresponds to the weighted sum of Diracs $\tilde{f}(t) = \sum_{n=-\infty}^{+\infty} f(nT)\delta(t-nT)$. Relating the Fourier transforms F and $\tilde{F}$ to each other can be used to show the

Theorem 2.0.1 (Sampling Theorem)**.** *If the support of $F_f(\omega)$ is included in $[-\pi/T, +\pi/T]$ then*

$$f(t) = \sum_{n=-\infty}^{+\infty} f(nT)h_T(t-nT) \tag{2.4}$$

with

$$h_T(t) = \frac{\sin(\pi t/T)}{\pi t/T}. \tag{2.5}$$

The sampling theorem tells us that if the Fourier transform $F_f(\omega)$ is band limited, thus $f(x)$ has no high variations between consecutive sample points, we can reconstruct the continuous function $f(x)$ from its samples $f(nT)$ by convolving these samples with the sinc-function $h_T(x)$. In practice *aliasing*[1] and *approximation* errors occur because the condition is often not satisfied. This can reveal itself in many ways, i.e. jagged edges and flickering. These errors occur because the sampling process is not able to capture all of the information from the continuously defined function. Further, the sampling interval T is often imposed by computation and/or storage constraints, therefore $F_f(\omega)$ is not band limited. In this case, the reconstruction formula (2.4) does not recover the original function $f(x)$. There are two ways to solve this problem. First, we can choose a high enough sampling rate (smaller intervals T) if the signal does not have an infinite spectrum. Second, we may filter the signal before sampling to remove all high frequencies above a threshold. Having solved this problem, another is still present, i.e. the support of the sinc-function $h_T(x)$ is infinite that in practice the reconstruction of a function becomes very difficult. This problem can be solved by using only reconstruction functions $\tilde{h}_T(x)$ in Equ. (2.4) with finite support. However, the problem of using an appropriate model for reconstructing the continuous function from its discrete sample points is still a challenging task.

Remark (Sampling Theory)**.** *Infinite sums have been investigated the first time by John Wallis (1616–1703) in his work "Arithmetica Infinitorum". Hence, today one attributes his name to the symbol for infinity (∞). He has also published the "Wallis's Product",*

[1] Aliasing is the phenomenon when high frequencies are masquerading as low frequencies in the reconstructed signal, i.e. the replicated copies of the frequency spectra overlap and during the reconstruction process high-frequency components are mixed in with low-frequency components from the original spectrum.

which can be used to compute the number π approximatively and has introduced the Latin verb "interpolare" (to interpolate) the first time in a mathematical sense. Edmund Taylor Whittaker (1873–1956) has studied the Newton-Gauss interpolation formula for an infinite number of equidistant abscissae on both sides of a given point. He has shown that – under certain conditions – the resulting interpolant converges to the so called "cardinal function". This consists of a linear combination of shifted functions of the form $\sin(x)/x$. Analyzing the frequency content of this interpolant, he has observed that all constituents of period 2ω are absent. Harry Nyquist (1889–1976) has pointed out the importance of sampling a function at twice of its highest frequency writing about telegraph transmission theory. The son of E.T. Whittaker, John Macnaughten Whittaker (1905–1984) has published in his textbook "Interpolatory Function Theory" more refined statements about the sampling theorem, or the "Cardinal Theorem of Interpolation Theory". Claude Elwood Shannon (1916–2001) has presented and prove the well known sampling theorem, referring to the works of J.M. Whittaker and H. Nyquist. He – also called the father of information theory – was the founder of the practical digital circuit design theory. Later, it has been found that similar theorems have been published earlier by Ogura, Kotelnikov (Russia), Raabe (Germany), Someya (Japan), and Weston.

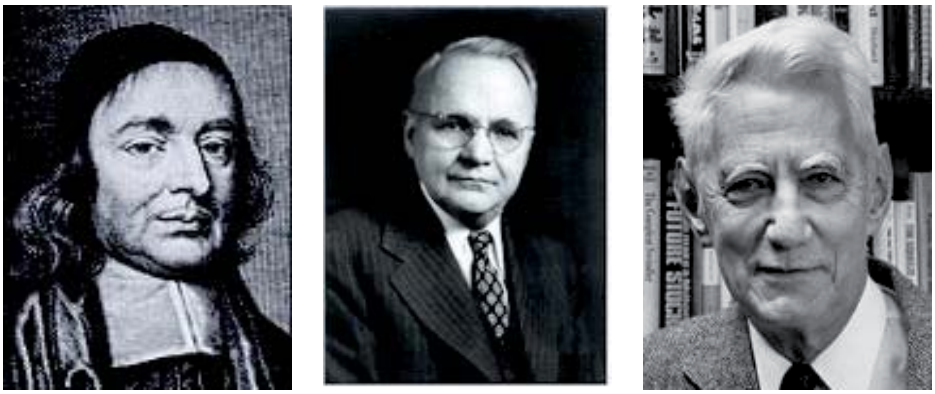


Figure 2.1: *Biographical profiles of John Wallis (left), Harry Nyquist (middle) and Claude Elwood Shannon (right). By courtesy of Wikipedia*

3 Volume Data

In computer graphics, usually three-dimensional surface meshes represented by triangles (polygons) are used to describe the shape of three-dimensional objects but not their internal properties. The triangles of a mesh are rendered by first projecting them onto a plane and afterwards occupy the resulting new triangles according to physical properties at the surface of the objects. This kind of representation and rendering is often used in modern computer games, not only because of the simplicity but also because of the huge hardware support.

In contrast, volume meshes are able to reconstruct physical properties of an object at any three-dimensional point, no matter where the point is located (at the surface or inside the object), i.e. they can be used to describe internal and external properties of solid objects (bones, tissue and skin) at the same time. Further, a volume mesh or grid allows the modeling of fluid and gaseous objects as well, e.g. natural phenomena like clouds, fog, fire and water.

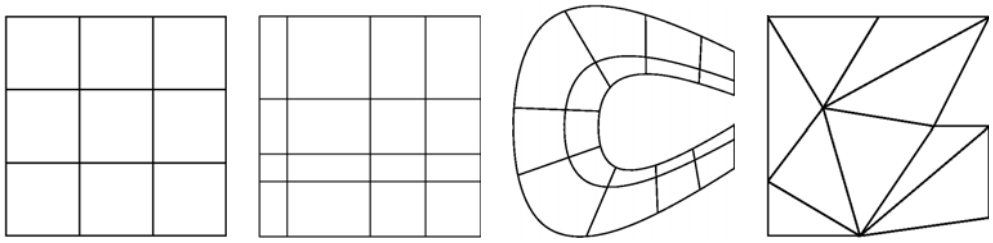


Figure 3.1: *An illustration of rectangular grids (middle left) which are similar to regular grids (left) but the size of the cells can vary within one coordinate system or grid. In structured or curvilinear grids (middle right), the cells are not rectangular any more, but still a regularity can be observed, where unstructured grids (right) have a complete unregular behavior.*

However, discrete volume data sets differ in the structure of the underlying sampling grid and are often classified into structured grids, i.e. a repetition of a structuring pattern can be observed, and unstructured grids, where no a priori organization of the mesh is assumed. In the most general case, heterogeneous grids can be made of arbitrary cells – also called *strongly heterogeneous* structures. In some other cases, there is a limited number of cell types in a grid, for example, tetrahedral and hexahedral cells. This type of grids are called *weakly heterogeneous*. Whereas homogeneous grids are made up of cells of the same type, e.g. from simple cubic or regular tetrahedral cells. Hence, the underlying grid structure stores discrete data values at (non-) equidistant grid positions only and can be considered as a discrete function or data set. An appropriate data reconstruction model for the considered grid has to be applied to extend these discrete data samples into a continuous function.

3.1 Domain Partitions or Grid Structures

3.1.1 Regular Partitions

In practice, due to their simple structure, uniform rectilinear grids (cf. Fig. 3.1) are widely used in various computations and are mostly encountered in indirect acquisition processes, such as magnetic resonance imaging (MRI), computed tomography (CT) and ultrasound (US) in medical imaging applications. However, since the output of such sensors is a continuous waveform whose amplitude and spatial behavior are related to the physical phenomenon being sensed, a regular scalar volume can be interpreted as a continuous three-dimensional function $f(\mathbf{x}) \in \mathbb{R}, \quad \mathbf{x} \in \mathbb{R}^3$. A two-step process, sampling and quantization, allows us to create a digital volume (discrete data set) from the continuous sensed data. In this case, a regular discrete volume data set, $f(\mathbf{k}) \in \mathbb{Z}, \quad \mathbf{k} \in \mathbb{N}^3$, with $\mathbf{k} = \mathbf{n}T, \quad \mathbf{n} \in \mathbb{N}^3$ and the interval T, is simply a three-dimensional array of cubic elements, also called *unit cubes* or *cells*. The voxels (or discrete data values) are located at the centers (or sometime at corners) of the corresponding unit cubes of size $[-0.5, +0.5]^3$. Hence, each voxel represents a constant amount of space as well as information. The information can be, for example, the material or other physical properties of the three-dimensional objects included in the volume. However, on one hand, there is almost no limit how fine we should sample the continuous function. Practical limits are established by imperfections in the optics used to focus on the sensor and the apparent fact of too big data sets. On the other hand, sampling theory (cf. Sec. 2) tells us when a function can be properly reconstructed from its samples. More information about schemes for dividing cubic cells into tetrahedra in three-dimensional space are reviewed in [CMS01], where by using test data geometric artifacts are disclosed, which arise from the subdivision schemes.

3.1.2 Irregular Partitions

Unstructured grids are found in many applications. For sparse data acquisition in geology e.g., one has a non-regular distribution of points that can be dense at one location and very sparse elsewhere. In simulations like in computational fluid dynamics (CFD) or finite element simulations (FEM) one often operates on unstructured tetrahedral grids due to their favorable properties for the numerical solution process, i.e. for stress or deformation computations. Using unstructured meshes allows not only more flexibility in the size of the cells, but also in their shape and topology. In some cases, e.g. for the calculation of the flow distribution around wings one selects cells that are aligned around the wing (curved cells) in order to have better solution properties of the differential equations. Volume rendering on unstructured grids requires only a few data samples, but the algorithms are not yet as efficient as for regular grids. In this thesis, regular grids are considered and discussed only. For references of volume rendering applied to non-regular grids, see e.g. the survey [SCBC05], where real-time rendering of large unstructured meshes is discussed in the context of graphics processing units (cf. also [CLP02] [CICS05]). In [RZNS04b] an algorithm for approximating huge general volumetric data sets is discussed. The data on arbitrarily shaped volumes is represented by cubic trivariate splines, i.e. piecewise polynomials of total degree three defined with respect to (w.r.t.) uniform type-6 tetrahedral partitions of the volumetric domain.

3.2 Reconstruction Models

3.2.1 Spline Models

In practice, often regular grids of discrete data values are given. The ideal three-dimensional sinc filter is usually replaced by either a box, tent or gaussian filter to reconstruct a function $\tilde{f}(\mathbf{x})$, which approximates or interpolates $f(\mathbf{k})$ at the grid points, and returns interpolated values otherwise, such that $\tilde{f}(\mathbf{x})$ approximates the total original function, i.e. $\tilde{f}(\mathbf{x}) \approx f(\mathbf{x})$. However, the choice of an interpolation filter impacts the analytic properties of the resulting continuous volume, i.e. nearest neighbor interpolation (box filter) is not continuous, hence it is not Lipschitz and results in sharp discontinuities between neighboring cells and a rather blocky appearance. The well known trilinear (tent filter) interpolation is continuous but it is not smooth across cell boundaries, that causes problems for typical root finding algorithms, because they require first derivative continuity. That means, piecewise trilinear filtering of rectilinear data interpolates the eight data values at the corners of a cell or a unit cube and yields an overall C^0-continuous approximation of a field. The resulting reconstruction of the normals have discontinuities on the shared boundary faces of each pair of neighboring cells. The resulting normal discontinuities are clearly visible in iso-surface as well as full volume rendering. Nevertheless, with some additional effort this method represents a good trade-off between computational cost and smoothness of the output signal. The next possible choice of functions for volume rendering are C^1-continuous piecewise triquadratic [MJC01] [BMDS02] filters which utilize the 27 data values in the local 3^3 neighborhood of the volume grid. This kind of functions generate smooth images at the cost of increased computation time. Hence, piecewise tricubic [LHJ99b] interpolation, where 64 data values of the local 4^3 neighborhood are utilized, is not suitable for the implementation of a fast rendering algorithm anymore. This is also one reason why in [LHJ99b] the intersection of a ray and the iso-surface is simplified by rotating and resampling the original data grid into another regular grid where the slices are parallel to the image plane. Trivariate spline models defined on type-6 tetrahedral partitions are another choice for volume reconstruction. They are defined analogously to the bivariate splines on a four-directional mesh in the two-dimensional space. However, in [RZNS03] [RZNS04a] [NRSZ04] a new approach to reconstruct non discrete models from gridded volume samples using quadratic trivariate *Super-Splines* on a uniform tetrahedral partition is discussed. A completely symmetric way using local averaging of data samples determines the approximating quadratic *Super-Splines*. Appropriate smoothness conditions are automatically satisfied. On each tetrahedron of the partition the quasi-interpolating spline is a polynomial of total degree two, and provides several advantages. The quadratic *Super-Splines* have appropriate smoothness properties necessary for the visualization process and the derivatives of the splines yield optimal approximation order for smooth data, while the theoretical error of the values is nearly optimal because of the averaging. Efficient computation using Bernstein-Bézier techniques (well known in CAGD [Far86] [Chu88] [HL93]) can be applied to compute and to evaluate the trivariate spline values and their gradients. Further, the volume data can be visualized efficiently, since along a ray the trivariate splines become univariate piecewise quadratic functions. Thus an exact intersection for a prescribed iso-value can be computed in an analytic way, further the volume rendering integral can be efficiently ap-

proximated. Other more detailed overviews of reconstruction filters, methods as well as their properties applied to signal processing can be found in [MN88] [UAE93a] [UAE93b] [ML94] [Dod97] [MMMY97b] and [LGS99] [TMG01]. Popular gradient estimation techniques from, for example, volume data are discussed in [MMMY97a] [BLM97] as well as in [NCKG00]. A method exploiting graphics hardware in order to achieve hardware accelerated high-quality filtering with arbitrary filter kernels can be found in [HTHG01] [HVTH02]. The evaluation of the filter convolution sum is reordered to accommodate the way the hardware works. Recently, a third order texture filtering has been presented [SH05]. Moreover, reconstructions with high smoothness are discussed in [MMK+98] [LM05], a mathematical framework using NURBS was developed in [MC01], and trivariate Coons patches were proposed in [HN00].

3.2.2 Radial Basis Functions

Radial Basis Functions (RBF) [CHH+03] [JWH+04] have been used to procedurally encode irregular grids of scalar data sets, where common choices are thin-plate splines, multi-quadrics or Gaussian functions. Mostly Gaussian functions are used, because of local support, robustness and regular and smooth behavior outside the fitting domain. This RBF encoding generates a complete, unified, functional representation of volume data, independent of the underlying topology. The original grid can be completely eliminated and during rendering only the hierarchical RBF representation is needed. Recently, this work has been extended for encoding vectors and multi-field data sets [WBH+05]. Efficient feature detection techniques were developed and the ability to refine regions of interest in the data. Further, graphics hardware was used to accelerate the reconstruction, rendering, and feature detection from this functional representation.

4 Rendering Requirements

The purpose in volume rendering is to directly create a two-dimensional image from a three-dimensional grid. Hence, except an accurate reconstruction model for volume data – as discussed in the previous sections – we need routines and functions allowing the manipulation of the geometry of objects, a possibly physically correct illumination model, and an accurate solution to the volume rendering integral. Further, mapping functions are used for a simple classification of data, where enhancement techniques, different shading models, and segmented data are often applied for a better understanding of the physical properties of objects. Some of these topics are discussed in the following sections.

4.1 Viewing and Modeling

In volume rendering often a three-dimensional world is defined first by a world coordinate system where objects with some geometrical properties are located. More specifically, a three-dimensional *world* or *scene* is usually defined by a Cartesian coordinate system. Within this coordinate system objects as, for example, lights, cameras, data sets (i.e. volume or polygonal data sets) and/or clipping planes are placed at arbitrary positions. For this, each object has usually an associated *affine* transformation which transforms that object from its local modeling (object) system into the world system by using homogenous coordinates. However, compared to the simpler two-dimensional viewing process the complexity of three-dimensional viewing comes in part by the added dimension and in another part by the fact that display devices are two-dimensional only. The mismatch between a three-dimensional world or an object and a two-dimensional display is resolved by introducing a *projection* which transforms an object onto a projection plane. Hence, the three-dimensional viewing process involves usually different coordinate systems where the corresponding (affine) transformations (or matrices) map coordinates from one system to another system. In this thesis we deal with the *object*, *world/scene*, *camera/eye*, *clip*, and *viewport* coordinate system, where the corresponding matrices $\mathbf{M}_{ow}$, $\mathbf{M}_{we}$, $\mathbf{M}_{ec}$, and $\mathbf{M}_{cv}$ define the transformations, respectively (cf. Fig. 4.1, 4.2, 4.3). The above mentioned transformations are used in our volume rendering algorithms, hence, we go into more detail and define each matrix separately. However, general discussions and transformations for viewing in three-dimensional space can be found in every good computer graphics book. We refer the interested reader to [FvDFH97] [WNDS99] [HZ03] [PH04].

4.1.1 Modeling Transformations

Modeling transformations of objects can be performed by any general matrix $\mathbf{M}$ which represents a valid (perspective) transformation of a coordinate system, i.e. if $\mathbf{M}$ is nonsingular and thus $\mathbf{M}^{-1}$ can be computed. However, in computer graphics often only

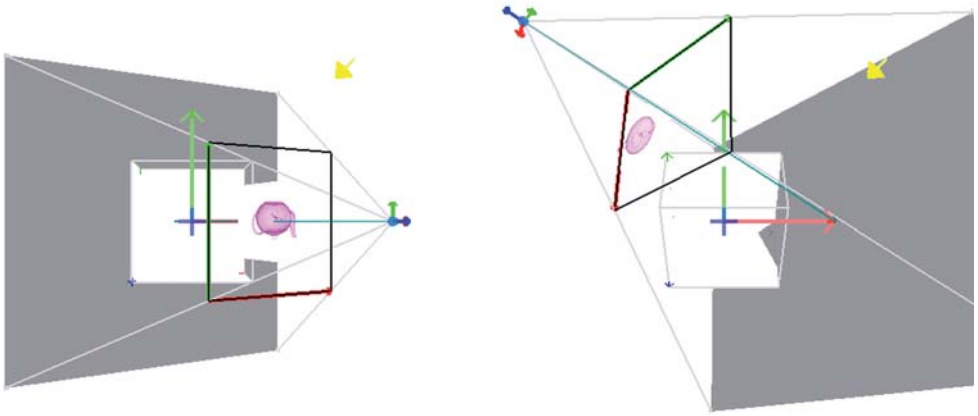


Figure 4.1: *Two different world to eye (camera) transformations* $\mathbf{M}_{we}$ *as well as object to world transformation* $\mathbf{M}_{ow}$ *are illustrated (the red, green, and blue arrows indicate the* x*,* y*, and* z *axis of the appropriate Euclidian coordinate system, respectively). Left: The parameters are* $\mathbf{eye} = (1,0,2)^\mathsf{T}$*,* $\mathbf{cen} = (0,0,0)^\mathsf{T}$*, and* $\mathbf{upv} = (0,1,0)^\mathsf{T}$*. Right: The parameters are* $\mathbf{eye} = (-1,1,2)^\mathsf{T}$*,* $\mathbf{cen} = (1,0,0)^\mathsf{T}$*, and* $\mathbf{upv} = (1,0,0)^\mathsf{T}$ *where additionally the object is rotated around the* x *axis by* $45°$*.*

a few (affine) matrices are defined which are particularly useful for transforming objects. In other words, the most basic transformations of objects are translations, scalings, and rotations. In the right hand coordinate system the particular translation and scaling matrices are defined as

$$\mathbf{M}_T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ and } \mathbf{M}_S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The inverse matrices $\mathbf{M}_T^{-1}$ and $\mathbf{M}_S^{-1}$ are obtained by substituting $-t_x, -t_y$, and $-t_z$ for t_x, t_y, and t_z and by substituting $1/s_x, 1/s_y$, and $1/s_z$ for s_x, s_y, and s_z, respectively. Of course s_x, s_y, and s_z have to be all nonzero. The rotation matrices about the x, y, and z axis of the current considered coordinate system are usually computed by

$$\mathbf{M}_{R_x} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & b & -a & 0 \\ 0 & a & b & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \mathbf{M}_{R_y} = \begin{pmatrix} b & 0 & a & 0 \\ 0 & 1 & 0 & 0 \\ -a & 0 & b & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \text{ and } \mathbf{M}_{R_z} = \begin{pmatrix} b & -a & 0 & 0 \\ a & b & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

with $a := \sin\alpha$, $b := \cos\alpha$, and $\alpha \in [0, 2\pi]$. The rotation about an arbitrary vector $\mathbf{v} := (x,y,z)^\mathsf{T}$ with $\mathbf{u} = \mathbf{v}/\|\mathbf{v}\| := (x',y',z')^\mathsf{T}$ can be defined as

$$\mathbf{M}_{R_v} = \begin{pmatrix} \mathbf{M}_{3\times3} & \mathbf{0} \\ \mathbf{0}^\mathsf{T} & 1 \end{pmatrix} \text{ with } \mathbf{S}_{3\times3} = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

and $\mathbf{M}_{3\times3} = \mathbf{u}\mathbf{u}^\mathsf{T} + b(\mathbf{I} - \mathbf{u}\mathbf{u}^\mathsf{T}) + a\mathbf{S}_{3\times3}$, where $\mathbf{0} := (0,0,0)^\mathsf{T}$ is the null vector, $\mathbf{I}$ the identity matrix and a, b as defined above. Note, here $\star^\mathsf{T}$ is the transpose operator.

However, $\mathbf{M}_{R_v}$ is always defined, the inverse $\mathbf{M}_{R_v}^{-1}$ as well as $\mathbf{M}_{R_x}^{-1}$, $\mathbf{M}_{R_y}^{-1}$, or $\mathbf{M}_{R_z}^{-1}$ can be obtained by substituting $-\alpha$ for α, and if $x = y = z = 0$ then all rotation matrices above become the identity matrix $\mathbf{I}$. When $x \neq 0$ and $y = z = 0$, $y \neq 0$ and $x = z = 0$, or $z \neq 0$ and $x = y = 0$ then instead of using $\mathbf{M}_{R_v}$ the special matrices $\mathbf{M}_{R_x}$, $\mathbf{M}_{R_y}$, or $\mathbf{M}_{R_z}$ can be considered only. However, more complex transformations can be defined by concatenations of the above matrices. For example, the object to world transformation $\mathbf{M}_{ow}$ usually places the corresponding object into the center of the world and is defined by a scaling, translation, and rotation matrix, i.e. as $\mathbf{M}_{ow} := \mathbf{M}_{R_v}\mathbf{M}_T\mathbf{M}_S$. The appropriate coefficients t_i, s_i with $i \in \{x, y, z\}$ depend on the size of the object where the axis $\mathbf{v}$ and rotation angle α are usually chosen by the user. Another example is the so called *look-at* or *modelview* matrix $\mathbf{M}_{we}$ which transforms world into camera or eye coordinates. It is important to place the camera such that the object is located within the viewing frustum (also called *view volume*) of the camera otherwise the object will be not visible. One can apply any of the above transformations to compute $\mathbf{M}_{we}$ or use the so called *look-at* matrix, i.e. $\mathbf{M}_{we} := \mathbf{M}_{lookat}$, where

$$\mathbf{M}_{lookat} = \begin{pmatrix} x_1 & y_1 & z_1 & 0 \\ x_2 & y_2 & z_2 & 0 \\ -x_3 & -y_3 & -z_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.1}$$

This matrix is determined from the viewer's eye position (or camera location) $\mathbf{eye} := (x_e, y_e, z_e)^\mathsf{T}$, the center position $\mathbf{cen} := (x_c, y_c, z_c)^\mathsf{T}$ (i.e. where the viewer is looking at), and the so called *up* vector $\mathbf{upv} := (x_u, y_u, z_u)^\mathsf{T}$ which defines the orientation of the viewer or camera in the three-dimensional world. Whereby the vectors $\mathbf{v}_i := (x_i, y_i, z_i)^\mathsf{T}$ with $\mathbf{v}_i = \mathbf{v}_i'/\|\mathbf{v}_i'\|$ for $i = 1, 2, 3$ are determined from $\mathbf{v}_3' = \mathbf{cen} - \mathbf{eye}$, $\mathbf{v}_2' = \mathbf{v}_3 \times (\mathbf{upv}/\|\mathbf{upv}\|)$, and $\mathbf{v}_1' = \mathbf{v}_2 \times \mathbf{v}_3$ (cf. Fig. 4.1).

4.1.2 Projection and Viewport Transformations

In three-dimensional viewing the projection matrices $\mathbf{M}_{ec}$ define the so called *view volume* in the world. All objects in the three-dimensional world are clipped against this view volume and are then projected onto the projection plane (see Fig. 4.2). In other words, the objects are transformed from world to eye and from eye (camera) coordinates to clipped coordinates where they are checked against the *view volume* and are afterwards projected into the projection plane. In eye (camera) coordinates the position of the camera (eye) is always located at the center of the world, i.e. the condition $\mathbf{0} = \mathbf{M}_{we}\mathbf{eye}$ should be satisfied. The projection matrices are defined as (other projection types, as e.g. the weak-perspective projection, can be found in [HZ03])

$$\mathbf{M}_{ortho} = \begin{pmatrix} +2/(r-l) & 0 & 0 & +(r+l)/(r-l) \\ 0 & +2/(t-b) & 0 & +(t+b)/(t-b) \\ 0 & 0 & -2/(f-n) & -(f+n)/(f-n) \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ and}$$

$$\mathbf{M}_{frustum} = \begin{pmatrix} +2n/(r-l) & 0 & +(r+l)/(r-l) & 0 \\ 0 & +2n/(t-b) & +(t+b)/(t-b) & 0 \\ 0 & 0 & -(f+n)/(f-n) & -2fn/(f-n) \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

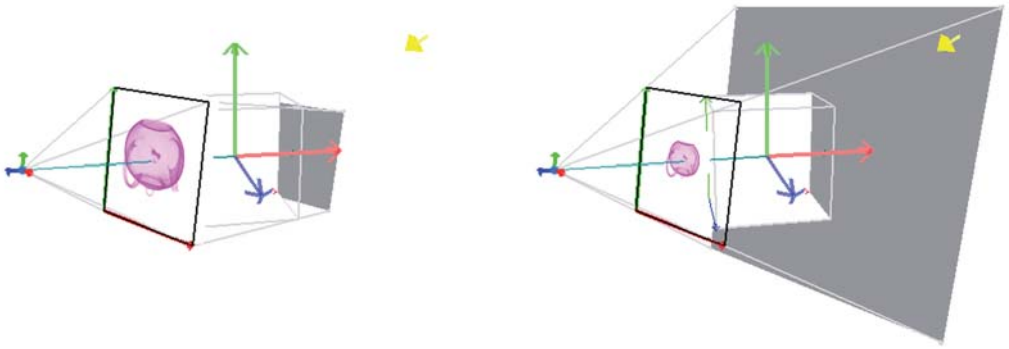


Figure 4.2: *Two different eye (camera) to clip transformations* $\mathbf{M}_{ec}$ *are illustrated. An orthographic projection matrix* $\mathbf{M}_{ec} := \mathbf{M}_{ortho}$ *(left) and a perspective projection matrix* $\mathbf{M}_{ec} := \mathbf{M}_{frustum}$ *are considered for rendering where the parameters each time are set to* $n = 1, f = 3, l = -0.5, r = +0.5, b = -0.5$*, and* $t = +0.5$*. Note, even though for an orthographic projection the eye is located at the infinity, in this illustration it is shown in world coordinates at the same position as it is defined in the perspective projection case.*

and determine the so called *view volume* by the parameters n, f, l, r, b, and t which are the *n*ear, *f*ar, *l*eft, *r*ight, *b*ottom, and *t*op clipping values. After the application of, for example, $\mathbf{M}_{ec} := \mathbf{M}_{ortho}$ (i.e. after transformation into clipping coordinate system) the near, far, left, right, bottom, and top values are located at $z = -1$, $z = +1$, $x = -1$, $x = +1$, $y = -1$, and $y = +1$, respectively. Now, after the perspective normalization, i.e. $(x/w, y/w, z/w, w/w)^{\mathsf{T}}$, the transformed object's coordinates can be easily clipped against these values and transformed onto the projection plane located at $z = -1$. The contents of the plane at $z = -1$ are transformed afterwards from the so called *window* into the display or image using the viewport matrix $\mathbf{M}_{cv} := \mathbf{M}_{viewport}$ with

$$\mathbf{M}_{viewport} = \begin{pmatrix} w/2 & 0 & 0 & w/2+x \\ 0 & h/2 & 0 & h/2+y \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where x, y define the lower left corner of the viewport and w, h specify the dimension of the viewport in the final display (or image).

4.2 Illumination Models

Many illumination models originate from the visualization of three-dimensional surface meshes (triangulations) which represent shapes of three-dimensional objects by hundreds of polygons. These models often describe some properties of light sources and object surfaces and how the interact with each other. However, directly illuminated objects are represented with warmer colors which include yellow, orange and red. This type of lighting is associated with light that comes directly from natural light sources which

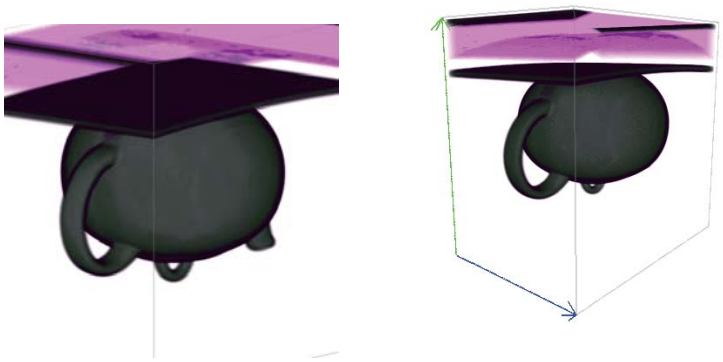


Figure 4.3: *Same configuration as in Fig. 4.2. The final images are obtained by transforming the contents of the projection planes (black rectangles in the above illustrations) by the viewport matrix* $\mathbf{M}_{cv}$ *into the final display of size* 512×512*. The parameters for the viewport matrix are* $x = y = 0$ *and* $w = h = 512$*. Hence, the viewport covers the whole display or final image.*

tend to emit warm light, like candles or the sun. Ambient lighting is typically shown using cooler colors like blue or purple. This is motivated by the fact that reflected light tends to be cooler in nature, like the light received from a blue sky rather than directly from the sun. In our algorithms, we have applied the well known Phong illumination model which is discussed next, afterwards we give also some references of other models.

4.2.1 Phong Model

The visual result of the image depends heavily on the illumination model [1] used in equation (4.6), i.e. the amount of light reflected to the viewpoint from a visible point on the surface as a function of the direction and strength of the light source, the position of the viewpoint and the orientation and properties of the surface. The Lambert's cosine law is a simple illumination model, where the intensity of the reflected light is proportional to the dot product of the surface normal and the light direction. An improved model was introduced in [Pho75], where ambient light, diffuse reflection and specular reflection are considered. This well known Phong illumination model at a specified position t is

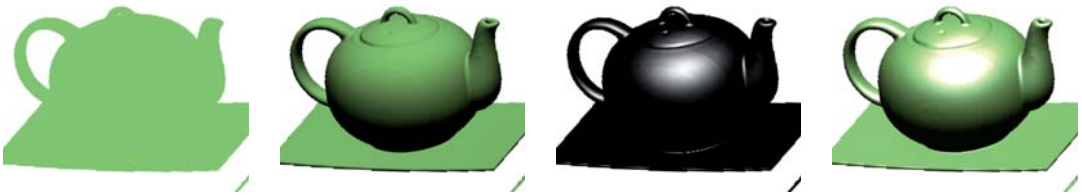


Figure 4.4: *The Phong illumination model. Teapot data set rendered (from left to right) with (constant) ambient, diffuse, and specular reflection, respectively. The right most image shows the different reflections combined by appropriate coefficients.*

$$q_\nu(t) = k_a L_{a\nu} O_{d\nu} + f_{at} L_{p\nu} \left[k_d O_{d\nu} (\mathbf{n}(t)\mathbf{l}(t)) + k_s O_{s\nu} (\mathbf{r}(t)\mathbf{v}(t))^n \right], \tag{4.2}$$

[1] Don't confuse the illumination model with the shading model.

where $q_\nu(t)$ is the resulting intensity (or color), k_a, k_d, and k_s are the ambient, diffuse, and specular reflection coefficients, $L_{a\nu}$ and $L_{p\nu}$ are the intensity of ambient light and the point light source's intensity, and $O_{d\nu}$ and $O_{s\nu}$ are the object's diffuse and specular colors. Further, f_{at} is the light-source attenuation factor, dependant on the distance of the light source from the current sample point. Finally, $\mathbf{n}$, $\mathbf{l}$, $\mathbf{r}$, $\mathbf{v}$ and n are the surface normal, light source direction, reflection direction of the light source, viewpoint direction, and the specular-reflection exponent, respectively. Where ν denotes the red, green and blue components of the color. However, this color model or its simplifications (e.g. omitting light intensities and object colors) are often used in volume rendering applications, even though it is a surface based model. In [Whi80] an extended model was proposed, where reflections, refractions, and shadows arising from interactions between objects in the scene are modeled and in [CT82] [FvDFH97] a more extended discussion of light models can be found.

4.2.2 Other Models

A technical illustration drawn by a human provides different geometric information compared to a Phong illuminated image. In [GGSC98] a non-photorealistic lighting model is presented that attempts to narrow this gap. This lighting model uses luminance and changes in hue to indicate surface orientation, it allows shading to occur only in midtones such that edge lines and highlights remain visually prominent (cf. also [BGKG05]). Other parametric models which are used to recover the Bidirectional Reflectance Distribution Function (BRDF) of three-dimensional non Lambertian surfaces can be found in [Geo03].

4.3 Volume Rendering Integral

The fundamental concept of many physically based rendering methods is the transport theory of light [Bli82] [Kaj86] [Sab88] [Kru90] [HHS93] [Max95]. Thus, volume rendering approaches in general use an emission absorption model, that leaves scattering out of account. Further, frequency dependance (variable ν) can also be safely ignored, i.e. $q_\nu(t) := q(t)$. Thus, using an emission-absorption model $\mathbf{n} \cdot (\nabla I) + \kappa\ I = q$, with $\mathbf{n}$ parallel to $\mathbf{r}$, $\|\mathbf{n}\| = 1$ and $I(0) = 0$, has the following analytic solution

$$I(\rho) = \int_0^\rho q(t)\ e^{-\sigma(0,t)} dt, \tag{4.3}$$

where I is the intensity at the position ρ along a ray $\mathbf{r}(\rho) = \mathbf{r}_s\ \rho + \mathbf{r}_d,\ \rho \in [0, B]$, in three-dimensional space ($\rho = B$ is the location of the background), q is a scattering function that is often identified with the Phong illumination model (cf. Sec. 4.2), and σ is the optical depth defined as

$$\sigma(t_1, t_2) = \int_{t_1}^{t_2} \kappa(\tau)\ d\tau, \tag{4.4}$$

where κ is the opacity function. In order to compute the integral equation, the interval $[0, B]$ is subdivided into small not necessarily equidistant subintervals $[t_k, t_{k+1}],\ k =$

$0, \ldots, N-1$, where $t_0 = 0$ and $t_N = B$, and hence

$$\begin{aligned} I(B) &= \sum_{k=0}^{N-1} \left(\int_{t_k}^{t_{k+1}} q(t)\ e^{-\sigma(t_k,t)} dt \right) \prod_{j=0}^{k-1} e^{-\sigma(t_j,t_{j+1})} \\ &= \sum_{k=0}^{N-1} C_k \prod_{j=0}^{k-1} (1-\alpha_j), \end{aligned} \tag{4.5}$$

whereby the voxel color C_k is

$$C_k = \int_{t_k}^{t_{k+1}} q(t)\ e^{-\sigma(t_k,t)} dt, \tag{4.6}$$

and the voxel opacity α_k is

$$\alpha_k = 1 - e^{-\sigma(t_k,t_{k+1})}. \tag{4.7}$$

The sum in (4.5) can now be rewritten into a recursive front-to-back compositing equation

$$\begin{aligned} \tilde{C}_k &= \tilde{C}_{k-1} + (1-\tilde{\alpha}_{k-1})\ C_k, \\ \tilde{\alpha}_k &= \tilde{\alpha}_{k-1} + (1-\tilde{\alpha}_{k-1})\ \alpha_k, \qquad k = 1, \ldots, N-1, \end{aligned} \tag{4.8}$$

where $\tilde{C}_k$ is the pixel color, $\tilde{\alpha}_k$ is the pixel opacity, and $\tilde{C}_0 = C_0, \tilde{\alpha}_0 = \alpha_0$. Obviously, $\tilde{C}_{N-1} = I(B)$. In order to compute a visualization of the whole voxel grid, the volume rendering equation is applied to each ray $\mathbf{r}$ cast into the volume. We can consider the volume as particles with certain mass density values, where all values in the integral (4.3) can be derived from the interpolated volume density function $f(\mathbf{x})$. The particles can contribute light to the ray in different ways, i.e emission [Sab88], transmission and reflection. Thus, an implementation of ray-casting would traverse the volume grid in front-to-back order, interpolate the density values and gradients at the sampling locations using the volume grid values (discrete data values), compute the color using Equ. (4.6) and opacity using a mapping function (transfer function) and Equ. (4.7), and finally weighting these values by the current accumulated transparency $(1-\tilde{\alpha}_{k-1})$ and adding them into the accumulated color $\tilde{C}_k$ and opacity $\tilde{\alpha}_k$ buffers, to prepare for the next sampled value along the ray. A nice property of this front-to-back composting is that the computation of the ray can be stopped once the accumulated alpha value $\tilde{\alpha}_k$ approaches a user-defined threshold, which means that light resulting from objects further back is completely blocked by the accumulated opaque material in front. This results in a standard acceleration technique called *early-ray-termination* (cf. Sec. 6). However, the volume rendering integral can be realized by many different means. The most common way is described above, i.e. the application of the front-to-back compositing equation (4.8) (also called *over-operator*) with different approximations or simplifications of equations (4.6), (4.7), and (4.2). However, the utilization of a data reconstruction model allows us often (in case of low order reconstructions, hence for low polynomial degrees) to define the formulas $q(t)$, κ, and σ in an explicit manner by using the form and values of reconstructed polynomial pieces. Then, the previously mentioned equations (4.6), (4.7), and (4.2) are well defined as well. Even though the color (4.6) have to be

evaluated by numerical approximations, since in general there exists no closed formula for the resulting integral because its integrand is the product of a polynomial with $e^{-\sigma}$. For the implementation of the approximative integrals associated with the voxel colors different integration rules can be applied, e.g. *Simpson integration rule* or *Gaussian quadrature rules*. Note that only quadrature rules which are accurate for exponential functions should be used.

4.4 Mapping Functions

Another important issue in volume rendering is the application of transfer functions ϕ (also called *mapping functions*). This allows users to interactively change the properties of the volume data set, e.g specifying semi-transparent (or transparent) material by mapping appropriate volume density values f to opacity values $g < 1.0$ (or $g = 0.0$). However, usually one or two-dimensional tables are used. Here, density values $f(\mathbf{r}(t_k))$ (typically in the range $[0, 255]$ for 8bit volume data) and gradient magnitudes $\|\nabla f(\mathbf{r}(t_k))\|$ along rays $\mathbf{r}$ at the sample positions t_k are used to determine optical properties of the data according to some mapping functions ϕ. Since the reconstructed densities $f(\mathbf{r}(t_k))$ are usually no integer values, a simple floor operator, i.e. $i = \lfloor f(\mathbf{r}(t_k)) \rfloor$, provides us with the index i into transfer function to obtain the opacity values $g := \phi(i)$. This is known as nearest neighbor interpolation. However, higher interpolation or approximation schemes can be applied here as well, e.g. univariate or bivariate spline models for one or two-dimensional transfer tables, respectively (cf. references in Sec. 3).
In the previous section, the volume data has been assumed as classified, i.e. in a pre-processing step the grid data values (densities) $f(\mathbf{k})$ are mapped by using transfer function(s) ϕ into opacity values $g(\mathbf{k})$ located on the same grid positions. The three-dimensional opacity function g (opacity volume) can be represented by trivariate piecewise spline models and the volume rendering integral could be evaluated according to the formulas (4.8), (4.6), (4.7), and (4.2). Since an evaluation of these equations is quite expensive, approximations are often used. A zero order approximation of the integral would be to sample the pre-classified volume g on equidistant positions $..., t_k, t_{k+1}, ...$ with intervals $d := t_{k+1} - t_k$ along a ray $\mathbf{r}$ and compute the contributions, i.e. considering the opacity along the ray as $\kappa = g|_{\mathbf{r}}$. Hence, the opacity and color could be computed by

$$\begin{aligned} \alpha_k &:= d\ g(\mathbf{r}(t_k)) \\ C_k &:= d\ q(t_k)\ \alpha_k. \end{aligned} \tag{4.9}$$

The absorption α_k is obtained from (4.7) by using $e^x \approx 1 - x$ and sampling the three-dimensional opacity function $g(\mathbf{x})$ along a ray. Similarly, the color C_k is computed from (4.6) by ignoring the self-attenuation (i.e. the exponential term) at first, sampling, and evaluating equation (4.2), i.e. $q(t) := q_\nu(t)$, for each color component ν. Note that all variables of (4.2) depending on the ray sampling position t should be computed appropriately. Here, the normal $\mathbf{n}(t)$ is set to the opacity gradient at the sample position t_k as $\mathbf{n}(t) := \nabla g(\mathbf{r}(t_k)) := \nabla g(\mathbf{x})|_{\mathbf{r}(t_k)} := (\partial g(\mathbf{x})|_{\mathbf{r}}/\partial x_1, \partial g(\mathbf{x})|_{\mathbf{r}}/\partial x_2, \partial g(\mathbf{x})|_{\mathbf{r}}/\partial x_3)^\mathsf{T}$. Finally, the simulation of self-attenuation can be realized by multiplying the color (the scattering part $q(t_k)$) with α_k. This approach is considered as the pre-classified volume rendering model.

However, volume data can be rendered using the well known pre-classified or post-classified rendering model. In the first approach, densities from the grid positions are mapped to colors and opacities prior to interpolation. Hence, in a pre-processing phase the colors are computed at each grid point by using the illumination equation (4.2), whereas the opacities are obtained from the density values at the grid locations using an one-dimensional (or two-dimensional) opacity transfer table ϕ. Then, during rendering the color vector and opacity values are interpolated and composited along a ray [Lev88] [Lev90]. This model leads to blurry images under magnification [MMC99]. In the second method the raw volume values (normal vectors and density values) are interpolated first, the interpolated samples along the ray are used to look up the color and opacity values, or to evaluate equation (4.2), and are finally composited into the image. The advantage of this model is that it generates sharper images, nevertheless post-classification does not solves all problems that come with high frequency transitions in the transfer functions. A narrow peak in the transfer function could be missed due to sampling the values at some sampling distance d along the ray. This makes pre-integrated transfer functions [EKE01] very interesting (cf. also Sec. 6) where the problem is solved by pre-computing a two-dimensional look-up table which saves the results of the volume rendering equation. Another important class of transfer functions for scalar data are three-dimensional tables [KKH01] based on data values, gradient magnitudes, and second directional derivatives (cf. also [Kin98] [KD98], and [KKH01]).

Figure 4.5: *Different settings of the opacity transfer function for full volume rendering. A linear function is used for rendering which also suppresses soft tissues, i.e. low density values are set to be fully transparent (left). The opacity function is set to make bones semi-transparent and to suppress soft tissues as before (right).*

4.5 Curvature and Silhouette Enhancement

Many applications in volume rendering rely on illumination models (cf. Sec. 4.2), where, first-order differential structure of the volume field (the gradient) is used to compute surface-based lighting results. Further, using second-order derivatives to compute cur-

vature information of the volume field gives additional insight into the underlying volume field [MH92], which comes from the geometry orthogonal to the iso-surfaces. In [MBF92] [MB95] Gaussian kernels, their first, second, and third derivatives were used to compute surface curvature characteristics. Ridge and valley lines were used in [IFP95] [IFP96] to simplify the representation of a surface in surgical planning. A perceptual motiva-

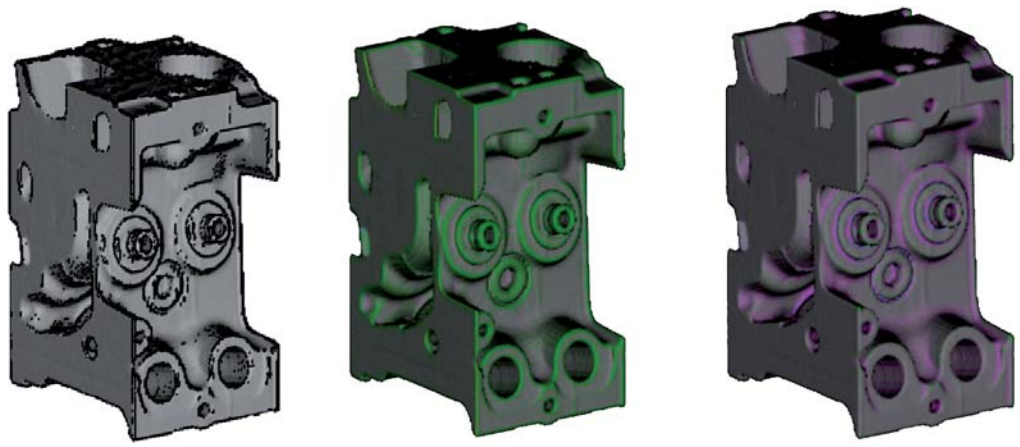


Figure 4.6: *Volume rendering using lookup tables and curvature information computed from our cubic type-6 spline model (cf. part II). Left image shows contours of the engine by using* λ_v *and* $\mathbf{v}^\mathsf{T}\mathbf{n}$ *as index into a two-dimensional lookup table with the thickness parameter* $\tau = 2.0$*. Middle and right images show curvature enhancement by using* λ_1 *and* λ_2*, respectively.*

tion and artistic inspiration for defining a stroke texture that is locally oriented in the direction of greatest normal curvature was given in [IFP97]. The aim is to use transparency for depicting multiple surfaces in a single image. In [Int97] a set of principal directions and principal curvatures are computed to define a natural flow over the surface of an object, and they can also be used to guide the placement of lines of a stroke texture that seeks to represent a three-dimensional shape in a perceptually intuitive way. In [HKG00] a concept of transfer function modification for direct volume rendering is presented, where transfer functions are defined in the domain of principal curvature magnitudes. Recently, a visualization technique (kinetic visualization) [LSM03] has been applied in volume rendering, that uses motion along a surface to assist in the perception of three-dimensional shape and structure of static objects. An intuitive method [KWTM03] to enhance ridges, valleys, and silhouettes of objects have been applied for iso-surface visualizations by computing curvature information from the volume data and using multi-dimensional transfer functions. High-quality curvature measurements using a combination of an implicit formulation of curvature with convolution-based filter reconstruction of the volume field has been proposed. Further, different high-order filters are compared with reference to the quality and accuracy of its reconstructed first and second derivatives. Finally, curvature-based transfer functions are used to extend the expressivity of volume rendering. This is shown by three different applications, i.e. non-photorealistic rendering, surface smoothing via anisotropic diffusion, and visualization of iso-surface uncertainty.

4.5.1 Curvature Estimation

The curvature of a surface [MH92] [KWTM03] is defined by the relationship between small positional changes on the surface, and the resulting changes in the surface normal. In volume data sets surfaces are implicitly represented as iso-surfaces of the reconstructed continuous data values $f(\mathbf{x})$. If in CT scans values of f increase when moving further inside objects, then the surface normal is defined as $\mathbf{n} = -\mathbf{g}/\|\mathbf{g}\|$ (otherwise the normal has a different sign, i.e. $\mathbf{n} = +\mathbf{g}/\|\mathbf{g}\|$) with the gradient $\mathbf{g} = \nabla f(\mathbf{x}) = (\partial f(\mathbf{x})/\partial x_1, \partial f(\mathbf{x})/\partial x_2, \partial f(\mathbf{x})/\partial x_3)^\mathsf{T}$, where $\mathbf{x} := (x_1, x_2, x_3)$ and $\|\mathbf{g}\| = \sqrt{g_1^2 + g_2^2 + g_3^2}$. Curvature information is contained in

$$\nabla \mathbf{n}^\mathsf{T} = -\frac{1}{\|\mathbf{g}\|}\mathbf{P}\mathbf{H} \tag{4.10}$$

with $\mathbf{P}$, and $\mathbf{H} \in \mathbb{R}^{3\times3}$ are the symmetric projection matrix which spans the tangent plane to the iso-surface, and the symmetric Hessian matrix, respectively. The expression $\mathbf{P} := (\mathbf{I} - \mathbf{n}\mathbf{n}^\mathsf{T})$ projects onto the orthogonal complement of the span of $\mathbf{n}$, i.e. the tangent plane to the iso-surface at the current considered point location, where $\mathbf{I}$, and $\mathbf{n}\mathbf{n}^\mathsf{T} \in \mathbb{R}^{3\times3}$ are the identity matrix, and a linear operator that projects onto the one-dimensional span of $\mathbf{n}$, respectively. From vector calculus we know that the Hessian matrix $\mathbf{H}$ encodes the changes of the gradient $\mathbf{g}$ according to infinitesimal changes of the position $\mathbf{x} \in \mathbb{R}^3$. Hence, $\mathbf{H}$ encodes changes along the gradient $\mathbf{g}$ (the length) as well as changes within the tangent plane (the direction). For curvature computations only the changes within the tangent plane are relevant and can be isolated by multiplication of $\mathbf{P}$. The scaling factor $-1/\|\mathbf{g}\|$ converts infinitesimal changes of the (un-normalized) gradient $\mathbf{g}$ into infinitesimal changes of the unit-length normal $\mathbf{n}$. However, it can be shown that the restriction of $\mathbf{H}$ to the tangent plane $\mathbf{P}$ is symmetric (cf. Equ. (4.10)). Hence, there is an orthonormal basis $\{\mathbf{v}_1, \mathbf{v}_2\}$ for the tangent plane which can be extended to an orthonormal basis in $\mathbb{R}^3$, i.e. $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{n}\}$. In this basis the derivative of the surface normal is

$$\nabla \mathbf{n}^\mathsf{T} = \begin{pmatrix} \lambda_1 & 0 & \sigma_1 \\ 0 & \lambda_2 & \sigma_2 \\ 0 & 0 & 0 \end{pmatrix}. \tag{4.11}$$

Here, $\mathbf{v}_1$ and $\mathbf{v}_2$ are the principal curvature directions and λ_1 and λ_2 are the principal curvature magnitudes. Since changes of position along the normal direction cannot change the length of the normal, the third row in equation (4.11) is zero. Where changes along the curvature directions make changes to the normal by $\lambda_{1,2}$. However, multiplying equation (4.10) by P from the right,

$$\mathbf{G} = \nabla \mathbf{n}^\mathsf{T}\mathbf{P} = -\frac{1}{\|\mathbf{g}\|}\mathbf{P}\mathbf{H}\mathbf{P} = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \tag{4.12}$$

has the effect of isolating $\lambda_{1,2}$ in the basis $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{n}\}$. This geometry tensor has a general form in practice according to the Euclidian coordinate system, i.e. concerning to the basis $\{(1,0,0),(0,1,0),(0,0,1)\}$. Here, a singular value decomposition of $\mathbf{G}$ will give the appropriate principal curvature directions (Eigenvectors) as well as the magnitudes (Eigenvalues). For more information see [KWTM03].

4.5.2 Silhouette Estimation

Many non-photorealistic approaches apply contour or silhouette enhancement by showing the transition between front and back-facing surfaces. For polygonal surfaces the sign of the dot product $\mathbf{v}^\mathsf{T}\mathbf{n}$ between the view direction and the normal of a polygon is used to find the orientation of a face [GG01]. Hence, if two faces with normals $\mathbf{n}_1$ and $\mathbf{n}_2$ have different visibility (one is oriented towards the user and the other not, respectively), then the dot products become $\mathbf{v}^\mathsf{T}\mathbf{n}_1 > 0$ and $\mathbf{v}^\mathsf{T}\mathbf{n}_2 < 0$ which is a sign for a silhouette. However, in volume rendering univariate functions $h(\mathbf{v}^\mathsf{T}\mathbf{n}) \in \mathbb{R}_+$ of the dot product and a threshold value τ can be used to enhance silhouettes, e.g. values of $h(\mathbf{v}^\mathsf{T}\mathbf{n}) < \tau$ with τ near to zero could be darkened. This proceeding leads to uncontrolled variation in the emerging thickness of contours. That means, on one side, at flat surfaces where large regions of surface normals are perpendicular to the view direction contours become very thick, on the other side, at fine structures they are often too thin. A recent method [KWTM03] regularizes the thickness of contours on iso-surfaces by using the curvature along the viewing direction.

4.6 Volume Shader Models

Volume rendering modes or computer visualization methods are, for example, x-ray projection (X-Ray) or maximum intensity projection (MIP). Applying these modes the volume cells (regular or irregular) can be processed (projected) in any order, since the volume rendering integral degenerates to a commutative function, i.e. the rays have not to be processed in a front-to-back manner. In contrast, for iso-surface or full volume rendering a depth ordering of the cells is required to satisfy the compositing order, thus rays have to be processed in a front-to-back manner, since the generalized volume rendering integral is not commutative.
However, each computer visualization method can be combined with any rending approach (cf. Sec. 5), e.g. ray-casting or shear-warp. Although a straight forward implementation of a volume rendering algorithm would apply ray-casting, the choice of the rendering method has great influence on the speed of the final algorithm as well as the resulting image quality. Similarly, for performance reasons or the requirements of the image quality, one can utilize different strategies to find the values along rays for the different modes:

- Analytical solution: Depending on the data reconstruction model, for each cube intersected by the ray a local polynomial of an appropriate degree has to be determined. According to the mode used, one has to compute the roots (for iso-surface visualizations), the maxima (for MIP) or the integral (for full volume rendering) of the polynomial pieces to determine the final result. Such an analytical computation and evaluation of the local polynomials is the most accurate but also the most expensive method.

- Sampling and interpolation: In this less expensive approach data values are sampled with a distance d along a ray using a data reconstruction model. No explicit polynomial pieces are considered (determined) within cubes, thus the cost of this approach depends only on the data reconstruction scheme and the sampling

distance d. However, the method can be greatly accelerated by using longer resampling distances d along rays, which, on the other side, produces lower quality images.

- Nearest neighbor interpolation: No real interpolation is performed at all, i.e. the data reconstruction model generates piecewise constant polynomials. Values on the data grid closest to the current considered samples along the ray are only considered for estimation. Since no interpolation is applied, the data grid structure becomes visible as aliasing or as staircase artifacts in the resulting images.

Figure 4.7: *Volume rendering modes reveal different information about the volume data set, e.g. maximum intensity projection (left), full volume rendering (middle), and iso-surface rendering with a gradient-based color transfer function (right) where high gradient magnitudes are mapped to red colors.*

4.6.1 Maximum Intensity Projection

Maximum intensity projection (MIP) [MGK99] [CKG99] is a computer visualization method that projects sample values into the projection plane with maximum intensity [MHG00] [MKG00] obtained along a ray from the three-dimensional data set [PHK+03] [ME05]. Applying a parallel (orthographic) projection method using MIP implies that two images obtained from opposite viewpoints are symmetrical. More over, the viewer cannot distinguish between left or right, front or back and even not if the object is rotated clockwise or not. On the one hand, within angiography data sets usually the data values of vascular structures are brighter than the values of the surrounding tissue, the structure of vessels contained in these data sets can be captured easily. On the other hand, an application of early-ray termination as acceleration technique (cf. Sec. 6) is not possible, making the standard MIP often even more expensive than full volume rendering or iso-surface rendering. Further, an image obtained by MIP does not contain any shading information, thus there is no depth and occlusion information, i.e. objects with brighter image data values appear to be in front – even though lying behind of objects corresponding to darker image values. A common way to simplify the interpretation of such images is to rotate (animate) the data. That means, the relative three-dimensional positions of objects are revealed by rendering the object's data sets from different viewing positions or directions. However, this improves the viewer's perception. In other words, it is easier for the viewer to reveal the relative three-dimensional positions of objects.

4.6.2 X-Ray Projection

Volume rendering can be considered as the inverse problem of tomographic reconstruction [Mal93] where the aim is to reconstruct the unknown volume density function $f(\mathbf{k})$ by rotating an X-ray emitter/dedector pair at some angle and collecting a set of measured projections. However, X-ray like images [Lev92] of a volume data set can be obtained with a rendering technique that evaluates the line integral in the frequency domain. Namely, the Fourier Projection-Slice Theorem allows to compute a two-dimensional X-ray like image from a three-dimensional data set by using a two-dimensional slice of that data in the frequency domain (cf. also [Max95]).

4.6.3 Iso-Surface Rendering

The goal in iso-surface rendering is to find the smallest intersection parameter $t \geq 0$ of each ray $\mathbf{r}(t) = \mathbf{r}_s + t\ \mathbf{r}_d$ going through the volume where the data has an user-defined iso-value. This global problem is usually split into smaller local problems. However, to find the correct intersection of a ray and a surface of the volume data using the trilinear reconstruction model is an essential task. Many available methods are accurate but slow or fast but for an approximate solution only. In [MKW+04] available techniques are compared, analyzed and a new intersection algorithm is presented, which is said to be three times faster and provides the same image quality and better numerical stability in opposite to previous methods.

4.6.4 Full Volume Rendering

In full volume rendering (FVR), usually the volume rendering integral is solved by appropriate evaluation of equations (4.6), (4.7), and (4.8) as already discussed in the previous sections. However, from the above mentioned integral one can derive an *absorption only* model where the volume is assumed to consist of perfectly black particles that absorb all the light that goes through them. In the *emission only* model the volume is assumed to be of particles that emit – but do not absorb – light. Other models take scattering and shading/shadowing into account (cf. [Max95]).

4.6.5 Non-Photorealistic

Volume illustration techniques, as, for example, presented in [TC00] [CMH+01] [LME+02] [LM02] [MHIL02], can be considered as a subclass of non-photorealistic rendering (NPR) [GGSC98]. The particular characteristic of illustrations [GSG+99] [ER00] is to emphasize important features or parts of an object and to advise the information to the viewer in the most efficient way. This characterization matches the requirements of traditional visualization techniques as well. However, in volume rendering transfer functions are often set such that objects are semi-transparent making spatial relationship between these objects difficult. Silhouette edge illustration (as discussed in section 4.5) can be particularly useful here (further non-photorealistic techniques can be found in [GGSC98], [GSG+99] [NSW02], [WKME03b]. The manipulation of color based on distance can also improve depth perception [FvDFH97]. For depth cues typically warmer hues are used for the foreground and cooler in the background and color values tend to become lighter and less intense with distance. In addition, non-linear fading of the alpha channel along

the view direction allows making closer material more transparent such that underlying features are more visible, but foreground material is still slightly preserved to provide context for the features of interest.

4.6.6 Focus and Context Techniques

Traditionally, objects within the volume data set are classified by optical properties like color and opacity using transfer functions. Another technique is to classify or segment the data during a pre-processing stage into different regions where each region specified by an identification number (id) represents an object of the volume. Now different optical properties can be assigned to each object such that some are enhanced and others not. An importance driven approach [VKG04] additionally assigns objects another dimension, which describes their importance. This scalar importance value encodes which objects are the most interesting ones and have the highest priority to be clearly visible. During rendering the model evaluates the visibility of each object according to its importance, i.e. such that less important objects are not occluding features that are more interesting. In other words, the less important ones are rendered more sparsely. Two approaches have been proposed using importance values, the first is called *maximum importance projection* (MImP), whereas the second *average importance projection*. In the former method for each ray the object with highest importance along the ray is determined. This object is displayed densely, whereas all the remaining objects along the ray are displayed with the highest level of sparseness, i.e. fully transparent. This approach can be considered as a cut-away view similar as in [SCC+04], where a cut-away technique for CT angiography of peripheral arteries in human legs is applied. The goal is to have a clear view on the vessels, which are partially segmented by their centerline.
A discussion of expressive visualization techniques, e.g. cut-away views, ghosted views, and exploded views, originating from technical illustration that expose the most important information in order to maximize the visual information of the underlying data can be found in [VG05] (cf. also [KTH+05] [BGKG05], and [BG05]).

4.7 Segmented Data

Usually transfer functions indexed by data values (densities) are used to define a simple segmentation of the volume and to assign different optical properties to different objects (as discussed above). However, for several regions (or objects) in a data set which have similar or even the same density values it is difficult or impossible to define appropriate mapping functions, such that one can discriminate between the objects. That means, it is often the case that a single rendering method or transfer function does not suffice in order to distinguish multiple objects of interest according to a users specific needs. Here another very powerful technique comes into play. This supports the perception of individual objects contained in a single volume data set by creating an explicit object membership identification number [UH99].
There are several ways of representing segmentation information of a volume. The simplest one is to specify this information for all objects contained in a single volume in another identification volume of the same size. However, usually only a few objects are contained in a single volume, hence identification numbers of 8bit resolution for each

voxel should be enough and are simply enumerated consecutively starting with one. In other words, each voxel contains an identification number of that object it belongs to. This number is stored at the appropriate position in the second identification volume[2]. Using this membership one is able to determine the transfer function, rendering, and compositing modes used for a given sample. A simple nearest neighbor look-up of the identification number for a given sample is trivial, but leads to artifacts. Whereas a filtering has to be performed carefully, since a direct interpolation of numerical identification numbers of objects leads to incorrectly interpolated intermediate values (see e.g. [HMBG00] [HBH03]).

[2]Objects (voxels) identified by separate masks can be easily combined in a pre-processing step into a single volume that contains a single object identification number for each voxel.

5 Rendering Methods

Volume rendering methods are algorithms applied for volume visualization. There are several methods and they can be split into image-order, object-order, hybrid or domain-based methods. In image-order algorithms (e.g. ray-casting) rays are cast from the eye location through each pixel in the image plane into the volume grid to determine the final value of the pixel (also called *a backward mapping scheme*). This technique (cf.

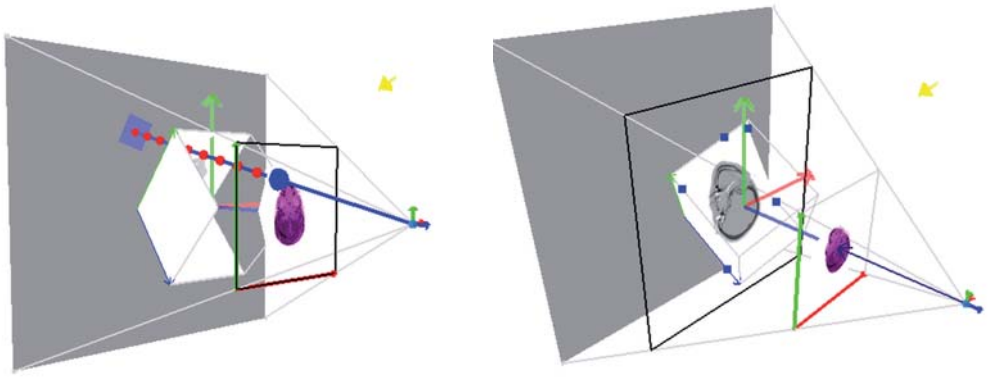


Figure 5.1: *Illustration of ray-casting (left) and slicing (right). In ray-casting rays (blue ray) cast from the eye (small blue dot) through each pixel (blue dot) in the image plane into the volume data set (light gray cube), where at (non-)equidistant sample positions (red dots) data values are reconstructed, shaded and composited into the final image pixel (blue dot). In slicing usually a slice or plane (black rectangle) aligned parallel with the projection plane (light gray rectangle) is shifted by the (non-)equidistant sampling distance through the volume. Data is reconstructed within the convex polygon defined by at most six intersection positions (blue solid quads) of the slice and the volume, and finally shaded and composited into the image plane.*

left illustration of Fig. 5.1) allows to apply early-ray termination for accelerated volume rendering. In object-order algorithms (e.g. splatting and cell-projection) volume data is projected directly onto the screen (also called *a forward mapping scheme*). Object-order techniques decompose the volume into a set of basis elements or basis functions which are individually projected to the screen and create the final image. These techniques allow easily applying space leaping for accelerated volume rendering. Hybrid methods try to combine the advantages of image-order as well as object-order methods, whereas domain-based techniques first transform the data from the spatial domain into another domain (e.g. frequency or wavelet domain) and afterwards directly determine the two-dimensional projection using the data from this domain. However, different partitioning of the algorithms are posible, one could differentiate between direct methods (e.g. direct iso-surface or full volume rendering) and indirect methods as, for example, marching

cubes. The former techniques directly visualize the volume data in one pass. Whereas the second kind of techniques previously extract some information from the volume grid, representing this in different form (e.g. as a triangulation of an iso-surface), and afterwards this new representation is visualized. We split the methods according to their underlying partitions used to represent volume data, i.e. whether the algorithms operate on regular or irregular grids.

5.1 Regular Grids

5.1.1 Ray-Casting

The term ray-casting arises from ray-tracing which describes the process of casting rays from the viewpoint (eye location) through specified pixels on the image plane [Gla84]. In computer graphics, usually the intersection of each ray with any object in the scene is computed and the pixel in the image plane is identified with the nearest intersection point and the reflected energy at this point. This ray is termed the primary ray, secondary rays are usually cast to model transparency, shadows or reflections [Lev88] [Lev90] [PH04]. However, applying ray-tracing (cf. [WMK04] [KL04] [Chr05]) with primary rays only, leads to the same outcome as using first-order approximation projection methods, e.g. ray-casting (cf. left illustration of Fig. 5.1). In this way ray-casting is a simulation of light-propagation through a scene using primary rays only, where the result is represented on the image plane. The simulation is based on the geometry of the objects, reflection, and refraction, thus the quality of the displayed image heavily depends on the physical models (cf. Sec. 4) used, e.g. the resolution of the scene (objects and image plane), the illumination model, the shader model, and the transfer functions (cf. [GTGB84] [CPC84] [Kaj86]). A brute force implementation of ray-tracing checks each ray against each object in the scene, thus it is very computation intensive and acceleration techniques are unavoidable, i.e. techniques which consider the coherence in a scene (cf. [Sam90]).
In volume rendering only primary rays[1] are considered because of the huge computation effort. Acceleration techniques as, for example, presented in [AW87] [SW91] [YK92] [RUL00] as well as special hardware [HMK+95] or graphics cards [KW03] [HQK05] [SSKE05] can be utilized to achieve interactive visualizations of the data set.
Recently, a fast software based algorithm [GBKG04a] [GBKG04b] has been developed by applying sophisticated caching strategies and data structures. On the one side, with such software-based algorithms one cannot obtain fast rendering times compared to hardware-accelerated algorithms. On the other side, the rigid architecture of the graphics processing unit (GPU) does not allow transporting the standard software-based volume rendering methods (especially with appropriate acceleration techniques) directly onto this hardware. Hence, recently more and more algorithms and appropriate extensions for the graphics hardware have been developed and discussed. A GPU-based object-order ray-casting algorithm for the rendering of large volume data sets has been published in [HQK05]. The volume data set is decomposed into small sub-volumes, which are further organized in a min-max octree data structure. Each leaf of that min-max octree stores the sub-volumes also called *cells*. After classification of that cells using mapping (transfer) functions, only visible cells are loaded into GPU memory. The cells are sorted

[1] Therefore the expression ray-casting.

into layers and all cells within the same layer are projected onto the image plane in front-to-back order using a volumetric ray-casting algorithm. Another framework [SSKE05] is based on a single pass volume ray-casting approach and is easily extensible in terms of new shader functionality as well as new reconstruction models.

5.1.2 Splatting

In the original object-based method [Wes89] [Wes90], the volume is represented by an array of overlapping, symmetric basis functions (gaussians kernels) in three-dimensional space. These basis functions are projected onto the screen to produce a two-dimensional image. A basic method only composites all kernels in front-to-back order onto the screen. Although this method is very fast, one disadvantage is that it can generate color bleeding, sparkling artifacts, and aliasing. One reason is due to the imperfect visibility ordering of the overlapping basis functions. However, advantages of splatting are that only unit cubes or basis functions have to be rendered (projected and rasterized), which are relevant in the resulting two-dimensional image. This method is attractive for rendering sparse data sets, it can handle irregular data sets and different grid topologies as well, i.e. body centered cubic grids [SM02]. In [Wes90] some quality improvements of a basic method has already been proposed, by summing up the basis functions (kernels in voxel space) within volume slices most parallel to the image plane (also called *sheet-buffer approach*). Nevertheless, this method generates brightness variations (popping artifacts) in animations of data sets, i.e. by rotating a data set. In a more recent method [MC98] [MSHC99] these disadvantages are removed by processing the basis functions within slabs, or sheet-buffers, aligned parallel to the image plane (also called *image-aligned sheet-buffered splatting approach*). The projection of the basis functions can be accelerated by the rasterization of a pre-computed two-dimensional footprint lookup table (cf. [HCSM00] as well). Each footprint table entry stores the analytically integrated basis function along a crossing ray. Whereas the aliasing problem was addressed in [SMM+97] [ZPvBG02], post-shaded rendering in [MMC99] and a perspective accurate splatting method was presented in [ZRB+04]. In other papers some performance improvements for software based splatting are given, like hierarchical splatting [LH99], three-dimensional adjacency data structures [OM01], and post-convolved [NM03] rendering. More recent methods utilize graphics hardware to accelerated splatting and were compared in [XC02]. Another hardware [NM05] based splatting approach exploits many of the new features of current graphics cards and is based on the sheet-buffered image aligned algorithm [MC98], which was initially developed to overcome the performance/quality concerns of the previous existing splatting algorithms.

5.1.3 Shear-Warp

The attractive original shear-warp approach [LL94] [Lac95] combines the advantages of image and object-order algorithms. This kind of algorithms are also called *hybrid* or *intermediate* methods. However, many extensions have been made to this method as well as a special hardware [PHK+99] has been developed. For more information on this technique as well as for a discussion of the state of the art see part III.

5.1.4 Slice-Based

The capability of rendering a volume on graphics hardware was already addressed in [CN94]. A slice based volume rendering (SBVR) algorithm was presented in [Ake93] [CCF94], which can be seen as an emulation of ray-casting, but where all rays are sampled at once on a slice. Here, volume rendering is performed by slicing the volume in object-aligned or image-aligned planes. These planes are rendered in front-to-back or in back-to-front order using three-dimensional textures during rasterization and by compositing them into the frame buffer (cf. right illustration of Fig. 5.1). A main drawback of a hardware based SBVR algorithm is that all fragments have to be processed from the three-dimensional texture even if they do not contribute to the final image. This decreases the rendering speed, in particular for complex shader programs with, for example, lighting and gradient computations. In more recent developments, the volumetric domain is split into small sub-volumes called *bricks* or *cells* [LMK03], where empty cells are removed and only the non-empty cells are rendered with the SBVR algorithm. This improves the rendering performance of large data sets. In [XZC05] iso-surfaces are visualized by a hardware accelerated algorithm with the slice-based volume rendering approach, where the early z-culling feature of current graphics hardware is utilized to obtain better performance.

5.2 Irregular Grids

5.2.1 Ray-Casting

A hardware-based ray-casting algorithm [WKME03a] for tetrahedral meshes is based on the method discussed in [Gar90] [PBMH02] [RGW$^+$03], where for each ray the irregular grid is traversed by following the links between neighboring tetrahedral cells. Pre-integrated volume rendering and early-ray termination are applied as acceleration techniques during ray integration. Modifications and extensions are presented [Pur04] as well. The algorithm can be subsumed as follows. First, the viewing ray is propagated in front-to-back order from cell to cell until the mesh has been abandoned. Thus, in an initialization phase the first intersection of the ray and the mesh is computed. Then, until the ray has not left the mesh, the exit point for the current cell is determined, the corresponding scalar value is interpolated, the ray integral within the cell is computed and blended into the fame buffer. Finally, the adjacent (neighboring) cell is found through the exit point.

5.2.2 Cell-Projection

Unstructured tetrahedral grids can be efficiently rendered using the projected tetrahedra (PT) algorithm [ST90]. In this algorithm the volume is decomposed into tetrahedral cells first. Scalar values are defined at each vertex of the tetrahedral cell. Inside each tetrahedral cell the density is assumed to be a linear combination of the data at the vertex values. Then the cells are sorted according to their visibility. Afterwards each tetrahedron is classified according to its projected profile and this is decomposed into triangles (cf. Fig. 5.2). Finally the color and opacity values for the triangle vertices in the original world coordinates are computed using ray integration and the triangles are

rendered (e.g. using graphics hardware). In other words, this algorithm visualizes and approximates a scalar function in three-dimensional space by rendering partially transparent tetrahedra, whereby color and opacity values are linearly interpolated between the triangle vertices (on the projected profile).

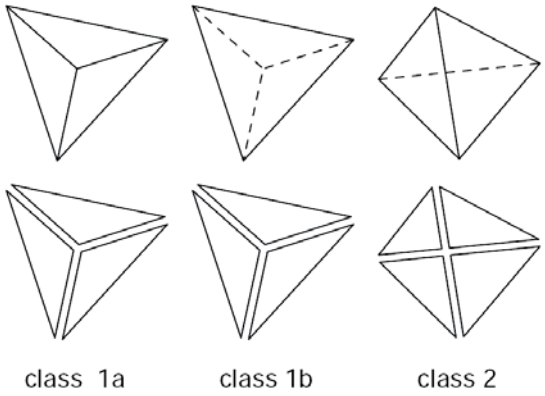


Figure 5.2: *Top: Classification of non-degenerate projected tetrahedra. Bottom: The corresponding decompositions (cf. [ST90]).*

However, this direct volume rendering algorithm has been invented several years ago and has bluntly accelerated the process of rendering unstructured tetrahedral grids, but leads to rendering artifacts [MBC93] [SBM94]. A further development has been presented [WMS98] in order to avoid some of these artifacts. This approach is restricted to a linearly varying opacity function and interpolates color linearly as well.

A generalization [RKE00] works for color and opacity with no restrictions on the transfer functions. These benefits are achieved by considering orthographic projections only and employing texture mapping, i.e. a three-dimensional texture map is set up which contains the color and opacity characterized by an intersection of a ray and a cell (tetrahedron).

Also vertex shaders on recent graphics hardware have been applied [WMFC02] to render the tetrahedral cells directly within the graphics card. A view independent cell projection approach has been given in [WKME03b]. This method is usful for all commutative blend functions, which in general allow the compositing in arbitrary order (e.g. in case of maximum intensity projection).

5.2.3 Slice-Based

In a slice-based approach [JWH+04] structured and unstructured volume data grids are encoded by radial basis functions (RBF). These compactly supported RBFs are used because of their limited spatial extent, explicit evaluation and smooth representation of noisy data. This work has been extended for encoding vector and multi-field data sets [WBH+05]. Efficient feature detection techniques have been developed and the ability to refine regions of interest in the data. Further, graphics hardware has been applied to accelerate the reconstruction, rendering, and feature detection process by using this functional representation.

5.3 Other Methods

5.3.1 Hardware or Texture Based

Many implementation issues for direct volume visualization algorithms applied on special purpose hardware [HMK+95] [MKS98] [PHK+99] have been solved [VHMK99], discussed and analyzed in the last years, e.g. how to efficiently implement acceleration techniques. Special purpose hardware is, due to small consumer markets, often very expensive, new developments in hardware lead sometimes to whole re-implementations of already available software packages, and due to missing standards and high level tools for development, the evolution of new applications becomes occasionally a tedious process. In this view, the inexpensiveness (due to the gamer's market) and programmability of recent graphics hardware make the development of volume rendering algorithms on this hardware more interesting and important. Once algorithms usable on graphics hardware have been developed they can be redistributed, and they profit from the higher computational power – recent graphics hardware is an order of magnitude faster than current processors – as well as the memory bandwidth of recent graphics cards compared to usual personal computers.
The task of visualizing a volume can be performed very fast by by means of two or three-dimensional textures. Due to the slightly rigid pipeline architecture of graphics processing units (GPU), it is often necessary to adapt existing as well as to develop new algorithms [RSEB+00] [KKH01] [KPHE02]. In the case of a huge data set, i.e. when the whole data does not fit into graphics memory, it has to be compressed or reloaded from the main memory [LMC01] [LK02] [KE02] [SW03]. However, volume rendering performed by graphics hardware and textures is usually done by slicing the texture block in back-to-front order with planes oriented parallel to the view plane (cf. right illustration in Fig. 5.1). An additional buffer, sometimes called *alpha buffer*, is needed in case of front-to-back rendering to store the accumulated opacity and color values. Whereas the most probable limitation of such texture based volume rendering is the huge amount of fragment and pixel operations, i.e texture access, interpolation, lighting calculation, and blending. These operations are performed no matter if voxels are transparent and do not contribute to the corresponding image pixels or these pixels are already opaque [RSEB+00].
A multi-pass approach is given in [KW03], where for each fragment rays are casted through the volume until an opacity threshold is reached or a selected iso-value is hit. However, before ray traversal, all necessary ray information (e.g. direction and start) according to the three-dimensional texture coordinates are pre-computed and stored in a two-dimensional texture. This is reused in upcoming passes. A similar process is referred as *Deferred Shading* (cf. [EHK+05]).
A discussion of modern and future graphics hardware and their programming models is presented [PBMH02]. In this work, a stream model for ray-tracing performed on programmable fragment processors is proposed, which takes advantage of parallel fragment units and high bandwidth to texture memory.
Hardware based methods are comparable in speed as well as in quality to optimized software based algorithms. Albeit it can be expected that graphics processing units will grow faster in performance compared to central processing units, and thus in several years graphics hardware maybe the main choice for implementing any kind of graphics

algorithms. However, in [OLG+05] techniques and summary are presented how general-purpose computations can be mapped to graphics hardware. Due to the development of new multi-core architecture concepts (see e.g. cell processor) and the significant consumer markets in this area, new, cheap, and flexible parallel architectures may become the reality. Where the development and the conversion of existing algorithms for such new, revolutionary architectures will be almost a crucial point.

5.3.2 Domain Based

The visualization of huge data sets (i.e. more than 1GB) is still a challenging task. Here, compressed data sets, on one side, will not occupy the main memory too much and on the other side, the bandwidth between the main memory and the central processing unit (CPU) will be spared. Compression methods as used in image and video processing [Gal91] are not directly applicable for volume data sets, because they often read and compress the entire images or slices of the video stream. In volume rendering, often only a fraction of the data is needed, because of the standard acceleration techniques, i.e. early-ray termination and space leaping. Therefore, a local, a random and a fast access to voxel data is needed. Hence, there are several goals for the encoding of a volume data set. First, a multi-resolution representation of the data for level of detail processing. Second, the effective utilization of redundancy in each direction of the three-dimensional volume data set. And third, a selective block, scan-line or cell compression method for fast random access.
Many approaches often use quantization [NH93], wavelet coefficients [IP99], hierarchies [BIPS00] [BIP01], and/or visualization-dependance [BPI01] to reduce volume data. In [GWGS02] each level of the octree stores some wavelet coefficients (cf. [CDSY97]) which are further compressed by run-length-Huffman or Lemple-Ziv-Welsh coding schemes for level of detail processing of the volume data. Similar approaches can be found in [Mur93] [Wes94] [LHJ99a] [RSEB+00] [WWH+00] [NS01]
Other approaches [Rod99] [GS01] follow a more conventional way to encode redundancy. They apply techniques well known in video compression and consider the volume slices as frames of a video stream.
Due to limited transmission bandwidth between the central processing unit and the graphics hardware yet other approaches [RSEB+00] [KKH01] use a very similar encoding of the data as discussed above. They decode first [KPHE02] or directly load chunks (cells) of the data set into the graphics texture memory. Afterwards they utilize the hardware for rendering and decompression for static data sets [KE02] [LK02] as well as for time varying data sets [LMC01] [SW03].
However, the shortcomings of such methods are obvious. Firstly, the data has to be decoded for visualization no matter if in hardware or software. Secondly, if the blocks are too small or too big the redundancy of the data is not well exploited. Thirdly, the image quality depends on the compression method and the allowed approximation error. It has been shown [GS01] that higher order wavelets not only reduce the size of compressed volumes while enlarging the peak-signal to noise ratio, but also significantly preserve features and improve the visual impression.
A rather theoretical description of a fast volume visualization system can be found in [SBS02]. Basically, a volumetric video system is discussed which is based on a performance server where large time-dependent data sets can be processed.

5.3.3 Indirect Methods

Indirect methods first extract some information from the given volume data set, e.g. the iso-surface, and represent the information by different means, e.g. as surface triangulation. One of the well known indirect visualization techniques applied for volume rendering is the marching cubes [LC87] method. This approach extracts surface information represented by triangles from a discrete three-dimensional data set. The basic principle behind the marching cubes algorithm is to subdivide a volume into a series of small unit cubes. The algorithm goes through each unit cube and tests the corner points (i.e. the particular 2^3 neighboring voxels) and replaces the cube by an appropriate set of polygons or triangles. The total amount of triangles approximates the specified iso-surface of the original data set. However, since one cube has 2^3 corners, there are $2^{2^3} = 256$ possible corner combinations (i.e. different states for a unit cube). But since there are twofold cell combinations under some conditions, the number of states can be reduced into a total of 15 combinations. This reduces the complexity of the algorithm in terms of programming effort as well as runtime and makes a computation of the polygon sets less expensive. The number of combinations can be further reduced (to three) by adapting the marching cubes algorithm to tetrahedral meshes [SFYC43]. The main problem of the marching cubes algorithm is that it is very expensive since all cells have to be considered. Hence, faster approaches [CMM+97] would first identify the unit cells intersected by the surface and then investigate only these cells (e.g. by using an octree representation of the volume) for further computations. Other problems of the marching cubes approach are that the polygon nets can contain holes (because of ambiguous faces) [Che95] and even with moderately sized volume data sets polygon meshes of several million polygons are obtained. The former difficulty leads to aliasing artifacts or wrong surfaces, but can be omitted [LLVT03]. The later issue can be solved by an application of polygon reduction schemes [KLS96] [LHSW03], since a real-time rendering of (several) million polygons is hard to realize even on modern graphics cards.

6 Acceleration Techniques

Volume rendering is a very time consuming process. Usually, the whole three-dimensional data set has to be processed to obtain a colored two-dimensional image of that volume. However, if one would like to animate (i.e. rotate, zoom) the volume data set in real time, then at least $1 - 10$ frames (colored images) per second have to be rendered. Without acceleration techniques this would be an impossible venture. There are of course many special acceleration techniques and algorithmic optimizations for the different kind of volume rendering algorithms (cf. Sec. 5). In the following only a few of them – the standard techniques – will be discussed, i.e. techniques which can be applied to many rendering algorithms without huge implementation efforts and which do not trade off image quality for speed.

6.1 Early Ray Termination

This acceleration technique has the nice property that, using front-to-back composting, it is possible that the computation of a ray can be stopped once the accumulated alpha value $\tilde{\alpha}_k$ (cf. Equ. (4.8)) approaches a user-defined threshold (cf. [Lev88] [Lev90]). That means, light resulting from objects further away is completely blocked by the accumulated opaque material in front. This technique can be easily applied in image-order volume rendering algorithms, where in object-order techniques often some special data structures have to be developed to realize this speed-up method.

6.2 Space Leaping

For the realization of this acceleration technique, there are many different and mostly special data structures available, depending on the rendering algorithm as well as the data used. However, physical theory shows that reflection occurs only where material parameters change, i.e. region boundaries are identified by large gradient magnitudes, whereas homogeneous regions in the data space have zero gradients and do not reflect diffuse and specular light. Hence, they can be skipped during rendering, because they do not contribute any information to the resulting image and the volume rendering process is speed-up enormously.
It has been shown [Lac95] that in real data sets with enhanced surfaces the amount of transparent voxels is often about 90–95%, i.e. for regions which are completely empty an approach called *empty space leaping* can be performed and is very efficient since even the calculation of absorption is not necessary. Empty space leaping [YS93] [LMK03] can be realized by several methods. Examples are distance coding or proximity clouds [ZKV92] [Š94] [PHK+99], recursive divide and conquer algorithms by using different (e.g. min-max) octree representations [WG90] [WG92], the Lipschitz octree data structure [SH94], and bounding boxes (cf. [YS93]). Further, multi-dimensional summed area

tables [Cro84] [Gla90] are applied in case of min-max octree representations of the data to compute an integral over a region (e.g. an octree node) in constant time, i.e. to quickly find if a node is empty and can be skipped.

In homogeneous but not empty regions, where absorption and reflection can occur, coherence encoding can be used to compute ray-light interactions. This means, real data sets contain, beside noise often clusters of cells or cubes that vary not only constantly, but also linearly or quadratically in the three-dimensional spatial domain. This data coherence can be for example encoded using piecewise linear polynomials [FS97] [CHM01] [Che01] as well as wavelets.

However, the space leaping technique is easily applicable to object-order rendering algorithms, although many different data structures have been published to realize this technique in image-order algorithms as well. In the next subsection, we are going to discuss spatial data structures for run length encoding, distance and coherence encoding to realize the space leaping technique.

6.2.1 Run Length Encoded Data

A run length data structure is used to encode piecewise constant data values, i.e. given a count and a value, where the count indicates how often the value should be repeated or reused. A pre-computed run-length encoding of the voxel scanlines can be applied for accelerating the process of volume rendering. This data structure allows skipping over transparent voxels during rendering. A run-length encoding of a scanline consists of a series of runs represented by a run length and a data value for each run (cf. chapter 17.7.1 in [FvDFH97] and [Lac95]).

6.2.2 Distance Encoded Data

A data structure for distance encoding assigns each cube (voxel or sub-cube) a value which encodes the distance to the next surface or the next non-transparent cube. The distance values stored in a separate data structure are used to compute the next sampling position along a specified ray within the volume [YS93] [HMK+95] [PHK+99], i.e. the next sample position $\mathbf{r}_{i+1}$ along a ray $\mathbf{r}(t)$ is obtained by $\mathbf{r}_{i+1} := \mathbf{r}(t_i) := \mathbf{r}_i + t_i\mathbf{r}_d$, where $\mathbf{r}_i$ is the last sampling position, t_i is the distance value at this location obtained from the distance data structure and $\mathbf{r}_d$ is the ray direction vector with Euclidian distance $\| \mathbf{r}_d \| = 1$. However, a pre-computation of distances for the whole volume data set requests that the distance values stored for each voxel can be applied to any ray passing through the volume, i.e. they must be view independent. Hence, the next possible non-transparent cube can be computed along a ray independently of the viewing direction. For the distance coding computation, different metrics can be used, e.g. the Euclidian metric, simpler metrics that approximate the Euclidian metric, or the Manhattan distance. However, the view independency can be fulfilled by applying a so called two or six pass algorithm. The volume is scanned from front-top-left to back-bottom-right (and vice versa) or from left to right (and vice versa) for each axis. During that passes the length of non transparent cubes is accumulated and stored in the distance volume.

6.2.3 Octree Data Structure

An octree data structure is used to represent volume data in a hierarchical manner (cf. Fig. 6.1) and it can be applied for space leaping as well as coherence encoding. However,

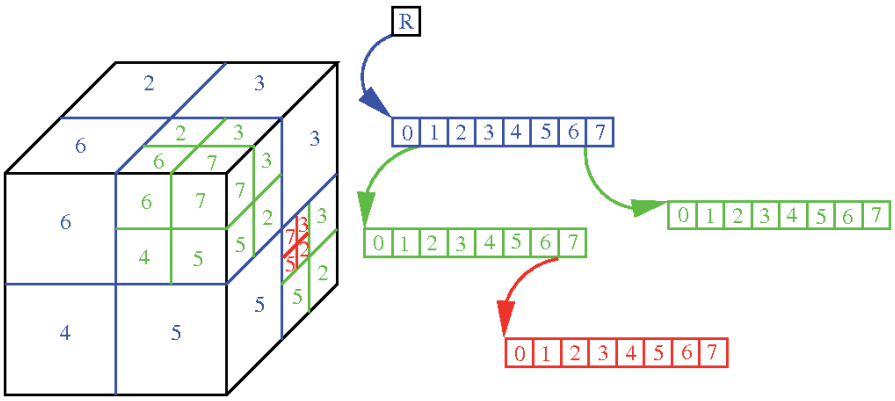


Figure 6.1: *Illustration of an octree data structure (right), where each node represents a corresponding sub cube of the volume data set (left) and stores links to it's eight children. The first root node (R) represents the whole volume (e.g. the min-max values).*

there are different methods on how to represent volume data by an octree. In this thesis, only a part of them is discussed.

Standard Octree

The empty-non-empty octree represents a data set by regions of empty and non-empty values. However, the root-node of the octree represents the whole data set, which is split into eight sub-nodes. The eight child-nodes of the root-node represent the corresponding eight sub regions of the data set, which are recursively subdivided into smaller regions represented by a child's child-nodes. If once a sub region is homogeneous (or empty) further subdivision is stopped and the corresponding node of the octree is marked as a leaf node.

During traversal or rendering one usually starts according to the considered ray with the root node and parses down the branches until encountering a leaf node. Then processing is stopped, if the leaf node represents a transparent voxel or sub-cube, otherwise the usual (but adapted) compositing is performed. That means, if the node is not transparent but a voxel, the contribution can be computed directly, otherwise one has to investigate which of the eight sub-nodes are intersected by the ray and in the correct processing order (e.g. front-to-back) each sub-node is processed.

Min-Max Octree

In order to incorporate faster mappings from discrete gray values to opacities by using transfer functions κ a min-max octree allows a rough estimation of transparent sub-cubes. For this an octree is created where now each node – associated with a sub region of the data set – contains the maximum and minimum values (e.g. density values) from the corresponding region of the data set. The min-max octree has some important features using transfer tables for data classification. First, it can be pre-computed and it is independent of the transfer function κ. Second, the minimum and maximum values

define a range of values within the respective sub-cube. Third, the voxels themselves remain in a three-dimensional array data structure making a fast random access possible. However, for a two-dimensional opacity transfer function $\kappa_{a,b} \in \mathbb{R}$ indexed by discrete values $a = 0, \ldots, N$ and $b = 0, \ldots, M$ (e.g. discrete gray values and gradient magnitudes) the corresponding summed area table $\sigma_{a,b} \in \mathbb{R}$ can be computed using

$$\gamma(a,b) = \sum_{i=0}^{a} \sum_{j=0}^{b} \kappa_{i,j}. \tag{6.1}$$

and the following algorithm – which can be rewritten in a recursive manner, i.e. previously computed results can be reused.

Algorithm 6.2.1 (Summed Area Table)**.** *The input to this algorithm is the two-dimensional transfer function κ, whereas the two-dimensional summed area table σ is returned.*

1: ***for*** $a = 0 < N$ ***do***
2: ***for*** $b = 0 < M$ ***do***
3: $\sigma_{a,b} = \gamma(a,b)$ {*Using equation (6.1).*}
4: ***end for***
5: ***end for***

During rendering the min-max values $a_{min}, a_{max} \in [0, N]$ and $b_{min}, b_{max} \in [0, M]$ of a sub cube stored in the octree are taken in order to find out whether the cube is empty or not. This is achieved if all values between min and max have opacity 0 and can be directly calculated from the summed area table by applying the following formula

$$\sum_{i=a_{min}}^{a_{max}} \sum_{j=b_{min}}^{b_{max}} \kappa_{i,j} = \sigma_{a_{max},b_{max}} - \sigma_{a_{max},b_{min}-1} - \sigma_{a_{min}-1,b_{max}} + \sigma_{a_{min}-1,b_{min}-1}. \tag{6.2}$$

The classification of the data using this table is done quickly. Whereas the processing of the summed area table is time consuming, it has to be recomputed only if the opacity transfer function is changed. The re-computation of the two-dimensional table is much faster than a re-classification of the whole three-dimensional data set. Nevertheless, one disadvantage of this approach is that the estimation of transparent regions is only approximative.
Aside from the computation of transparent and non-transparent sub-cubes there are alternatives to octrees that may be more efficient in finding transparent regions. Those are for example k-d-trees or binary space partition trees which allow a more tighter approximation of the transparent regions at the cost of more complex ray intersection calculations.

Lipschitz Octree

The Lipschitz octree data structure [SH94] combines the speed enhancements of distance coding with the threshold flexibility of min-max octrees. The discrete voxel data is redefined as a continuous scalar field, over which a local Lipschitz bound can be computed. These local Lipschitz bounds are repeatedly merged to create an octree of Lipschitz

bounds over variously sized domains, ending at the top-level with a global Lipschitz bound on the entire volume.
However, for trilinear interpolation an eight neighborhood of voxels has to be considered at a time – sometimes denoted as a *cell* or an *unit cube*, where all values within this neighborhood or cell can be reconstructed by the following formula

$$f(x,y,z) = \sum_{i,j,k\in\{0,1\}} (1-x)^{1-i}x^i(1-y)^{1-j}y^j(1-z)^{1-k}z^k f_{ijk}, \tag{6.3}$$

where $x, y, z \in [0,1]$ and $f_{ijk}, \quad i,j,k \in \{0,1\}$ are the eight voxel values at the eight corners of the unit cube or cell. Now, the Lipschitz constant l_c over the domain of the unit cube is defined as the maximum gradient magnitude within that cell. This is bound by the sum of the individual gradient magnitudes maxima

$$l_c \leq \sqrt{\sum_{\xi\in\{x,y,z\}} \max(\partial v/\partial\xi)^2}, \tag{6.4}$$

where the maximum of each of these partial derivatives along the three directions x, y, and z is subsequently bounded by one-side differences

$$\begin{aligned} \max|\partial v/\partial x| &\leq \max_{j,k\in\{0,1\}} |f_{1jk} - f_{0jk}| \\ \max|\partial v/\partial y| &\leq \max_{i,k\in\{0,1\}} |f_{i1k} - f_{i0k}| \\ \max|\partial v/\partial z| &\leq \max_{i,j\in\{0,1\}} |f_{ij1} - f_{ij0}|. \end{aligned} \tag{6.5}$$

The construction of a Lipschitz hierarchy (octree) of bounds is built bottom up by using the equations above. Hence, for each cell or unit cube of the original volume the Lipschitz bounds are computed and organized in the corresponding nodes on the base level L_0 of the octree. Then, for each (parent) node on level L_1 the Lipschitz bounds are determined as the maximum of the Lipschitz bounds of the corresponding eight child nodes using the formulae from above. The top level L_N of the octree contains a node, which represents the whole volume.
During rendering efficient octree traversal methods – discussed in the following – can be applied together with the Lipschitz bounds stored in that hierarchy to realize a fast surface based rendering algorithm. For more information about the Lipschitz approach the interested reader is referred to [SH94] and the next subsection.

Efficient Octree Traversal

In volume rendering one of the most frequent operations is the computation of the octree nodes intersected by a straight line, i.e. ray-object intersection tests for the visualization of hierarchical density models by ray-casting. As we saw in the last subsections an octree node has pointers to subsets of objects or voxels that it intersects or contains. Then, by traversing the octree it is possible to restrict the ray-object intersection tests to this set of objects or voxels that the octree node comprises. An improvement on rendering time is only achieved when the traversal process is much faster than the test on all objects or if we can leap over empty regions or process efficiently homogenous regions in the volume

data represented by an octree node. There exist several algorithms for octree traversal, which can be classified into two groups, according to the order in which pierced voxels are obtained:

- Bottom-up methods: Traversing starts at the first leaf node intersected by the ray and a neighbor finding process is used to find the adjacent (next) leaf node [Gla84] [Sam90].
- Top-down methods: Traversing starts at the root node and a (recursive) procedure is used to traverse the descendants intersected by the ray [SW91] [RUL00]. These methods avoid the neighbor finding process, thus saving memory and computation time.

The first of the top-down methods is called the spatial measure for accelerated ray-tracing (SMART) [SW91]. It applies horizontal and vertical steps, i.e. two decision vectors H_{SMART} and V_{SMART}, for the navigation along a ray within the corresponding octree nodes. It is numerically stable and is performed in integer space. However, additional effort has to be carried out for accurate intersection computations.
The second more intuitive algorithm [RUL00] is based on a parametric representation of the ray, thus ray-plane intersections are obtained nearly for free and its representation allows mapping real values to ray points. The algorithm computes the parameter values at which the ray intersects the root octree node. These values are then propagated to the child octree nodes by incrementally computing the new values using additions and multiplications. Due to the employment of comparisons on previously obtained parameter values, it is possible to find the child octree nodes easily. That means, once the first child octree node has been found, the sequence of the remaining traversed child nodes is obtained. This is done by an automaton whose states correspond to the nodes and whose transitions are associated with the movements by the ray, i.e. the current child node and the exit face of the ray according to that node determine the adjacent child node or to return to the parent node (cf. Fig. 6.2).

6.3 Pre-Integration

The evaluation of the rendering integral (cf. Sec. 4) is very time consuming for many volume rendering applications, pre-integration [RKE00] has been published as a technique to avoid high sampling rates by splitting the volume rendering integral into several parts. A similar technique has been first presented in [MHC90] and later pre-integration has been introduced in the context of the volume rendering integral. This acceleration method can be utilized in different volume rendering algorithms, i.e. ray-casting, cell projection, and shear-warp (cf. [EKE01] [GRS+02] [GWGS02]). In this technique and its modifications (cf. [RE02] [MG02] [SKLE03] [WKME03b]) lookup tables are pre-computed for emitted colors and extinction coefficients based on the volume rendering integral. This integral is simplified in a way that it depends only on two or three scalar values (enter, exit density values and the length between enter and exit points) and the color and opacity transfer functions (tables), which itself depend on the scalar values. Thus, a two- or three-dimensional texture is used to store the results of this volume rendering integral, but the look up table (texture) has to be recomputed whenever one

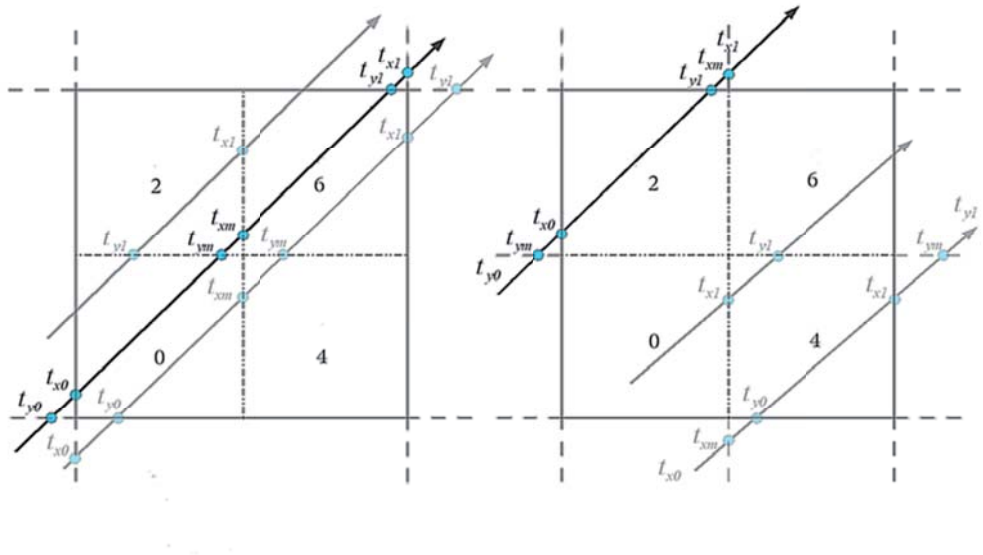


Figure 6.2: *Node selection for efficient octree traversal by ray parameters t.*

of the transfer functions will get changed. Thus, an adaptive pre-integration method has been introduced in [WKME03a]. Here the assumption is, that if the look up table has been computed for length values smaller than or equal to l, then further computation for length values $\tilde{l} = l + d$ are carried out by splitting the integral into two parts. That means, the new entries in the look up table greater than l can be calculated by re-using the values already computed and applying interpolations of that tabulated integrals, a blending operation.

7 Software

A re-invention of the wheel should be avioded, therefore, developers often built their own algorithms (or modifications) on top of already existing packages, i.e. they integrate their new methods into open source or commercial software systems. We shortly summarize some interesting and existing software packages, which can be used for exactly this purpose. We first give some links on commercial volume rendering systems as well as open source packages.

The the volume graphics library (VGL) is a software development environment which integrates three-dimensional surface-based as well as volume-based models into one framework using the well known OpenGL polygon graphic system as well as fast software (e.g. volume ray-tracer) and hardware accelerated volume renderers, respectively. Further, the VGL is a C++ class graphics library which offers a sophisticated application programming interface (API) and a powerful set of algorithms for the visualization as well as the manipulation of three-dimensional voxel data sets. A data visualization and analysis system (GUI), called *VGStudio Max*, is built on top of this library and provides users with interactive rendering and three-dimensional image processing capabilities of data sets up to several GBytes.

Another three-dimensional modular imaging framework is called *MedicView* 3D Graphic Station and supports many common features for a typical three-dimensional medical imaging solution (e.g. segmentation tools, two and three-dimensional measurement tools, analysis tools, and volume and iso-surface rendering).

The *MRIcro* standalone program allows users to view medical images. It includes tools to complement statistical parametric mapping (SPM) and to identify regions of interest as well as for the analysis, efficient viewing and exporting of brain imaging data sequences.

The medical imaging toolkit (*MITK*) is a C++ library for medical image processing as well as analyzing and is inspired by the success of open source softwares VTK and ITK. It is a free software and can be used freely for research and education purpose. Some of its features are, for example, surface reconstruction (enhanced marching cubes algorithm) and rendering, multiple surface rendering, volume rendering with ray-casting, texture-based volume rendering, various segmentation and registration algorithms.

The visualization toolkit [1] (*VTK*) is an open source, freely available software system for three-dimensional computer graphics, image processing, and visualization. It consists of a C++ class library, and several interpreted interface layers including Tcl/Tk, Java, and Python. The toolkit contains a wide range of visualization algorithms, for example, scalar, vector, tensor, texture, and volumetric methods. Further, advanced modeling techniques such as implicit modeling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation are supported.

The following libraries are good candidates for study purposes as well as for further developments of these volume rendering approaches, because all of them are available

[1] Note, professional support and products for VTK are provided by *Kitware*, Inc. .

as source code packages and are not too overloaded with many other sophisticated algorithms.
However, the *OpenQVis* project has the focus on implementing methods for interactive high-quality volume visualization on general purpose hardware, i.e. on desktop computers with state of the art graphics cards. The goal of this project is to obtain high image quality results comparable to traditional ray-casting approaches at interactive frame rates (cf. [RSEH05]). Other source code packages which do fit into this philosophy implement GPU-based ray-casting algorithms (see Fig. 7.1) and can be used to study new features of state of the art graphics hardware architectures (cf. [KL04] [Chr05] [SSKE05]). The shear-warp rendering algorithm can be studied using the original [LL94] package and the volume rendering software *Volsh* which is based on a parallel implementation of the original shear-warp factorization. Note, the shear-warp algorithm is also included in the visualization toolkit (VTK).

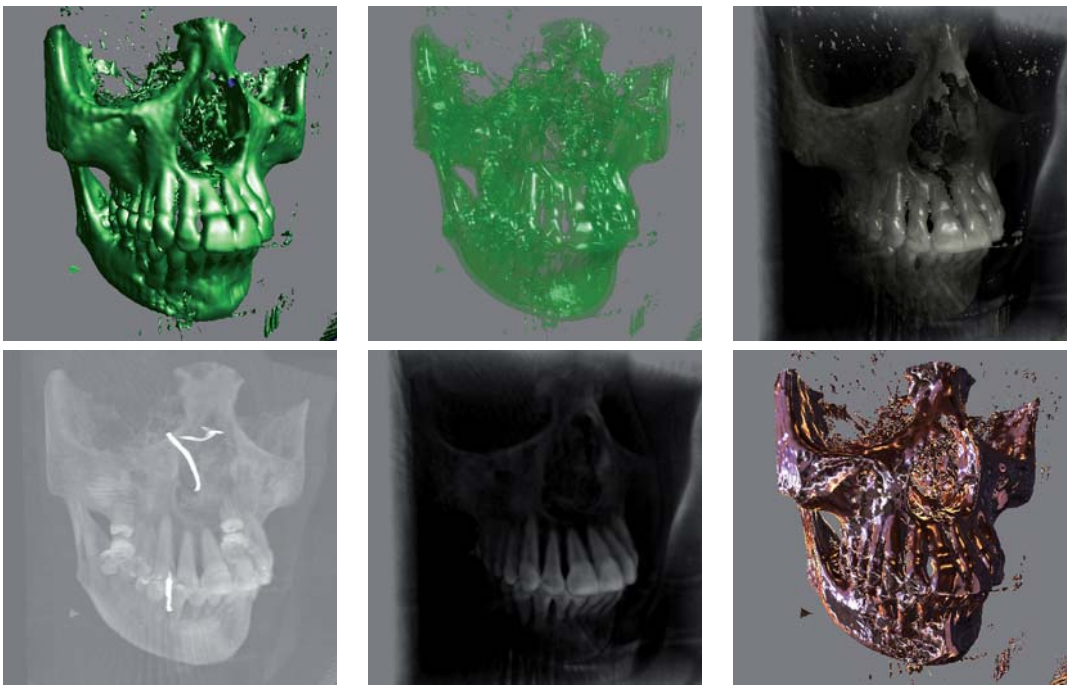


Figure 7.1: *GPU-based ray-casting using the source from [SSKE05]. From left top to right bottom, iso surface rendering, transparent iso surface rendering, volume iso surface rendering, maximum intensity projection, full volume rendering, and iso surface with sphere texture mapping.*

Part II

Spline Models For Volume Reconstruction

1 Introduction

1.1 Related Work

Sampling theories, especially Fourier analysis, give us a tool at hand, which can be used to reconstruct a continuous function from a set of discrete data samples in an optimal way. However, as discussed in part 2 this global approach is difficult to realize in practice. Hence, local reconstruction of functions has been of interest in signal processing for example [MN88] [UAE93a] [UAE93b] [Dod97] and [LGS99] for image and [ML94] [MMMY97b] [MMK+98] [TMG01] for volume reconstruction. On one hand, it is obvious that the overhead is acceptable for local data reconstruction methods (i.e. for piecewise polynomials). These methods are often easy to implement and very efficient as well. On the other hand, they should produce satisfying reconstruction results compared to the above mentioned optimal approach. A general statement here is that local methods yield better reconstructions from a given set of discrete data values if the local support of the filters is increased.
In this context, a piecewise constant model for reconstruction is the simplest model, but the most inaccurate one as well, since it considers only the closest data value or an averaging of some data values in the neighborhood. A more popular model in signal reconstruction is the trilinear interpolation model $\sum_{i,j,k=0}^{1} \mathbf{a}_{ijk} x^i y^j z^k$, where $\mathbf{a}_{ijk} \in \mathbb{R}^n$, i.e. piecewise polynomials of total degree three. This trilinear interpolation allows us to directly obtain gradients from the model. It has the property that the gradients vary bilinearly only. In this view it is not very convenient to use it for high quality shading, because the gradients are not continuous over the piecewise defined region and it usually generates stripe artifacts. That is one reason why approaches based on this model often use central differences or the Sobel operator (see for example [GW02]) to compute gradients. However, other gradient estimation techniques can be found for example in [MMMY97a] [BLM97] [NCKG00] [LM05]. The next natural choice are models of type $\sum_{i,j,k=0}^{m} \mathbf{a}_{ijk} x^i y^j z^k$, where $\mathbf{a}_{ijk} \in \mathbb{R}^n$. If $m = 2, 3$ we have a triquadratic [MJC01] or a tricubic [LHJ99b] model where the polynomial pieces are of total degree six or nine, respectively. The advantages of higher degree models are the smoothness properties, i.e. the appropriate polynomial pieces are C^1 or C^2 continuous, hence, the gradients are directly available from the model and are continuous or even smooth. The main disadvantages of these two models are the increasing data stencils of 3^3 and 4^3 grid points, respectively, and the high total degree of the polynomial pieces (see also [UAE93a] [UAE93b]).
Besides these so called *tensor product splines* defined on rectilinear, volumetric domains Ω, however, there are trivariate spline models defined on so-called *type-6* partitions $\triangle$ which are tetrahedral partitions constructed directly by slicing the rectilinear, volumetric partition $\diamondsuit$ in an appropriate way. Trivariate linear polynomials defined on tetrahedral volumetric domains Δ are in the form of $\sum_{i+j+k\leq 1} \mathbf{a}_{ijk} x^i y^j z^k$, where $\mathbf{a}_{ijk} \in \mathbb{R}^n$ (see [CMS01]). Similar to tensor product splines the next natural choice for trivariate

polynomials of higher degrees are quadratic [RZNS03] [RZNS04a] [NRSZ04] and cubic polynomials [SZ05] of the form $\sum_{i+j+k\leq m} \mathbf{a}_{ijk} x^i y^j z^k$, where $\mathbf{a}_{ijk} \in \mathbb{R}^n$ and $m = 2, 3$. The next sections are organized as follows. We first introduce some basics about Bernstein polynomials and Bézier curves. This material is the foundation for the following sections, even though readers already familiar to that topic can skip that introduction. In Sec. 2 we are going to review tensor product splines in Bézier form defined on volumetric (*type-0*) partitions $\diamondsuit$, some of their properties, techniques and algorithms which are useful, elegant and necessary for volume reconstruction. Sec. 3 is devoted to the counterpart of trivariate spline models in Bézier form defined on *type-6* partitions $\triangle$. We will also discuss properties of these splines, especially in the context of quadratic Super-Splines [RZNS03] [RZNS04a] [NRSZ04] and trivariate cubic splines [SZ05].

1.2 Bernstein Polynomials and Bézier Curves

For the parametrization of curves, surfaces and solids a special basis of the Bernstein polynomials defined on a parameter interval $[a, b]$ is used. Since we can transform each interval $[a, b] := \{t \in \mathbb{R} | a \leq t \leq b\}$ into the unit interval $I := [0, 1] := \{\lambda \in \mathbb{R} | 0 \leq \lambda \leq 1\}$ by an affine transformation $\lambda = (t - a)/(b - a)$ we consider the definition and properties of these special polynomials on this unit interval.

Definition 1.2.1 (Bernstein Polynomials)**.** *The i'th Bernstein polynomial of degree n on the interval I is defined as (cf. Fig. 1.1)*

$$B_i^n(\lambda) \quad = \quad \frac{n!}{(n-i)!i!}(1-\lambda)^{n-i}\lambda^i, \quad i = 0, 1, \ldots, n. \tag{1.1}$$

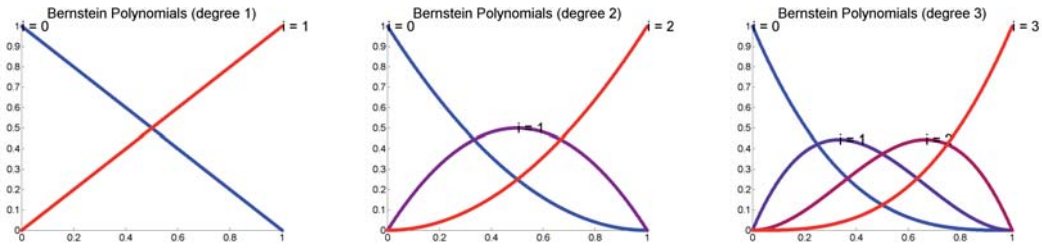


Figure 1.1: *Bernstein Polynomials of degree one (left), two (middle), and three (right), where the colors (from blue to red) indicate the ith polynomial.*

There are some properties of the Bernstein polynomials which turned out to be useful for application and proofs. However, the most important ones are symmetry, boundedness and the maximum principle on the unit interval. Finally the recursion formula is stated in

Theorem 1.2.1 (Bernstein Polynomial Recursion Formula)**.** *The Bernstein polynomials satisfy the recursion*

$$B_i^n(\lambda) \quad = \quad (1-\lambda)B_i^{n-1}(\lambda) + \lambda B_{i-1}^{n-1}(\lambda), \quad \lambda \in \mathbb{R}, \quad i = 1, 2, \ldots, n. \tag{1.2}$$

and the explicit derivatives are specified in

Theorem 1.2.2 (Bernstein Polynomial Derivatives)**.** *The derivatives of the Bernstein polynomials are given by*

$$\frac{\partial}{\partial\lambda}B_i^n(\lambda) \;=\; \begin{cases} -nB_0^{n-1}(\lambda) & \text{if} \quad i=0, \\ +n[B_{i-1}^{n-1}(\lambda) - B_i^{n-1}(\lambda)] & \text{if} \quad i=1,2,\ldots,n-1, \\ +nB_{n-1}^{n-1}(\lambda) & \text{if} \quad i=n. \end{cases} \tag{1.3}$$

These Bernstein polynomials define the basis functions for Bézier curves. The geometric properties of Bézier curves were developed independently by de Casteljau and Bézier and later it was found by Forrest that there is a connection between Bézier curves and the original Bernstein polynomials (cf. notes below and [Far02]). However, we arrive at the so called *Bernstein-Bézier form*

$$s|_I(\lambda) \;=\; \sum_{i=0}^{n} \mathbf{b}_i B_i^n(\lambda), \quad \lambda \in I \tag{1.4}$$

of a polynomial curve, where $\mathbf{b}_i \in \mathbb{R}^d$ are called Bézier points and λ the local coordinate. Together with the Bernstein polynomials the Bernstein-Bézier curve $s|_I(\lambda)$ is defined. The Bézier points as well as the curve have an illustrative and geometric meaning. That means, if the Bézier points become $\mathbf{b}_i := (i/n) \in I$ and are connected according to the natural order of their indices, then we obtain the associated Bézier grid (or net), which defines the convex hull of the polynomial piece (cf. Fig. 1.2). However, if the Bézier points become $\mathbf{b}_i := (i/n, a_i) \in I \times \mathbb{R}$ the resulting spline can be considered as a function-valued formula and the coefficients $a_i \in \mathbb{R}$ as ordinates associated with abscissae $(i/n) \in I$, i.e $(\lambda, s|_I)$ defines a surface in $I \times \mathbb{R}$. Such Bézier splines can be used to approximate a one-dimensional function $f(t)$ as shown in Fig. 1.2. Bézier points of the form $\mathbf{b}_i := (i/n, \mathbf{c}_i) \in I \times \mathbb{R}^l$ can be used to describe a vector-valued function $\mathbf{f}(t) := (f_{x_1}(t), f_{x_2}(t), \ldots, f_{x_l}(t)) \in \mathbb{R}^l$.

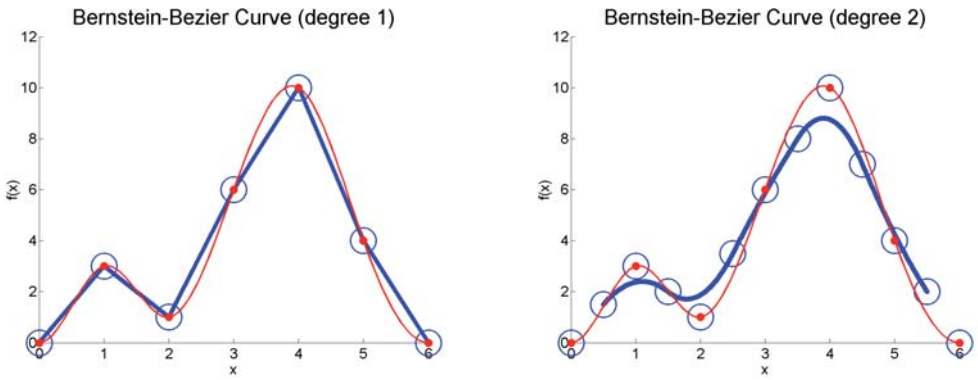


Figure 1.2: *Bernstein-Bézier curve (left) using Bernstein polynomials of degree two (right). The red dots represent some data (e.g. obtained from an explicit polynomial), whereas the blue circles represent the Bézier points usually computed from the data points and the blue curve was reconstructed using Equ. (1.4).*

These curves in Bernstein-Bézier form have some interesting and useful properties as well, which result often directly from the Bernstein polynomials. One should have in mind the most important properties, when developing algorithms using this kind of piecewise polynomials. First, the 1'st derivative of Equ. 1.4 can be computed by

$$\frac{\partial}{\partial\lambda}s|_I(\lambda) \quad = \quad n\sum_{i=0}^{n-1}(\mathbf{b}_{i+1}-\mathbf{b}_i)B_i^{n-1}(\lambda), \tag{1.5}$$

where the 2'nd derivative is written as

$$\frac{\partial^2}{\partial^2\lambda}s|_I(\lambda) \quad = \quad n(n-1)\sum_{i=0}^{n-2}(\mathbf{b}_{i+2}-2\mathbf{b}_{i+1}+\mathbf{b}_i)B_i^{n-2}(\lambda), \tag{1.6}$$

and both equations directly follow from Equ. 1.3. Then, the convex hull property, which is one of the most important properties, speeding up intersection computations between diffenent curves.

Theorem 1.2.3 (Convex Hull Property)**.** *The set of all Bézier points*

$$M \quad := \quad \left\{ s|_I(\lambda)=\sum_{i=0}^{n}\mathbf{b}_iB_i^n(\lambda)| \quad \lambda\in I \right\} \tag{1.7}$$

is contained in the convex hull of the $n+1$ *Bézier points* $\mathbf{b}_i\in\mathbb{R}^d, \quad i=0,1,\ldots,n.$

Smoothness properties are very important and can be used to connect piecewise polynomials in Bernstein-Bézier form, i.e. to set neighboring Bézier points of two different pieces in a way that for example one obtains a total C^1-continuous curve (see Fig. 1.2). In special we have – without loss of generality – two Bernstein-Bézier polynomials $s|_I$ and $s|_J$ defined on two intervals $I:=[a,b]$ and $J:=[c,d]$, where $h_b:=b-a$ and $h_d=d-c$. Then, if $\mathbf{b}_{n,b}=\mathbf{b}_{0,d}$ and $\mathbf{b}_{n,b}=\frac{h_d}{h_b+h_d}\mathbf{b}_{n-1,b}+\frac{h_b}{h_b+h_d}\mathbf{b}_{1,d}$ we obtain a C^1-continuous patch which is built up by these two piecewise polynomials. The final property which is useful to evaluate piecewise polynomials in Bernstein-Bézier form is the *de Casteljau* algorithm. This algorithm is numerically stable and directly results from the recursion formula (1.2).

Algorithm 1.2.1 (Univariate de Casteljau)**.** *The input to this algorithm is an array of* $n+1$ *Bézier points* $\mathbf{b}_i\in\mathbb{R}^d, \quad i=0,1,\ldots,n$ *(i.e. the polynomial piece is of total degree* n*) and the local coordinate* $\lambda\in I$*. Here* $\mathbf{b}_i^0:=\mathbf{b}_i$ *are the Bézier points from Equ. (1.4) defining the convex hull of the polynomial piece and the algorithm to compute the value* $s|_I(\lambda)$ *is*

1: ***for*** $j=1<=n$ ***do***
2: ***for*** $i=0<=n-j$ ***do***
3: {*Compute second derivative.*}
4: ***if*** *j==n-1* ***then***
5: $\frac{\partial^2}{\partial^2\lambda}s|_I(\lambda)=n(n-1)(\mathbf{b}_{i+2}^{j-1}-2\mathbf{b}_{i+1}^{j-1}+\mathbf{b}_i^{j-1})$
6: ***end if***
{*Compute first derivative.*}
7: ***if*** *j==n* ***then***

8: $\frac{\partial}{\partial\lambda} s|_I(\lambda) = n(\mathbf{b}_{i+1}^{j-1} - \mathbf{b}_i^{j-1})$
9: ***end if***
{*Compute value.*}
10: ***if*** $j==n$ ***then***
11: $s|_I(\lambda) = (1-\lambda)\mathbf{b}_i^{j-1} + \lambda\mathbf{b}_{i+1}^{j-1}$
12: ***end if***
{*Recursion to compute new values on layer* j.}
13: $\mathbf{b}_i^j = (1-\lambda)\mathbf{b}_i^{j-1} + \lambda\mathbf{b}_{i+1}^{j-1}$
14: ***end for***
15: ***end for***

As output we obtain the result $s|_I(\lambda) = \mathbf{b}_0^n$*, i.e. the value on the one-dimensional curve, and the first and second partial derivatives (see also Equ. 1.5 and 1.6). Note that in step* j *the old values* $\mathbf{b}_i^{j-1}$ *and* $\mathbf{b}_{i+1}^{j-1}$ *in the array can be replaced by the newly computed values* $\mathbf{b}_i^j$*. There is no need to allocate space for another temporary array.*

However, the Bernstein polynomials are used to construct one-dimensional curves in Bernstein-Bézier form. Similarly, but with little variation of the notation, the same polynomials are utilized for the parametrization of surfaces and solids defined over triangular and tetrahedral partitions. For the one-dimensional case the Bernstein polynomials can be rewritten as follows.

Definition 1.2.2 (Bernstein Polynomials)**.** *The Bernstein polynomials of degree* n *on the interval* I *are defined as*

$$B_{\tau_0,\tau_1}^n(\lambda_0,\lambda_1) = \frac{n!}{\tau_0!\tau_1!}\lambda_0^{\tau_0}\lambda_1^{\tau_1}, \quad |\tau_0+\tau_1| = n, \tag{1.8}$$

where $B_{\tau_0,\tau_1}^n(\lambda_0,\lambda_1) = 0$ *if* $\tau_\nu \notin \{0,1,\ldots,n\}$ *for some* $\nu \in \{0,1\}$*,* $\lambda_\nu \geq 0$*,* $\sum_{\nu=0}^1 \lambda_\nu = 1$*, and* λ_ν *are called the barycentric coordinates on the interval* I*. Finally, if* $\tau_0 := (n-i)$*,* $\tau_1 := i$*,* $\lambda_0 := (1-\lambda)$*, and* $\lambda_1 := \lambda$ *we arrive at Def. 1.2.1.*

Therefore, given an interval I with $\{\lambda_\nu \in \mathbb{R}, \quad \nu \in \{0,1\} \mid 0 \leq \lambda_\nu \leq 1, \quad \sum_{\nu=0}^1 \lambda_\nu = 1\}$ and two points p_0, p_1 at the interval boundaries any point on a line can be expressed in terms of barycentric coordinates λ_ν with respect to this non degenerate interval I as $p = \sum_\nu \lambda_\nu p_\nu$. Now, Equ. (1.4) can be rewritten as

$$s|_I(\lambda_0,\lambda_1) = \sum_{|\tau_0+\tau_1|=n} \mathbf{b}_{\tau_0,\tau_1} B_{\tau_0,\tau_1}^n(\lambda_0,\lambda_1) \tag{1.9}$$

$$= \sum_{|\tau_0+\tau_1|=n} \mathbf{b}_{\tau_0,\tau_1} \frac{n!}{\tau_0!\tau_1!}\lambda_0^{\tau_0}\lambda_1^{\tau_1}, \tag{1.10}$$

where $\mathbf{b}_{\tau_0,\tau_1} \in \mathbb{R}^2$ are called the Bézier points defined as $\mathbf{b}_{\tau_0,\tau_1} := ((\tau_0,\tau_1)/n, a_{\tau_0,\tau_1})$, where a_{τ_0,τ_1} are the Bézier ordinates associated with the abscissas $((\tau_0,\tau_1)/n)$.
Note, the classical *Horner* scheme – also numerically stable – is an analogous method to evaluate a polynomial at a given point location $\lambda = \lambda_0$ which is written in terms of the monomial basis. Whereas, the *de Boor* algorithm is a generalization of *de Casteljau's* algorithm for evaluating B-spline curves, which is fast and numerically stable as well.

Remark (Bernstein-Bézier Curves)**.** *Sergei Natanovich Bernstein an Ukrainian mathematician (1880-1968) has introduced new polynomials in a constructive proof of the "Stone-Weierstrass" approximation theorem. These "Bernstein" polynomials have been used later by Paul de Casteljau (1910 - 1999) – an engineer at Citroän – for the approximation of curves. At the same time Pierre Bézier (1910-1999) came to the same curves working for the competitor Renault. Robin Forrest realized that de Casteljau and Bézier did the same job. Then, the noun "Bernstein-Bézier" curve appears, and due to the progress of computer graphics these curves become more and more popular (cf. [Far02]).*

Figure 1.3: *Biographical profiles of Sergei Natanovich Bernstein (left), and Pierre Bézier (right). By courtesy of Wikipedia*

2 Tensor Product Bézier Splines

Tensor product splines in Bernstein-Bézier form are usually defined on finite rectilinear, volumetric domains $\Omega \subset \mathbb{R}^3$.

2.1 Uniform Cube Partition

In the following we denote a uniform rectilinear, volumetric partition $\diamondsuit$ also as a *type-0* uniform cube partition, because there are no oblique slicing planes which further subdivide the volumetric partition. However, a uniform partition $\diamondsuit$ of the cubic domain Ω is obtained as follows. Basically, the volumetric domain $\mathbb{R}^3$ is restricted to a rectangular domain $\Omega := [0, L] \times [0, M] \times [0, N] \subset \mathbb{R}^3$. This is further split into a uniform rectilinear, volumetric partition $\diamondsuit$, where every cube $Q \in \diamondsuit$ has side length of 1 (cf. Fig. 2.1). Volume data can be considered as a set of $(L + 2) \times (M + 2) \times (N + 2)$

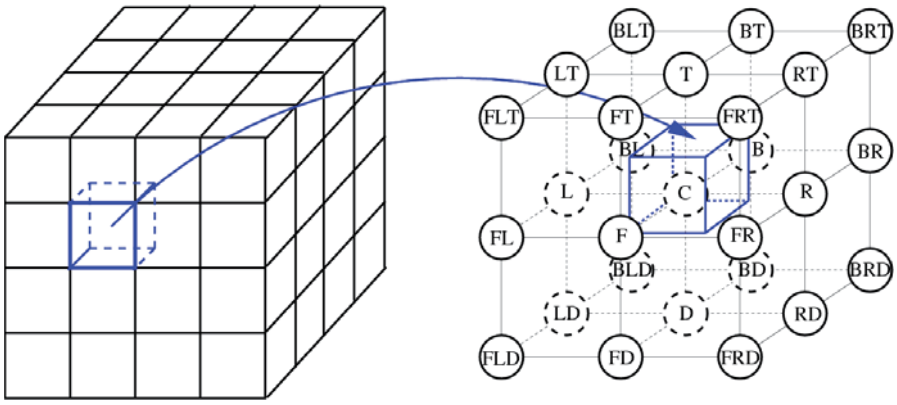


Figure 2.1: *The illustration of a uniform cube partition, where the domain $\mathbb{R}^3$ is restricted into the cubic domain Ω, which is further split into uniform cube partition $\diamondsuit$. Each cube $Q_{i,j,k} \in \diamondsuit$ has side length 1 and the corresponding spline $s|_Q$ is defined in the domain Ω (left). The currently considered unit cube $Q_0 = [-0.5, +0.5]^3$ (blue cube) with its data value $f_{i,j,k}$ located at the **C**enter $\mathbf{v}_{i,j,k}$ of that unit cube. The local neighborhood of data values $f_{i+i_0,j+j_0,k+k_0}$ with basic lablels **F**ront, **B**ack, **L**eft, **R**ight, **D**own, and **T**op is located on the center positions $\mathbf{v}_{i+i_0,j+j_0,k+k_0}$ (white dots) of the corresponding unit cubes $Q_{i+i_0,j+j_0,k+k_0}$, $i_0, j_0, k_0 \in \{-1, 0, +1\}$ (right).*

grid points of the form $\mathbf{v}_{i,j,k} = (\frac{2i+1}{2}, \frac{2j+1}{2}, \frac{2k+1}{2}) \in \Omega$ with corresponding data values $f_{i,j,k} := f(\mathbf{v}_{i,j,k}) \in \mathbb{R}$, $i = -1, \ldots, L$, $j = -1, \ldots, M$, $k = -1, \ldots, N$. Each interior grid point $\mathbf{v}_{i,j,k}$, i.e. where $i \notin \{-1, L\}$, $j \notin \{-1, M\}$, $k \notin \{-1, N\}$, is located at the center of a corresponding cube

$$Q_{i,j,k} \quad = \quad [i, i+1] \times [j, j+1] \times [k, k+1]. \tag{2.1}$$

For each cube some Bernstein-Bézier coefficients located on the corresponding Bézier points on $Q_{i,j,k}$ – where exactly depends on the degree of the splines – are obtained from the volume data. These Bézier points and their coefficients – usually 8, 27, or 64 on $Q_{i,j,k}$ – define piecewise linear, quadratic, or cubic splines in Bernstein-Bézier form, where the polynomial pieces have total degree of 3, 6, or 9 on $Q_{i,j,k}$, respectively. As we already know from the one-dimensional case, appropriate smoothness conditions are used to determine the Bernstein-Bézier coefficients for the corresponding Bézier points. Both together define a continuous patch build up by several piecewise splines. The same procedure can be applied in the volumetric domain Ω, where the smoothness conditions have to be satisfied along each of the three directions and between all cubes $Q_{i,j,k} \in \diamondsuit$. Here, the coefficients for the piecewise spline representation (2.5) are determined by repeated averaging of the data values while satisfying natural appropriate smoothness conditions (see for example [MJC01]). Once the location of the Bézier points on the cubes $Q_{i,j,k}$ and the averaging rules – which are symmetric – to compute the coefficients are determined and because of the uniform structure of the partition $\diamondsuit$, each cube can be considered separately. That means, a considered location in the domain Ω can be used to find the position – the indices i, j, k – in the partition $\diamondsuit$ and thus each cube $Q_{i,j,k}$ can be transformed into an *unit cube* $Q_0 = [-0.5, +0.5]^3$ [1], which is confined by six planes of the general form

$$P_\nu(x, y, z, d) = ax + by + cz + d = 0, \tag{2.2}$$

where the planes for $\nu = 6, \ldots, 11$ are

$$\begin{aligned}
P_6^Q(x, y, z, d) &= +1x + 0y + 0z - 1d, \\
P_7^Q(x, y, z, d) &= +0x + 1y + 0z - 1d, \\
P_8^Q(x, y, z, d) &= +0x + 0y + 1z - 1d, \\
P_9^Q(x, y, z, d) &= +1x + 0y + 0z + 1d, \\
P_{10}^Q(x, y, z, d) &= +0x + 1y + 0z + 1d, \\
P_{11}^Q(x, y, z, d) &= +0x + 0y + 1z + 1d,
\end{aligned} \tag{2.3}$$

with $d = 0.5$. Similarly the grid points $\mathbf{v}_{i+i_0,j+j_0,k+k_0}$ with their corresponding data values $f_{i+i_0,j+j_0,k+k_0}$, $i_0, j_0, k_0 \in \{-1, 0, +1\}$, which are in the local neighborhood of the current considered cube $Q_{i,j,k}$ can be transformed into the local neighborhood of the unit cube Q_0 as well (cf. Fig. 2.1). These are used to determine the coefficients of the polynomial piece (2.5) by some averaging formula. The following notation (also illustrated in Fig. 2.1) is necessary to keep these formula short, and for a better geometric understanding. The value $f_{i,j,k}$ at the **C**enter $\mathbf{v}_{i,j,k}$ of the cube $Q_0 := Q_{i,j,k}$ is defined as

$$C := f_{i,j,k}.$$

The given values in its **F**ront, **B**ack, **L**eft, **R**ight, **D**own, and **T**op neighboring boxes ($Q_{i-1,j,k}$, $Q_{i+1,j,k}$, $Q_{i,j-1,k}$, $Q_{i,j+1,k}$, $Q_{i,j,k-1}$, and $Q_{i,j,k+1}$), respectively, are defined as

$$\begin{aligned}
F &:= f_{i-1,j,k}, \; B := f_{i+1,j,k}, \\
L &:= f_{i,j-1,k}, \; R := f_{i,j+1,k}, \\
D &:= f_{i,j,k-1}, \; T := f_{i,j,k+1}.
\end{aligned}$$

[1] This interval $[-0.5, +0.5]^3$ can be easily transformed by an affine transformation into the new interval $[0, 1]^3$ if necessary.

Similarly, the given values in the **F**ront-**L**eft, **F**ront-**R**ight, **F**ront-**D**own, and **F**ront-**T**op boxes ($Q_{i-1,j-1,k}$, $Q_{i-1,j+1,k}$, $Q_{i-1,j,k-1}$, and $Q_{i-1,j,k+1}$) of Q_0, respectively, are defined as

$$\begin{aligned} FL &:= f_{i-1,j-1,k},\ FR := f_{i-1,j+1,k}, \\ FD &:= f_{i-1,j,k-1},\ FT := f_{i-1,j,k+1}. \end{aligned}$$

Finally, applying the same scheme as above all other values located at the centers of the remaining neighboring boxes of Q_0 are defined as

$$\begin{aligned} BL &:= f_{i+1,j-1,k},\ BR := f_{i+1,j+1,k}, \\ BD &:= f_{i+1,j,k-1},\ BT := f_{i+1,j,k+1}, \\ LD &:= f_{i,j-1,k-1},\ LT := f_{i,j-1,k+1}, \\ RD &:= f_{i,j+1,k-1},\ RT := f_{i,j+1,k+1}, \end{aligned}$$

and

$$\begin{aligned} FLD &:= f_{i-1,j-1,k-1},\ FLT := f_{i-1,j-1,k+1}, \\ FRD &:= f_{i-1,j+1,k-1},\ FRT := f_{i-1,j+1,k+1}, \\ BLD &:= f_{i+1,j-1,k-1},\ BLT := f_{i+1,j-1,k+1}, \\ BRD &:= f_{i+1,j+1,k-1},\ BRT := f_{i+1,j+1,k+1}. \end{aligned}$$

2.2 Bézier Form

The step from piecewise curves in Bernstein-Bézier form to surfaces or volumes in the same form is straight forward. Three parameters are necessary to describe a volume and are now denoted as $\lambda_0, \lambda_1, \lambda_2 \in [0,1]$. The basic idea of the definition of Bernstein-Bézier models for volumes is to start with three curves in Bernstein-Bézier form and vary them along each parameter direction independently.
Piecewise continuous splines on uniform cube partitions which may satisfy some smoothness conditions are called here *type-0 splines* better known as *tensor product splines*. The space of these splines with respect to $\diamondsuit$ is defined by

$$\mathcal{S}_{l+m+n}(\diamondsuit) = \{s \in C^\pi(\Omega) : s|_Q \in \mathcal{P}_{l+m+n}, \forall Q \in \diamondsuit\}, \tag{2.4}$$

where $\mathcal{P}_{l+m+n} := \text{span}\{x^i y^j z^k : i,j,k \geq 0, i+j+k \leq l+m+n\}$ denotes the X-dimensional space of $l+m+n$ degree polynomials, i.e. the space of trivariate polynomials of total degree $l+m+n$ and $C^\pi(\Omega)$ is the set of π-times continuously differentiable functions on Ω.
Piecewise tensor product splines of degree (l,m,n) are now defined on each unit cube Q and can be written as

$$s|_Q(\lambda_0,\lambda_1,\lambda_2) = \sum_{i,j,k=0}^{l,m,n} \mathbf{b}_{i,j,k} B_i^l(\lambda_0) B_j^m(\lambda_1) B_k^n(\lambda_2), \tag{2.5}$$

where $\lambda_\nu \in [0,1]$, $\nu \in \{0,1,2\}$ are local coordinates. If the Bézier points become $\mathbf{b}_{i,j,k} := (i/l, j/m, k/n) \in Q$ and are connected according to the natural order of their

indices, then we obtain the associated Bézier grid, which defines the convex hull of the polynomial piece. In case the Bézier points become $\mathbf{b}_{i,j,k} := (i/l, j/m, k/n, a_{i,j,k}) \in Q \times \mathbb{R}$ the resulting spline can be considered as a function-valued formula and the coefficients $a_{i,j,k} \in \mathbb{R}$ as ordinates associated with abscissae $\vartheta^{l,m,n}_{i,j,k} := (i/l, j/m, k/n) \in Q$. Now $(\lambda_0, \lambda_1, \lambda_2, s|_Q)$ defines a hyper-surface in $Q \times \mathbb{R}$. On one side, this definition of tensor product splines, i.e. such a hyper-surface, can be used to describe a spatial density or opacity field in three-dimensional domain. On the other side a vector-valued formula where $\mathbf{b}_{i,j,k} := (\vartheta^{l,m,n}_{i,j,k}, \mathbf{c}_{i,j,k}) \in Q \times \mathbb{R}^d$ can be used to describe vector-valued field, i.e. velocity or color, in a three-dimensional domain.
However, the trivariate version of Alg. 1.2.1 to evaluate the polynomial piece $p = s|_Q \in \mathcal{P}_{l+m+n}$ in the above formulation can be described as follows. Strictly speaking, the univariate algorithm is applied along each direction λ_ν by always reusing the previously computed results. In other words, from a given tensor array of $(l+1) \times (m+1) \times (n+1)$ Bézier points new intermediate points are computed by a linear combination of the appropriate points along the parametric line described by λ_0, i.e along the l dimension of the tensor array. This results in a matrix of $(m+1) \times (n+1)$ intermediate Bézier points, where the application of the univariate algorithm with λ_1 along the m dimension of that matrix will give us new intermediate points stored in a vector of size $n+1$. In the last step the result is found by utilizing the univariate algorithm with λ_2 along the vector of intermediate Bézier points. There are different orders as well as techniques [HL93] on how to evaluate Equ. 2.5, one is shown in the following algorithm (see also [MD95]).

Algorithm 2.2.1 (Trivariate de Casteljau on Type-0 Partitions)**.** *The input to this algorithm is a three-dimensional array of* $(l+1) \times (m+1) \times (n+1)$ *Bézier points* $\mathbf{b}_{i,j,k} \in \mathbb{R}^d$ *(i.e. the polynomial piece is of total degree* $l+m+n$*) and the local coordinates* $(\lambda_0, \lambda_1, \lambda_2) \in Q$*. Here* $\mathbf{b}^{0,0,0}_{i,j,k} := \mathbf{b}_{i,j,k}$ *are the Bézier points from Equ. (2.5) defining the convex hull of the polynomial piece and the algorithm to compute the value* $s|_Q(\lambda_0, \lambda_1, \lambda_2)$ *is*

1: {*Averaging along* λ_0 *direction.*}
2: $p = q = 0$
3: ***for*** $k = 0 <= n$ ***do***
4: ***for*** $j = 0 <= m$ ***do***
5: ***for*** $o = 1 <= l$ ***do***
6: ***for*** $i = 0 <= l - o$ ***do***
7: $\mathbf{b}^{o,p,q}_{i,j,k} = (1-\lambda_0)\mathbf{b}^{o-1,p,q}_{i,j,k} + \lambda_0 \mathbf{b}^{o-1,p,q}_{i+1,j,k}$
8: ***end for***
9: ***end for***
10: ***end for***
11: ***end for***
{*Averaging along* λ_1 *direction.*}
12: $q = 0, o = l, i = 0$
13: ***for*** $k = 0 <= n$ ***do***
14: ***for*** $p = 1 <= m$ ***do***
15: ***for*** $j = 0 <= m - p$ ***do***
16: $\mathbf{b}^{o,p,q}_{i,j,k} = (1-\lambda_1)\mathbf{b}^{o,p-1,q}_{i,j,k} + \lambda_1 \mathbf{b}^{o,p-1,q}_{i,j+1,k}$
17: ***end for***

18: ***end for***
19: ***end for***
{*Averaging along* λ_2 *direction.*}
20: $o = l, p = m, i = j = 0$
21: ***for*** $q = 1 <= n$ ***do***
22: ***for*** $k = 0 <= n - q$ ***do***
23: $\mathbf{b}_{i,j,k}^{o,p,q} = (1 - \lambda_2)\mathbf{b}_{i,j,k}^{o,p,q-1} + \lambda_2 \mathbf{b}_{i,j,k+1}^{o,p,q-1}$
24: ***end for***
25: ***end for***

As output we obtain the result $s|_I(\lambda_0, \lambda_1, \lambda_2) = \mathbf{b}_{0,0,0}^{l,m,n}$*, and the first and second partial derivatives (see also Equ. 2.6). For reasons of clearness the partial derivative computations are not shown here. Note, the superscripts* o, p, q *are used to indicate that the new or intermediate Bézier points are stored in a new place, i.e. in a temporary tensor array. This is of course not necessary and as in the univariate case the original values can be overwritten.*

The 1'st partial derivatives of Equ. (2.5) are given according to Equ. (1.3) as

$$\begin{aligned}
\frac{\partial}{\partial\lambda_0} s|_Q(\lambda_0, \lambda_1, \lambda_2) &= l \sum_{i,j,k=0}^{l-1,m,n} (\mathbf{b}_{i+1,j,k} - \mathbf{b}_{i,j,k}) B_i^{l-1}(\lambda_0) B_j^m(\lambda_1) B_k^n(\lambda_2), \\
\frac{\partial}{\partial\lambda_1} s|_Q(\lambda_0, \lambda_1, \lambda_2) &= m \sum_{i,j,k=0}^{l,m-1,n} (\mathbf{b}_{i,j+1,k} - \mathbf{b}_{i,j,k}) B_i^l(\lambda_0) B_j^{m-1}(\lambda_1) B_k^n(\lambda_2), \\
\frac{\partial}{\partial\lambda_2} s|_Q(\lambda_0, \lambda_1, \lambda_2) &= n \sum_{i,j,k=0}^{l,m,n-1} (\mathbf{b}_{i,j,k+1} - \mathbf{b}_{i,j,k}) B_i^l(\lambda_0) B_j^m(\lambda_1) B_k^{n-1}(\lambda_2). \qquad (2.6)
\end{aligned}$$

Once we have determined the cube Q_0 which contains a point $\mathbf{q} \in \Omega$ and its local coordinates $\lambda_\nu(\mathbf{q})$, $\nu = 1, 2, 3$ according to Q_0 are computed – this will be discussed in the following sections. We are able to evaluate the polynomial piece (2.5) and its derivatives at $\mathbf{q}$ using λ_ν, the Bézier points $\mathbf{b}_{i,j,k}$ [2] and Alg. 2.2.1. For shading of surfaces obtained from the volume model at a point $\mathbf{q}$ it is necessary to compute the 1'st partial derivatives (the gradient)

$$(\nabla s|_Q)(\mathbf{q}) = \left(\frac{\partial}{\partial x} s|_Q(\mathbf{q}), \frac{\partial}{\partial y} s|_Q(\mathbf{q}), \frac{\partial}{\partial z} s|_Q(\mathbf{q}) \right)$$

at the same location $\mathbf{q}$. Thus, if we assume ξ_ν, $\nu \in \{0, 1, 2\}$ to be the parametric lines along the three directions of the three main edges of Q_0 with length 1, then the partial derivative of $s|_Q$ according to $\mathbf{q}$ along the line ξ_ν is denoted by $\frac{\partial}{\partial\xi_\nu} s|_Q \in \mathcal{P}_{n-1}$ and is given by Equ. 2.6, where $\frac{\partial}{\partial x} s|_Q(\mathbf{q}) := \frac{\partial}{\partial\lambda_0} s|_Q(\lambda_0, \lambda_1, \lambda_2)$, $\frac{\partial}{\partial y} s|_Q(\mathbf{q}) := \frac{\partial}{\partial\lambda_1} s|_Q(\lambda_0, \lambda_1, \lambda_2)$, and $\frac{\partial}{\partial z} s|_Q(\mathbf{q}) := \frac{\partial}{\partial\lambda_2} s|_Q(\lambda_0, \lambda_1, \lambda_2)$.

2.3 Point Location and Local Coordinates

Before we are able to evaluate the polynomial piece (2.5), i.e. compute the value and the derivatives of the spline $s|_Q$ at a given point $\mathbf{q} \in \Omega$, we need to know the location of $\mathbf{q}$

[2] Once the cube Q_0 is found, all its Bézier points $\mathbf{b}_{i,j,k}$ are known as well.

in the partition $\diamondsuit$ and its local coordinates. Hence, we have to determine a cube $Q \in \diamondsuit$ with $\mathbf{q} \in Q$ and the local coordinates of $\mathbf{q}$ with respect to the cube Q. The indices of the cube Q are found by simple rounding the coordinates of $\mathbf{q} = (q_x, q_y, q_z)$, i.e. we have $Q_0 = Q_{\lfloor q_x \rfloor, \lfloor q_y \rfloor, \lfloor q_z \rfloor}$, where $\lfloor \cdot \rfloor$ denotes the *floor* operator [3]. The local coordinates $\lambda_\nu, \quad \nu \in \{0, 1, 2\}$ with respect to the cube Q are found using $\mathbf{q}$ and are associated with $\lambda_0 := q_x - \lfloor q_x \rfloor$, $\lambda_1 := q_y - \lfloor q_y \rfloor$, and $\lambda_2 := q_z - \lfloor q_z \rfloor$.

2.4 Piecewise Linear Splines

Piecewise linear splines are continuous but not continuously differentiable – here such splines are called *linear type-0 splines*. The space of these splines with respect to $\diamondsuit$ is defined by

$$\mathcal{S}_{1+1+1}(\diamondsuit) \quad = \quad \{s \in C^0(\Omega) : s|_Q \in \mathcal{P}_{1+1+1}, \forall Q \in \diamondsuit\}, \tag{2.7}$$

where $\mathcal{P}_{1+1+1} := \text{span}\{x^i y^j z^k : i, j, k \geq 0, i + j + k \leq 3\}$ denotes the 19-dimensional space of cubic polynomials, i.e. the space of trivariate polynomials of total degree three. A spline $s \in \mathcal{S}_{1+1+1}$ can be written in its piecewise Bernstein-Bézier form (2.5), i.e. in that equation we set $l = m = n = 1$.

2.4.1 Bernstein-Bézier Coefficients

For piecewise linear splines the 8 Bernstein-Bézier coefficients for an arbitrary unit cube $Q \in \diamondsuit$ are determined as follows. Each Bernstein-Bézier coefficient $a_{i,j,k}$ and its corresponding domain point $\vartheta^{1,1,1}_{i,j,k}$ located on Q define the appropriate Bézier point $\mathbf{b}_{i,j,k} := (\vartheta^{1,1,1}_{i,j,k}, a_{i,j,k})$. Further, every corner vertex $\mathbf{v}_\nu$ of Q (blue circles in right drawing of Fig. 2.2) corresponds to a domain point, i.e. $\vartheta^{1,1,1}_{1,0,1} := \mathbf{v}_0, \vartheta^{1,1,1}_{1,1,1} := \mathbf{v}_1, \vartheta^{1,1,1}_{1,0,0} := \mathbf{v}_2, \vartheta^{1,1,1}_{1,1,0} := \mathbf{v}_3, \vartheta^{1,1,1}_{0,0,1} := \mathbf{v}_4, \vartheta^{1,1,1}_{0,1,1} := \mathbf{v}_5, \vartheta^{1,1,1}_{0,0,0} := \mathbf{v}_6$, and $\vartheta^{1,1,1}_{0,1,0} := \mathbf{v}_7$. The appropriate four front coefficients $a_{1,0,1}, a_{1,1,1}, a_{1,0,0}$, and $a_{1,1,0}$ are determined by

$$\begin{aligned}
a_{1,0,1} &= \frac{1}{8}(C + F + L + T + LT + FL + FT + FLT), \\
a_{1,1,1} &= \frac{1}{8}(C + F + R + T + RT + FR + FT + FRT), \\
a_{1,0,0} &= \frac{1}{8}(C + F + L + D + LD + FL + FD + FLD), \\
a_{1,1,0} &= \frac{1}{8}(C + F + R + D + RD + FR + FD + FRD),
\end{aligned} \tag{2.8}$$

where the other coefficients in the back face of the cube are computed accordingly. However, a setting of the coefficients as discussed above leads to an overall piecewise approximating spline function, i.e. the splines do not interpolate the original data values such as for example C, F, etc. . Nevertheless, interpolating splines can be easily constructed by shifting the unit cube by the half of its size in each direction. This has of course to be done for each cube of the volume in the same manner. In other words the above coefficients located on the domain points in the front face of the unit cube could be

[3] That means $\lfloor b \rfloor$ is the maximal integer $\leq b$.

set to $a_{1,0,1} = FL, a_{1,1,1} = F, a_{1,0,0} = FLD$, and $a_{1,1,0} = FD$, where the coefficients for the back facing domain points could be simply set as $a_{0,0,1} = L, a_{0,1,1} = C, a_{0,0,0} = LD$, and $a_{0,1,0} = D$.

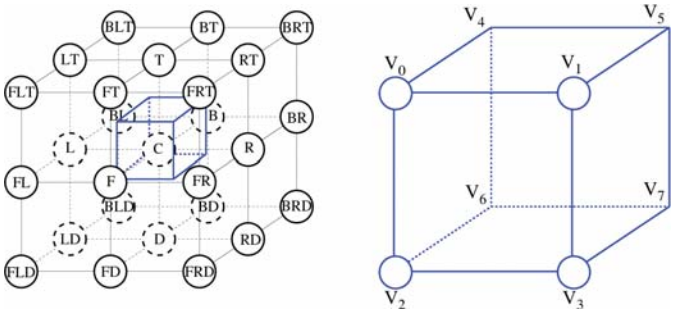


Figure 2.2: *The data values (white circles in the left drawing) in the local* 27 *neighborhood of a considered cube* Q_0 *(blue cube) and appropriate averaging rules – for linear splines – give the* 4 *Bernstein-Bézier coefficients located on its Bézier points (blue circles in the right illustration) of a unit cube* Q*. The other four coefficients located in the back face of* Q_0 *are computed by symmetry and are not shown here.*

2.4.2 Evaluation of Polynomial Pieces and its Derivatives

For the evaluation of polynomial pieces $s|_Q \in \mathcal{P}_3$ we have to consider 8 Bézier points $\mathbf{b}_{i,j,k}$ (blue circles in Fig. 2.3) on the unit cube Q. To obtain the value or the derivative of the polynomial piece $s|_Q(\lambda_0, \lambda_1, \lambda_2)$ at the Bézier point located at $\mathbf{q}$ (green dot in Fig. 2.3) we proceed as follows. Each pair of the Bézier points is linearly combined in the first direction (e.g. along the local x axis of the unit cube) using the appropriate local coordinates $\lambda_0, 1-\lambda_0$ according to Q (indicated as arrows in Fig. 2.3). This results in four new Bézier points (cyan dots in Fig. 2.3). These points undergo the same procedure along the second direction using different local coordinates (i.e. $\lambda_1, 1-\lambda_1$) which results in only one pair of new Bézier points (yellow dots in Fig. 2.3). Now, to obtain the value of $s|_Q$ at position $\mathbf{q}$ (green dot in Fig. 2.3), in the final step once more this pair is linearly combined along the third direction using the last two local coordinates $(\lambda_2, 1-\lambda_2)$. On the other hand, to obtain the directional derivative $\frac{\partial}{\partial\lambda_2} s|_Q(\mathbf{q}) \in \mathcal{P}_2$ along the edge of λ_2 of $s|_Q$ at position $\mathbf{q}$, in the final step the difference of the pair of Bézier points (yellow dots in Fig. 2.3) is taken. Similarly, to obtain the directional derivative in the other direction, e.g. $\frac{\partial}{\partial\lambda_0} s|_Q(\mathbf{q}) \in \mathcal{P}_2$. We first apply Alg. 1.2.1 or Alg. 2.2.1 along the directions corresponding to the local coordinates $\lambda_2, 1-\lambda_2$ and $\lambda_1, 1-\lambda_1$ or vice verse and afterwards take the difference of the final pair of Bézier points along the direction corresponding to $\lambda_0, 1-\lambda_0$.

2.5 Piecewise Quadratic Splines

Piecewise quadratic splines do satisfy additional smoothness conditions in comparison to the previously defined linear splines – in this thesis such splines are called *quadratic*

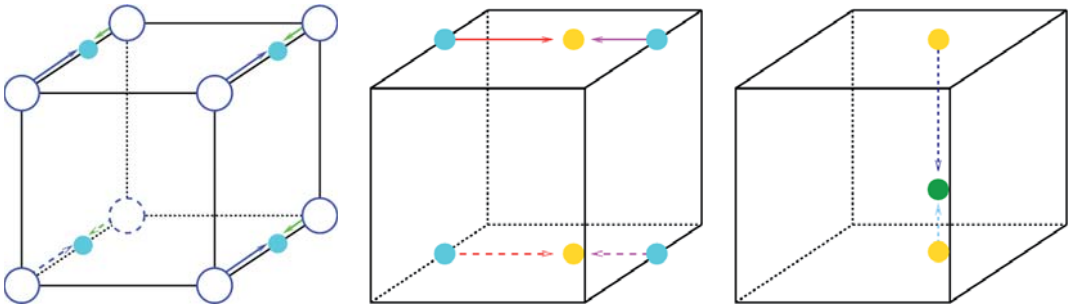


Figure 2.3: *The evaluation of a polynomial piece $s|_Q \in \mathcal{P}_3$ at a point $\mathbf{q}$ (green dot) with the de Casteljau algorithm 2.2.1. The 8 blue circles show the Bézier points $\mathbf{b}_{i,j,k}$ of Equ. 2.5. The arrows show which of the points are combined by weighting with the local coordinates of $\mathbf{q}$ w.r.t. Q. The different colors represent the local coordinates $\lambda_\nu(\mathbf{q})$, $\nu \in \{0,1,2\}$, where $\lambda_0(\mathbf{q}), (1-\lambda_0(\mathbf{q}))$ are indicated by blue and green arrows, $\lambda_1(\mathbf{q}), (1-\lambda_1(\mathbf{q}))$ by red and magenta arrows, and $\lambda_2(\mathbf{q}), (1-\lambda_2(\mathbf{q}))$ by light and dark blue arrows, respectively.*

type-0 splines. The space of these splines with respect to $\diamondsuit$ is defined by

$$\mathcal{S}_{2+2+2}(\diamondsuit) \quad = \quad \{s \in C^1(\Omega) : s|_Q \in \mathcal{P}_{2+2+2}, \forall Q \in \diamondsuit\}, \tag{2.9}$$

where $\mathcal{P}_{2+2+2} := \text{span}\{x^i y^j z^k : i,j,k \geq 0, i+j+k \leq 6\}$ denotes the X-dimensional space of six degree polynomials, i.e. the space of trivariate polynomials of total degree six. A spline $s \in \mathcal{S}_{2+2+2}$ can be written in its piecewise Bernstein-Bézier form (cf. Equ. 2.5), i.e. we set $l = m = n = 2$ in that equation.

2.5.1 Bernstein-Bézier Coefficients

For piecewise quadratic splines 27 Bernstein-Bézier coefficients for an arbitrary cube $Q \in \diamondsuit$ have to be determined. Once more each Bernstein-Bézier coefficient $a_{i,j,k}$ and its corresponding domain point $\vartheta^{2,2,2}_{i,j,k}$ located on Q define the appropriate Bézier point $\mathbf{b}_{i,j,k} := (\vartheta^{2,2,2}_{i,j,k}, a_{i,j,k})$. Here also every corner vertex $\mathbf{v}_\nu$ of Q (blue circles in right drawing of Fig. 2.4) corresponds to a domain point, that means now we define the front domain points as $\vartheta^{2,2,2}_{2,0,2} := \mathbf{v}_0$, $\vartheta^{2,2,2}_{2,2,2} := \mathbf{v}_1$, $\vartheta^{2,2,2}_{2,0,0} := \mathbf{v}_2$, and $\vartheta^{2,2,2}_{2,2,0} := \mathbf{v}_3$. And the back domain points are set to $\vartheta^{2,2,2}_{0,0,2} := \mathbf{v}_4$, $\vartheta^{2,2,2}_{0,2,2} := \mathbf{v}_5$, $\vartheta^{2,2,2}_{0,0,0} := \mathbf{v}_6$, and $\vartheta^{2,2,2}_{0,2,0} := \mathbf{v}_7$. The appropriate four coefficients $a_{2,0,2}, a_{2,2,2}, a_{2,0,0}$, and $a_{2,2,0}$ at the front face of Q are determined by the same formulas as shown above in case of *linear type-0 splines*. However, the next four coefficients $a_{2,1,2}, a_{2,2,1}, a_{2,1,0}$, and $a_{2,0,1}$ are determined by

$$\begin{aligned} a_{2,1,2} &= \frac{1}{4}(C+F+T+FT), \\ a_{2,2,1} &= \frac{1}{4}(C+F+R+FR), \\ a_{2,1,0} &= \frac{1}{4}(C+F+D+FD), \\ a_{2,0,1} &= \frac{1}{4}(C+F+L+FL). \end{aligned} \tag{2.10}$$

The corresponding domain points $\vartheta^{2,2,2}_{2,1,2} := (\mathbf{v}_0 + \mathbf{v}_1)/2, \vartheta^{2,2,2}_{2,2,1} := (\mathbf{v}_1 + \mathbf{v}_3)/2, \vartheta^{2,2,2}_{2,1,0} := (\mathbf{v}_3 + \mathbf{v}_2)/2$, and $\vartheta^{2,2,2}_{2,0,1} := (\mathbf{v}_2 + \mathbf{v}_0)/2$ are located at the midpoints (red circles in right drawing of Fig. 2.4) of the edges connecting two adjacent vertices $\mathbf{v}_\nu$ of a unit cube. The next domain point $\vartheta^{2,2,2}_{2,1,1} := (\mathbf{v}_0 + \mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3)/4$ is located at the center of a face (green circle in right drawing of Fig. 2.4) of a unit cube, where the appropriate coefficient is determined from

$$a_{2,1,1} \quad = \quad \frac{1}{2}(C + F). \tag{2.11}$$

Finally, the Bernstein-Bézier coefficient $a_{1,1,1}$ associated with the domain point $\vartheta^{2,2,2}_{1,1,1}$ located in the center of the cube Q (cyan circle in right drawing of Fig. 2.4) is simply set to $a_{1,1,1} = C$, where $C := f_{i,j,k}$ is the data value itself located in the center of a cube Q. The remaining 17 coefficients follow directly from symmetry, hence the formulas are not given here, but can be easily constructed from figure 2.4.

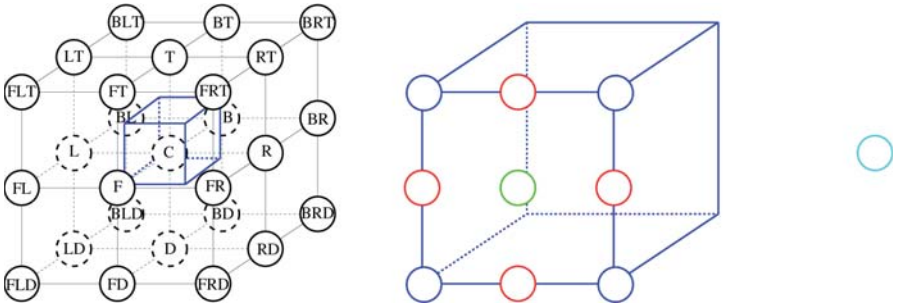


Figure 2.4: *The data values (white circles in the left drawing) in the local* 27 *neighborhood of a considered cube Q_0 (blue cube) and appropriate averaging rules – for quadratic splines – give the* 10 *Bernstein-Bézier coefficients located on its Bézier points (blue, red, green and cyan circles in the right illustration) of a unit cube Q. The other* 17 *coefficients are computed by symmetry and are not shown here for the sake of clarity.*

2.5.2 Evaluation of Polynomial Pieces and its Derivatives

For the evaluation of the polynomial pieces $s|_Q \in \mathcal{P}_6$ we have to consider the 27 Bézier points (blue, red, green and cyan circles in Fig. 2.5) on a unit cube Q. The procedure is very similar applied for piecewise linear polynomials. First, we linearly combine all pairs of Bézier points along the first direction using the appropriate local coordinates (i.e. $\lambda_0, 1 - \lambda_0$). The resulting 18 new Bézier points (cyan dots in Fig. 2.5) are then pairwise linearly combined along the second direction using local coordinates $\lambda_1, 1 - \lambda_1$, and finally the 12 new Bézier points (yellow dots in Fig. 2.5) are linearly combined along the third direction using $\lambda_2, 1 - \lambda_2$. This results in 8 Bézier points on layer $l = 1$ (green dots in Fig. 2.5) to which again a very similar procedure is applied. However, basically the Bézier points on layer $l = 1$ can be supplied to the algorithm used for piecewise linear polynomials. Note, as mentioned before there are many different ways how to implement the de Casteljau algorithm, i.e. the procedure discussed here is slightly different compared to Alg. 2.2.1. However, the reason is that this approach allows faster

reconstructions of the data especially if partial derivatives are needed for high quality shading computations.

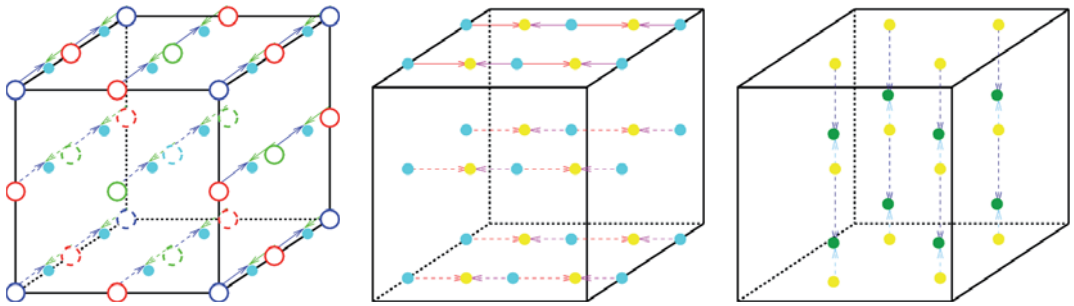


Figure 2.5: *Evaluation of a polynomial piece $s|_Q \in \mathcal{P}_6$ at a point $\mathbf{q}$ with the de Casteljau algorithm. Top row: The 27 blue, red, green and cyan circles show the Bézier points $\mathbf{b}_{i,j,k}$ of Equ. 2.5. The arrows show which of the points are combined by weighting with the local coordinates of $\mathbf{q}$ w.r.t. Q. The different colors represent the local coordinates $\lambda_\nu(\mathbf{q})$, $\nu \in \{0,1,2\}$, where $\lambda_0(\mathbf{q}), (1-\lambda_0(\mathbf{q}))$ are indicated by blue and green arrows, $\lambda_1(\mathbf{q}), (1-\lambda_1(\mathbf{q}))$ by red and magenta arrows, and $\lambda_2(\mathbf{q}), (1-\lambda_2(\mathbf{q}))$ by light and dark blue arrows, respectively. The resulting 8 new points $\mathbf{c}_{i,j,k}$ on level $l = 1$ indicated as green dots and the local coordinates $\lambda_\nu(\mathbf{q})$, $\nu \in \{0,1,2\}$ are the input values for the next iteration, i.e. basically the algorithm for piecewise linear polynomials can be applied (cf. Fig. 2.3).*

2.6 Simple Ray-Casting

2.6.1 Intersection Computations

In standard ray-casting algorithms rays

$$\mathbf{r}_{\nu,\mu}(t) \quad = \quad \mathbf{r}_s^{\nu,\mu} + t\, \mathbf{r}_d^{\nu,\mu} \tag{2.12}$$

are cast into the volume (object space) from each image pixel $\nu := 0, 1, \ldots, N$ and $\mu := 0, 1, \ldots, M$, where here $N \times M$ is the size of the image. Further, $\mathbf{r}_s^{\nu,\mu}$ and $\mathbf{r}_d^{\nu,\mu} \in \mathbb{R}^3$ are the ray start and direction in object space of the corresponding pixel ν, μ. The ray start position is located in the projection plane, whereas the (normalized) ray direction is obtained from the difference of the current ray start and the position of the viewer $\mathbf{r}_o$ located in object space as well, i.e. $\mathbf{r}_d^{\nu,\mu} := \mathbf{r}_s^{\nu,\mu} - \mathbf{r}_o$ [4]. Now, an arbitrary ray casted through Ω results in two intersection points, which are called the enter point $\mathbf{q}_1 \in \Omega$ and the exit point $\mathbf{q}_2 \in \Omega$ of a ray $\mathbf{r}_{\nu,\mu}(t)$ coming from pixel ν, μ. This is computed by intersecting the ray with the six planes limiting Ω (cf. Fig. 2.6). If the ray does not intersect Ω no contribution from the volume is computed nor stored into the appropriate pixel ν, μ of that ray. Otherwise we must process all the cubes $Q \in \diamondsuit$ which are intersected by the ray. Thus by using the entrance point $\mathbf{q}_1$ we determine the first unit cube $\tilde{Q} := Q_1$ with $\mathbf{q}_1 \in \tilde{Q}$ and the first local coordinates (in fact the local ray start

[4] If the viewer position is infinitely far away from the object itself, then the ray direction remain the same for all rays casted into object space for a fixed view transformation.

according to $\tilde{Q}$) $\tilde{\mathbf{r}}_s := \tilde{\lambda} := (\tilde{\lambda}_1, \tilde{\lambda}_2, \tilde{\lambda}_3)$ as described above. Setting the ray direction as $\tilde{\mathbf{r}}_d := \mathbf{r}_d^{\nu,\mu}$ we obtain a local ray

$$\tilde{\mathbf{r}}(t) \quad = \quad \tilde{\mathbf{r}}_s + t\ \tilde{\mathbf{r}}_d, \tag{2.13}$$

according to $\tilde{Q}$ with $\tilde{\mathbf{r}}_s := (\tilde{r}_{s,x}, \tilde{r}_{s,y}, \tilde{r}_{s,z})$ and $\tilde{\mathbf{r}}_d := (\tilde{r}_{d,x}, \tilde{r}_{d,y}, \tilde{r}_{d,z})$. From the intersection of this local ray $\tilde{\mathbf{r}}(t)$ and the general plane equation, see Equ. (2.2), the ray parameter t can be found by

$$t \quad = \quad -\frac{d + a\ \tilde{r}_{s,x} + b\ \tilde{r}_{s,y} + c\ \tilde{r}_{s,z}}{a\ \tilde{r}_{d,x} + b\ \tilde{r}_{d,y} + c\ \tilde{r}_{d,z}}, \tag{2.14}$$

where if $a\ \tilde{r}_{d,x} + b\ \tilde{r}_{d,y} + c\ \tilde{r}_{d,z} = 0$ no intersection occurs, thus the ray is parallel to the considered plane. Applying each of the six planes of Equ. (2.3) to the above formula (2.14) – i.e. setting the parameters a, b, c and d appropriately – gives us six ray parameters $t_i,\ i = 6, \ldots, 11$. Then using Equ. (2.13) and $\tilde{t} = \min_i(t_i)$ the nearest exit intersection $\tilde{\mathbf{r}}_e$ of the ray and the current unit cube $\tilde{Q}$ is found as well as the exit plane $P_i^{\tilde{Q}}$ (see Fig.2.6). Once the current unit cube $\tilde{Q}$ is determined – thus also the corresponding Bernstein-Bézier coefficients – as well as the local ray enter $\tilde{\mathbf{r}}_s$ and exit $\tilde{\mathbf{r}}_e$ positions, and the ray direction $\tilde{\mathbf{r}}_d$, the evaluation of the polynomial pieces and its derivatives along the ray can be applied.
Of course, all contributions of the unit cubes intersected by the ray $\mathbf{r}_{\nu,\mu}(t)$ should be determined in the local way described above. Here, basically the idea of *Bresenham's* line drawing algorithm is applied (see also Siddon's method [Sid85] [FRD06] often applied for fast voxel and polygon ray-tracing algorithms in intensity modulated radiation therapy treatment planning). Once we have found the first unit cube $\tilde{Q} := Q_1$, the enter and exit planes $P_j^{Q_1}, P_i^{Q_1}$ intersected by a ray, we can determine easily from the exit plane $P_i^{Q_1}$ using a look-up table which unit cube should be considered next. Similarly, the new local ray start in the next unit cube Q_2 can be found from the exit position $\tilde{\mathbf{r}}_e$ using modulo computation, i.e. if we assume the exit plane in cube Q_1 as $P_i^{Q_1} := P_6^Q$, then the enter plane in cube Q_2 would be $P_j^{Q_2} := P_9^Q$, where the new ray start position in cube Q_2 would be computed as $\tilde{\mathbf{r}}_s := (\tilde{r}_{e,x} - 1, \tilde{r}_{e,y}, \tilde{r}_{e,z})$ using the exit position $\tilde{\mathbf{r}}_e$ in cube Q_1 (cf. Fig. 2.6)

2.6.2 Univariate Polynomial Pieces

For real volume or iso-surface rendering the local polynomial pieces according to Q have to be evaluated. Thus if $\tilde{\mathbf{r}}_s = \tilde{\mathbf{r}}(t_s)$ and $\tilde{\mathbf{r}}_e = \tilde{\mathbf{r}}(t_e)$, where $t_s := t_0 := 0 < t_e := t_1 := \tilde{t}$, are two intersection points of $\tilde{\mathbf{r}}(t)$ and Q. The restriction of the polynomial piece p to the line segment $[\tilde{\mathbf{r}}_s, \tilde{\mathbf{r}}_e]$ is a univariate polynomial of degree 3 or 6 (cf. right images in Fig. 2.6). The necessary equations of 3rd ($N = 2$) or 6th ($N = 5$) degree are set up by computing the values $\tilde{w}_s$, $\tilde{w}_i$, and $\tilde{w}_e$ at positions $\tilde{\mathbf{r}}_s$, $\tilde{\mathbf{r}}_i = \tilde{\mathbf{r}}(i/(N+1))$, and $\tilde{\mathbf{r}}_e$, where $i = 1, \ldots, N$ and $N \in \{2, 5\}$. By using the Newton interpolation form, the unique polynomials

$$f(\tau) \quad = \quad \alpha_{N+1}\tau^{N+1} + \alpha_N\tau^N + \ldots + \alpha_0\tau^0 \tag{2.15}$$

on the appropriate interval $[0, t_e]$, which interpolates the values $\tilde{w}_s$, $\tilde{w}_i$, and $\tilde{w}_e$ at the points $0, (i/(N+1))$, and t_e can be easily found. In general one has to solve a linear

system of equations to obtain the coefficients $\alpha_j,\ j = 0, \ldots, N+1$ of the polynomial above. However, these coefficients can be pre-computed (i.e. expressed by explicit equations) for low polynomial degrees.

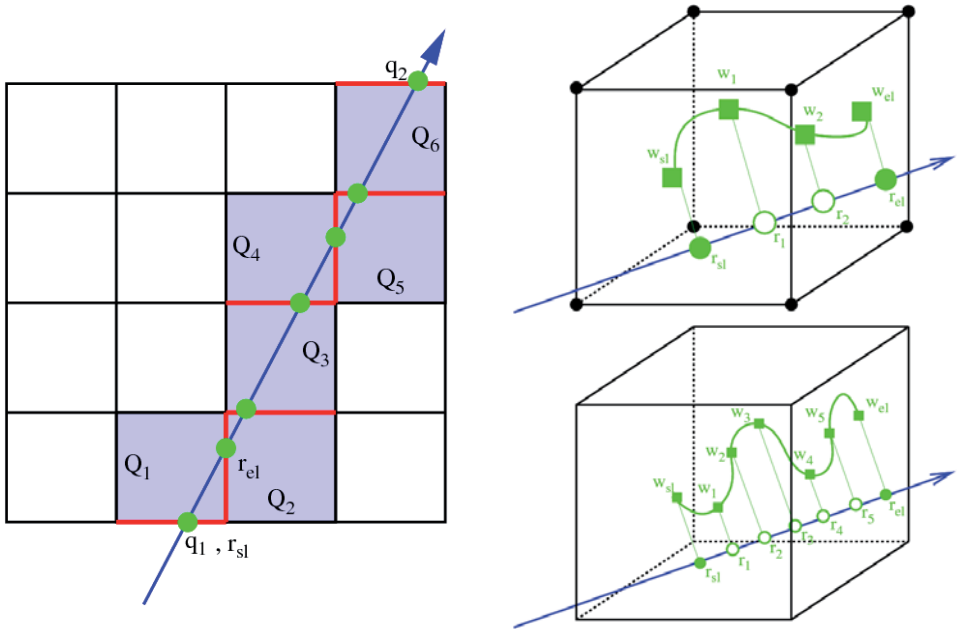


Figure 2.6: *Left: Ray-Casting a volumetric cube partition (simplified two-dimensional view). The intersection positions (green dots) of the (blue) ray and the unit cubes (blue rectangles) define the intersection planes (red lines), which are used to find the next local start position of the ray segment according to the next local unit cube. Right: On each unit cube univariate polynomial pieces of total degree* 3 *(top) and* 6 *(bottom) are defined. The green solid dots* $\mathbf{r}_{sl}$ *and* $\mathbf{r}_{el}$ *indicate the intersections of the ray and the unit cube (as in left image), where* w_{sl} *and* w_{el} *are the corresponding values (green rectangles) as obtained by the de Casteljau algorithm. The green circles* $\mathbf{r}_{sl} < \mathbf{r}_i < \mathbf{r}_{el},\ i = 1, \ldots, N$ *are intermediate positions along the ray, where values (green rectangles)* w_i *are obtained by the de Casteljau algorithm as well. All values are used to define polynomials in Newton form of total degree* 3 *(right top image) and* 6 *(right bottom image).*

3 Trivariate Bézier Splines

3.1 Uniform Tetrahedral Partition

A suitable uniform tetrahedral partition $\triangle$ is obtained from $\diamondsuit$. Here, basically all unit cubes $Q_{i,j,k} \in \diamondsuit$ are subdivided by first drawing two diagonals into each of the six faces of $Q_{i,j,k}$. Then the center $\mathbf{v}_{i,j,k}$ of the unit cube $Q_{i,j,k}$ is connected with the eight vertices as well as with the intersection positions of the two diagonals drawn into the six faces of $Q_{i,j,k}$. Thus each unit cube is split into six pyramids, where each pyramid is further subdivided into four tetrahedra of the same form. However, this procedure decomposes each unit cube $Q_{i,j,k}$ into 24 uniform tetrahedra $T_{i,j,k,l}$ yielding a tetrahedral partition $\triangle \subseteq \diamondsuit$ of the domain Ω, i.e. $T_{i,j,k,l} \subset Q_{i,j,k}, \forall l \in \{0,\ldots,23\}$. This is called a *type-6* tetrahedral partition because it is alternatively described as the result of slicing each box $Q_{i,j,k}$ with six different planes of the general form (2.2), where the planes for $\nu = 0,\ldots,5$ are

$$\begin{aligned} P_0^Q(x,y,z,d) &= +1x + 1y + 0z + 0d, \\ P_1^Q(x,y,z,d) &= +0x + 1y + 1z + 0d, \\ P_2^Q(x,y,z,d) &= +1x + 0y + 1z + 0d, \\ P_3^Q(x,y,z,d) &= +1x - 1y + 0z + 0d, \\ P_4^Q(x,y,z,d) &= +0x + 1y - 1z + 0d, \\ P_5^Q(x,y,z,d) &= -1x + 0y + 1z + 0d, \end{aligned} \tag{3.1}$$

with $d = 0.0$, and for each of the 24 tetrahedra $T \subseteq Q$ some Bernstein-Bézier coefficients located on the corresponding Bézier points are associated as well. These Bézier points and their coefficients – usually 4(15), 10(65), or 20(175) on $T(Q)$ – define piecewise linear, quadratic, or cubic splines in Bernstein-Bézier form, where the polynomial pieces have total degree 1, 2, or 3 on T, respectively. The Bézier points are located on $T(Q)$ – where exactly depends on the degree of the splines – and are obtained from the volume data. Similar to tensor product splines appropriate smoothness conditions have to be satisfied along each direction of the nine-directional mesh [1] and between all tetrahedra $T \in \triangle$. Once the location of the Bézier points on T and the averaging rules – which are symmetric as well – to compute the coefficients are found and because of the uniform structure of the partition $\triangle$ and $\diamondsuit$, each cube Q containing 24 congruent tetrahedra T can be considered separately (cf. Fig. 3.1).
Thus, if the location in the partition $\diamondsuit$ (the indices i,j,k) from the current location $\mathbf{q}$ in the domain Ω is found – which gives the current unit cube $Q_0 = Q_{i,j,k}$ – and the local coordinates of $\mathbf{q}$ according to Q_0 are computed (see Sec. 2.3), the tetrahedral partition

[1] This can be considered as a trivariate generalization of the four-directional mesh well-known in the bivariate spline theory.

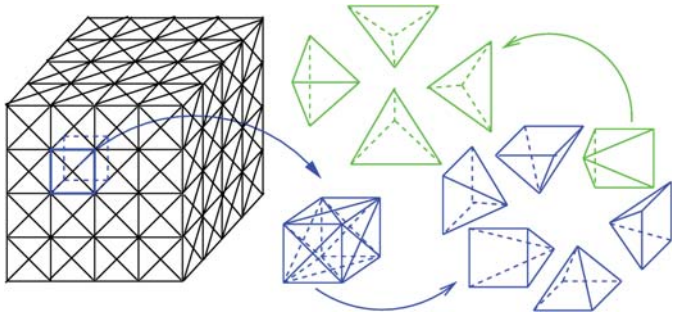


Figure 3.1: *The limitation of the domain $\mathbb{R}^3$ into the cubic domain Ω. This is split into a uniform cubic partition $\diamondsuit$ and further into a uniform tetrahedral partition $\triangle$. Each cube $Q_{i,j,k} \in \diamondsuit$ has side length 1, where each spline s_T of the corresponding tetrahedron $T_{i,j,k,l} \in \triangle \subseteq \diamondsuit \ni Q_{i,j,k}$ with $T_{i,j,k,l} \subset Q_{i,j,k}$ is defined in the tetrahedral domain $\Delta \subseteq \Omega$ (left). The currently considered unit cube $Q_0 = [-0.5, +0.5]^3$ (where the local neighborhood of data values is determined as well, cf. Fig. 2.1) can be viewed as being subdivided into six pyramides, where each pyramid is further decomposed into four tetrahedra (right).*

of such a unit cube Q_0 can be considered only (cf. Fig. 3.2). In other words, each tetrahedron $T_{i_0},\ i_0 = 0, \ldots, 23$ can be considered as a subset of the unit cube $Q_0 = [-0.5, +0.5]^3$, i.e $T_{i_0} \subset Q_0,\ \forall i_0$. This allows defining several lookup tables. The first stores the transition from a tetrahedron T_{i_0} to its three (or four) neighboring tetrahedra

$$T_{i_0} \quad \rightarrow \quad [T_{i_1}, T_{i_2}, T_{i_3}(, T_{i_4})] \tag{3.2}$$

with $i_0, i_1, i_2, i_3, i_4 \in \{0, \ldots, 23\}$, where $i_0 \neq i_1 \neq i_2 \neq i_3 \neq i_4$. Here, T_{i_4} is the adjacent tetrahedron of T_{i_0}, which is located in the neighboring unit cube $\tilde{Q}_0$ of Q_0. The second saves the transition from a tetrahedron T_{i_0} to its four confining planes as defined in equations (2.3) and (3.1)

$$T_{i_0} \quad \rightarrow \quad [P^Q_{j_1}, P^Q_{j_2}, P^Q_{j_3}, P^Q_{j_4}] \tag{3.3}$$

with $j_1, j_2, j_3, j_4 \in \{0, \ldots, 11\}$, where $j_1 \neq j_2 \neq j_3 \neq j_4$. Here, $P^Q_{j_4}$ is always one of the planes from (2.3), i.e. each tetrahedron $T_{i_0},\ i_0 = 0, \ldots, 23$ is confined by three planes from (3.1) and one plane from (2.3), thus one face of a tetrahedron coincides with a face of the unit cube Q_0. The next table stores the transition from a tetrahedron T_{i_0} and its plane $P^Q_{j_0}$ (one of its four confining planes) to the adjacent tetrahedron

$$[T_{i_0}, P^Q_{j_0}] \quad \rightarrow \quad [T_{i_1}] \tag{3.4}$$

with $i_0, i_1 \in \{0, \ldots, 23\}$ and $j_0 \in \{0, \ldots, 11\}$, where $i_0 \neq i_1$. The last table defines a transition from a tetrahedron T_{i_0} to its four vertices

$$T_{i_0} \quad \rightarrow \quad [\mathbf{v}_{k_0}, \mathbf{v}_{k_1}, \mathbf{v}_{k_2}, \mathbf{v}_{k_3}], \tag{3.5}$$

where the vertex $\mathbf{v}_{k_0}$ is located at the center of the unit cube Q_0, thus by definition it is set to $\mathbf{v}_{k_0} := (0, 0, 0)$ (since the local coordinate system is placed in the center of the unit cube). Here, the vertex $\mathbf{v}_{k_1}$ is of the form $(\omega_1, \omega_2, \omega_3)$, where $\omega_\alpha = \omega_\beta = 0$ and $\omega_\gamma = \pm 0.5$ for some $\alpha, \beta, \gamma \in \{1, 2, 3\}$, where once more $\alpha \neq \beta \neq \gamma$ and $\mathbf{v}_{k_2}, \mathbf{v}_{k_3}$ are the vertices located on the corners of the unit cube Q_0 (cf. Fig. 3.2).

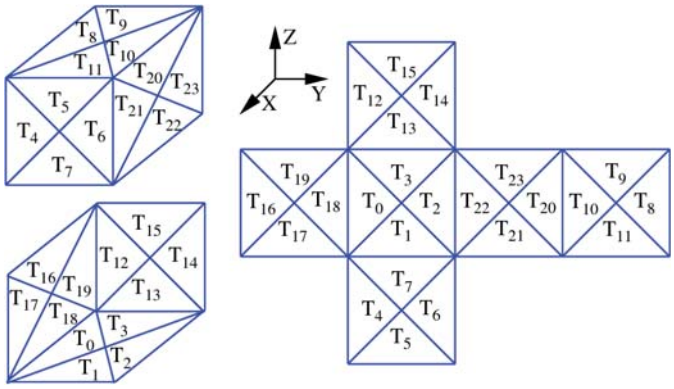


Figure 3.2: *Volumetric uniform tetrahedral partition of the unit cube Q_0 into 24 congruent tetrahedra, where each tetrahedron is numbered appropriately (left). Each face of the unit cube is flipped open (right).*

3.2 Bézier Form

Piecewise continuous splines on uniform tetrahedral partitions which may satisfy some smoothness conditions are called here *type-6 splines*. The space of these splines with respect to $\triangle$ is defined by

$$\mathcal{S}_n(\triangle) \quad = \quad \{s \in C^\pi(\Omega) : s|_T \in \mathcal{P}_n, \forall T \in \triangle\}, \tag{3.6}$$

where $\mathcal{P}_n := \text{span}\{x^i y^j z^k : i,j,k \geq 0, i+j+k \leq n\}$ denotes the X-dimensional space of n degree polynomials, i.e. the space of trivariate polynomials of total degree n and $C^\pi(\Omega)$ is the set of π-times continuously differentiable functions on Ω.
A spline $s \in \mathcal{S}_n$ can be written in its piecewise Bernstein-Bézier form (cf. Equ. 1.10 and [Far86]) and is now defined on each tetrahedron $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] \in \triangle$ on the domain Δ as

$$p = s|_T(\lambda_0, \lambda_1, \lambda_2, \lambda_3) = \sum_{|\tau_0+\tau_1+\tau_2+\tau_3|=n} \mathbf{b}_{\tau_0,\tau_1,\tau_2,\tau_3} B^n_{\tau_0,\tau_1,\tau_2,\tau_3}(\lambda_0, \lambda_1, \lambda_2, \lambda_3), \tag{3.7}$$

where the sum goes over $(\tau_0, \tau_1, \tau_2, \tau_3)$ which satisfy $|\tau_0 + \tau_1 + \tau_2 + \tau_3| = n$ and $\tau_0 + \tau_1 + \tau_2 + \tau_3 \geq 0$. The coefficients $\mathbf{b}_{\tau_0,\tau_1,\tau_2,\tau_3} := (\vartheta^n_{\tau_0,\tau_1,\tau_2,\tau_3}, a_{\tau_0,\tau_1,\tau_2,\tau_3}) \in T \times \mathbb{R}$ are called the Bézier points of s on T, where $\vartheta^n_{\tau_0,\tau_1,\tau_2,\tau_3} := (\tau_0\mathbf{v}_0 + \tau_1\mathbf{v}_1 + \tau_2\mathbf{v}_2 + \tau_3\mathbf{v}_3)/n$ in T are called the domain points and $a_{\tau_0,\tau_1,\tau_2,\tau_3}$ are the corresponding Bernstein-Bézier coefficients. According to Def. 1.2.2 the Bernstein polynomials of degree n become now

$$B^n_{\tau_0,\tau_1,\tau_2,\tau_3}(\lambda_0, \lambda_1, \lambda_2, \lambda_3) = \frac{n!}{(\tau_0!\tau_1!\tau_2!\tau_3!)} \lambda_0^{\tau_0}\lambda_1^{\tau_1}\lambda_2^{\tau_2}\lambda_3^{\tau_3} \in \mathcal{P}_n, \tag{3.8}$$

where $\lambda_\nu \in \mathcal{P}_1 = \text{span}\{1, x, y, z\}$, $\nu = 0, 1, 2, 3$ are the barycentric coordinates with respect to T and are determined by interpolation conditions $\lambda_\nu(\mathbf{v}_\mu) = \delta_{\nu,\mu}$, $\mu = 0, 1, 2, 3$ (where $\delta_{\nu,\mu}$ denotes Kroneckers symbol), i.e. for any point $\mathbf{q} \in \mathbb{R}^3$ the barycentric coordinates $\lambda_\nu(\mathbf{v}_\mu) \in \mathbb{R}$ according to $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$ are determined as the solution of the 4×4 linear system

$$\begin{pmatrix} \mathbf{q} \\ 1 \end{pmatrix} \quad = \quad \lambda_0(\mathbf{q}) \begin{pmatrix} \mathbf{v}_0 \\ 1 \end{pmatrix} + \lambda_1(\mathbf{q}) \begin{pmatrix} \mathbf{v}_1 \\ 1 \end{pmatrix} + \lambda_2(\mathbf{q}) \begin{pmatrix} \mathbf{v}_2 \\ 1 \end{pmatrix} + \lambda_3(\mathbf{q}) \begin{pmatrix} \mathbf{v}_3 \\ 1 \end{pmatrix}, \tag{3.9}$$

where e.g. $\mathbf{v}_0 := \mathbf{v}_{i,j,k}$ is the center of Q, $\mathbf{v}_1 = (\mathbf{v}_{i,j,k} + \mathbf{v}_{i-1,j,k})/2$ is the center of a face F of Q (i.e. the intersection position of the two diagonals drawn into that face F) and $\mathbf{v}_2 = (\mathbf{v}_{i,j,k} + \mathbf{v}_{i-1,j-1,k+1})/2, \mathbf{v}_3 = (\mathbf{v}_{i,j,k} + \mathbf{v}_{i-1,j+1,k+1})/2$ are the two vertices of an edge of that face F. In this sense the *type-6* partition is symmetric and using the above Bernstein-Bézier form one can conveniently describe smoothness conditions across common triangular faces of tetrahedrons T and $\tilde{T}$. From practical point of view splines satisfying some smoothness conditions have great importance, e.g. for high quality visualizations.
The Bernstein-Bézier coefficients $a_{\tau_0,\tau_1,\tau_2,\tau_3}$ in the piecewise form (3.7) of an n degree spline s_f on $\triangle$ of a continuous function f are directly obtained from the data values $f_{i,j,k}$ at the corresponding points $\mathbf{v}_{i,j,k}$ lying at the centers of the corresponding cubes $Q_{i,j,k}$. Thus, to obtain the unique coefficient $a_{\tau_0,\tau_1,\tau_2,\tau_3}$ for each domain point $\vartheta^n_{\tau_0,\tau_1,\tau_2,\tau_3}$ in the tetrahedron $T \subset Q := Q_{i,j,k} \in \diamondsuit$ the 27 neighboring data values $f_{i+i_0,j+j_0,k+k_0} := f(\mathbf{v}_{i+i_0,j+j_0,k+k_0})$ of the continuous function f at the points $\mathbf{v}_{i+i_0,j+j_0,k+k_0}, i_0, j_0, k_0 \in \{-1, 0, +1\}$ are averaged in an appropriate way, i.e.

$$a_{\tau_0,\tau_1,\tau_2,\tau_3} \quad = \quad \sum_{i_0,j_0,k_0 \in \{-1,0,+1\}} \omega_{i_0,j_0,k_0} f_{i+i_0,j+j_0,k+k_0}, \tag{3.10}$$

where ω_{i_0,j_0,k_0} are non-negative weights and are independent on the current considered cube $Q := Q_{i,j,k}$ and tetrahedron $T \subset Q$. Now, since the scheme is completely symmetric as well as the partition $\triangle$, it is sufficient to consider only one tetrahedron $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$ and show how to set the Bernstein-Bézier coefficients $a_{\tau_0,\tau_1,\tau_2,\tau_3}$ of $s|_T$ for that particular tetrahedron T on the $n+1$ different layers

$$\mathcal{L}^n_{\tau_0} \quad = \quad \{\vartheta^n_{\tau_0,\tau_1,\tau_2,\tau_3} \in T : \tau_1 + \tau_2 + \tau_3 = n - \tau_0\},\ \tau_0 = 0, \ldots, n, \tag{3.11}$$

of domain points in T. Finally, all other Bernstein-Bézier coefficients of s_f associated with domain points on the different layers in the remaining tetrahedra in Q immediately follow from symmetry and thus, for all tetrahedra in $\triangle$.
The trivariate version of Alg. 1.2.1 to evaluate the polynomial piece $p = s|_T \in \mathcal{P}_n$ in the above form can be defined as follows.

Algorithm 3.2.1 (Trivariate de Casteljau on Type-6 Partitions)**.** *For $j = 1, \ldots, n$ and $\tau_0 + \tau_1 + \tau_2 + \tau_3 = n - j$ compute*

$$\mathbf{b}^j_{\tau_0+\tau_1+\tau_2+\tau_3} \quad = \quad \lambda_0(\mathbf{q})\mathbf{b}^{j-1}_{\tau_0+1,\tau_1,\tau_2,\tau_3} + \lambda_1(\mathbf{q})\mathbf{b}^{j-1}_{\tau_0,\tau_1+1,\tau_2,\tau_3} + \tag{3.12}$$

$$\lambda_2(\mathbf{q})\mathbf{b}^{j-1}_{\tau_0,\tau_1,\tau_2+1,\tau_3} + \lambda_3(\mathbf{q})\mathbf{b}^{j-1}_{\tau_0,\tau_1,\tau_2,\tau_3+1} \tag{3.13}$$

and we obtain the result as $s|_T = \mathbf{b}^n_{0,0,0,0}$, where the $\mathbf{b}^0_{\tau_0,\tau_1,\tau_2,\tau_3} = \mathbf{b}_{\tau_0,\tau_1,\tau_2,\tau_3}$, $\tau_0+\tau_1+\tau_2+\tau_3 = n$ are the Bézier points from Equ.(3.7) defining the convex hull of the polynomial piece.

This algorithm degenerates to its bivariate and univariate versions if one or two of the barycentric coordinates of $\mathbf{q}$ vanish, respectively. In these cases, $\mathbf{q}$ lies in the interior of a triangular face of T or on an edge of T, and the number of necessary arithmetic operations reduces accordingly.
Once we have determined the cube Q_0 which contains a point $\mathbf{q} \in \Omega$ and its local coordinates $\lambda_\nu(\mathbf{q}),\ \nu = 1, 2, 3$ according to Q_0 are computed – this will be discussed

in the following sections. We are able to evaluate the polynomial piece (2.5) and its derivatives at $\mathbf{q}$ using λ_ν, the Bézier points $\mathbf{b}_{i,j,k}$ [2] and Alg. 2.2.1. For shading of surfaces obtained from the volume model at a point $\mathbf{q}$ it is necessary to compute the 1'st partial derivatives (the gradient)

$$(\nabla s|_Q)(\mathbf{q}) \quad = \quad \left(\frac{\partial}{\partial x}s|_Q(\mathbf{q}), \frac{\partial}{\partial y}s|_Q(\mathbf{q}), \frac{\partial}{\partial z}s|_Q(\mathbf{q})\right)$$

at the same location $\mathbf{q}$.

3.3 Point Location and Barycentric Coordinates

Before we are able to evaluate the polynomial piece (3.7), i.e. compute the value and the derivatives of the spline $s|_T$ at a given point $\mathbf{q} \in \Omega$, we need to know the location of $\mathbf{q} = (q_x, q_y, q_z)$ in the partition $\triangle$ and its barycentric coordinates according to a tetrahedron T. Thus, first we have to find a cube $Q \in \diamondsuit$ with $\mathbf{q} \in Q$ (cf. Sec. 2.3), a tetrahedron $T \in \triangle$ with $\mathbf{q} \in T$, where $T \subset Q$, and the barycentric coordinates $\lambda_\nu(\mathbf{q}) \in \mathbb{R},\ \nu = 0, 1, 2, 3$ according to T. As we know, the uniformity of $\triangle$ allows a translation of $\mathbf{q}$ such that the remaining computations can be performed for (the tetrahedral partition of) the unit cube $Q_0 = [-0.5, +0.5]$ (see Fig. 3.1).
However, to find the tetrahedron T the observation from Sec. 3.1 is used, where each cube Q is split into 24 congruent tetrahedra by slicing it with six planes (3.1). Thus, the orientation of $\mathbf{q}$ with respect to these planes is found from $P_\nu(q_x, q_y, q_z, d),\ \nu = 0, \ldots, 5$ followed by a sign check for each of the six planes. This gives a six-bit binary code for the orientation of $\mathbf{q}$ and finally the tetrahedron $T \subset Q$ with $\mathbf{q} \in T$ is found by a simple table lookup.
The barycentric coordinates $\lambda_\nu(\mathbf{q}) \in \mathbb{R},\ \nu = 0, 1, 2, 3$ according to $T \subset Q$ are computed from (3.9). First, the vertices of a tetrahedron $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$ have to be organized. For example, if the first vertex $\mathbf{v}_0$ is identified with the origin of the unit cube Q_0, the second vertex $\mathbf{v}_1$ with the intersection point of the two diagonals drawn into a face F of Q, the third vertex $\mathbf{v}_2$ and fourth vertex $\mathbf{v}_3$ are identified with the two corners of that face F. Then, this allows generating another lookup table with 24 entries of the pre-computed solutions. Thus, the barycentric coordinates of $\mathbf{q}$ with respect to T are computed by

$$\begin{aligned}
\lambda_0(\mathbf{q}) &= 1 + L_2 + L_2, \\
\lambda_1(\mathbf{q}) &= 1 + L_3 + L_4, \\
\lambda_2(\mathbf{q}) &= 1 + L_0 + L_4, \\
\lambda_3(\mathbf{q}) &= 1 - \lambda_0(\mathbf{q}) - \lambda_1(\mathbf{q}) - \lambda_2(\mathbf{q}),
\end{aligned} \tag{3.14}$$

where $L_{\nu=0}^5 := [+q_x, +q_y, +q_z, -q_x, -q_y, -q_z]$ is an ordered list. Both tables can be determined by analyzing all of the 24 possible solutions of (3.9) and, therefore, the barycentric coordinates of $\mathbf{q}$ are found without performing expensive rotation or transformation operations, branching over the 24 cases and any multiplication. Note that the identification of the vertices $\mathbf{v}_i,\ i = 0, 1, 2, 3$ of the tetrahedron T with the appropriate vertices of Q defines the lookup table with 24 entries of the pre-computed solutions, and thus has to be performed carefully.

[2] Once the cube Q_0 is found, all its Bézier points $\mathbf{b}_{i,j,k}$ are known as well.

3.4 Piecewise Linear Splines

Piecewise linear splines defined on a tetrahedral partition $\triangle$ are called in this thesis *linear type-6 splines*. These polynomials are also continuous but not smooth over the volume domain Ω (similar to the *linear type-0 splines*). The space of these splines with respect to $\triangle$ is defined by

$$\mathcal{S}_1(\triangle) \quad = \quad \{s \in C^0(\Omega) : s|_T \in \mathcal{P}_1, \forall T \in \triangle\}, \tag{3.15}$$

where $\mathcal{P}_1 := \text{span}\{x^i y^j z^k : i, j, k \geq 0, i + j + k \leq 1\}$ denotes the four-dimensional space of linear polynomials, i.e. the space of trivariate polynomials of total degree one.
A spline $s \in \mathcal{S}_1$ can be written in its piecewise Bernstein-Bézier form (3.7), i.e. in that equation we set $n = 1$.

3.4.1 Bernstein-Bézier Coefficients

For piecewise linear splines the 4(15) Bernstein-Bézier coefficients for an arbitrary tetrahedron $T \in \triangle(Q \in \diamondsuit)$ are determined as follows. First, according to Equ. 3.11 two different layers $\mathcal{L}^1_\nu$, $\nu = 0, 1$ have to be considered. The coefficients of $s|_T$ associated with its corresponding Bézier points (shown as left blue, right blue, and green circle in Fig. 3.3, respectively) $\mathbf{b}_{0,0,1,0} = (\vartheta^1_{0,0,1,0}, a_{0,0,1,0}), \mathbf{b}_{0,0,0,1} = (\vartheta^1_{0,0,0,1}, a_{0,0,0,1})$, and $\mathbf{b}_{0,1,0,0} = (\vartheta^1_{0,2,0,0}, a_{0,1,0,0})$ at the domain points $\vartheta^1_{0,0,1,0} = \mathbf{v}_2, \vartheta^1_{0,0,0,1} = \mathbf{v}_3$, and $\vartheta^1_{0,1,0,0} = \mathbf{v}_1$ in layer $\mathcal{L}^1_0$ are determined by

$$\begin{aligned} a_{0,0,1,0} &= \frac{1}{8}(C + F + L + T + LT + FL + FT + FLT), \\ a_{0,0,0,1} &= \frac{1}{8}(C + F + R + T + RT + FR + FT + FRT), \\ a_{0,1,0,0} &= \frac{1}{2}(C + F). \end{aligned} \tag{3.16}$$

Then, the last Bernstein-Bézier coefficient $a_{1,0,0,0}$ associated with the Bézier point $\mathbf{b}_{1,0,0,0} = (\vartheta^1_{1,0,0,0}, a_{1,0,0,0})$ located at the center of the cube $Q_{i,j,k}$ in layer $\mathcal{L}^1_1$, i.e. at the domain point $\vartheta^1_{1,0,0,0} = \mathbf{v}_0$ (cyan circle in right drawing of Fig. 3.3) is simply set to

$$a_{1,0,0,0} \quad = \quad C, \tag{3.17}$$

where C is the data value itself located in the center of a cube $Q_{i,j,k}$.

3.4.2 Evaluation of Polynomial Pieces and its Derivatives

Once the cube $Q_0 := Q_{i,j,k} \in \diamondsuit$ and the tetrahedron $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] \subset Q_0$ with the point $\mathbf{q} \in T$ are determined and the local coordinates $\lambda_\nu(\mathbf{q})$, $\nu = 0, 1, 2, 3$ of $\mathbf{q}$ according to T are computed. The polynomial piece (3.7) and its derivatives at $\mathbf{q}$ can be determined. Note that when the cube and tetrahedron T are found, all Bézier points according to T are known as well and the polynomial piece (3.7) can be evaluated applying Alg. 3.2.1 (see Fig. 3.4).

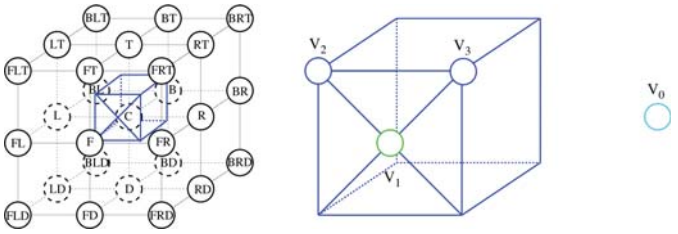


Figure 3.3: *The data values (white circles in the left drawing) in the local* 27 *neighborhood of a considered cube* Q_0 *(blue cube) and appropriate averaging rules – for linear splines – give the* 4 *Bernstein-Bézier coefficients located on its Bézier points (colored circles in the right illustration) on two different layers* $\mathcal{L}^1_\nu,\ \nu = 0, 1$ *of a tetrahedron* T *(right figure). The other coefficients for the remaining* 23 *tetrahedra of* Q_0 *are computed by symmetry and are not shown here.*

3.5 Piecewise Quadratic Splines

In this section we describe piecewise continuous splines which satisfy additional smoothness conditions in comparison to the previously defined linear splines – such splines are called *quadratic type-6 splines*, also better known as *Super-Splines* (see [RZNS03] [RZNS04a]). The accurate smoothness conditions can be found in [NRSZ04]. The space of these splines with respect to $\triangle$ is defined by

$$\mathcal{S}_2(\triangle) \quad = \quad \{s \in C^1(\Omega) : s|_T \in \mathcal{P}_2, \forall T \in \triangle, s \text{ smooth at } \mathbf{v}, \forall \mathbf{v} \text{ of } \diamondsuit\}, \qquad (3.18)$$

where $\mathcal{P}_2 := \text{span}\{x^i y^j z^k : i, j, k \geq 0, i + j + k \leq 2\}$ denotes the 10-dimensional space of quadratic polynomials, i.e. the space of trivariate polynomials of total degree two. Note, this splines are not everywhere on Ω continuously differentiable.
Once again a spline $s \in \mathcal{S}_2$ can be written in its piecewise Bernstein-Bézier form (3.7) with $n = 2$.

3.5.1 Bernstein-Bézier Coefficients

For piecewise quadratic splines the 10(65) Bernstein-Bézier coefficients for an arbitrary tetrahedron $T \in \triangle(Q \in \diamondsuit)$ are determined as follows. First, according to Equ. 3.11 three different layers $\mathcal{L}^2_\nu,\ \nu = 0, 1, 2$ have to be considered now. The six coefficients of $s|_T$ associated with its corresponding points in layer $\mathcal{L}^2_0$ of T (also located on Q) are determined by

$$\begin{aligned}
a_{0,0,2,0} &= \frac{1}{8}(C + F + L + T + LT + FL + FT + FLT),\\
a_{0,0,0,2} &= \frac{1}{8}(C + F + R + T + RT + FR + FT + FRT),\\
a_{0,0,1,1} &= \frac{1}{4}(C + F + T + FT),\\
a_{0,1,1,0} &= \frac{1}{4}(C + F) + \frac{1}{8}(L + T + FL + FT),\\
a_{0,1,0,1} &= \frac{1}{4}(C + F) + \frac{1}{8}(R + T + FR + FT),\\
a_{0,2,0,0} &= \frac{1}{4}(C + F) + \frac{1}{16}(D + L + R + T + FD + FL + FR + FT), \qquad (3.19)
\end{aligned}$$

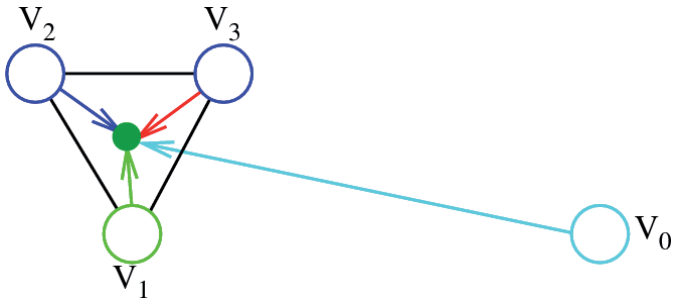


Figure 3.4: *The evaluation of a polynomial piece $s|_T \in \mathcal{P}_1$ at a point $\mathbf{q}$ (green dot) with the de Casteljau algorithm (3.2.1) using the Bézier points (colored circles) on the two different layers $\mathcal{L}^1_\nu$, $\nu = 0, 1$ of the tetrahedron T (as in right drawing of Fig. 3.3). These four colored dots show the Bézier points $\mathbf{b}^0_{0,0,1,0} := \mathbf{b}_{0,0,1,0} = (\vartheta^1_{0,0,1,0}, a_{0,0,1,0}), \mathbf{b}^0_{0,0,0,1} := \mathbf{b}_{0,0,0,1} = (\vartheta^1_{0,0,0,1}, a_{0,0,0,1}), \mathbf{b}^0_{0,1,0,0} := \mathbf{b}_{0,1,0,0} = (\vartheta^1_{0,2,0,0}, a_{0,1,0,0})$, and $\mathbf{b}^0_{1,0,0,0} := \mathbf{b}_{1,0,0,0} = (\vartheta^1_{1,0,0,0}, a_{1,0,0,0})$ at the domain points $\vartheta^1_{0,0,1,0} = \mathbf{v}_2, \vartheta^1_{0,0,0,1} = \mathbf{v}_3, \vartheta^1_{0,1,0,0} = \mathbf{v}_1$, and $\vartheta^1_{1,0,0,0} = \mathbf{v}_0$ and the associated coefficients $a_{\tau_0,\tau_1,\tau_2,\tau_3}$, $\tau_0+\tau_1+\tau_2+\tau_3 = 1$ on level $\ell = 0$. The new Bézier point (green dot) $s|_T(\mathbf{q}) = \mathbf{b}^1_{0,0,0,0}$ and its associated coefficient on level $\ell = 1$ is determined from an affine combination by weighting with the barycentric coordinates of $\mathbf{q}$ w.r.t. $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$ and is the final result. The arrows show which points are involved, where the different colors represent the barycentric coordinates $\lambda_\nu(\mathbf{q})$, $\nu = 0, 1, 2, 3$. Note, the coefficients $a^0_{\tau_0,\tau_1,\tau_2,\tau_3}$ of the corresponding points (colored circles) already define the partial derivatives $\frac{\partial}{\partial \xi_\nu} s|_T \in \mathcal{P}_0$ along the edges of the tetrahedron T.*

where the Bernstein-Bézier coefficients $a_{0,0,2,0}$ and $a_{0,0,0,2}$ located on the domain points $\vartheta^2_{0,0,2,0} = \mathbf{v}_2$ and $\vartheta^2_{0,0,0,2} = \mathbf{v}_3$ define the Bézier points $\mathbf{b}_{0,0,2,0} = (\vartheta^2_{0,0,2,0}, a_{0,0,2,0})$ and $\mathbf{b}_{0,0,0,2} = (\vartheta^2_{0,0,0,2}, a_{0,0,0,2})$ (show as left blue and right blue circles in Fig. 3.5), respectively. Accordingly the coefficients $a_{0,0,1,1}, a_{0,1,1,0}, a_{0,1,0,1}$ and their domain points give us the Bézier points $\mathbf{b}_{0,0,1,1} = (\vartheta^2_{0,0,1,1}, a_{0,0,1,1}), \mathbf{b}_{0,1,1,0} = (\vartheta^2_{0,1,1,0}, a_{0,1,1,0}), \mathbf{b}_{0,1,0,1} = (\vartheta^2_{0,1,0,1}, a_{0,1,0,1})$ (shown as red, left yellow, right yellow circles in Fig. 3.5), respectively. The last coefficient $a_{0,2,0,0}$ in this layer and its associated domain point $\vartheta^2_{0,2,0,0} = \mathbf{v}_1$ results in the Bézier point $\mathbf{b}_{0,2,0,0} = (\vartheta^2_{0,2,0,0}, a_{0,2,0,0})$ and is depicted by a green circle in Fig. 3.5. Then, the next three coefficients of $s|_T$ associated with its corresponding points $\vartheta^2_{1,0,1,0}, \vartheta^2_{1,0,0,1}, \vartheta^2_{1,1,0,0}$ in layer $\mathcal{L}^2_1$ of T (located inside Q) are determined by

$$
\begin{aligned}
a_{1,0,1,0} &= \frac{5}{16}(C) + \frac{3}{16}(F+T+L) + \frac{1}{16}(FL+FT+LT-FLT),\\
a_{1,0,0,1} &= \frac{5}{16}(C) + \frac{3}{16}(F+T+R) + \frac{1}{16}(FR+FT+RT-FRT),\\
a_{1,1,0,0} &= \frac{20}{64}(C) + \frac{12}{64}(F) + \frac{6}{64}(D+L+R+T) + \frac{2}{64}(FD+FL+FR+FT) +\\
&\quad \frac{1}{64}(LD+LT+RD+RT-FLD-FLT-FRD-FRT),
\end{aligned}
\tag{3.20}
$$

where the Bézier points $\mathbf{b}_{1,0,1,0} = (\vartheta^2_{1,0,1,0}, a_{1,0,1,0}), \mathbf{b}_{1,0,0,1} = (\vartheta^2_{1,0,0,1}, a_{1,0,0,1})$, and $\mathbf{b}_{1,1,0,0} = (\vartheta^2_{1,1,0,0}, a_{1,1,0,0})$ are shown as left brown, right brown, and magenta circles in Fig. 3.5. Finally, the Bernstein-Bézier coefficient $a_{2,0,0,0}$ associated with the Bézier point $\mathbf{b}_{2,0,0,0} = (\vartheta^2_{2,0,0,0}, a_{2,0,0,0})$ (cyan circle in right drawing of Fig. 3.3) located at the center $\vartheta^2_{2,0,0,0} =$

$\mathbf{v}_0$ of the cube Q in layer $\mathcal{L}_2^2$ of T is computed by

$$\begin{aligned} a_{2,0,0,0} &= \frac{40}{128}(C) + \frac{12}{128}(B+D+F+R+T+L) + \\ &\quad \frac{2}{128}(BD+BL+BR+BT+FD+FL+FR+FT+LD+LT+RD+RT) - \\ &\quad \frac{1}{128}(BLD+BLT+BRD+BRT+FLD+FLT+FRD+FRT). \end{aligned} \tag{3.21}$$

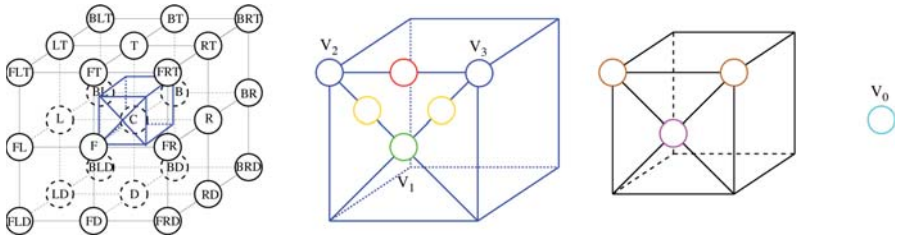


Figure 3.5: *The data values (white circles in the left drawing) in the local* 27 *neighborhood of a considered cube* Q_0 *(blue cube) and appropriate averaging rules – for quadratic splines – give the* 10 *Bernstein-Bézier coefficients located on its Bézier points (colored circles in the right illustration) on three different layers* $\mathcal{L}_\nu^2,\ \nu = 0,1,2$ *of a tetrahedron* T *(right figure). The other coefficients for the remaining* 23 *tetrahedra of* Q_0 *are computed by symmetry and are not shown here.*

3.5.2 Evaluation of Polynomial Pieces and its Derivatives

Once the cube $Q_0 := Q_{i,j,k} \in \diamondsuit$ and the tetrahedron $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] \subset Q_0$ with the point $\mathbf{q} \in T$ have been determined and the local coordinates $\lambda_\nu(\mathbf{q}),\ \nu = 0,1,2,3$ of $\mathbf{q}$ according to T are computed. The polynomial piece (3.7) and its derivatives at $\mathbf{q}$ can be determined. Once the cube and tetrahedron T are found, all Bézier points according to T are known as well and the polynomial piece (3.7) can be evaluated applying Alg. 3.2.1 (see Fig. 3.6).

3.6 Piecewise Cubic Splines

Piecewise cubic splines (i.e. *cubic type-6 splines*) defined on a tetrahedral partition $\triangle$ are a further development of the previously discussed *Super-Splines*. For more information on the smoothness conditions see [SZ05]. The space of these splines with respect to $\triangle$ is defined by

$$\mathcal{S}_3(\triangle) = \{s \in C^1(\Omega) : s|_T \in \mathcal{P}_3, \forall T \in \triangle\}, \tag{3.22}$$

where $\mathcal{P}_3 := \text{span}\{x^i y^j z^k : i,j,k \geq 0, i+j+k \leq 3\}$ denotes the 19-dimensional space of cubic polynomials, i.e. the space of trivariate polynomials of total degree three. Note, this splines are everywhere on Ω continuously differentiable.
Again a spline $s \in \mathcal{S}_3$ can be written in its piecewise Bernstein-Bézier form (3.7) with now $n = 3$.

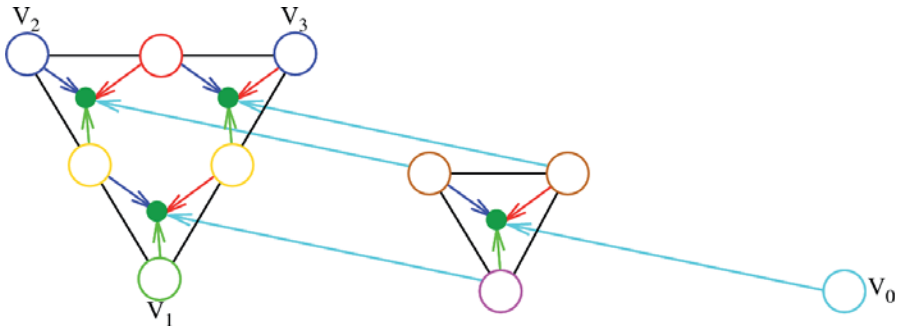


Figure 3.6: *The evaluation of a polynomial piece $s|_T \in \mathcal{P}_2$ at a point $\mathbf{q}$ is now performed by the repeated application of de Casteljau algorithm (3.2.1). First, on level $\ell = 0$ the ten Bézier points (circles in the same color as in Fig. 3.5) $\mathbf{b}^0_{\tau_0,\tau_1,\tau_2,\tau_3} := \mathbf{b}_{\tau_0,\tau_1,\tau_2,\tau_3} = (\vartheta^2_{\tau_0,\tau_1,\tau_2,\tau_3}, a_{\tau_0,\tau_1,\tau_2,\tau_3})$, $\tau_0+\tau_1+\tau_2+\tau_3 = 2$ on the three different layers $\mathcal{L}^2_\nu$, $\nu = 0,1,2$ of the tetrahedron T are used to obtain the intermediate result (green dots) $\mathbf{b}^1_{\tau_0,\tau_1,\tau_2,\tau_3}$, $\tau_0+\tau_1+\tau_2+\tau_3 = 1$ on level $\ell = 1$ from the affine combinations by weighting with the barycentric coordinates of $\mathbf{q}$ w.r.t. $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$. Here the four Bézier points on level $\ell = 0$ associate with the vertices of the tetrahedron T (i.e. the domain points become once more $\vartheta^2_{0,0,2,0} = \mathbf{v}_2, \vartheta^2_{0,0,0,2} = \mathbf{v}_3, \vartheta^2_{0,2,0,0} = \mathbf{v}_1$, and $\vartheta^2_{2,0,0,0} = \mathbf{v}_0$) are $\mathbf{b}^0_{0,0,2,0} := \mathbf{b}_{0,0,2,0} = (\vartheta^2_{0,0,2,0}, a_{0,0,2,0})$, $\mathbf{b}^0_{0,0,0,2} := \mathbf{b}_{0,0,0,2} = (\vartheta^2_{0,0,0,2}, a_{0,0,0,2})$, $\mathbf{b}^0_{0,2,0,0} := \mathbf{b}_{0,2,0,0} = (\vartheta^2_{0,2,0,0}, a_{0,2,0,0})$, and $\mathbf{b}^0_{2,0,0,0} := \mathbf{b}_{2,0,0,0} = (\vartheta^2_{2,0,0,0}, a_{2,0,0,0})$. Again the arrows show which points are involved, where the different colors represent the barycentric coordinates $\lambda_\nu(\mathbf{q})$, $\nu = 0,1,2,3$. Note, this time the coefficients $a^1_{\tau_0,\tau_1,\tau_2,\tau_3}$ of the corresponding points (green dots) define the partial derivatives $\frac{\partial}{\partial \xi_\nu} s|_T \in \mathcal{P}_1$ along the edges of the tetrahedron T. The final result $s|_T(\mathbf{q}) = \mathbf{b}^2_{0,0,0,0}$ is obtained from the repeated application of de Casteljau algorithm by using the intermediate result (green dots) $\mathbf{b}^1_{\tau_0,\tau_1,\tau_2,\tau_3}$ as input into the scheme described in Fig. 3.4.*

3.6.1 Bernstein-Bézier Coefficients

The general idea and the scheme on how to determine the Bernstein-Bézier coefficients for an arbitrary tetrahedron $T \in \triangle$ to obtain piecewise cubic splines on the domain Δ is very similar to that used for linear and quadratic splines. Here 20(175) Bernstein-Bézier coefficients for an arbitrary tetrahedron $T \in \triangle(Q \in \diamondsuit)$ have to be determined, where according to Equ. 3.11 four different layers $\mathcal{L}^3_\nu$, $\nu = 0,1,2,3$ have to be considered now. The first ten coefficients of $s|_T$ associated with its corresponding points in layer $\mathcal{L}^3_0$ of T (also located on Q, depicted as a blue cube in Fig. 3.7) are determined by

$$\begin{aligned}
a_{0,0,3,0} &= \frac{1}{8}(C+F+L+T+LT+FL+FT+FLT),\\
a_{0,0,0,3} &= \frac{1}{8}(C+F+R+T+RT+FR+FT+FRT),\\
a_{0,0,2,1} &= \frac{5}{24}(C+F+T+FT)+\frac{1}{24}(L+FL+LT+FLT),\\
a_{0,0,1,2} &= \frac{5}{24}(C+F+T+FT)+\frac{1}{24}(R+FR+RT+FRT),\\
a_{0,1,2,0} &= \frac{5}{24}(C+F)+\frac{1}{8}(L+T+FL+FT)+\frac{1}{24}(LT+FLT),\\
a_{0,1,0,2} &= \frac{5}{24}(C+F)+\frac{1}{8}(R+T+FR+FT)+\frac{1}{24}(RT+FRT),
\end{aligned} \tag{3.23}$$

and

$$\begin{aligned}
a_{0,1,1,1} &= \frac{13}{48}(C+F)+\frac{1}{32}(L+R+FL+FR)+\frac{7}{48}(T+FT)+\\
&\quad \frac{1}{96}(LT+RT+FLT+FRT),\\
a_{0,2,1,0} &= \frac{13}{48}(C+F)+\frac{17}{192}(L+T+FL+FT)+\frac{1}{64}(R+D+FR+FD)+\\
&\quad \frac{1}{96}(LT+FLT)+\frac{1}{192}(RT+LD+FRT+FLD),\\
a_{0,2,0,1} &= \frac{13}{48}(C+F)+\frac{17}{192}(R+T+FR+FT)+\frac{1}{64}(L+D+FL+FD)+\\
&\quad \frac{1}{96}(RT+FRT)+\frac{1}{192}(RD+LT+FLT+FRD),\\
a_{0,3,0,0} &= \frac{13}{48}(C+F)+\frac{5}{96}(L+R+T+D+FL+FR+FT+FD)+\\
&\quad \frac{1}{192}(RT+RD+LT+LD+FRT+FRD+FLT+FLD). \qquad (3.24)
\end{aligned}$$

The Bernstein-Bézier coefficients $a_{0,0,3,0}$ and $a_{0,0,0,3}$ located on the domain points $\vartheta^3_{0,0,3,0} = \mathbf{v}_2$ and $\vartheta^3_{0,0,0,3} = \mathbf{v}_3$ define the Bézier points $\mathbf{b}_{0,0,3,0} = (\vartheta^3_{0,0,3,0}, a_{0,0,3,0})$ and $\mathbf{b}_{0,0,0,3} = (\vartheta^3_{0,0,0,3}, a_{0,0,0,3})$, respectively, and are shown as left blue and right blue dots in Fig. 3.7. Accordingly the other Bézier points $\mathbf{b}_{0,0,2,1}, \mathbf{b}_{0,0,1,2}, \mathbf{b}_{0,1,2,0}, \mathbf{b}_{0,1,0,2}, \mathbf{b}_{0,1,1,1}, \mathbf{b}_{0,2,1,0}$, and $\mathbf{b}_{0,2,0,1}$ and their coefficients are shown as left red, right red, left yellow, right yellow, center yellow, left pink, and right pink dots in Fig. 3.7, respectively, where the last point $\mathbf{b}_{0,3,0,0} = (\vartheta^3_{0,3,0,0}, a_{0,3,0,0})$ with $\vartheta^3_{0,3,0,0} = \mathbf{v}_1$ is drawn as a green dot. Next, the six Bernstein-Bézier coefficients of $s|_T$ associated with points in $\mathcal{L}^3_1$ of T (located inside Q) are defined by

$$\begin{aligned}
a_{1,0,2,0} &= \frac{1}{4}C+\frac{1}{6}(F+L+T)+\frac{1}{12}(LT+FL+FT),\\
a_{1,0,0,2} &= \frac{1}{4}C+\frac{1}{6}(F+R+T)+\frac{1}{12}(RT+FR+FT),\\
a_{1,0,1,1} &= \frac{1}{3}C+\frac{5}{24}(F+T)+\frac{1}{12}FT+\frac{1}{24}(L+R)+\frac{1}{48}(LT+RT+FL+FR),\\
a_{1,1,1,0} &= \frac{1}{3}C+\frac{5}{24}F+\frac{1}{48}(D+R+LT)+\frac{1}{8}(L+T)+\frac{5}{96}(FL+FT)+\\
&\quad \frac{1}{96}(FD+LD+RT+FR),\\
a_{1,1,0,1} &= \frac{1}{3}C+\frac{5}{24}F+\frac{1}{48}(D+L+RT)+\frac{1}{8}(R+T)+\frac{5}{96}(FR+FT)+\\
&\quad \frac{1}{96}(FD+LT+RD+FL),\\
a_{1,2,0,0} &= \frac{1}{3}C+\frac{5}{24}F+\frac{7}{96}(L+R+T+D)+\frac{1}{32}(FL+FR+FT+FD)+\\
&\quad \frac{1}{96}(RT+RD+LT+LD). \qquad (3.25)
\end{aligned}$$

The corresponding Bézier points $\mathbf{b}_{1,0,2,0}, \mathbf{b}_{1,0,0,2}, \mathbf{b}_{1,0,1,1}, \mathbf{b}_{1,1,1,0}, \mathbf{b}_{1,1,0,1}$, and $\mathbf{b}_{1,2,0,0}$ are the left blue, right blue, red, left yellow, right yellow, and green circles, respectively.

Further, the Bernstein-Bézier coefficients of $s|_T$ associated with points in $\mathcal{L}_2^3$ of T (located inside Q as well) are set to

$$\begin{aligned}
a_{2,0,1,0} &= \frac{3}{8}C + \frac{7}{48}(F+T+L) + \frac{1}{48}(R+D+B+LT+FL+FT) + \\
&\quad \frac{1}{96}(RT+BT+FR+FD+LD+BL), \\
a_{2,0,0,1} &= \frac{3}{8}C + \frac{7}{48}(F+T+R) + \frac{1}{48}(L+D+B+RT+FR+FT) + \\
&\quad \frac{1}{96}(LT+BT+FL+FD+RD+BR), \\
a_{2,1,0,0} &= \frac{3}{8}C + \frac{1}{12}(T+R+L+D) + \frac{1}{64}(FT+FR+FL+FD) + \frac{7}{48}F + \frac{1}{48}B + \\
&\quad \frac{1}{96}(RT+LD+LT+RD) + \frac{1}{192}(BT+BR+BL+BD),
\end{aligned} \tag{3.26}$$

where the corresponding Bézier points $\mathbf{b}_{2,0,1,0}, \mathbf{b}_{2,0,0,1}$, and $\mathbf{b}_{2,1,0,0}$ are the left brown, right brown, and magenta circles, respectively. Finally, the Bernstein-Bézier coefficient of $s|_T$ associated with the point in $\mathcal{L}_3^3$ of T (i.e. the domain point $\vartheta_{3,0,0,0}^3 = \mathbf{v}_0$ is located in the center of Q) is set to

$$\begin{aligned}
a_{3,0,0,0} &= \frac{3}{8}C + \frac{1}{12}(T+F+L+R+D+B) + \frac{1}{96}(LT+FL+FT+RT+BT+BD) + \\
&\quad \frac{1}{96}(FR+FD+LD+RD+BR+BL)
\end{aligned} \tag{3.27}$$

and is shown as the right most cyan circle in Fig. 3.7.

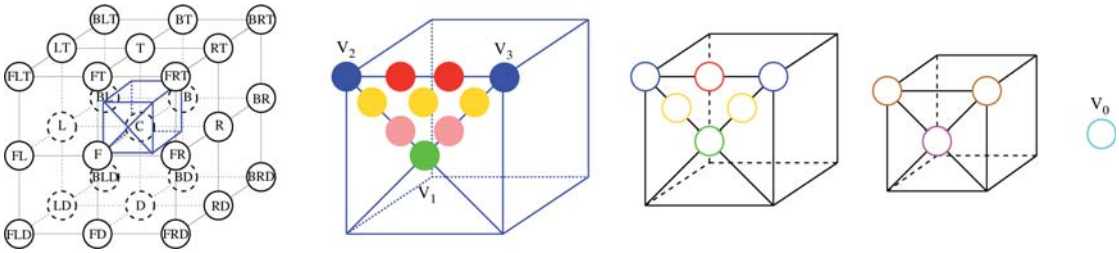


Figure 3.7: *The data values (white circles in the left most drawing) in the local* 27 *neighborhood of a considered cube* Q_0 *(blue cube) and appropriate averaging rules – for cubic splines – give the* 20 *Bernstein-Bézier coefficients located on its Bézier points (colored circles and dots in the right illustrations) on four different layers* $\mathcal{L}_\nu^3$, $\nu = 0,1,2,3$ *of a tetrahedron* T *(right figures). The other coefficients for the remaining* 23 *tetrahedra of* Q_0 *are computed by symmetry and are not shown here for simplicity.*

3.6.2 Evaluation of Polynomial Pieces and its Derivatives

Once the cube $Q_0 := Q_{i,j,k} \in \diamondsuit$ and the tetrahedron $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] \subset Q_0$ with the point $\mathbf{q} \in T$ have been determined and the local coordinates $\lambda_\nu(\mathbf{q})$, $\nu = 0,1,2,3$ of $\mathbf{q}$ according to T are computed. The polynomial piece (3.7) and its derivatives at $\mathbf{q}$ can be determined. Note that once again the cube and tetrahedron T are found, all Bézier

points according to T are known as well and the polynomial piece (3.7) can be evaluated applying Alg. 3.2.1 (see Fig. 3.8).

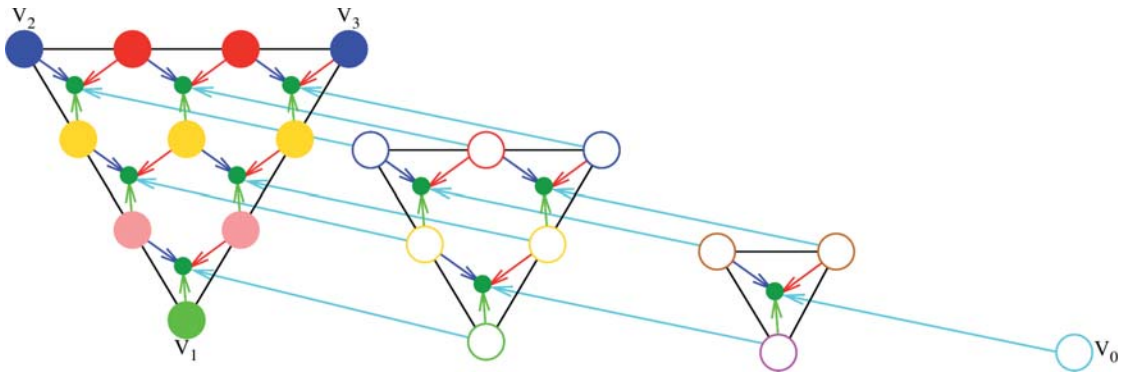


Figure 3.8: *The evaluation of a polynomial piece $s|_T \in \mathcal{P}_3$ at a point $\mathbf{q}$ is here performed by the repeated application of de Casteljau algorithm (3.2.1) as well. First, on level $\ell = 0$ the twenty Bézier points (colored dots and circles) $\mathbf{b}^0_{\tau_0,\tau_1,\tau_2,\tau_3} := \mathbf{b}_{\tau_0,\tau_1,\tau_2,\tau_3} = (\vartheta^3_{\tau_0,\tau_1,\tau_2,\tau_3}, a_{\tau_0,\tau_1,\tau_2,\tau_3})$, $\tau_0 + \tau_1 + \tau_2 + \tau_3 = 3$ on the four different layers $\mathcal{L}^3_\nu$, $\nu = 0,1,2,3$ of the tetrahedron T are used to obtain the intermediate result (green dots) $\mathbf{b}^1_{\tau_0,\tau_1,\tau_2,\tau_3}$, $\tau_0+\tau_1+\tau_2+\tau_3 = 2$ on level $\ell = 1$ from the affine combinations by weighting with the barycentric coordinates of $\mathbf{q}$ w.r.t. $T = [\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3]$. Here the four Bézier points on level $\ell = 0$ associated with the vertices of the tetrahedron T (i.e. the domain points become once more $\vartheta^3_{0,0,3,0} = \mathbf{v}_2, \vartheta^3_{0,0,0,3} = \mathbf{v}_3, \vartheta^3_{0,3,0,0} = \mathbf{v}_1$, and $\vartheta^3_{3,0,0,0} = \mathbf{v}_0$) are $\mathbf{b}^0_{0,0,3,0} := \mathbf{b}_{0,0,3,0} = (\vartheta^3_{0,0,3,0}, a_{0,0,3,0})$, $\mathbf{b}^0_{0,0,0,3} := \mathbf{b}_{0,0,0,3} = (\vartheta^3_{0,0,0,3}, a_{0,0,0,3})$, $\mathbf{b}^0_{0,3,0,0} := \mathbf{b}_{0,3,0,0} = (\vartheta^3_{0,3,0,0}, a_{0,3,0,0})$, and $\mathbf{b}^0_{3,0,0,0} := \mathbf{b}_{3,0,0,0} = (\vartheta^3_{3,0,0,0}, a_{3,0,0,0})$. Again the arrows show which points are involved, where the different colors represent the barycentric coordinates $\lambda_\nu(\mathbf{q})$, $\nu = 0,1,2,3$. Note, this time the coefficients $a^1_{\tau_0,\tau_1,\tau_2,\tau_3}$ of the corresponding points (green dots) define the second partial derivatives along and across the edges of the tetrahedron T. The final result $s|_T(\mathbf{q}) = \mathbf{b}^3_{0,0,0,0}$ is obtained from the repeated application of de Casteljau algorithm by using the intermediate result (green dots) $\mathbf{b}^1_{\tau_0,\tau_1,\tau_2,\tau_3}$ as input and follow the description in Fig. 3.6*

3.7 Simple Ray-Casting

3.7.1 Intersection Computations

Basically, the configuration and computation of intersection information is very similar to the method described in Sec. 2.6, which deals with a uniform cubic partition $\diamondsuit$ only. The difference here comes from the subdivision of this partition into a uniform tetrahedral partition $\triangle \subseteq \diamondsuit$ as discussed in Sec. 3.1. Thus, similarly an arbitrary ray casted through $\Delta \subseteq \Omega$ results first in two intersection points, which are called the enter point $\mathbf{q}_1 \in \Delta$ and the exit point $\mathbf{q}_2 \in \Delta$ of a ray $\mathbf{r}_{\nu,\mu}(t)$ coming from pixel ν, μ (cf. Equ. (2.12)). Once more, this is computed by intersecting the ray with the six planes limiting Ω (cf. Fig. 3.9) and if the ray does not intersect Ω no contribution from the volume is computed nor stored into the appropriate pixel ν, μ of that ray. Otherwise, here we must process all the tetrahedra $T \in \triangle \subseteq \diamondsuit$ which are intersected by the ray.
For this, the entrance point $\mathbf{q}_1$ is used to determine the first unit cube $\tilde{Q} := Q_1$ with $\mathbf{q}_1 \in \tilde{Q}$ (see Sec. 2.3), the first local ray start $\tilde{\mathbf{r}}_s$ and the first entrance plane $P^{\tilde{Q}}_j$ according

to $\tilde{Q}$ (cf. Sec. 2.6). The first entrance plane $P_j^{\tilde{T}} := P_j^{\tilde{Q}}$ is also one of the four planes confining the first tetrahedron $\tilde{T} \subset \tilde{Q}$. Now, using the global ray direction $\mathbf{r}_d^{\nu,\mu}$ (as in Equ. 2.12), we define once more a local ray (as in Equ. 2.13) according to $\tilde{Q}$, where $\tilde{\mathbf{r}}_s := (\tilde{r}_{s,x}, \tilde{r}_{s,y}, \tilde{r}_{s,z})$ is the local ray start and $\tilde{\mathbf{r}}_d := (\tilde{r}_{d,x}, \tilde{r}_{d,y}, \tilde{r}_{d,z})$ the local (global) ray direction. Now all tetrahedra $T \subset \tilde{Q}$ which are intersected by the local ray have to be processed. Thus, the first tetrahedron $\tilde{T} \subset \tilde{Q}$ has to be found by using the local ray start $\tilde{\mathbf{r}}_s$ (as it has been discussed in Sec. 3.3). This and the two lookup tables (cf. Sec. 3.1) complete the information needed for further intersection computations and we proceed as follows. According to the current tetrahedron $\tilde{T} \subset \tilde{Q}$, the first plane $P_j^{\tilde{T}} := P_j^{\tilde{Q}}$ of $\tilde{Q}$ (which confines the tetrahedron $\tilde{T}$ as well as the cube $\tilde{Q}$) intersected by the local ray and the first lookup table, we know which of the three remaining planes from the set $\{P_0^T, P_1^T, P_2^T, P_3^T, P_4^T, P_5^T\}$ confining the current tetrahedron has to be intersected to find the next intersection position $\tilde{\mathbf{r}}_e$ of the local ray and thus the next tetrahedron. For this, we apply the parameters a, b, c and d of the remaining three planes into Equ. (2.14). Hereby, using Equ. (2.13) and $\tilde{t} = \min_{i=0,1,2}(t_i)$, the nearest exit intersection $\tilde{\mathbf{r}}_e$ of the local ray and the current tetrahedron $\tilde{T} \subset \tilde{Q}$ as well as the exit plane $P_i^{\tilde{T}}$ of $\tilde{T}$ is found (see Fig.3.9). Then, until the local ray does not leave the current cube $\tilde{Q}$, i.e. the exit plane $P_i^{\tilde{T}} \notin \{P_6^Q, P_7^Q, P_8^Q, P_9^Q, P_{10}^Q, P_{11}^Q\}$ we determine the next tetrahedron $\hat{T} \subset \tilde{Q}$ by using the current tetrahedron $\tilde{T}$ as well as the current exit plane $P_i^{\tilde{T}}$ as indices into the second lookup table. Whereas the exit intersection $\tilde{\mathbf{r}}_e$ and the exit plane $P_i^{\tilde{T}}$ according to the current tetrahedron $\tilde{T}$ become the enter intersection $\hat{\mathbf{r}}_s$ and the enter plane $P_j^{\hat{T}}$ according to the next tetrahedron $\hat{T}$, respectively, then we proceed in a recursive manner. Once the current tetrahedron $\tilde{T}$ is determined – thus also the corresponding Bernstein-Bézier coefficients – as well as the local ray enter $\tilde{\mathbf{r}}_s$ and exit $\tilde{\mathbf{r}}_e$ positions, and the ray direction $\tilde{\mathbf{r}}_d$, the evaluation of the polynomial pieces and its derivatives along the ray can be applied.

Once more, all the contributions of the different tetrahedra intersected by the global ray $\mathbf{r}_{\nu,\mu}(t)$ should be determined in the local way described above. Here, the idea of *Bresenham's* line drawing algorithm is applied as well (cf. Fig. 3.9). Thus, once the first unit cube $\tilde{Q} := Q_1$ together with the other information is found, we apply the local process as discussed in the previous paragraph, until the exit plane of the current considered tetrahedron $\tilde{T}$ becomes $P_i^{\tilde{T}} \in \{P_6^Q, P_7^Q, P_8^Q, P_9^Q, P_{10}^Q, P_{11}^Q\}$. This exit plane confines the current tetrahedra $\tilde{T}$ as well as the current cube $\tilde{Q}$ and thus can be used together with another lookup table to find the next cube $\hat{Q}$ along the global ray. Again, the new local ray start in the next unit cube $\hat{Q} := Q_2$ can be easily found from last exit position $\tilde{\mathbf{r}}_e$ using modulo computation (see Sec. 2.6). Whereas the first tetrahedron in the next cube $\hat{Q}$ is found by using the last considered tetrahedron $\tilde{T} \subset \tilde{Q}$, the exit plane $P_i^{\tilde{T}}$ and the second lookup table.

3.7.2 Univariate Polynomial Pieces

Here, local polynomial pieces according to T have to be evaluated for real volume or iso-surface rendering. Thus, if $\tilde{\mathbf{r}}_s = \tilde{\mathbf{r}}(t_s)$ and $\tilde{\mathbf{r}}_e = \tilde{\mathbf{r}}(t_e)$, where $t_s := t_0 := 0 < t_e := t_1 := \tilde{t}$, are two intersection points of $\tilde{\mathbf{r}}(t)$ and T. Then, the restriction of the polynomial piece p to the line segment $[\tilde{\mathbf{r}}_s, \tilde{\mathbf{r}}_e]$ is a univariate polynomial of degree 1, 2 or 3 (cf. right images

in Fig. 2.6). The necessary equations of 1st ($N = 0$), 2nd ($N = 1$) or 3rd ($N = 2$) degree are set up by computing the values $\tilde{w}_s$, $\tilde{w}_i$, and $\tilde{w}_e$ at positions $\tilde{\mathbf{r}}_s$, $\tilde{\mathbf{r}}_i = \tilde{\mathbf{r}}(i/(N+1))$, and $\tilde{\mathbf{r}}_e$, where $i = 1, \ldots, N$ and $N \in \{0, 1, 2\}$ (cf. Sec. 2).

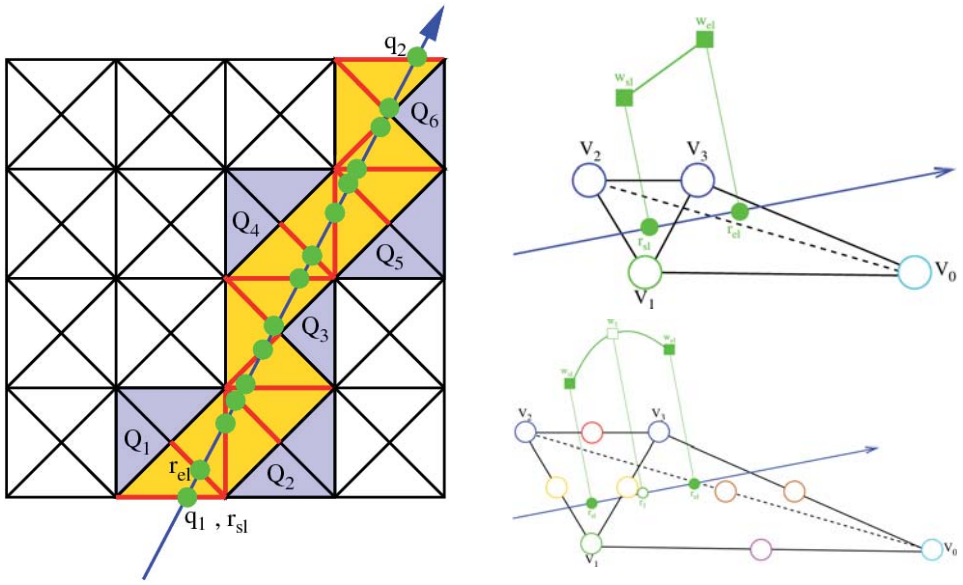


Figure 3.9: *Left: Ray-Casting a uniform tetrahedral partition (simplified two-dimensional view). The intersection positions (green dots) of the ray (blue ray) and the tetrahedra (blue triangles) define the intersection planes (red lines), which are used to find the next local start position of the ray segment according to the next tetrahedron or local unit cube. Right: On each tetrahedron univariate polynomial pieces of total degree 1 (top) and 2 (bottom) are defined. The green solid dots* $\mathbf{r}_{sl}$ *and* $\mathbf{r}_{el}$ *indicate the intersections of the ray and the tetrahedron (as in left image), where* w_{sl} *and* w_{el} *are the corresponding values (green rectangles) as obtained by the de Casteljau algorithm. The green circles* $\mathbf{r}_{sl} < \mathbf{r}_i < \mathbf{r}_{el}$, $i = 1, \ldots, N$ *are intermediate position along the ray, where values (green rectangles)* w_i *are obtained by the de Casteljau algorithm as well. All values are used to defined polynomials in Newton form of total degree* 1 *(right top image) and* 2 *(right bottom image).*

Part III

Fast Shear-Warp Algorithm

1 Introduction

1.1 Related Work

In this section we summarize the state of the art based on the shear-warp algorithm, where the original method [Lac95] is still considered as one of the fastest software based volume rendering implementations. We also briefly recall the basic ideas making such an efficient implementation possible, and finally in the following sections we further develop this method into our new implementation. However, the original shear-warp method [LL94] [Lac95] belongs to the so called intermediate algorithms. It efficiently utilizes both acceleration techniques, i.e. space leaping and early-ray termination (see part I). It has been shown that this algorithm is very efficient due to its ability to optimally use the cache architecture found in modern computers. In [SNL01] a perspective shear-warp algorithm in a virtual environment based on the specifications in [Lac95] has been developed. Later, this algorithm has been parallelized [SL02]. Artifacts arising in the original implementation led to modifications and essential further developments of the shear-warp approach. In [SM02] a sophisticated improvement using intermediate slices was introduced to reduce the overall sampling distance, which is often considered as a major source for artifacts visible on the screen. A similar approach using *Hermite* curves was given in [SHM04]. Recently, [SKLE03] included pre-integrated volume rendering in the shear-warp algorithm for parallel projection to further reduce potential problems connected with the classification step in the rendering. Our approach presented below is orthogonal to pre-integrated volume rendering, since it uses linear univariate models of the opacity and color function defined within a cubic cell, whereas we consider trivariate piecewise polynomials on tetrahedral and cubic partitions. Therefore, we can easily extend this method using pre-integrated rendering as well. In addition, the three-fold overhead for coding the volume data has been solved by reducing the coding to two [SM02] and one coded volume(s) [SHM04], respectively.

1.2 Overview of the Shear-Warp Algorithm

Let us first summarize the original method, which uses three run-length encoded data sets[1] one for each main viewing direction. This is done by classification based on the density values, i.e. mapping the scalar values (density) to opacities and colors and encoding these values using appropriate data structures into run-length encoded data sets. Each run-length encoded data item (voxel) consists of two entities, i.e. an opacity value and a shading index representing the local gradient. Next, just before rendering the *total* transformation matrix $\mathbf{T}$ is factorized into a *shear* $\mathbf{S}$ and a *warp* $\mathbf{W}$ matrix. The major viewing axis is determined from $\mathbf{T}$ and the appropriate run-length-encoding

[1] A data reduction which encodes runs of voxels with opacities smaller than a user-defined threshold by the length of the run.

is chosen. Then, the sheared volume slices are scaled, re-sampled and projected in front-to-back order into the *intermediate* image, aligned with the volume slice most parallel to the final image plane. The grid spacing in this *intermediate* image equals that of the volume being rendered – only for the parallel projection case. Rays perpendicular to the *intermediate* image are cast from the pixels of the *intermediate* image plane into the sheared volume. During rendering two adjacent run-length encoded scan-lines are traversed simultaneously. *Space-leaping*, i.e. skipping transparent (low opacity) data values (voxels), is realized efficiently because of the run-length encoded scan-lines of the volume. Each time a non-transparent data (a voxel) is detected in one of the two voxel runs the corresponding pixel in the scan-line of the *intermediate* image is updated. If this pixel is already opaque, i.e. the value of the accumulated opacity in this pixel is above a user-defined threshold, then all adjacent opaque pixels are skipped as well, and the dynamic run-length encoded data structure of the *intermediate* image is updated. Otherwise, if this pixel is not opaque the shading indices of the run-length encoded data are used to compute the colors by using a shading lookup table. The colors as well as the opacities are bi-linearly interpolated and the resulting new color and opacity are then composite with the corresponding *intermediate* image pixel colors and opacities. Finally, after rendering all slices, the two-dimensional transformation is applied to the *intermediate* image by using the *warp* matrix.

1.3 Basic Idea of the Shear-Warp Factorization

The basic idea of the shear-warp algorithm relates the to standard ray-casting algorithm. Here, basically rays

$$\mathbf{r}_{\nu,\mu}(t) \;=\; \mathbf{r}_s^{\nu,\mu} + t\ \mathbf{r}_d^{\nu,\mu} \tag{1.1}$$

are cast into the volume from each image pixel $\nu := 0, 1, \ldots, N$ and $\mu := 0, 1, \ldots, M$, where $N \times M$ is the size of the image (or size the viewport) and $\mathbf{r}_s^{\nu,\mu}, \mathbf{r}_d^{\nu,\mu} \in \mathbb{R}^3$ are the ray start and direction in image space, respectively. These rays can be determined and manipulated by a projection matrix, which transforms points from object space into the image space by

$$\begin{aligned} \mathbf{x_i} &= \mathbf{M}_v\ \mathbf{M}_p\ \mathbf{M}_l\ \mathbf{M}_t\ \mathbf{x_o}, \\ \mathbf{x_i} &= \mathbf{M}\ \mathbf{x_o}, \end{aligned} \tag{1.2}$$

where $\mathbf{M}_t, \mathbf{M}_l, \mathbf{M}_p$, and $\mathbf{M}_v \in \mathbb{R}^{4\times4}$ are the object to world (*t*ransform), world to eye or camera (*l*ook-at), eye or camera to clip (*p*rojection) and clip to image (*v*iew-port) transforms (matrices), respectively. The homogeneous coordinates (point locations) in object and image space are denoted as $\mathbf{x_o}, \mathbf{x_i} \in \mathbb{R}^4$ and $\mathbf{M}$ is called the *total* transformation matrix. The idea now is to decompose this matrix as

$$\mathbf{M} \;=\; \mathbf{M}_{warp}\ \mathbf{M}_{shear}\ \mathbf{M}_{perm}, \tag{1.3}$$

where $\mathbf{M}_{perm}, \mathbf{M}_{shear}$, and $\mathbf{M}_{warp} \in \mathbb{R}^{4\times4}$ are the *perm*utation, *shear*, and *warp* matrices, respectively. This results – as can be seen in Fig. 1.1 and 1.2 – in rays which are perpendicular to one face of the volume and the so called *intermediate* image. In other words, in intermediate image space all rays going through the intermediate image pixels

are perpendicular to that image. In this sense, all slices of the volume can be sheared, scaled, re-sampled and projected into the intermediate image according to this shear matrix where according to the permutation matrix or the main viewing direction the respective data set to be projected is chosen. Note, to satisfy the compositing order which results from the volume rendering integral the slices are processed in a front-to-back manner. Finally, the intermediate image is transformed by the warp matrix into the final image, which is a two-dimensional transformation only and can be efficiently realized by graphics hardware. However, in the next section we will recall the decomposition of the total transformation matrix $\mathbf{M}$ into its parts $\mathbf{M}_{warp}, \mathbf{M}_{shear}$, and $\mathbf{M}_{perm}$ in a more detailed way.

1.4 Factorization of the Transformation Matrix

1.4.1 Parallel Projection Case

In this case the projection matrix $\mathbf{M}_p$ of Equ. (1.2) is assumed to be a parallel projection matrix $\mathbf{M}_p := \mathbf{M}_{ortho}$, i.e. the viewer (or viewpoint) is infinitely far away from the object and the rays become parallel (cf. Fig. 1.1). Now, the total transformation $\mathbf{M} \in \mathbb{R}^{4\times4}$ becomes an affine viewing transformation matrix which transforms points from object space to image space with no perspective scale. The goal is to factor this matrix according to Equ. (1.3).
The shear matrix transforms standard object space coordinates into sheared object space coordinates and by definition the standard object space coordinate system is sheared until the viewing direction becomes perpendicular to the slices of the volume. However, in image space the viewing direction vector is defined as $\mathbf{v}_i := (0, 0, 1, 0)^\mathsf{T}$ and by applying the linear system of equations (cf. Equ. 1.2) we obtain

$$\begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & m_{44} \end{pmatrix} \begin{pmatrix} v_{o,x} \\ v_{o,y} \\ v_{o,z} \\ 0 \end{pmatrix}. \tag{1.4}$$

Note that alternatively we could use the viewing direction in camera or eye space, i.e. $(0, 0, -1, 0)^\mathsf{T}$, and the appropriate transformations to compute $\mathbf{v}_o$. However, the viewing direction vector $\mathbf{v}_o := (v_{o,x}, v_{o,y}, v_{o,z}, 0)$ in object space can be easily found using *Cramer's* rule or the inverse $\mathbf{M}^{-1}$. Now, the principal viewing axis is defined as the object-space axis which is most parallel to the viewing direction $\mathbf{v}_o$. Hence, to avoid special cases a permutation matrix has to be defined as well, which relates to the principal viewing axis. This transforms object space coordinates into standard object space coordinates by $\mathbf{x}_{so} = \mathbf{M}_{perm}\mathbf{x}_o$. The application of the inverse of this permutation matrix to the transformation matrix $\mathbf{M}$ gives a new permuted transformation matrix $\mathbf{M}' = \mathbf{M}\mathbf{M}_{perm}^{-1}$. This transforms standard object coordinates into image coordinates, i.e. $\mathbf{x}_i = \mathbf{M}'\mathbf{x}_{so}$. In standard object coordinates the z axis is always the principle viewing axis. Thus, using $\mathbf{M}'$ and $\mathbf{v}_{so} = \mathbf{M}_{perm}\mathbf{v}_o$ the necessary shear factors in x, y directions in standard object space are

$$s_x = -v_{so,x}/v_{so,z}, \tag{1.5}$$
$$s_y = -v_{so,y}/v_{so,z} \tag{1.6}$$

and according to Equ. (1.3) we obtain

$$\mathbf{M}' = \underbrace{\begin{pmatrix} m'_{11} & m'_{12} & (m'_{13} - s_x m'_{11} - s_y m'_{12}) & m'_{14} \\ m'_{21} & m'_{22} & (m'_{23} - s_x m'_{21} - s_y m'_{22}) & m'_{24} \\ m'_{31} & m'_{32} & (m'_{33} - s_x m'_{31} - s_y m'_{32}) & m'_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{M}'_{warp}} \underbrace{\begin{pmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{M}'_{shear}}, \quad (1.7)$$

$$= \mathbf{M}'_{warp} \mathbf{M}'_{shear}, \quad (1.8)$$

where the shear matrix $\mathbf{M}'_{shear}$ projects each slice of the volume onto the slice located at $z = 0$. The projected and composited slices define the intermediate image, where the current coordinate system is not convenient because the origin is not located at a corner of the intermediate image. For this, the coordinate system of the intermediate image can be defined by projecting the eight corners of the volume into the slice located at $z = 0$ by using the above shear transformation $\mathbf{M}'_{shear}$. Then, the minimal x, y coordinates of the

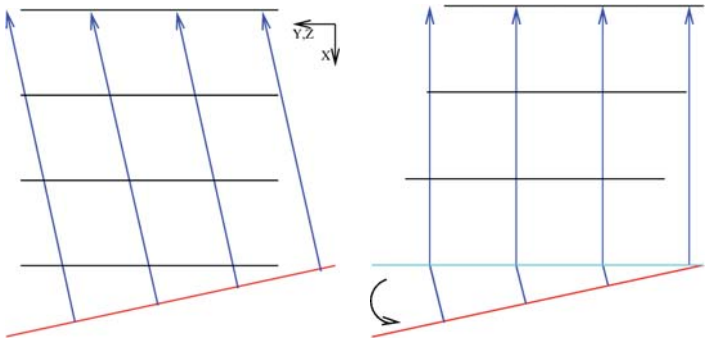


Figure 1.1: *Left: Ray-Casting using a parallel projection matrix. Right: Shear-Warp factorization. After decomposition the rays are perpendicular to the volume slices (lines in black color) and the* intermediate *image (line in cyan color), which coincides with the first slice of the volume. All volume slices are sheared, re-sampled and projected into the* intermediate *image by using the* shear *matrix. Finally, this image is transformed using the* warp *matrix into the final image (line in red color).*

projected volume are repositioned to the lower-left corner (the origin) of the intermediate image, which gives the translation offsets t_x, t_y. The shear and translation information defines the intermediate image coordinate system, thus the new shear matrix $\mathbf{M}_{shear}$ can be defined and be used to project and composite each slice of the volume into the intermediate image. The compositing order is another important issue. Thus, a front-to-back traversal of the volume slices has to be satisfied, but this can correspond to looping through the slices in increasing order or in decreasing order depending on the viewing direction. This stacking order of the slices is found by examining the component of the viewing direction vector corresponding to the principal viewing axis, i.e. $v_{so,z}$. If $v_{so,z} \geq 0$, then the slice at $z = 0$ is the front slice, otherwise the slice located at $z = z_{max}$ (where z_{max} is the size of the volume in depth direction in the standard coordinate

system). Finally, the shear matrix equals,

$$\mathbf{M}_{shear} = \begin{pmatrix} 1 & 0 & s_x & t_x \\ 0 & 1 & s_y & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.9}$$

and transforms standard object coordinates to intermediate image coordinates, where the warp matrix can be computed from

$$\mathbf{M}_{warp} = \begin{pmatrix} m'_{11} & m'_{12} & (m'_{13} - s_x m'_{11} - s_y m'_{12}) & m'_{14} \\ m'_{21} & m'_{22} & (m'_{23} - s_x m'_{21} - s_y m'_{22}) & m'_{24} \\ m'_{31} & m'_{32} & (m'_{33} - s_x m'_{31} - s_y m'_{32}) & m'_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.10}$$

and transforms intermediate image coordinates to the final image coordinates. Thus, the shear-warp factorization of an arbitrary affine viewing transformation matrix $\mathbf{M}$ includes a permutation $\mathbf{M}_{perm}$, a shear $\mathbf{M}_{shear}$, and a warp $\mathbf{M}_{warp}$ matrix and can be written as $\mathbf{M} = \mathbf{M}_{warp}\mathbf{M}_{shear}\mathbf{M}_{perm}$ (cf. Equ. 1.3).

1.4.2 Perspective Projection Case

Similar to the parallel projection case the matrix $\mathbf{M}_p$ of Equ. (1.2) is now assumed to be a perspective projection matrix $\mathbf{M}_p := \mathbf{M}_{frustum}$, i.e. the viewer (or viewpoint) is located at a finite distance from the object (cf. Fig. 1.2). Now, the total transformation matrix $\mathbf{M} \in \mathbb{R}^{4\times4}$ becomes a perspective viewing transformation matrix which transforms points from object space to image space using a perspective scale of the homogenous coordinates. Note that this transformation matrix needs not to be singular. The goal is once more to factor this matrix according to Equ. (1.3). However, the first step is to find the eye location in object space $\mathbf{e}_o$ according to Equ. (1.2), where the the eye position in image space is defined to be $\mathbf{e}_i := (0, 0, -1, 0)$. Hence, once more the eye position can be found by solving the linear system of equations

$$\begin{pmatrix} 0 \\ 0 \\ -1 \\ 0 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} e_{o,x} \\ e_{o,y} \\ e_{o,z} \\ e_{o,w} \end{pmatrix}. \tag{1.11}$$

In the second step we need to find the principal viewing direction. Basically, it is the same procedure as in the parallel projection case, i.e. the principal viewing axis is defined as the object-space axis which is most parallel to the object-space viewing direction. But, since now we do not have a unique viewing direction, several procedures are possible. The first possibility is to define eight vectors, i.e. from the eye position $\mathbf{e}_o$ to each corner of the volume $\mathbf{p}_{o,i}$, where $i = 0, \ldots, 7$ (note, in this case i is an index). Then, for each vector $\mathbf{p}_{o,i} - \mathbf{e}_o$ the same procedure is used as for an affine case to find the corresponding principal viewing axis, i.e. the largest component of this vector determines the axis. If for all vectors we get the same principal viewing axis, then, we are fine and one permutation matrix can be chosen to avoid spacial cases. Otherwise, the volume has to be split into at most eight sub-volumes which are rendered from different principal

viewing directions (axes), i.e. the factorization is proceeded for each of the required principal viewing directions. Another possibility is to apply the same procedure used for the affine case to find the principle viewing axis. In this simpler but faster case we avoid the splitting of the volume into several parts at the cost of reduced rendering quality if the eye gets too close to the object. However, once the permutation matrix is chosen – based on the principal viewing axis – the transformation from standard object space to image space can be computed as $\mathbf{M}' = \mathbf{M}\mathbf{M}_{perm}^{-1}\mathbf{M}_{trans}^{-1}$, where $\mathbf{M}_{trans}$ is a translation matrix, that translates the origin of the volume to avoid divisions by zero. Thus, again using $\mathbf{M}'$ and $\mathbf{e}_{so} = \mathbf{M}_{trans}\mathbf{M}_{perm}\mathbf{e}_o$ the necessary shear factors in x, y directions are

$$s_x = -e_{so,x}/e_{so,z}, \tag{1.12}$$

$$s_y = -e_{so,y}/e_{so,z} \tag{1.13}$$

whereby now we obtain a perspective scaling transformations

$$s_z = -e_{so,w}/e_{so,z} \tag{1.14}$$

as well. According to Equ. (1.3) we obtain

$$\mathbf{M}' = \mathbf{M}'_{warp}\mathbf{M}'_{shear} \tag{1.15}$$

$$= \mathbf{M}'_{warp} \begin{pmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & s_z & 1 \end{pmatrix}, \tag{1.16}$$

The shear matrix $\mathbf{M}'_{shear}$ derived above guarantees that the volume slice located in the $z = 0$ plane has a scale factor of one. Depending on the stacking order of the volume

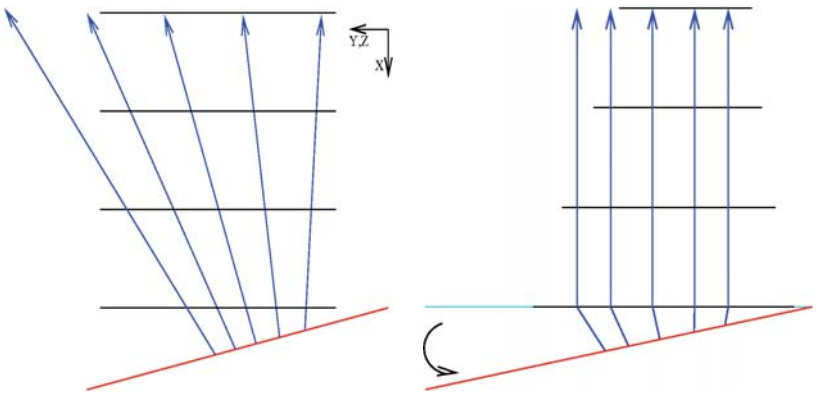


Figure 1.2: *Left: Ray-Casting using a perspective projection matrix. Right: Shear-Warp factorization. After decomposition rays are perpendicular to the volume slices (lines in black color) and the* intermediate *image (line in cyan color), which coincides with the first slice of the volume. All volume slices are sheared, scaled, re-sampled and projected into the* intermediate *image by using the* shear *matrix. Finally, this image is transformed using the* warp *matrix into the final image (line in red color).*

slices – as in the affine case – that slice is not necessarily the front volume slice. Thus,

another uniform scale has to be defined so that the front-most volume slice is scaled by a factor of one. Then, a translation is chosen to position the origin of the projected volume cube at the bottom-left corner of the intermediate image. Hence, the final shear transformation equals

$$\mathbf{M}_{shear} = \begin{pmatrix} f & 0 & fs_x + t_x s_z & t_x \\ 0 & f & fs_y + t_y s_z & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & s_z & 1 \end{pmatrix} \tag{1.17}$$

and transforms standard object coordinates to intermediate image coordinates, where

$$f = \begin{cases} 1 & \text{if slice } k = 0 \text{ is in front} \\ 1 - z_{max} s_z & \text{otherwise} \end{cases} \tag{1.18}$$

and $z_{max} := N - 1$ with N the size (dimension) of the volume data set (as before) along the z direction (in standard object space). Here, once more, t_x, t_y are the translation factors which translates the deformed space (the lower left corner) into the intermediate image space (origin). The warp matrix is given again by

$$\mathbf{M}_{warp} = \mathbf{M}' \mathbf{M}_{shear}^{-1} \tag{1.19}$$

which transforms intermediate image coordinates to the final coordinates. Hence, the shear-warp factorization of an arbitrary perspective viewing transformation matrix $\mathbf{M}$ includes a permutation matrix $\mathbf{M}_{perm}$, a shift of the origin matrix $\mathbf{M}_{trans}$ (only if the eye point $\mathbf{e}_o$ in object space lies in the $z = 0$ slice of the volume), a three-dimensional shear, perspective scale, and a translation matrix $\mathbf{M}_{shear}$, and a perspective warp matrix $\mathbf{M}_{warp}$ and can be written as $\mathbf{M} = \mathbf{M}_{warp} \mathbf{M}_{shear} \mathbf{M}_{trans} \mathbf{M}_{perm}$.

1.4.3 Properties

The projection from the object space to the intermediate image space has several well known geometric properties that simplify the rendering algorithm. First, scan-lines in the intermediate image are parallel to scan-lines in the volume slices. Second, each volume slice is scaled by the same factor, whereas in parallel projection case this factor can be chosen arbitrarily[2]. Third, in the parallel projection case the interpolation weights for the voxels within a slice are the same, whereas in the perspective projection case they have to be computed on the fly (cf. Fig. 1.3).

1.5 Data Structures

1.5.1 Run-Length Encoded Volume

First, let us review some data structures used for run-length encoding of the raw data set and process itself. This is important for further discussion and for our new extensions done to overcome the data redundancy.

[2] In fact, a unity scale factor is chosen so that for a given voxel scan-line there is a one-to-one mapping to the intermediate-image pixels.

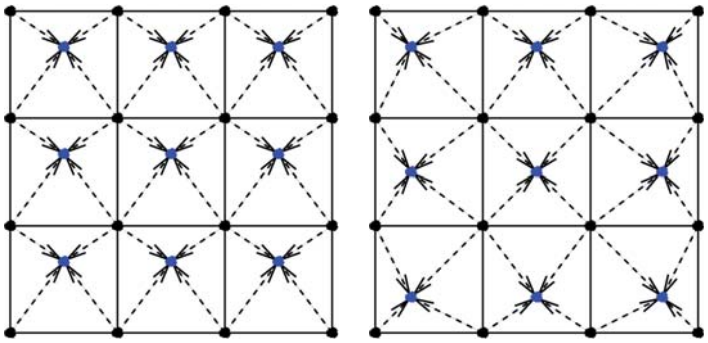


Figure 1.3: *Left: The interpolation weights are the same for each voxel within a slice in the parallel projection case. Right: The interpolation weights have to be recomputed for each voxel within a slice in the perspective projection case. In both cases the black dots are the original data values located on a volume slice, whereas the blue dots are the interpolated values located on a volume slice as well, but at positions where the rays intersect the corresponding slice.*

The run-length encoded volume is constructed in a preprocessing step. For this, the volume is traversed in object order and a user-specified opacity function is used to classify each voxel, i.e. using an opacity transfer function each raw data value (density) – usually stored in 8 bit – is classified, giving the opacity. Then, using also a user-specified threshold each classified voxel (i.e. the voxel's opacity) is used to determine the transparency, i.e. whether to save this voxel in the run data structure[3] or not. Thus, for all non-transparent voxels the lengths of the runs and the data itself is stored, whereas for the transparent voxels only the lengths of the runs are stored. The local gradients are computed by using central differences, Sobel operator, or higher order approximations from the classified data (opacities) or the original raw data values (densities). Applying a quantization method to the gradients gives an index. This index and the opacity (or density) are stored in the data array of the run-length encoded data set (as one needs). The gradient itself is saved in another lookup table at the index position for later use (i.e. for shading during rendering by using Phong illumination model). Note, in the original method a different table is used, where at the gradient's index position the result of the shading using the local gradient is stored. This accelerates the algorithm enormously. However, one has to make notes in the run-length array of the run-length encoded data set, i.e. how many values are encoded and/or skipped. Only together with this information one is able to represent the raw data set as a run-length encoded set.

For the representation of the volume as a run-length encoded volume a data structure of three arrays is necessary. The first array contains the non-transparent data after classification[4], whereas the second one encloses the run-length data, i.e. how many (non-)empty data values are already processed and stored. Further, another array is necessary which combines the data and the run-length array of the encoded data set, and this one stores pointers into the first two arrays and is used to find the beginning of each slice or scan-line of a slice of the original volume (cf. Fig. 1.4). We can say that this array accounts for the synchronization of the other two arrays. In the original method

[3] A run is a sequence of contiguous voxels that are all transparent or all non-transparent.
[4] Classified data below a user-defined threshold is not stored in the data array.

this is a simple one-dimensional array (cf. left drawing in Fig. 1.5), where each entry contains a pointer to the run-length data (i.e. the number of (non-)empty data within a slice) and a pointer to the data values itself (i.e. the opacities and indices, which are used for the gradient- or color lookup tables). Each entry in this one-dimensional array is associated with the beginning of a slice in the run-length encoded volume (cf. black marked array in left drawing of Fig. 1.5).

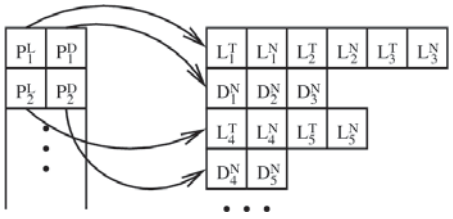


Figure 1.4: *Left: An array of pointers, where each entry P_i^L and P_i^D is a pointer used to find the beginning of a slice or a scan-line with run lengths L and data D values, respectively. Right: The run length array ($L_j^\star$) contains alternating entries of transparent (L_j^T) and non-transparent (L_j^N) run lengths. Each array starts with a transparent run and ends with a non-transparent run. The corresponding data array (D_j^N) contains only data values (as for example the opacity, density, and color values) of the non-transparent runs (L_j^N). Note that all arrays are continuous in memory, i.e. the run-length entries are stored as $L_1^\star, L_2^\star, \ldots, L_k^\star$ and the data values as $D_1^N, D_2^N, \ldots, D_k^N$ and only the pointers P_i^L and P_i^D are set appropriately.*

The 3-Fold Redundancy

The original shear-warp approach [Lac95] comes with a 3-fold redundancy of run-length encoded data sets. That means, three run-length encoded data sets are pre-computed, one for each principle viewing direction x, y and z, respectively. During rendering one of those data sets – dependent on the principle viewing direction (cf. Sec. 1.4) – is chosen to be projected it onto intermediate image, where the run-length encoded slices are processed in front-to-back manner. In the following sections, we are going to describe how to remove two of the three data sets without sacrificing the efficiency of the original method significantly.

The 2-Fold Redundancy

From the left drawing in Fig. 1.5 one can observe that the two individual run-length encoded data sets, i.e. used for the principal z and y viewing directions are basically identical. The main difference is the order in which they are used, i.e. the order how the original data set is traversed dependent on the principle viewing direction. With this observation two operations have to be realized. First, the run-length encoded data set used for the principle z viewing direction is still maintained[5]. Then, a two-dimensional array is used to store pointers into each scan-line of a slice of the run-length encoded volume data set. Previously, a one-dimensional pointer array to the run-length encoded slices has been used only. This modification allows reusing the run-length encoded Z

[5] Any one of the three data sets could be retained as well.

data set for both the z and y principle viewing direction. Next, the permutation matrix (cf. Sec. 1.4) has to be adapted accordingly, i.e. when viewing along the principle y direction. The advantage is that the run-length encoded Y data set has neither to be computed nor to be encoded, i.e. a total of $O(K/3)$ of all non empty data values K stored in the run-length encoded data structure can be saved as well as the pointer and run-length data structures. Whereby the size of the whole run-length encoded data structure is increased now by using a two-dimensional array of size $N \times M$ with pointers into scan-lines of volume slices. The original method uses a one-dimensional array of size N with pointer into the slices itself only[6].
However, this small modification allows still for efficient space leaping and early-ray termination, both remaining unchanged from the original algorithm. Even the scan-lines used during rendering are usually sufficiently long, which means that the cache is filled often, and, therefore, degradation in performance by cache misses is unlikely.

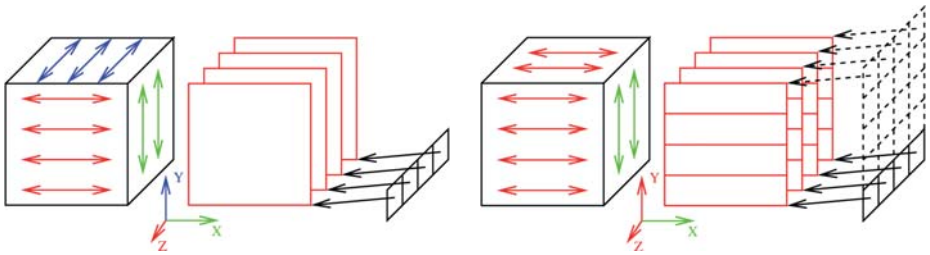


Figure 1.5: *Left: Three run-length encoded (RLE) data sets (original, cf. [Lac95]). The RLE Z, Y, and X data sets are marked in red, blue, and green colors and are used for the z, y, and x principle viewing directions marked in the same colors, respectively. The slices of the RLE Z data set are highlighted as red, where a one-dimensional array (drawn as black colored array rightmost in this left figure) is used to store pointers to the run-length data and the data itself of each slice of this run-length encoded data set. This encoding is done for each of the three data sets separately. Right: Two run-length encoded (RLE) data sets (cf. [SM02] [SHM04]). The RLE Z and X data sets are marked in red and green colors and are used for the z, y, and x principle viewing directions marked in the same colors, respectively. Once more the slices of the RLE Z data set are highlighted as red, where now an two-dimensional array (drawn as black colored array rightmost in the right figure) is used to store pointers to the run-length data and the data itself of each scan-line of a slice. This encoding is done for each of the two data sets separately. Note, the run-length encoded Y data set (marked in blue color) has neither to be computed nor to be encoded.*

Non-Redundancy

The second run-length-encoded volume, i.e. that used for the principal x viewing direction, can not be removed without sacrificing the efficiency of the shear-warp method. However, the basic idea to eliminate the second run-length encoded data set is to use new data structures and a combination of ray-casting and shear-warp. Before we go into more details in Sec. 2, let us outline the problem. By simply removing the run-length encoded X data set (shown in green color in Fig. 1.5) one has to care about the consequences. That means, considering the main z and y viewing directions the scan-lines

[6]Here N and M are the dimensions of the raw data volume in z and y directions, respectively.

of the Z data set are still processed in a parallel manner according to the intermediate image scan-lines. In contrary to this, if the principle viewing direction becomes the x direction we have to process the volume scan-lines in a perpendicular way to the intermediate image, i.e. we process the Z data set still in storage order (object order) but the intermediate image scan-lines are now processed in an oblique manner. For this we have to develop new data structures, which allow us to project the scan-lines oriented perpendicular to the intermediate image in an appropriate way. Moreover, the two standard acceleration techniques (space-leaping and early-ray-termination) should be applicable and efficient (cf. Fig. 1.6) as well.

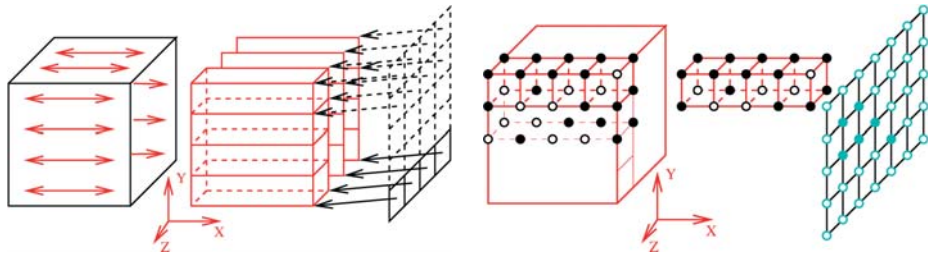


Figure 1.6: *Left: The same configuration as in right drawing of Fig. 1.5, where here the RLE X data set has been removed. Right: Using the run-length-encoded (RLE) Z data set only (cf. [SHM04] [SHZ+05]), still a two-dimensional array of pointers into run-length encoded scan-lines is used (see left figure). Removing the RLE X data set one has to process the the scan-lines of the RLE Z data set, which are perpendicular to the intermediate image during rendering. Thus, the intermediate image scan-lines are processed in an oblique fashion. In other words, here the intermediate image is aligned with the y, z plane of the coordinate system and the appropriate volume face. The volume scan-lines are encoded along the x-direction of the coordinate system. The grid constant, i.e. the spacing between the data values in the volume (voxels) and the intermediate image (pixels) is equal. In the center of the right image a* column *consisting of four run-length encoded scan-lines of the Z data set is emphasized. Here empty data values (i.e. where opacity is smaller than a threshold) are marked as black circles, whereas non empty data values are drawn as solid black dots. Now each such* column *in the run-length encoded Z data set has to be processed, and the contributions are projected onto the appropriate intermediate pixels, where solid cyan dots denote opaque pixels and cyan circles (semi-)transparent pixels.*

1.5.2 Coherence Encoded Volume

Run-length data structures can be applied to describe empty space or better chunks of equal data values (i.e. coherence) along a direction. This, for example, can be the case within a volume scan-line. Here, an empty or a constant run of data values is encoded by a number (the sum of equal data values) and the data value itself, no matter of the underlying data type (i.e. whether we deal with opacity, density or color values). Hence, this data structure encodes piecewise constant functions very well, i.e. homogeneity within the local neighborhood [FS97] (where emptiness is a special case). However, higher degree polynomials can be applied here as well to encode linear (cf. [CHM01] [Che01]), quadratic or cubic functions within the data along one direction. In other words, an arbitrary one-dimensional function is subdivided into several (not necessarily) equidistant intervals, where each interval is represented by a linear, quadratic or cubic

function. Note that one has to carefully put the piecewise linear, quadratic or cubic functions together, i.e. by considering appropriate smoothness conditions across the intervales of the whole function space.
However, the algorithm to compute the coherence encoding of a volume scanline (a one-dimensional function) or better the breakpoints is applied to our shear-warp method and can be described as follows.
The linearization of an arbitrary function[7] (or curve) $f(x) \in \mathbb{R}$ can be performed by an error-based criterion [Che01]

$$\text{err}(c_1, c_0) \quad = \quad \frac{1}{i_e - i_s + 1} \sum_{i=i_s+1}^{i_e-1} |f(x_i) - g(x_i)| \tag{1.20}$$

with $g(x) = c_1 x + c_0$ the piecewise linear function. Here $c_1 = (f_{i_e} - f_{i_s})/(x_{i_e} - x_{i_s})$ and $c_0 = (f_{i_e} x_{i_s} - f_{i_s} x_{i_e})/(x_{i_s} - x_{i_e})$ are the parameters of a linear function $g(x)$ defined by the current considered two boundary positions x_{i_s} and x_{i_e} and their values $f_{i_s} := f(x_{i_s})$ and $f_{i_e} := f(x_{i_e})$ with interval $h := x_{i+1} - x_i$ taken from the original function $f(x)$. The number of sample points including the boundary values is $i_e - i_s + 1$. Note, in this coherence encoding approach the goal is not to directly approximate an arbitrary continuous function or curve, but only some sample values obtained possibly from such a function. The approximation of a continuous function f using this error-based criterion depends on the sampling interval h and the strategy would be a little bit different, i.e. smaller the interval, better the approximation and thus smaller the error between the original function and the linearization. However, in real world data sets or applications we often deal with discrete data samples only, mostly we even do not know the continuous function the samples are taken from and only assume for example a linear, quadratic, etc. reconstruction model for the data. Hence, linear coherence encoding here means that an array of discrete data samples is split into several nonequal sized sub-arrays, where all the samples in such a sub-array can be represented by a linear function defined by the boundary samples of the corresponding sub-array. That means, only the boundary samples have to be stored, all other values in between can be reconstructed(cf. Fig. 1.7). In other words, some breakpoints are determined in the array of discrete data samples, which further are used to define piecewise linear functions, such that all intermediate data samples can be reconstructed (or approximated when assuming a small error threshold value) by these piecewise functions and thus have not to be stored by using an appropriate data structure.
A straight forward generalization to quadratic coherence encoding using piecewise quadratic functions would be to replace the linear function by a quadratic function of the form $g(x) = c_2 x^2 + c_1 x + c_0$. Hence, the parameters c_2, c_1 and c_0 of $g(x)$ would be computed from the two boundary points as well, i.e. located at positions x_{i_s} and x_{i_e}, and their values $f_{i_s} := f(x_{i_s})$ and $f_{i_e} := f(x_{i_e})$. The third point needed to determine all parameters of $g(x)$ in this case would be any point between the boundary samples, i.e. for example at position x_{i_i} with $i_i := \lfloor (i_s + i_e)/2 \rfloor$[10] and $f_{i_i} := f(x_{i_i})$. Note that now it is not enough to store the boundary values only. We need to save the value at the intermediate position

[7] Here, for example x is the spacial position along a scan-line in the volume with $f(x)$ the opacity or density value at this location x.
[10] Here, $\lfloor b \rfloor$ is the maximal integer $\leq b$, i.e. the *floor* operator.

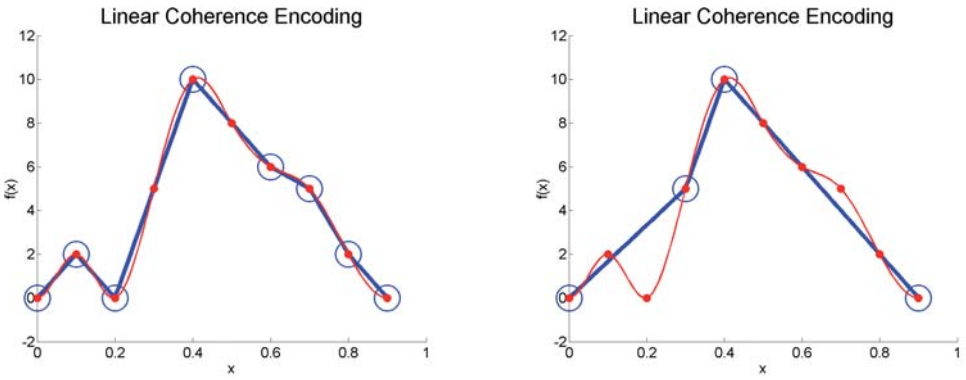


Figure 1.7: *In linear coherence encoding of (red dots) data value taken at discrete position from the (red) original continuous curve some (blue circles) breakpoints are computed. These breakpoints define (blue) piecewise linear functions which can be used to reconstruct data in between and thus possibly to save some (red dots) data samples. The encoding of data samples with* $\epsilon = 0$ *(left figure) and* $\epsilon = 1$ *(right figure) are depicted, where eight and four breakpoints are needed, and two and six samples (values) are saved, respectively. In both figures the error*[9] *between the sample values of the original function, the function itself and the piecewise linear representation is depicted. However, in the left image the related error becomes* 0 *comparing with the sample points and* 0.25 *comparing with the original function, respectively. Whereas in the right figure it is* 1.08 *comparing with the sample points (i.e.* 0.92 *between breakpoints with indices* $i_s = 0$ *and* $i_e = 3$ *and* 0.17 *between breakpoints with indices* $i_s = 4$ *and* $i_e = 9$*) and* 0.72 *compared to the original function.*

x_{i_i} as well to be able to reconstruct the other intermediate data values later. Alternatively the parameters of the piecewise function representing this interval could be stored. Note that appropriate smoothness conditions between the piecewise quadratic functions should be introduced, such that the overall encoding will be continuously differentiable.

1.5.3 Run-Length Encoded Intermediate Image

The intermediate image is represented by a run-length encoded data structure as well. This encodes runs of opaque and non-opaque pixels. Nevertheless, the requirements here are different compared to the run-length encoded volume data structure, which can be pre-computed. The intermediate image data structure has to be modified during rendering, thus a dynamic generation of opaque pixel runs, a merging of adjacent runs, and a fast method to find the end of a run is necessary here. All this allows a fast implementation of early-ray termination, i.e. voxel computations in adjacent slices can be skipped if the corresponding pixels are already opaque.
The intermediate image data structure is a two-dimensional array of pixels, where each pixel contains a color, an opacity, and a relative offset. The size of this two-dimensional array is determined from the shear-warp factorization. The offset stores the number of pixels to skip to reach the end of the current opaque pixel run, whereas an offset equals to zero means that this pixel is non-opaque (cf. Fig. 1.8).
The intermediate image data structure is used as follows. Just after the shear-warp factorization it is initialized, i.e. all pixel opacities are made transparent and all offsets are set to zero. Then, during rendering but before any computations are done, it is

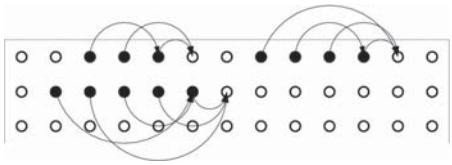


Figure 1.8: *The black circles are identified with non-opaque intermediate image pixels where the corresponding offsets are set to zero. Analogously, the black dots are identified with opaque pixels and the offsets are updated by means to point to the first non-opaque pixel in the same intermediate image scan-line.*

checked, whether the current pixel's offset is non-zero. If it is the case, then the end of the opaque pixel run is found, thus some pixels as well as voxels can be skipped and the corresponding computation operations are omitted. Otherwise, voxel data is interpolated and composited into the corresponding intermediate image pixel. When after compositing the new pixel's opacity exceeds the maximum opacity threshold, then the pixel's offset is set to one[11] and possibly some adjacent opaque pixel runs are merged with the new one. In other words the dynamic run-length encoding of the intermediate image scan-lines is updated.

1.6 Volume Reconstruction

For the transformation of a slice into sheared object space (or intermediate image space) it is necessary to reconstruct values at the intersection positions of the rays coming from the intermediate image and the slices by using the stored data values on the volume grid (or slice). However, since the run-length array accounts for transparent voxels, space-leaping within the scan-lines can be applied easily before any interpolation is done. The original implementation uses a bilinear interpolation filter. Thus, there are two possibilities on how to produce the required interpolated voxel scan-line. First, the backward projection method, where two input scan-lines are traversed and decoded simultaneously. Here each voxel scan-line has to be considered twice, once for each interpolated voxel scan-line to which it contributes. Second, the forward projection method, where each input voxel scan-line is considered only once and it's contributions are distributed into the two interpolated voxel scan-lines. Thus, these partial results have to be stored in a temporary buffer until using the next adjacent input voxel scan-line the final results can be computed. The second method in not very convenient, because the temporary buffer need to be run-length encoded to conceive the benefits of the other coherence data structures. However, during rendering transparent regions are skipped using the run-length array of the encoded data set, whereas the dynamically run-length encoded intermediate image scan-lines allows skipping over occluded voxels. Thus, only voxels which are non-transparent and non-occluded are processed. In the perspective projection case a reconstruction and a low-pass filter is used, because of the divergent rays. However, the implementation differs from the parallel projections case. First, the filter footprint may cover more than two voxel scan-lines. Thus, several input voxel scan-lines have to be considered to generate one interpolated scan-line. Second,

[11] That means a new opaque pixel run is created which contains only the current pixel.

since the slices are scaled not only by unity, the image and volume scan-lines can not be traversed at the same rate. Third, the reconstruction weights have to be recomputed for each sample point.

1.7 Opacity Correction

The volume rendering integral is the physical basis on how to compute color and opacity along a viewing ray from the given volume data, the look-up tables used and how to accumulate the result in the final image. Here, the spacing between two sample points along a viewing ray is considered as constant. Using the shear-warp factorization this spacing is constant in sheared object space, but varies in image space depending on the current view transformation. Thus, colors and opacities computed in object space have to be corrected to account for the different spacings in image and object space. Otherwise, images obtained from an oblique view onto the volume would be about 30% more transparent than images obtained from a perpendicular view. Thus, the opacity α_d computed using an opacity transfer function ϕ, is given by

$$\alpha_d = 1 - \exp(-\phi d), \tag{1.21}$$

where d is the width of the voxel and the initial sample spacing. Then for some other sample spacing d' the corrected opacity can be computed by

$$\alpha_{d'} = 1 - (1 - \alpha_d)^{d'/d}. \tag{1.22}$$

The corrected opacity is a function of the initial opacity and the spacing ratio d'/d. In the parallel projection case this function is the same for every voxel, because of the unit viewing rays. In the perspective projection case the opacity correction is more difficult because of the divergent rays not every voxel requires the same correction. However, the spacing ratio can be computed for each intermediate image pixel just after the viewing transformation has been changed. During rendering the ratios stored at the intermediate images pixel positions can be applied to computed the opacity correction for the sampled values along the corresponding rays.

1.8 Introducing Intermediate Slices

The well known problem arising in the original implementation of the shear-warp algorithm is the sampling distance, which depends on the viewing direction and varies between 1 (for a perpendicular view) and $\sqrt{3}$ (for an oblique view onto the volume). This becomes visible as stripe artifacts, which occur in the resulting images. A partial solution for that is to use *intermediate* slices (cf. Fig. 1.9 and [SM02]). This reduces the artifacts compared to the original implementation, but they are still visible, if using high frequent opacity transfer functions. A way to get rid of this is to perform over-sampling, i.e. introducing as many *intermediate* slices as necessary to satisfy the sampling theorem (see part I).
However, the advantage of using intermediate slices is it's simplicity and the fact, that we do not have to run-length encode any intermediate slices, i.e. they are interpolated

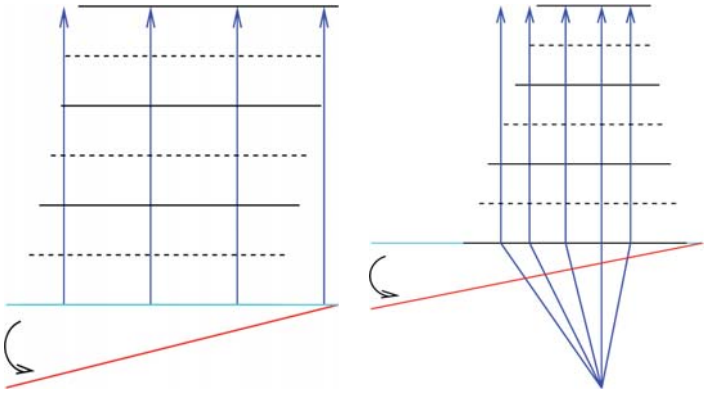


Figure 1.9: *Left: Shear-Warp factorization for the parallel case using intermediate slices. Right: Shear-Warp factorization for the perspective case using intermediate slices.*

and rendered on the fly using the existing run-length encoded slices. The formula to reconstruct values f by bilinear interpolation within the intermediate slices can be written as (see Fig.1.10)

$$f = (\mathbf{s}_1\tilde{u} + \mathbf{s}_2 u)\tilde{v} + (\mathbf{s}_3\tilde{u} + \mathbf{s}_4 u)v \tag{1.23}$$

where $u, v, \tilde{u} = 1 - u, \tilde{v} = 1 - v$ are the interpolation weights of the intermediate slice located at position $k + w,\ w \in [0, 1]$, i.e. between slice k and $k + 1$ and $\mathbf{s}_i,\ i = 1, 2, 3, 4$ are the values on that intermediate slice. These values are reconstructed on the fly by $\mathbf{s}_i = \mathbf{g}_i\tilde{w} + \mathbf{g}_{i+4}w$, thus we obtain the result by using the trilinear interpolation formula the result (see Fig. 1.10)

$$\begin{aligned}
\mathbf{f} &= ((\mathbf{g}_1\tilde{w} + \mathbf{g}_5 w)\tilde{u} + (\mathbf{g}_2\tilde{w} + \mathbf{g}_6 w)u)\,\tilde{v} && (1.24)\\
&+ ((\mathbf{g}_3\tilde{w} + \mathbf{g}_7 w)\tilde{u} + (\mathbf{g}_4\tilde{w} + \mathbf{g}_8 w)u)\,v, && (1.25)\\
&= (\mathbf{g}_1\tilde{u}\tilde{v} + \mathbf{g}_2 u\tilde{v} + \mathbf{g}_3\tilde{u}v + \mathbf{g}_4 uv)\tilde{w} && (1.26)\\
&+ (\mathbf{g}_5\tilde{u}\tilde{v} + \mathbf{g}_6 u\tilde{v} + \mathbf{g}_7\tilde{u}v + \mathbf{g}_8 uv)w, && (1.27)\\
&= \mathbf{f}_1\tilde{w} + \mathbf{f}_2 w, && (1.28)
\end{aligned}$$

where $w = 0.5, \tilde{w} = 1 - w,\ \mathbf{g}_i,\ i = 1, \ldots, 4$ are the four values located on slice k and $\mathbf{g}_{i+4},\ i = 1, \ldots, 4$ are the values located on slice $k + 1$. The variables $\mathbf{g}$ can be identified with colors, gradients, opacities or densities obtained from the run-length encoded data set.

Another technique to suppress artifacts [SHM04] is a numerical continuation of the discrete data set. A *Hermite* polynomial is placed in between sample points in the two adjacent volume slices. It is determined by constraints on two sample points $\mathbf{v}_1 := (u_1, v_1, w_1)$, $\mathbf{v}_2 := (u_2, v_2, w_2)$ and their gradient vectors $\mathbf{f}_1, \mathbf{f}_2$ at these sample points

$$\begin{aligned}
f(t) &= (2t^3 - 3t^2 + 1)\mathbf{v}_1 + (-2t^3 + 3t^2)\mathbf{v}_2\\
&+ (t^3 - 2t^2 + t)\mathbf{f}_1 + (t^3 - t^2)\mathbf{f}_2, && (1.29)
\end{aligned}$$

where the parameter $t \in [0, 1]$ specifies an intermediate point and the tangent slope at the curve can be found by $df(t)/dt$.

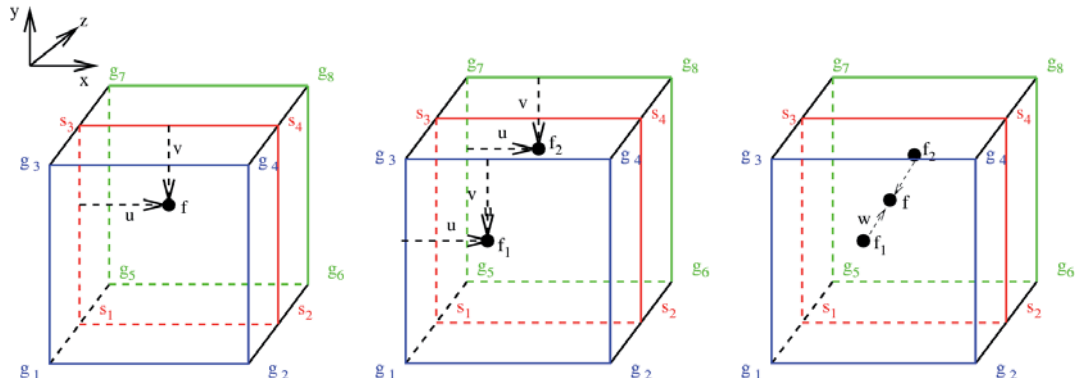


Figure 1.10: *Left: Compute the interpolation weights u, v on the intermediate slice located at position $k + w$ (red rectangle) to reconstruct value f. Middle: Reconstruct values f_1, f_2 located at slices k (blue rectangle) and $k + 1$ (green rectangle), respectively, by using the previously computed weights u, v and the appropriate data values $g_i,\ i = 1, \ldots, 8$. Right: Reconstruct value f located on the intermediate slice from values f_1, f_2 using the weight w. Thus, the data values $s_i,\ i = 1, \ldots, 4$ have not to be encoded nor reconstructed.*

The intermediate slice located at position $k + w$ can be composite into the intermediate image in several ways. First, as a 2-step process. In the first pass running through slice k once more by considering two run-length encoded volume scan-lines at a time, where now the values are weighted by a factor of $\tilde{w}$ and the interpolation weights $u, v, \tilde{u}, \tilde{v}$ are set appropriately for slice $k + w$. In the second pass going through slice $k + 1$, with identical interpolation weights but using the weighting factor w. Once the contributions of both slices are added together, the result can be composite with the current intermediate image in the usual way. Second, by simultaneously compositing slice k and constructing the partial results as in the first pass above, and do the same for slice $k+1$ and the second pass above. However, the speedup may be small by the fact that run traversal would be sub-optimal, since the pixels just occluded by slice k would still be considered for slice $k + w$ and $k + 1$. In both cases a temporary buffer of the size of the intermediate image is necessary to summarize the partial results and composite them afterwards. Another possibility is to consider four run-length encoded volume scan-lines at a time when rendering slice $k + w$. This processing is sub-optimal as well, because of space-leaping. Since only the minimum number of voxels of four instead of two volume scan-lines could be skipped. However, since real volume data sets are varying smoothly a degradation in rendering speed is unlikely, whereas a temporary buffer for partial results is not necessary here.

2 Combination of Ray-Casting and Shear-Warp

In this section some new data structures will be presented which allow us to combine ray-casting and the shear-warp algorithm. Due to these data structures one is able to remove the threefold redundancy of run-length encoded data sets (see Sec. 1.5) and apply different reconstruction models (cf. Sec. 2 and 3) for data reconstruction, which further allows solving the volume rendering integral in a more accurate way. Nevertheless, the algorithm is still performed in object order and utilizes the efficient standard acceleration techniques as well, i.e. fast space-leaping and early-ray termination. That means, the well known advantages of the original shear-warp algorithm are preserved, whereby the quality of the resulting images is dramatically improved by using a ray-casting like approach and smooth reconstruction models, compared to existing methods based on the shear-warp factorization.
First, we give the basic idea of our approach for the parallel projection case and show step by step more details. Afterward the somewhat more difficult case of perspective projection will be discussed.

2.1 Parallel Projection Case

2.1.1 Basic Idea

The basic idea of our new approach is visualized in the right picture of Fig. 2.1, whereas in the left drawing the original shear-warp algorithm (cf. Sec.1) is depicted. First, as we have discussed, in the original method rays are considered to be perpendicular to the sheared slices and the intermediate image. At the intersections positions of the rays and the sheared slices values are bi-linearly reconstructed using the data from the local neighborhood. Note that the coefficients used for bi-linear reconstruction has to be computed once per slice only, which are then applied to the local neighborhood of four data values within a slice to reconstruct a value between them. Thus, after shearing a slice, i.e. computing the position (projection) of that slice in the intermediate image and after determining the interpolation weights, space-leaping within the considered voxel scan-lines of a slice synchronously with early-ray termination by using the corresponding scan-lines of the intermediate image is applied. The bi-linearly interpolated values are composited in front-to-back manner into the appropriate pixels of the intermediate image. Although this approach is quite fast it discriminates the quality of the resulting images. However, by applying the more time-consuming intermediate slices approach as discussed in Sec. 1 it is possible to generate high-quality images. On the other hand some questions remain open. How many intermediate slices have to be chosen to obtain high-quality visual results? Does this depend on the underlying data? Since this approach applies a trilinear model (namely the intermediate slices) for the reconstruction

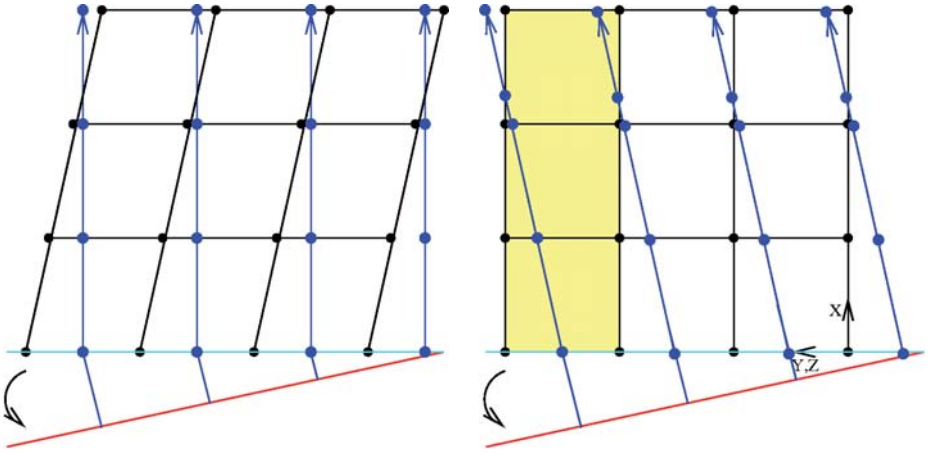


Figure 2.1: *Simplified two-dimensional view of the new shear-warp factorization for orthographic projections. The red lines denote the final image, whereas the cyan lines denote the intermediate image. The black grid is associated with the volume and the cross points (black dots) are the data values. Left: The original shear-warp algorithm. Rays are considered perpendicular to the slices and the intermediate image. At the intersections positions of the rays and the slices (blue dots) values are bi-linearly interpolated (reconstructed). Right: In the new shear-warp technique rays may vary up to* 45° *according to the main viewing axis. The intersections positions of the rays and the grid planes (blue dots) are pre-computed, the values at these positions and anywhere else along a ray can be reconstructed by using several reconstruction models for the data. Since a parallel projection matrix is used all intersection position of the rays and the grid may be represented by shifting the* column *template (yellow highlighted), which stores the pre-computed positions.*

of the volume data, other questions arise immediately. Is that model accurate enough? Does it represent the data appropriately? What about different reconstruction models, i.e. can we use piecewise quadratic or cubic spline models as well? Can they be easily introduced into the shear-warp method? What about a more accurate approximation of the volume rendering integral? Then, another goal is to remove the threefold redundancy of run-length encoded data sets used in the shear-warp approach, because more and more data sets obtained from imaging systems like CT, MRI, etc. grow to several gigabytes. This can be achieved only by developing some new data structures as well.
On that score in our new shear-warp approach (cf. left drawing in Fig. 2.1) we follow a somewhat different technique. Instead of shearing the slices we let the rays vary up to 45° according to the main viewing direction[1], both is determined from the model matrix $\mathbf{M}$ of Equ. (1.2). If the angle between the current ray direction and the current main viewing direction exceeds 45° then the shear-warp factorization automatically accounts for this and selects another main viewing axis, such that the current ray direction will always stay below this threshold value. Thus, only the processing order of the data set is different. First of all, the intersection positions (blue dots in Fig. 2.1) of the rays and the local grid planes are pre-computed and we further allow different data reconstruction models to obtain data values between the grid points (i.e. piecewise linear-, quadratic-, and cubic models defined on *type-0* and *type-6* partitions of the volume). This setting

[1] A main viewing direction is always the direction parallel to one of the three main viewing axis.

permits us to easily reconstruct values at these intersection positions and anywhere else along the rays (but of course not only on the rays). Then, on each interval (i.e. between two consecutive intersection positions) along different rays $\mathbf{r}_{\nu,\mu}$[2] we approximate the volume rendering integral, compute the iso-surface or the maximum intensity projection and composite the result in front-to-back order into the appropriate pixels ν, μ of the intermediate image. We can observe from the right drawing of Fig. 2.1, that only one such called *column* template is necessary to represent the intersection positions of all rays with the local grid planes. During rendering this template is shifted through the volume grid to quickly find the intersection positions and the other information necessary for visualization of the volume data.

2.1.2 Column Template

Type-0 Partitions

The *column* template allows combining ray-casting with the shear-warp algorithm. Let us now have a look at the details on how to create such a template. The data structure developed in this section is only applicable for parallel rays, i.e. if the projection matrix in equation (1.2) becomes a parallel projection matrix ($\mathbf{P} := \mathbf{P}_{\mathbf{parallel}}$). However, in this case it suffices to store only a single such data structure for the whole volume, thus it can be reused for the other columns in the volume by shifting it with the grid constant. The information stored in this template is dependent on the main viewing direction, further we apply this information during rendering in different sequence (which depends on the viewing direction as well) and the memory consumption is negligible (i.e. the data structure takes less than 200 kilobytes of memory for volumes of size N^3). Further, this new data structure describes a bijective mapping between object space (voxels) and intermediate image space (pixels) (see also [YK92]). For further discussion of the details we assume, without restriction of the general case, that the main viewing axis becomes the x axis.
In section 2 we have been confronted with the unit cube limited by six planes (2.3). Similarly, a concatenation of several such unit cubes results in an unit column, which is defined by six planes as well. Thus, a unit column of a rectangular domain Ω (volume) can be bounded by the planes

$$
\begin{aligned}
P_6^C(x,y,z,d) &= P_6^Q(x,y,z,d) - D, && (2.1)\\
P_7^C(x,y,z,d) &= P_7^Q(x,y,z,d), && (2.2)\\
P_8^C(x,y,z,d) &= P_8^Q(x,y,z,d), && (2.3)\\
P_9^C(x,y,z,d) &= P_9^Q(x,y,z,d), && (2.4)\\
P_{10}^C(x,y,z,d) &= P_{10}^Q(x,y,z,d), && (2.5)\\
P_{11}^C(x,y,z,d) &= P_{11}^Q(x,y,z,d), && (2.6)
\end{aligned}
$$

where $D := L - 2d$ with L the size of the rectangular domain (i.e. the volume data set) in x direction and $d = 0.5$. Thus, a *column* template represents all information affecting such an unit column, and can be split into a more local description by considering only

[2] Here the rays in object space $\mathbf{r}_{\nu,\mu}$ are identified with the corresponding intermediate image pixels ν, μ and not as in ray-casting with the pixels of the final image.

a *cube* template that is associated with an unit cube Q as defined in Sec. 2 (cf. Fig. 2.2). From a local point of view the *cube* template contains all necessary information – except the data values (or spline coefficients), which are stored in the run-length encoded volume – to reconstruct data along the rays $\mathbf{r}_{\nu,\mu}(t) = \mathbf{r}_s^{\nu,\mu} + t\ \mathbf{r}_d$ coming from intermediate image pixels ν, μ and intersecting the current considered unit cube. Now, each *cube* template of

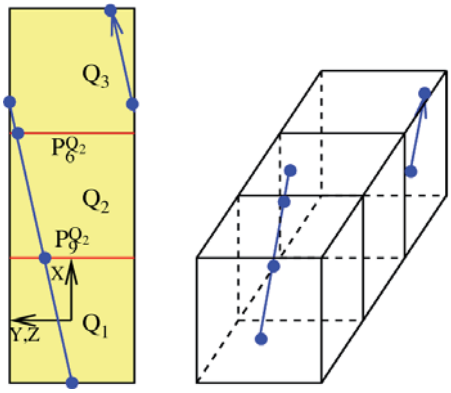


Figure 2.2: *A* column *template usable with type-0 partitions of the volumetric domain and affine projection matrices only. Left: The* column *template as in Fig. 2.1 with yellow highlighted unit cubes Q_i, $i = 1, 2, 3$. The bottom and top red lines denote the planes $P_9^{Q_2}$ and $P_6^{Q_2}$, respectively, where the other planes follow directly from Equ. (2.3). The local coordinate system of the unit column C is located at the center of the first unit cube Q_1, where the local systems of the cubes Q_i are placed at their center positions as well. Right: The three-dimensional analog. The intersection positions of the ray segments with the different planes of the unit cubes are marked as solid blue dots.*

the *column* template stores the following information. First, the number of ray segments passing through that unit cube, i.e. there are at most three ray segments (in the parallel projection case). This results from the shear-warp factorization and the affine viewing transformation matrix. The at most three ray segments passing through such a unit cube are associated with three different intermediate image pixels by using relative offset values. Second, a number (an identifier) for the first ray segment passing through the current considered cube, i.e. each ray segment passing through the unit column has a unique identifier and all ray segments are sorted according to the x coordinates of it's intersection positions with the unit column (the sorting comes for free using a parallel projection matrix). Finally, for each ray segment, the intersection positions with the unit cube are stored, i.e. with the planes (2.3) affected by a ray segment as well as the plane numbers itself.
The generation of this template is straight forward and is done in an recursive way using modulo like operations for the intersection computation of a ray and the unit column. For this, we first need a local ray start and direction according to the unit column to compute all the intersections with the different planes. Just before rendering, we determine the current ray direction $\mathbf{r}_d$ in object space according to the main viewing direction (or axis) determined from the shear-warp factorization, i.e. for the current total transformation matrix $\mathbf{T}$. The ray direction remains the same for all of the different rays coming from the different intermediate image pixels ν, μ (because of an orthographic transformation). The ray start $\mathbf{r}_s^{\nu,\mu}$ in object space depends on the position of the intermediate image

pixel ν, μ. Since the intermediate image and the slice located at position $x = 0$ coincide, the ray start can be easily computed using the shear-warp factorization. This object space ray start position is used to determine the column, thus further a unit column can be considered only. Then, the global ray start position $\mathbf{r}_s^{\nu,\mu}$ has to be transformed into the local coordinate system of the unit column (both procedures are very similar to those to find a unit cube and the local coordinates in the volumetric cubic partition $\diamondsuit$, which has been described in Sec. 2). This local ray start position $\mathbf{r}_s^C$ according to a unit column C remains constant independent of the intermediate image pixel ν, μ. Since both parameters (ray start and direction) needed for the generation of this *column* template are constant, only one such data structure have to be setup. During rendering for each intermediate image pixel ν, μ the column has to be determined only (i.e. the data – or spline coefficients), whereas the information from the *column* template associated with the unit column can be reused.

Once we have the local ray start $\mathbf{r}_s^C$ and direction according to the unit column C we proceed as follows – note, this procedure has to be applied once per view only. We consider the first unit cube Q_1 in the unit column, then, by definition (and by assumption that the main viewing axis is the x axis), the first local ray start is located in the entry plane $P_9^{Q_1}$ of Q_1 as well as in the plane P_9^C of C, i.e. $\mathbf{r}_s^{Q_1} := \mathbf{r}_s^C$ (similar to Fig. 2.6 where no restrictions are made regarding the projection and main viewing axis). This defines the first intersection position of the first ray segment R_1 with the current unit cube Q_1 as well as an intersection with the unit column C. This information is stored in the current *cube* template associated with Q_1. Next, dependent on the ray direction (and the main viewing axis, which is fixed here) we know which three of the remaining five planes $P_6^Q, P_7^Q, P_8^Q, P_{10}^Q$, and P_{11}^Q have to be intersected[3] with ray segment R_1 to find the next (closest) intersection position with Q_1 (this has been discussed in Sec. 2.6 as well). Considering the example of Fig. 2.2 the next intersected plane by the ray segment R_1 is $P_6^{Q_1}$. This intersection plane as well as the exit position $\mathbf{r}_e^{Q_1}$ are stored in the current *cube* template. Further, we save relative offsets associated with the ray segments going through a unit cube. These are used to determine the intermediate image pixel position during rendering where the ray segment's contribution has to be stored. In other words, the offsets of the first ray segment R_1 are always zero, because its (ray) start position is located in the intermediate image thus at the global position $x = 0$ (according to the unit column at $\mathbf{r}_s^C = (x_e^C, y_e^C, z_e^C) = (-0.5, y_e^C, z_e^C)$). Now, when the ray segment R_1 leaves the current considered unit cube Q_1 (as in Fig. 2.2), the next *cube* template associated with its corresponding unit cube Q_2 has to be taken into consideration. If this unit cube is not part of the unit column C, processing is stopped, otherwise we proceed as follows. Since R_1 leaves Q_1 in plane $P_6^{Q_1}$ the exit position is of type $\mathbf{r}_e^{Q_1} = (x_e^{Q_1}, y_e^{Q_1}, z_e^{Q_1}) = (+0.5, y_e^{Q_1}, z_e^{Q_1})$ and is used to determine the entry location of R_1 in the next unit cube Q_2 as $\mathbf{r}_s^{Q_2} = \mathbf{r}_e^{Q_1} + (-1, 0, 0)$. According to the exit plane $P_6^{Q_1}$ of R_1 in Q_1 the entry plane of R_1 in Q_2 becomes $P_9^{Q_2}$ now. According to our example from Fig. 2.2 we have the same situation in the unit cube Q_2. Therefore we proceed here as discussed above. At the outset, the situation in Q_3 is once more the same, i.e. as before the local (ray) start position of R_1 according to Q_3 becomes $\mathbf{r}_s^{Q_3} = (-0.5, y_e^{Q_3}, z_e^{Q_3})$ with the corresponding entry plane $P_9^{Q_3}$. We compute again the intersections of R_1 and the remaining planes and find this time the nearest

[3]This can be pre-computed since the ray direction is constant for each pixel per view.

exit intersection, for example $\mathbf{r}_e^{Q_3} = (x_e^{Q_3}, +0.5, z_e^{Q_3})$ (or $\mathbf{r}_e^{Q_3} = (x_e^{Q_3}, y_e^{Q_3}, +0.5)$) with the appropriate exit plane $P_7^{Q_3}$ (or $P_8^{Q_3}$). Since R_1 does not leave the current considered unit cube in plane P_6, a new ray segment R_2 has to be considered now. The consequence is that there are two ray segments (R_1 and R_2) going through this cube Q_3. The relative offsets of these ray segments used to determine the corresponding intermediate image pixels during rendering are still zero for R_1, whereas for second ray segment R_2 these offsets become, for example, $(1, 0)$ (or $(0, 1)$). After that, again modulo like operations are applied. Thus, proceeding in that way the entry position of R_2 oriented towards Q_3 becomes $\mathbf{r}_s^{Q_3} = \mathbf{r}_e^{Q_3} + (0, -1, 0)$ (or $\mathbf{r}_s^{Q_3} = \mathbf{r}_e^{Q_3} + (0, 0, -1)$) with entry plane $P_{10}^{Q_3}$ (or $P_{11}^{Q_3}$). Finally, the last time the intersection of R_2 with Q_3 is computed, i.e. the ray segment's exit intersection $\mathbf{r}_e^{Q_3} = (+0.5, y_e^{Q_3}, z_e^{Q_3})$ with the plane $P_6^{Q_3}$ is determined. The next unit cube Q_4 to consider is not part of the unit column C, thus the pre-computation is stopped here.

The *column* template can be generated by the following algorithm.

Algorithm 2.1.1 (Generate Column Template Type-0)**.** *The input to this algorithm are the ray start* $\mathbf{r}_s^C$*, the ray direction* $\mathbf{r}_d$*, and the first intersected plane* P_i^C *according to a unit column* C *as well as the maximal size of the volume data set, i.e.* $D := \max(L, M, N)$.

1: {*Check ray start, direction and first intersection plane.*}
2: *check*();
3: {*Initialize entry, exit plane flags and the entry intersection information.*}
4: *initialize*();
5: $j = k = 1$;
6: ***while*** $k \leq D$ ***do***
7: {*Compute adjacent intersection information, i.e. the exit intersection of the current ray segment* R_j *according to the current unit cube* Q_k.}
8: $f = next\_intersection()$;
9: ***if*** $f ==$ *plane_exit* ***then***
10: {*Ray segment* R_j *exits the unit cube* Q_k*. Save exit intersection information for this cube* Q_k *and prepare entry intersection information for next cube* Q_{k+1}.}
11: *save_intersection*();
12: *prepare_next_cube*();
13: $k = k + 1$;
14: ***else***
15: {*Ray segment* R_j *does not exit the unit cube* Q_k*. Save exit intersection information for this cube* Q_k *and prepare entry intersection information for next ray segment* R_{j+1} *in this cube* Q_k.}
16: *save_intersection*();
17: *prepare_next_rseg*();
18: $j = j + 1$;
19: ***end if***
20: ***end while***

Type-6 Partitions

The *column* template for this partition type is quite similar to that specified in the last section (i.e. the differences are due to the partition). Thus, similarly to Sec. 3 we have to consider the tetrahedral partition of the unit column C as well as the unit cube Q. More precisely, the differences are: First, additional six plane equations (3.1) have to be taken into account for the intermediate intersection computations, which results totally in more intersection positions defined by the ray segments and the different cubes (see Fig. 2.3). However, for this the data structure itself has not to be changed, the pre-computation is done in the recursive manner by using the modulo like operations as above, and thus the treatment of the cubes remains the same. Hence, considering the first unit cube Q_1 as well as the ray segment R_1 in Fig. 2.3 a straight forward implementation could be as follows. Given the enter plane and the entry intersection position (at $t_0 := 0$) of R_1 according to Q_1, one could first compute the next, nearest exit intersection (e.g. at $t_1 = d$) by using R_1, the six planes (2.3) and equation (2.14) as already discussed above. Then the other six planes (3.1) could be used together with R_1 and Equ. (2.14) as well to find another six intermediate intersections (e.g. at $s_i,\ i = 1, \ldots, 6$ marked as dark green dots in Fig. 2.3). The next step would be to discard all s_i with $s_i < t_0$ or $s_i > t_1$ since they are not in the currently considered interval. The remaining intermediate intersections have to be sorted according their parameter s_i – of course only if $i > 1$ – to satisfy the order of the subintervals (i.e. the compositing order or iso-surface computations during rendering). Finally, the positions corresponding to

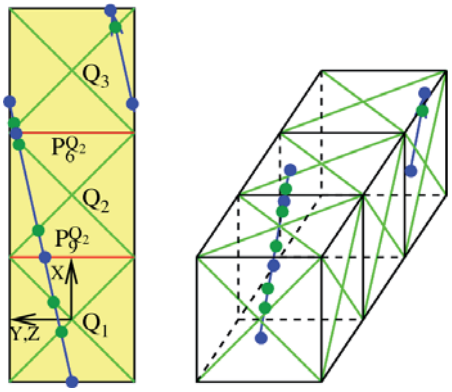


Figure 2.3: *A* column *template usable with type-6 partitions of the volumetric domain and affine projection matrices only. The same configuration as in Fig. 2.2. For the intermediate intersection computations another six different plane equations (a part of those are marked as green lines) have to be considered (cf. Equ. (3.1)). Thus, in this* column *template additionally the positions (dark green dots) and the corresponding planes have to be stored as well.*

the remaining s_i as well as the appropriate (intersected) planes have to be stored in the *column* template. However, this implementation is not very elegant. First, we have to compute for each ray segment going through a unit cube the intersections with all nine (or eleven, ignoring the dependency of the ray direction) planes no matter if this segment intersects only two of them (excluding the entry intersection or plane). Second, we have introduced a sorting step. Hence, our implementation here to determine the intermediate

positions is a bit different, but the same as the method presented in Sec. 3.7. The main difference here is that we apply modulo like computations as used for the generation of the type-0 *column* template (discussed in the previous section).
Hence, we do not discuss the procedure again, instead we give a similar algorithm compared to Alg. 2.1.1, which now generates the *column* template for type-6 tetrahedral partitions.

Algorithm 2.1.2 (Generate Column Template Type-6)**.** *The input to this algorithm are the ray start* $\mathbf{r}_s^C$*, the ray direction* $\mathbf{r}_d$*, and the first intersected plane* P_i^C *according to a unit column* C *as well as the maximal size of the volume data set, i.e.* $D := \max(L, M, N)$*.*

1: {*Check ray start, direction and first intersection plane.*}
2: $check()$;
3: {*Initialize entry, exit plane flags and the entry intersection information.*}
4: $initialize()$;
5: $j = k = 1$;
6: ***while*** $k \leq D$ ***do***
7: {*Find first tetrahedron* T_1 *in current cube* Q_k*.*}
8: $find\_first\_tetrahedron()$;
9: $l = 1$;
10: {*Compute adjacent intersection information, i.e. the exit intersection of the current ray segment* R_j *according to the current unit cube* Q_k *and the current tetrahedron* T_l*.*}
11: $f = next\_intersection()$;
12: ***while*** $f == plane\_internal$ ***do***
13: {*Save intersection information and find next tetrahedron* T_{l+1} *using current exit plane* $P_\star^{T_l}$*, the current tetrahedron* T_l*, and the appropriate lookup table from Sec. 3.1, finally compute modulo intersection information according to the next tetrahedron* T_{l+1}*.*}
14: $save\_intersection()$;
15: $find\_next\_tetrahedron()$;
16: $l = l + 1$;
17: $compute\_modulo\_informations()$;
18: {*Compute adjacent intersection information, i.e. the exit intersection of the current ray segment* R_j *according to the current unit cube* Q_k *and the current tetrahedron* T_l*.*}
19: $f = next\_intersection()$;
20: ***end while***
21: **assert**$(l \leq 7)$;
22: ***if*** $f == plane\_exit$ ***then***
23: {*Ray segment* R_j *exits the unit cube* Q_k*. Save exit intersection information for this cube* Q_k *and prepare entry intersection information for next cube* Q_{k+1}*.*}
24: $save\_intersection()$;
25: $prepare\_next\_cube()$;
26: $k = k + 1$;
27: ***else***
28: {*Ray segment* R_j *does not exit the unit cube* Q_k*. Save exit intersection infor-*

mation for this cube Q_k and prepare entry intersection information for next ray segment R_{j+1} in this cube Q_k.}

29: *save_intersection*();
30: *prepare_next_rseg*();
31: $j = j + 1$;
32: ***end if***
33: ***end while***

2.1.3 Algorithm and Acceleration Techniques

The computation as well as the usage of the *column* template is dependent on the principle viewing directions. Thus, we not only have to process the data set (visit the voxels) mainly in three different ways (in general their are four cases for each face of the data set, i.e. in total we have 24 different processing orders of the original data set here, because this processing dependents on the sign of the ray direction itself as well), we also have to adapt the generation of the template (i.e. the planes intersected with the different ray segments – see previous section – have to be permuted according to the permutation matrix in Sec. 1 and the signs of the principle viewing directions). This template is then applied during rendering in two different manners as well. This comes from the fact, that we use one run-length encoded data set for all viewing directions only (see Sec. 1.5). Hence, the acceleration techniques have also to be adapted to some extent. However, space leaping remains more or less unchanged, no matter of the principle viewing direction. Whereas, early-ray termination has to be applied in the right way, i.e. in accordance to the usage of the *column* template (or adequately to the principle viewing directions). This is discussed next.

The Principle y, z Directions

For both main viewing directions we apply the run-length encoded Z data set only. However, the run-length encoded data scan-lines as well as the dynamically run-length encoded intermediate image scan-lines are in both cases parallel to each other (cf. Sec. 1.5, Fig. 1.5, and Fig. 1.6). The stacking order of the slices (the sign of the principle viewing direction), i.e. whether we are looking down the positive or the negative direction of the corresponding axis, determines the processing order of the pointer array (cf. Fig. 1.4, Fig. 1.5, and Fig. 1.6), i.e. which scan-lines or slices are processed first. Whereas from the global ray direction (each ray corresponding to an intermediate image pixel has the same direction) we determine the order on how to go through the intermediate image, i.e. from top-left to bottom-right, from top-right to bottom-left or vice versa, and how to visit the data values in the run-length encoded voxel scan-lines, i.e. from left to right or vice versa, respectively. This is necessary to satisfy the compositing order. The rendering algorithm can be described as follows. Just before rendering – as in original method – we determine the main axis (in this case we consider the y or z-axis only) and the stacking order of the slices. Then the global (constant) ray direction and the local (constant) ray start according to the unit column are determined from the shear-warp factorization. The *column* template is pre-computed considering the permutation matrix respectively the appropriate plane equations. During rendering all slices are processed scan-line by scan-line in a front to back manner from left to right, top to down or vice versa to satisfy

compositing order (see above). This is dependent on the global (constant) ray direction. Then, for all unit cubes[4] of the first slice we use the pre-computed information from the first *cube* template of the *column* template, for all unit cubes of the second slice the information from the second *cube* template of *column* is used and so on, i.e. the pre-computed intersection and projection information are reused from the template and remain the same during the processing of a slice (in the original method the re-sampling weights for the different voxels within a slice are the same). However, space leaping of the unit cubes comes for free by using the run-length encoded data structure, i.e. we skip the appropriate data values (or spline coefficients) within scan-lines of a slice. In other words, we simply skip the processing of all cubes within a scan-line of a slice, where the ray segments going through the considered unit cubes do not contribute to the appropriate intermediate image pixels because the corresponding data values (taken from the run-length encoded data set) are empty. For early-ray termination we apply dynamic run-length encoding of the intermediate image scan-lines, where at most three, two or only one scan-line(s) have to be considered at a time. This results from at most three ray segments going through a unit cube, which affect at most three intermediate image pixel in two different scan-lines (cf. Fig. 2.4).

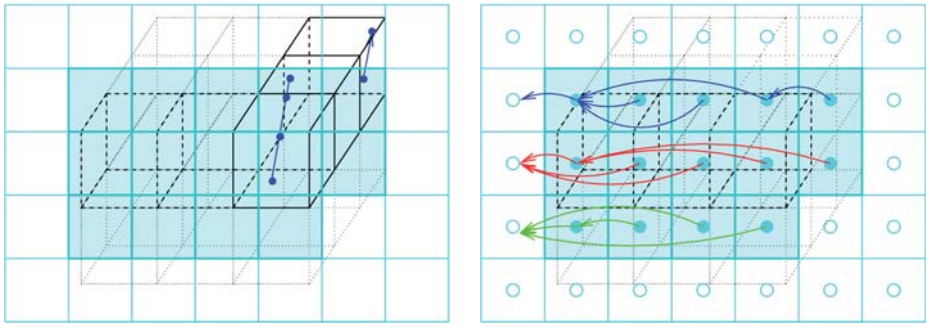


Figure 2.4: *Left: The projection of a (black dotted) voxel scan-line onto at most three (marked as cyan squares) intermediate image scan-lines. The pre-computed (solid black) unit column can be considered perpendicular to the voxel scan-line, where the information of the unit cubes corresponds to the appropriate slices. Right: The same projection, where at most three dynamically run-length encoded intermediate image scan-lines have to be considered for early-ray termination. Each transparent (cyan circle) or opaque (cyan dot) intermediate image pixel is located at the center of the unit square. The intermediate image domain is subdivided by unit squares into a uniform planar partition.*

The general rendering algorithm for these principle viewing directions can be written as follows.

Algorithm 2.1.3 (Render Orthographic Y,Z Axis)**.** *The input to the shear-warp algorithm are the run-length encoded data set, the viewing specification, and the image buffers where to store the result.*

1: {*Compute factorization, initialize variables and intermediate image, ...*}
2: *initialization*();

[4]The data values are located in the center or at the corners of the unit cubes, dependent on the reconstruction method used. But they are stored in the run-length encoded data set.

```
3: {Generate column template using Alg. 2.1.1 or 2.1.2}
4: column_template_generate();
5: {For each slice of the volume in front-to-back manner.}
6: for k = 0 < k_max do
7:    {Read the appropriate and necessary intersection information from column template.}
8:    cinfo = column_template_read_cube_info(k);
9:    {For each scan-line(s) of slice k in appropriate order.}
10:   for i = 0 < i_max do
11:      {Get the appropriate voxel and image scan-line(s).}
12:      get_scanlines(i, k)
13:      {Until not the end of scan-line(s).}
14:      while j < j_max do
15:         {Space leaping of empty voxels and the corresponding pixels (in a parallel fashion).}
16:         j = space_leaping(j);
17:         {Early-ray termination, skipping of opaque pixels (in a parallel fashion) and the the corresponding voxels in the voxel scan-lines.}
18:         j = early_ray_termination(j);
19:         {Compute the run of non-empty and non-occluded voxels.}
20:         j1 = get_run(j);
21:         {Compositing of the run into the appropriate intermediate image pixels of the non-empty and non-occluded voxels by first reading data from the run-length encoded volume (e.g. spline coefficients) and reusing the information from the column template.}
22:         for jh = j < j1 do
23:            dinfo = get_data(i, jh, k);
24:            pinfo = get_pixel(i, jh, k, cinfo);
25:            compositing(pinfo, dinfo, cinfo);
26:         end for
27:      end while
28:   end for
29: end for
30: warp_intermediate_image();
```

The Principle x Direction

For this principle viewing direction we also relate to the run-length encoded Z data set. Hence, the run-length encoded data scan-lines and the dynamically run-length encoded intermediate image scan-lines are not parallel anymore to each other (cf. Sec. 1.5 and Fig. 1.6). However, now the sign of the principle viewing direction (the stacking order of the slices), i.e. whether we are looking down the positive or the negative direction of the axis, determines the processing order of the run-length encoded voxel scan-lines, i.e. whether they have to be processed from the front to the back or vice versa, respectively[5]. Whereas, now the global ray direction gives us the order how to

[5]This is different compared to the other two principle viewing axes.

go through the intermediate image, i.e. from top-left to bottom-right, from top-right to bottom-left or vice versa[6], and in which order the pointers from the pointer array (cf. Fig. 1.4 and Fig. 1.6) have to be used[7]. This is necessary to satisfy the compositing order. The rendering algorithm can be described as follows. Before rendering – as in the last section – we determine the main axis (in this case we consider the x-axis only), the sign of this main viewing direction or axis, the global (constant) ray direction and the local (constant) ray start according to the unit column from the shear-warp factorization. This allows processing the volume data as well as running through the intermediate image in an appropriate manner. The *column* template is pre-computed without having to consider the permutation, thus we can apply the method from Sec. 2.1.2 directly. During rendering we process the pointers in the pointer array, i.e the run-length encoded voxel scan-lines, as well as the intermediate image from top-left to bottom-right, top-right to bottom-left, or vice versa dependent on the global ray direction. Since now all voxel scan-lines are processed perpendicularly to the intermediate image and all the ray segments going through a unit column (which is reused for each voxel scan-line by shifting it's intersection and projection information) project onto an oblique line in the intermediate image, early-ray termination have to be adapted accordingly (see Fig. 2.5). The consequence is that the dynamically run-length encoded scan-lines of the intermediate image are now encoded in an oblique fashion, whereas space leaping has not to be changed.

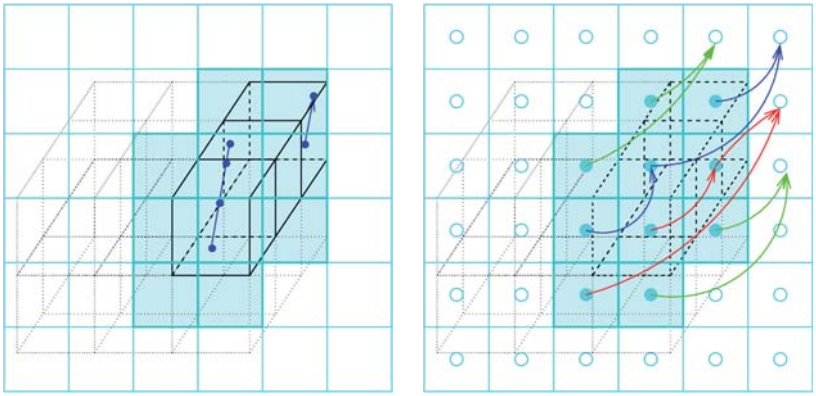


Figure 2.5: *Left: The projection of a (solid black) voxel scan-line onto at most four (marked as cyan squares) intermediate image scan-lines. The pre-computed (solid black) unit column can be considered perpendicular to the intermediate image and parallel to the voxel scan-line, where the information of the unit cubes corresponds to the appropriate voxel data. Right: The same projection, where at most four oblique dynamically run-length encoded intermediate image scan-lines have to be considered for early-ray termination.*

The general rendering algorithm for these principle viewing directions can be written as follows.

Algorithm 2.1.4 (Render Orthographic X Axis)**.** *The input to the shear-warp algorithm are as before the run-length encoded data set, the viewing specification, and the image buffers where to store the result.*

[6]Similar to the other principle viewing axes.
[7]This is different compared to the other two principle viewing axes as well.

```
1: {Compute factorization, initialize variables and intermediate image, ... .}
2: initialization();
3: {Generate column template using Alg. 2.1.1 or 2.1.2}
4: column_template_generate();
5: {For each pixel of the intermediate image in appropriate order.}
6: for i = 0 < i_max do
7:    for j = 0 < j_max do
8:       {Get the appropriate voxel scan-line(s).}
9:       get_scanlines(i, j)
10:      {Until not the end of scan-line(s).}
11:      while k < k_max do
12:         {Space leaping of empty voxels and the corresponding pixels (in an oblique
            fashion).}
13:         k = space_leaping(k);
14:         {Early-ray termination, skipping of opaque pixels (in an oblique fashion) and
            the the corresponding voxels in the voxel scan-line(s).}
15:         k = early_ray_termination_oblique(k);
16:         {Compute the run of non-empty and non-occluded voxels.}
17:         k1 = get_run(k);
18:         {Compositing of the run into the appropriate intermediate image pixels of the
            non-empty and non-occluded voxels by first reading data from the run-length
            encoded volume (e.g. spline coefficients) and the information from the column
            template.}
19:         for kh = k < k1 do
20:            dinfo = get_data(i, j, kh);
21:            cinfo = column_template_read_cube_info(kh);
22:            pinfo = get_pixel(i, j, kh, cinfo);
23:            compositing(pinfo, dinfo, cinfo);
24:         end for
25:      end while
26:   end for
27: end for
28: warp_intermediate_image();
```

2.2 Perspective Projection Case

2.2.1 Basic Idea

In this case we also alternate the original perspective projection shear-warp method. Slices are sheared as well as scaled such that the rays coming from the intermediate image plane can be considered to be perpendicular to that plane (cf. left picture of Fig. 2.6). In our new method once more we do not modify the slices but vary the rays according to the main viewing direction, i.e. rays still have their source in the intermediate image plane (not in the image plane) but are not perpendicular to that anymore. The angle between a ray and the main viewing direction is not limited to 45° anymore, also it varies from ray to ray (i.e. it is not constant per view as before). This

is due to the perspective factorization (cf. Sec. 1.4), i.e. when the location of the eye is close to the object or better close to the projection plane, the rays are going to diverge more, thus many rays constitute a higher angle with the main viewing direction and the variance of that angle increases as well. Hence, on one side there could arise some volume cubes, which are intersected by more than three rays, thus the appropriate *cube* template would contain more than three ray segments as well. We have to consider this in our *column* template (the data structure). On the other side, there are unit cubes of the volume which are not intersected by any ray at all, i.e. some *cube* templates in a *column* could be empty, which has to be noted as well. Next, if the location of the eye is far away from the object or the projection plane, then the different rays become more and more parallel to each other, i.e. the angle between a ray and the main viewing direction stays more likely within the 45° limit as well as its variance will decrease. When the variance of the angle is close to zero, we can consider the parallel projection case. However, that is why we can not directly pre-compute all the intersections of the different rays arising from the intermediate image plane with all unit columns of a volume, i.e. coherence between the different rays is unlikely. Nevertheless, the *column* template (see Sec. 2.1.2) – the data structure itself – can be reused in a way that the changes to the rendering procedures afterwards are minimized. In the following the differences according to the parallel projection case are discussed only.

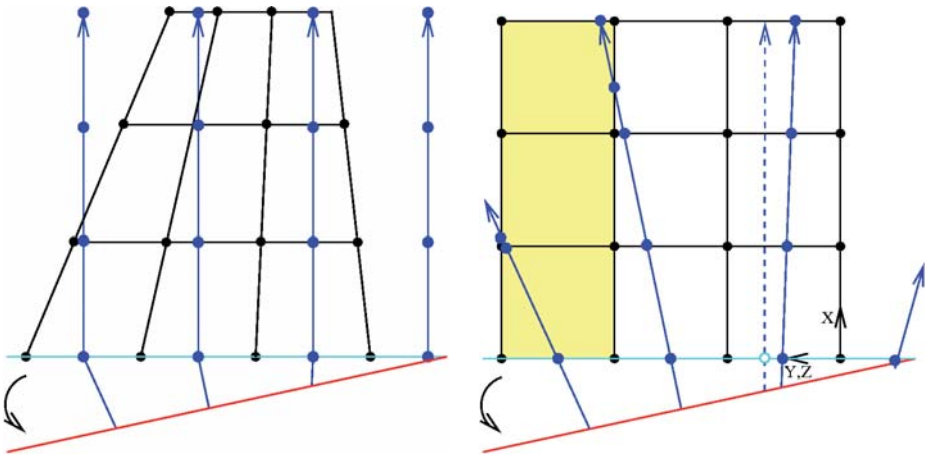


Figure 2.6: *A simplified two-dimensional view of the new shear-warp factorization for a perspective projection. As before the red lines denote the final image, whereas the cyan lines denote the intermediate image. The black grid is associated with the volume and the cross points (black dots) are the data values. Left: The original shear-warp algorithm. Rays are considered perpendicular to the sheared and now also scaled slices and the intermediate image. At the intersection positions of the rays and the slices (blue dots) values are bi-linearly interpolated (reconstructed). Right: In the new shear-warp technique rays may vary according to the main viewing axis. The intersections positions of the rays and the grid planes (blue dots) are pre-computed, the values at these positions and anywhere else along a ray can be reconstructed by using several reconstruction models for the data. Since a perspective projection matrix is used here the* column *template (yellow highlighted) has to be recomputed for each column of the volume.*

2.2.2 Algorithm and Acceleration Techniques

The generation and the application of the *column* or *cube* template is dependent on the main viewing direction (or axis) as before. The processing order of the original data set and the intermediate image have to be taken into account as well, both depend now on the so called *divergence point* (cyan circle in right picture of Fig. 2.6). The location of this point could be any position in the intermediate image plane[8], but is not necessarily inside the intermediate image domain.
However, the point is computed by first transforming the eye position into object space. Then, a line or ray (dotted blue ray in right picture of Fig. 2.6), defined by this object space eye position and the normal according to the intermediate image plane, is used to intersect the intermediate plane and thus to find the *divergence point*. Hence, the ray direction corresponding to the *divergence point* is perpendicular to the intermediate image plane as well and the four adjacent but discrete intermediate image pixel positions define the start pixels in the intermediate image plane. That means, all the pixels (as well as the corresponding columns of the volume) are processed from these start pixels to outer pixels of the intermediate image. In other words, after the *divergence point* is found, the intermediate image is subdivided into at most four sub-regions, where each sub-region is processed independently[9]. The number of sub-regions, which can be four, two or one, is depended on the position of the clipped *divergence point*, i.e. whether it is located inside the image domain, on an edge or a corner of the image, respectively. If it is outside of the image domain clipping is applied to find the start pixels, which then are always located on an edge or a corner of the intermediate image. The processing of the image is started from that start pixels or pixels near to the *divergence point*, i.e. where the corresponding ray directions form a smaller angle to the normal of the intermediate image, and is proceeded to the far pixels, i.e. where the corresponding ray directions form a bigger angle to that normal. This is necessary to satisfy the compositing order, i.e. ray intervals (along rays or ray segments) which are nearer to the eye location have to be precessed and rendered first.
However, it is not possible to reuse the intersection information stored in this *column* or *cube* template for neighboring columns or cubes within the volume, since there is no such coherence as for the orthographic projection between neighboring rays. Hence, the intersection information is recomputed for each column or cube, whereas the data structure itself is reused to minimize changes to the rendering procedure or algorithm. Note that the permutation matrix as well as the signs of the principle viewing direction have to be considered here as well to account for correct intersection computations. The run-length encoded Z data set is considered for reconstruction only, no matter which principle axis will be considered.

The Principle y, z Directions

The encoded data scan-lines as well as the intermediate image scan-lines are in both cases parallel to each other (cf. Sec. 1.5, Fig. 1.5, and Fig. 1.6), as previously. The stacking order of the slices (the sign of the principle viewing direction) determines the

[8] The intermediate image plane coincides with a face of the volume as well as with the first slice of the volume.
[9] The sub-regions can be processed in parallel.

processing order of the pointer array (cf. Fig. 1.4, Fig. 1.5, and Fig. 1.6), i.e. which scan-lines or slices are processed first to account for correct compositing. Space leaping and early-ray termination is very similar to the parallel projection case. However, one difference is in the rate of traversal of the image and volume scan-lines (or columns). This results from the non-affine (perspective) projection matrix and involves a scaling factor. The algorithm is as follows. Just before rendering we determine – as before – the main axis and the stacking order of the slices. Then the *divergence point* is computed and the processing of the volume cubes as well as the intermediate image pixels starts at this point, i.e. at the nearby pixels, as discussed above. Space leaping is maintained, whereas early-ray termination is very similar to the parallel projection case, where we once more apply the dynamically encoded intermediate image scan-lines. However, since we do not have a pre-computed *cube* template, we don't know if and how many rays will intersect the considered volume cube, we simply project all eight vertices of that cube into the intermediate image and determine the convex hull. All intermediate image pixels inside this convex hull are taken into account to determine the skipping offset for early-ray termination. We skip only the minimum number of pixels possible (if more than one pixel scan-line have to be considered) and synchronize between the image and volume scan-lines because of the afore mentioned scaling factor. Once we have found a non empty volume cube where early-ray termination will not apply as well, the eight corner vertices are projected onto the intermediate image as well. Now, all intermediate image pixels as well as the corresponding ray information are determined located in the convex hull of these eight projected vertices. Then, we compute the intersection information of those rays corresponding to the appropriate pixels with the considered volume cube and store this information in our *cube* template. This together with the values (i.e. the spline coefficients) from the run-length encoded data set are used to evaluate the appropriate splines (i.e. to reconstruct the data along the rays) as well as the volume rendering integral (or the iso-surface).

The Principle x Direction

For this principle viewing direction – as in parallel projection case – we also relate to the run-length encoded Z data set. Hence, the volume scan-lines and the intermediate image scan-lines once more are not parallel to each other (cf. Sec. 1.5 and Fig. 1.6). The processing order of the data set as well as the intermediate image is dependent on the same parameters as discussed in the previous sections. However, a pre-computation of intersection information, which is stored in the *column* template and afterwards reused during rendering, is not possible as well – as for the other two directions. Nevertheless, we reuse the data structure for the reasons described above. Basically, two approaches are discussed here. The first one is very similar to the algorithm discussed in the previous section, whereas the second is a propagation scheme.
We are always interested to process the data in object order to take advantage of the underlying computer architecture, i.e. to maximize cache performance. Hence, the first algorithm starts at the *divergence point* as before, but processes each column of the volume which are now considered perpendicularly to the intermediate image. In contrast to the method for the y and z directions, we project the whole column of a volume into the intermediate image space to determine the convex hull and the intermediate image pixel affected by this projected column. Then, we determine the corresponding

rays and compute the intersection information with that column, but only for rays with corresponding non opaque intermediate image pixels. The information is further transformed into the unit column and saved into the *column* template, which is used during rendering as in the parallel projection case.
The other approach is to propagate the necessary intersection information from column to column, where once an intermediate image pixel has become opaque the appropriate ray is terminated and the corresponding intersection information is not propagated to adjacent columns. Thus, we process each of the at most four sub-regions of the intermediate image independently. We start in a sub-region with that intermediate image pixel as well as the corresponding volume column near to the *divergence point* as well. Here, we do not project the column into the intermediate plane, since for the first column and the corresponding pixel we also know the first ray parameters, which directly allows us to compute the intersection information with the first volume column. We know that no other rays arising from the nearby discrete intermediate image pixels exist which also intersect the same current considered column. However, considering adjacent pixels or volume columns in a way discussed above, i.e. such that the compositing order is satisfied, the dynamically run-length encoded intermediate image scan-lines used during early-ray termination have to be encoded in an oblique fashion, where all scan-lines together look like a star structure on the intermediate image. In the orthographic projection case the scan-lines are encoded in an oblique fashion as well, but since the ray direction is constant for each ray coming from the appropriate intermediate image pixel, adjacent image scan-lines are encoded in a parallel way to each other. Hence, altogether they look like a rib structure. However, previously computed intersection information of previous rays with previous columns have to be maintained, i.e. saved so that it remains available for the current considered columns. This allows paring down intersection computations as well as the projection of the columns onto the intermediate image domain. Another advantage is that there is no need to transform the intersections of a volume column into the unit column, since they are already in this local space. Hence, the generation of the *column* template is initiated, which is later applied as in the parallel projection case. One drawback of the propagation method is the order on how the information is feed forward, i.e. previously computed intersections (ray segments) propagated to adjacent columns may not stay sorted according to their deep coordinates (in this case the deep coordinates are considered along the x-axis). Thus, before we generate the *column* template a sorting of all ray segments going through a unit column has to be performed.

Part IV

Hierarchical Data Encoding and Visualization

1 Introduction

1.1 Related Work

Volume visualization algorithms projecting wavelet transformed data sets into two-dimensional planes are often ranked to the *domain based rendering methods*. However, a comprehensive state of the art discussion of several approaches can be found in the introductory part I. In the following we suggest a new method based on wavelet decomposition of the volume data, a representation of the sparse data set by an octree structure and splines and a new shear-warp like visualization algorithm.

1.2 Wavelets for Volumetric Data

Wavelet theory provides an elegant multiresolution hierarchy framework based on multiresolution signal analysis which decomposes a function into a smooth approximation of the original function and a set of detailed information at different resolutions. The generalized wavelet series expansion, the discrete, and the continuous wavelet transform are tools used to compute this multiresolution hierarchy of a signal or a function and are closely related to the Fourier series expansion, the discrete, and the continuous Fourier transform. In practice the fast (inverse) wavelet transform implements an efficient algorithm to compute the discrete wavelet transform of a signal, which is very similar to the fast (inverse) Fourier transform as well as a two-band analysis (synthesis) subband (de)coding scheme [Mal99] [GW02].
However, before applying the discrete wavelet transform one needs to define appropriate basis[1] functions. There exist two families of functions, called the scaling functions $\{\phi_{j,k}(t)\}_{(j,k)}$ and the wavelet functions $\{\psi_{j,k}(t)\}_{(j,k)}$. Both functions sets should be square integrable and should constitute the basis of V_j (which caries the low-frequency approximation) and W_j (which caries the detail approximation), respectively, of an arbitrary function $f \in \mathbf{L}^2(\mathbb{R})$. The projection of a function f onto the other subspaces V_j, W_j is considered as $\mathbf{L}^2(\mathbb{R}) \rightarrow V_j, W_j$. With an appropriate inner product defined on each vector space a multiresolution analysis framework can be defined. In this sense W_j can be defined to be the orthogonal complement of V_j. Since then the approximations of f at scales 2^j and 2^{j+1} are equivalent to the orthogonal projections V_j and V_{j+1} with $V_j \subseteq V_{j+1}$, one obtains the next finer approximation of f by $V_{j+1} = V_j \oplus W_j$[2]. In this view orthogonal wavelets carry the details necessary to increase the resolution of a signal approximation. Therefore, each basis function ϕ should be orthogonal to each basis ψ under the chosen inner product, the basis functions ϕ, ψ of V_j, W_j should form a basis

[1] A basis consists of a minimum set of vectors or functions from which all other vectors or functions in the vector or functions space can be generated by linear combinations.
[2] Here, $\oplus$ is the union of spaces.

for V_{j+1}, and each basis $\psi, \hat{\psi}$ of W_j, W_i should be orthogonal to each other (if the last condition is not true the resulting wavelets are sometimes called *pre-wavelets*).

Wavelet bases often utilize their ability to efficiently approximate special classes of functions with only several non-zero wavelet coefficients. Application of these wavelet bases are for example data compression, noise removal or edge enhancement. The design of a wavelet ψ has therefor to be optimized to the considered task or function as well as to generate a maximum number of wavelet coefficients that are close to zero. However, this depends highly on the regularity of a function f, the number of vanishing moments of the wavelet ψ, the size of its support and regularity.

Daubechies wavelets are the optimal choice, because they have minimum support for a considered number of vanishing moments ([Dau88] [Dau92]). Other wavelets, as for example the Haar wavelets, the Mexican Hat, Shannon, Meyer, or Battle-Lemarié wavelets, often make a tradeoff between number of vanishing moments and the size of the support. For a very regular function f with only a few singularities a wavelet with many vanishing moments should be preferred to generate a large number of small coefficients. Functions with many singularities are better represented by wavelets with less support at the cost of fewer vanishing moments.

Further, a method to verify the correctness of the analysis and synthesis algorithms is to first compute the forward and backward transform of a signal and afterwards to compare the results with the original function. For a perfect analysis and synthesis of a signal the applied filters have to satisfy some conditions, e.g. the analysis and synthesis filters should be time-reversed versions of each other (cf. [Vet86] [VK95]).

However, unlike cos and sin waves used by the Fourier transform, wavelets have local decay in both spatial domain and frequency domain, hence they are well localized. One similarity between the windowed Fourier and the wavelet transform is that both representations taking inner products of f with a family of functions $g_{\omega,t} = g(x-t)e^{-i\omega x}$ and $\psi_{u,s} = \psi((t-u)/s)/s$, respectively. Using discrete multirate filter bank algorithms a fast orthogonal wavelet transform that needs only $O(N)$ operations for a signal of size N can be implemented. In higher dimensions the wavelet bases of $\mathbf{L}^2(\mathbb{R}^d)$ are constructed by tensor products of separable one-dimensional wavelet functions (cf. [Mal99]). Hence, we have

$$\begin{aligned} \phi(x,y,z) &= \phi(x)\phi(y)\phi(z) && (1.1)\\ \psi^1(x,y,z) &= \psi(x)\phi(y)\phi(z) && (1.2)\\ &\vdots && (1.3)\\ \psi^8(x,y,z) &= \psi(x)\psi(y)\psi(z). && (1.4) \end{aligned}$$

These eight wavelet functions measure functional variations along the different directions, i.e. intensity or density variations.

However, for given separable three-dimensional scaling and wavelet functions

$$\begin{aligned} \phi_{j,l,n,m}(x,y,z) &= 2^{j/2}\phi(2^j x - l, 2^j y - m, 2^j z - n) && (1.5)\\ \psi_{j,l,n,m}(x,y,z) &= 2^{j/2}\psi^k(2^j x - l, 2^j y - m, 2^j z - n), \quad k \in \{1,\ldots,8\} && (1.6)\\ & && (1.7) \end{aligned}$$

the extension of the one-dimensional discrete wavelet transform can be defined as

$$W_f^{\phi}(j_0,l,m,n) = \frac{1}{\sqrt{LMN}} \sum_{x=0}^{L-1} \sum_{y=0}^{M-1} \sum_{z=0}^{N-1} f(x,y,z)\phi_{j_0,l,m,n}(x,y,z) \tag{1.8}$$

$$W_f^{\psi^k}(j,l,m,n) = \frac{1}{\sqrt{LMN}} \sum_{x=0}^{L-1} \sum_{y=0}^{M-1} \sum_{z=0}^{N-1} f(x,y,z)\psi_{j,l,m,n}^{k}(x,y,z) \tag{1.9}$$

for $k = 1, \ldots, 7$. As in the one-dimensional case we start with an arbitrary scale j_0. The coefficients $W_f^{\phi}(j_0,l,m,n)$ and $W_f^{\psi^k}(j,l,m,n)$ define the approximation of the three-dimensional function $f(x,y,z)$ at this starting scale j_0 and the detail at higher scales $j \geq j_0$. Usually, $j_0 = 0$ and $L = M = N = 2^J$ such that $j = 0, 1, \ldots, J-1$ and $l, m, n = 0, 1, \ldots, 2^J - 1$. Hence, at the finest scale J usually $W_f^{\phi}(J,l,m,n)$ is identified with the discrete volume grid samples $f(x,y,z)$. The three-dimensional wavelet transform can be implemented by using digital analysis filters and by down sampling the resulting data. Even more easily, once the one-dimensional transform has been implemented, it can be applied along the different directions of the volume data set separately, i.e. first we compute the one-dimensional transform along the x direction of the volume data set, by reusing the result another one-dimensional transform along the y direction is performed, and finally this result is taken as input to the last call of the one-dimensional wavelet transform along the z direction. This is possible if separable filters are used and results in eight different frequency bands, one approximation band represented by the scale coefficients $W_f^{\phi}(j_0,l,m,n)$ and seven sets of detail coefficients represented by $W_f^{\psi^k}(j,l,m,n)$ for $k = 1, \ldots, 7$. The inverse wavelet transform reconstructs the function $f(x,y,z)$ from its wavelet coefficient representation by

$$f(x,y,z) = \frac{1}{\sqrt{LMN}} \sum_{l} \sum_{n} \sum_{m} W_f^{\phi}(j_0,l,m,n)\phi_{j_0,l,m,n}(x,y,z) \tag{1.10}$$

$$+ \frac{1}{\sqrt{LMN}} \sum_{k=1}^{8} \sum_{j=j_0}^{\infty} \sum_{l} \sum_{n} \sum_{m} W_f^{\psi^k}(j,l,m,n)\psi_{j,l,m,n}^{k}(x,y,z) \tag{1.11}$$

and as could be expected, the reconstruction algorithm is similar to the one-dimensional case as well, i.e. at each scale j we have eight sub-volumes representing the eight sub-bands. Each sub-volume is up-sampled by a factor of 2 – zero values are inserted at the new positions – and convolved with the inverse (synthesis) filters.
One should note that in case of finite functions f the boundary of the corresponding discrete signal $c_j(n)$ has to be treated differently from the interior part samples. That means, the convolution of a signal close to its boundary with the low and high pass filters requires to know the values beyond the boundary of the signal. With that boundary problems is usually dealt by appropriately designing boundary wavelets, by zero padding of the signal, by symmetrization of the signal (mirroring), or by periodization of the signal.

2 Hierarchical Encoding

In the following, we describe implementation details of our hierarchical volume rendering approach. At first, we discuss aspects, which are different from the standard approach to generate the images. Next, we focus on details on pre-computing the run-length encoding for the spline representation of the hierarchical data in use and on how a *qube* template is defined that both serves for speeding up intersection computations and allows for minimizing memory accesses by visiting each voxel only once.

2.1 Wavelet Coding Scheme and Octree Representation

In order to be able to include the spline model (see part II) and the numerical integration along a ray (see part I) a new data structure used during run-length encoding of the shear-warp type renderer of part III is required: Quadratic or cubic type-0 or type-6 splines represent the opacity function within a cube $\tilde{Q}$. More specifically, they represent the opacity function for each partition of $\tilde{Q}$, which may be a 24-tetrahedral partition $\triangle$ for $\tilde{Q}$ at the finest level as described in Sect. 3 or variations thereof (for details, see Fig. 2.1).
However, the new hierarchical encoding operates on scalar data on a Cartesian grid of dyadic size. For coding purposes, we perform the following operations in a preprocessing context. The classified volume is transformed into the wavelet space, whereby we implement the *Haar* and a *linear wavelet type* according to [SS96] [CDSB03] although for discussing the results we restrict ourselves to Haar wavelets since they have shown better performance for the type of data given. Each node of an octree data structure stores an average value of the considered sub cube which we denote as *average data*, i.e. $W_f^{\phi}[j,k]$. High frequency components (*detail data* or $W_f^{\psi}[j,k]$) are computed at each level in order to be able to prune some octree nodes. These are classified as *transparent*, *homogeneous* or *heterogeneous* depending on the average and detail data of the decomposed volume. If all detail data of an octree node are below a user-defined threshold and all child nodes of the current octree node are homogeneous then the current node of the octree becomes homogeneous as well. In the same way we identify transparent nodes. All nodes in the octree, whose parent node is either transparent or homogeneous, are pruned. For each leaf of the octree (which can be on any level j) the determining Bernstein-Bézier coefficients (see Sec. 2 and 3) are computed from a 2^j sub sampling of the original grid.
Further, computing the contribution of the voxel to the final image requires determining the intersection of all rays with these partitions. These intersection points (entrance and exit points of each partition belonging to a voxel) are stored in a pre-computed template called *qube* or *column* template (see Sec. 2) in order to speed up computing. It should be emphasized here, that by considering several rays passing through the cube $\tilde{Q}$ allows reducing memory accesses by up to a factor of 8. Visiting each voxel only once has

been the characteristic of splatting so far but now can be used for shear-warp by this technique as well.

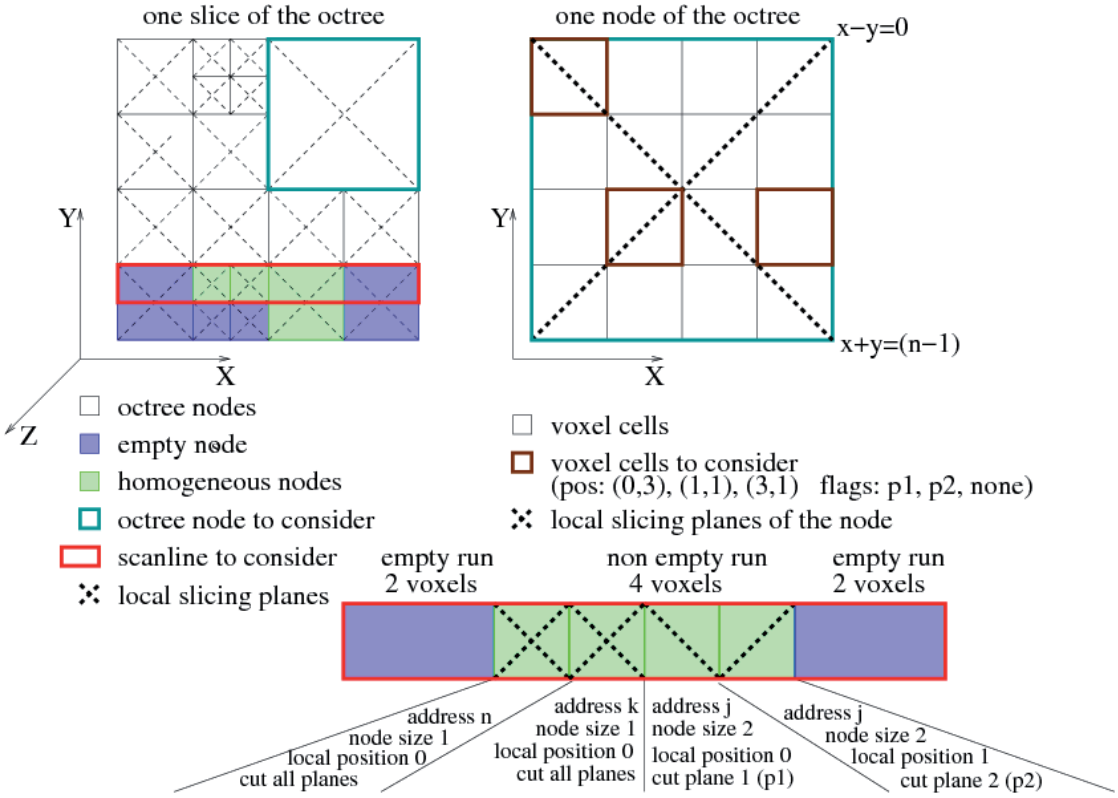


Figure 2.1: *Top left: One slice where the different leafs are subdivided by six slicing planes defining the tetrahedral partition of that leafs. Further, each leaf can be transparent, homogeneous or heterogeneous and stores an address to its quadratic or cubic spline coefficients. Top right: One octree leaf of size 4 and its subdivision by the slicing planes. This leaf is representative for 4^3 voxel cubes. For each voxel cube (addressed by local position coordinates inside the leaf) there is a 6 bit flag, which stores the plane numbers going through this cube. Bottom: Scanline created from octree, where the blue and green areas are transparent and non-transparent runs. For each non transparent run each voxel stores the address of the spline coefficients (4 bytes), the octree leaf size and the local coordinates of the voxel cube respective the leaf (3 bytes) and a flag indicating the planes to consider at current voxel cube (1 byte).*

2.2 Piecewise Splines defined on Octree Nodes

Usually our splines are defined on each unit cube $\tilde{Q}$ of a volumetric partition $\diamondsuit$ (or on each tetrahedron of $\triangle$) which normally corresponds to the finest level of an octree. However, in our hierarchical encoding we basically compute for each homogeneous leaf node of the octree the Bernstein-Bézier coefficients from a 2^j sub sampling of the original grid. However, the considered quadratic or cubic splines are in general smooth between neighboring cubes (or tetrahedrons) on a single layer. This is usually not the case when two adjacent layers of an octree have to be taken into account for reconstruction or rendering. In other words, considering two different sized neighboring cubes or leaf nodes from the two adjacent octree layers, i.e. which have a face in common, the corresponding poly-

nomial pieces (splines) also defined on two adjacent scales do not necessarily define an overall smooth function. Hence, this approach is not the most accurate one, because the resulting non smooth data transitions between the nodes may become visible depending on the compression rations used for the data (see results in part V) and the considered data sets itself. However, this approach allows us to generate the necessary run-length-encoded data sets from the octree structure and the spline hierarchy to be visualized with the shear-warp algorithm. There are only a few modifications which have to be done to the shear-warp method discussed in part III (see also Fig. 2.1). Nevertheless, a hierarchical encoding of volume data using for example an octree data structure where splines from different scales are able to reconstruct an overall smooth function which has to be examined in future. In general this is possible by considering and encoding the differences between the splines on two adjacent scales and additionally considering a narrow band of cubes at scale j around an octree node at scale $j-1$.

2.3 Generation of Run-Length-Encoded Data Sets

Once the octree data structure is created (see previous subsection), the run-length encoding for two main viewing directions has to be pre-computed. The third run-length encoding can be avoided using pointers to each scan line of the two other run-length encodings[1] and changing the processing order of the scan lines during rendering (see [SM02] [SHM04] and part III). However, to create a run-length encoded scan line, we start at the root node of the octree and visit only those leafs that are intersected by the current scan line. Whenever a transparent leaf is found the length of the voxel run within this leaf is written into the *run-length structure*. In the case of a homogeneous leaf for each voxel along the scan line within the current homogeneous leaf we store a pointer to the spline coefficients of the respective homogeneous leaf of the octree. Further, we store the size of the leaf and the local position of each voxel in the current octree leaf. Finally, we compute partitions for the cube $\tilde{Q}$ associated with the voxel. If the leaf belongs to the finest level of the octree we select the usual 24-tetrahedral partition. Otherwise, in each voxel a flag indicates which of the six slicing planes defining the tetrahedral partition of the homogeneous octree leaf subdivides $\tilde{Q}$ and the respective partitions are stored (see Fig. 2.1 for a 2D-explanation). Note, considering the usual volumetric partition $\diamondsuit$ the slicing planes which further subdivide the partition into $\triangle$ have not to be stored.

[1] A reduction to one run-length encoding is possible as well [SHM04] to further reduce the amount of data.

3 Visualization

3.1 Run-Length-Encoded Data Sets

For the visualization of the run-length encoded data sets obtained from the hierarchy we process the slices as before, i.e. space-leaping is performed in the same way as in the original algorithm mentioned in part III.
However, before determining the contribution of a slice to the intermediate image, all ray intersections with every partition (associated with a voxel on the slice) are computed (see Fig. 2.1). The intersection points of rays with the partitions define ray segments that are the subdivisions of the volume rendering integral mentioned in Equ. (4.8). All intersection points are stored in the so-called *qube* template. Since we consider parallel projection and since the shear-warp implementation assumes rays starting from a grid having the same grid constant as the volume, the *qube* template is identical for each voxel in the same slice and therefore has to be computed only once per slice.
Further, in order to minimize accesses to the main memory, we improved and further modified the processing scheme of the shear-warp algorithm. Instead of operating on two *voxel scanlines* and one *pixel scanline*, we consider one voxel and therefore two pixel scanlines (see also Sec. 2). However, by this rearrangement, only one access to voxel data is necessary per frame. Hereby, we directly compute all contributions of a cube $\tilde{Q}$ (belonging to a voxel) to the intermediate image (which are 4 pixels lying in both considered pixel scanlines) and thus achieve a significant speedup (independent from the choice of the non-discrete model representing the data). As a consequence, for *early-ray-termination* two pixel scanlines are considered in parallel. Whenever a non-transparent voxel and a non-opaque pixel (in at least one of both scanlines) is found, *compositing* is carried out for each ray segment intersecting the partitions of the voxel. For computing the contribution for each ray segment we use the Bernstein-Bézier coefficients of the splines stored in the memory and further determine the opacity and gradient values derived from the quadratic or the cubic splines. These values serve for computing the numerical approximation of the volume rendering integral (see part I). The numerical approximation allows controlling the visual quality and the numerical error for the rendering integral of (4.3). This is a technique still not found in typical implementations of volume rendering.

3.2 Hierarchy

In the section above we have discussed how to render the run-length encoded scanlines (or the rle volume) pre-computed from the octree hierarchy. We know that this algorithm has linear complexity ($O(N)$) in the number of non-transparent and non-occluded voxels N contained in the data set, as our previous algorithm described in part III. However, an approach which directly renders the octree nodes during its traversal (with complexity

$O(N \log N)$) gains redress only if the nodes of the octree data structure are going to change after each rendering (i.e. when the state of some nodes is changed due to, for example, a new opacity classification function). Even then the approach can only be satisfied when the (pre-) computation and the rendering of all the rle scanlines requires more time than the traversal of the octree and the rendering of its non-empty nodes. However, we can constitute two main stages where the pre-processing and rendering time of the above algorithm is spent. First, we have to traverse the octree, i.e. all non-empty nodes, and set up the rle volume during this traversal. Second, we have to render that rle volume. A direct rendering of the hierarchy has only one stage. That means, we have to visit all non-empty nodes in the hierarchy and directly visualize that nodes. Now, if all the stages in the first approach need together more time than the step in the hierarchical method, then this algorithm discussed in this section is eligible. A final answer, however, can only be given after an implementation of that hierarchical approach and a comparison to the method above. Nevertheless, it seems that with a careful design of some new data structures and methods required for that hierarchical algorithm (e.g. for early-ray termination) one could achieve better performance compared to the previous method. In other words, the (pre-) computation of the rle volume from the hierarchical data structure is rather time consuming (see part V). This is because we can only investigate the empty space in the hierarchical data during the computation of the rle volume. Rendering is performed in a subsequent step, thus the acceleration technique early-ray termination can not already be applied in the former stage. And hence a rle volume of the whole hierarchy has to be calculated first. Afterwards the rendering of the rle volume is performed quite quickly (see also part V). Hence, if the data classification function is going to change very frequently, at that given times one has to recompute at first the whole rle volume and at second render it. In that view the complexity of both algorithms is $O(N \log N)$ due to the traversal of the hierarchical data structure. However, in a direct rendering of the data, i.e. performed during the traversal of the hierarchy, one could investigate the empty space as well as early-ray termination at the same time. Thus, one can not only omit the generation of the data structures used for the rle volume, one also can skip non-empty nodes due to the possible application of early-ray termination [1].

In the following we outline the new hierarchical rendering algorithm. To simplify matter we restrict the discussion to quadtrees (see Fig. 3.1). The presented algorithm, however, can be extended to octrees in a straight forward manner. In the right illustration of Fig. 3.1 we show four different cases how a quadtree is traversed depending on the sign of main viewing axis and the viewing or ray direction (blue arrows) to satisfy compositing order of the nodes, i.e. such that the volume rendering integral can be solved correctly. In fact eight $(2 * 4)$ or twenty four $(4 * 6)$ different cases have to be considered for a quadtree or an octree representation of the data, respectively. However, the numbers shown in the appropriate nodes indicate the order of traversal of the corresponding nodes. That means, when the main viewing axis becomes the positive y-axis and the x and y components of ray direction $\mathbf{r}_d$ (blue arrow) have positive signs, a recursive depth first traversal of the quadtree starts always at the bottom-left child node of the current considered parent node and then visits the bottom-right, top-left, and top-right nodes, respectively. This traversal results in the numbering of the nodes shown in the

[1] This, of course, depends on the data and the classification function.

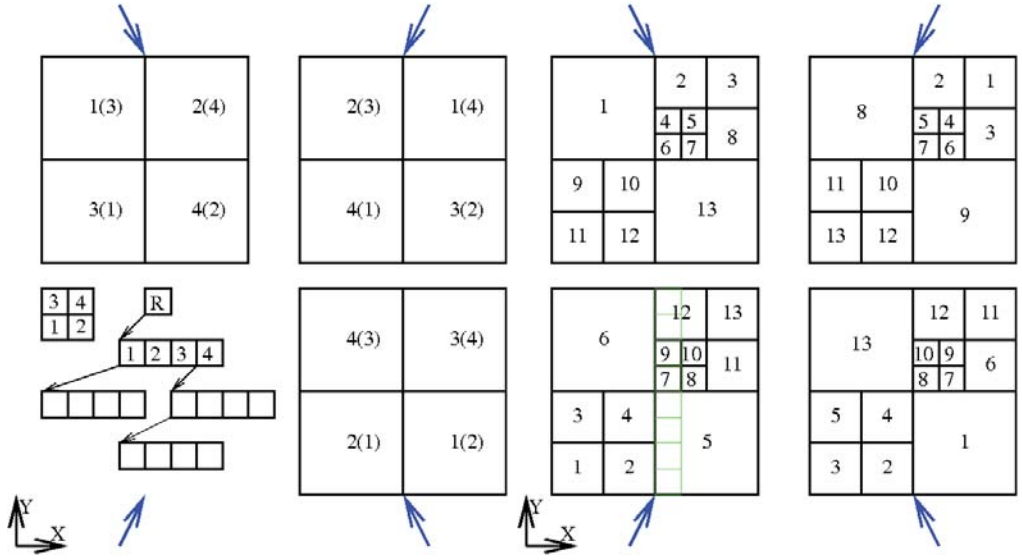


Figure 3.1: *Left: Look-up tables (black values) for a correct access of the child nodes (black quads) according to the different considered main viewing axes and ray directions (blue rays) to satisfy compositing order of the data. The bottom-left figure shows the quadtree data structure with the default numbering of the child nodes. Right: The traversal order of the quadtree nodes according to the look-up table values at the left.*

bottom-left sub-image of the right illustration of Fig. 3.1. When, for example, the main viewing axis becomes the negative y-axis and the x and y components of ray direction $\mathbf{r}_d$ have positive and negative signs, respectively. The recursive depth first traversal of the quadtree starts at the top-left child node and then visits the top-right, bottom-left, and bottom-right nodes (in that order) which results in the numbering of the nodes shown in the top-left sub-image of the right illustration of Fig. 3.1. The two other cases are also visualized, i.e. for the positive main y-axis and negative x and positive y components of the ray direction (bottom-right sub-image) as well as the negative main y-axis and negative x and y components of the ray (top-right sub-image), respectively. Note, once the main viewing axis is determined, for example, as the positive y-axis. The y component of the ray direction can not have a negative sign any more. In other words, the main axis and its sign are computed from the ray direction.

A straight forward implementation of a rendering algorithm which traverses and visualizes the nodes depending on the viewing direction would require to store eight reorganized copies of the same quadtree to satisfy the compositing order of the nodes. Hence, the traversal of the different quadtrees remains always the same (i.e. fixed) and one has to switch between the different quadtree data structures depending on the case (i.e. ray direction). However, we propose to use eight look-up tables which store the order the children have to be visited for the considered case and one quadtree data structure only. In fact seven tables are enough, since one transition would be an identity mapping which can be omitted (have not to be stored). This dramatically reduces the data consumption and makes the algorithm more convenient. In the sub-images of the left illustration of Fig. 3.1 we show the look-up tables corresponding to the four different cases shown on

the right illustration. More precisely, the lower-left sub-image in the left illustration shows the default quadtree data structure (i.e. its organization in main memory) and the default traversal order of the child nodes as well as the corresponding sub-cubes of the data for the first case (i.e. when the main viewing direction becomes the y-axis with positive x and y components of the ray direction). In this case we do not need to store any look-up table, since the child nodes can be visited in the order they are stored in main memory (identity mapping). For the other three cases the numbers in brackets give the indices into the look-up table where the neighbouring numbers are stored at the corresponding positions in that table and are used as indices into the array of child nodes in the default quadtree data structure to visit the nodes in correct compositing order (as shown in the right illustration of Fig. 3.1). This basic approach can be easily generalized to include the other cases and to octrees not shown here.
Once the look-up based traversal of the octree is defined and satisfies the compositing order of the nodes, we can visualize the nodes or better the corresponding sub-cubes of the data set. This rendering step is very similar to our original algorithm in part III. The main distinction is that often different sized sub-cubes (sub data sets) have to be visualized which project onto sub-regions of the intermediate image and the sub-cubes contain only non-transparent voxels. Therefore, space leaping is performed during the traversal of the octree where early-ray termination takes place during the visualization step, i.e. when a non-empty sub-cube is rendered. However, the algorithm starts with the root node and recursively visits all non-empty octree nodes (only once per rendered image) in the order discussed above. Once a homogenous node in the octree is found we also know the size and position of the corresponding sub-cube in the volume data set. From its size and position we determine the region of projection (region of interest) of that sub-cube in the intermediate image. Similarly, using the position and the size we are able to find the necessary intersections, ray segments, etc. passing through the considered voxel of that sub-volume from our *column* template. This gives us the required information at hand for rendering the data represented by a sub-cube. Note, we can render the original data, i.e. for each voxel in the sub-volume we have to consider different data (or spline coefficients). Or we can render the averaged data represented by the considered octree node. This reduces the reload or the re-computation of the spline coefficients. However, as mentioned above, the rendering or compositing process is except some small changes the same as in our original algorithm from part III. The differences are: First, we do not perform space leaping, since the data in the sub-volume is always non-transparent. Second, before compositing a voxel[2] early-ray termination is applied. Then, for example, if we have to skip some pixels (and of course the appropriate voxels as well), we also have to check if the next pixel to consider is still in the region of interest. That means, if the next pixel is still in region on the intermediate image of the projected sub-cube. Similarly, we could check if the next voxel is still in the currently considered sub-volume. If this is the case, we proceed with the compositing, otherwise adjacent (sub-volume) scanlines and the corresponding image scanlines have to be considered until the whole sub-cube is processed.

[2]We consider the same number of voxel and image scanlines as in our original algorithm.

Part V

Results On Reconstruction and Visualization

1 Prerequisites

We have implemented the algorithms in C++ using a 3.0 GHz PentiumIV PC with 1 GB RAM or a 2.0 GHz PentiumIV M Laptop with 2 GB RAM and VGL[1] as underlying library. Different 8 bit data sets on a Cartesian grid are chosen for our experiments or illustrations:

- Engine (CT), courtesy: General Electric. CT scan of two cylinders of an engine block, ($256 \times 256 \times 128$ voxels).
- Teapot (artificial CT), courtesy: Terarecon Inc., MERL, Brigham and Women's Hospital. CT scan of the SIGGRAPH 1989 teapot with a small version of the AVS lobster inside, ($256 \times 256 \times 178$ voxels).
- Head (MRI), courtesy: Brain Development Lab, University of Oregon. A T1 weighted MRI of the head of Mark Dow. Recorded at the Martinez, CA, VA Hospital on a Picker 1.5T system, ($187 \times 236 \times 253$ voxels).
- Bonsai (CT), courtesy: Stefan Röttger, VIS, University of Stuttgart, Germany. CT scan of a bonsai tree, ($256 \times 256 \times 256$ voxels)
- Foot (X-Ray), courtesy: Philips Research, Hamburg, Germany. Rotational C-arm x-ray scan of a human foot with tissue and bone present in the data set, ($256 \times 256 \times 256$ voxels).
- Skull (CT), courtesy: Siemens Medical Solutions, Forchheim, Germany. Rotational C-arm x-ray scan of phantom of a human skull, ($256 \times 256 \times 256$ voxels).

However, many example data sets can be downloaded from the web page of *Gr*aphisch–*I*nteraktive *S*ysteme (GRIS, see also the VolVis organization and the Visualization Lab of the Computer Science Department at Stony Brook University) and the page of the Robert and Beverly Lewis Center for NeuroImaging. Another page of GRIS provides several more recent data sets in raw format. Additional medical data sets with 8bit as well as 16bit voxel resolution are provided on the web page of the University of Erlangen where the data is represented on regular grids mainly coming from CT or MRI scanners. The data is stored in the special file format (PVM) which contains information about the grid size, bit depth, and the cell spacing of a data set. The Visible Human Project creates a complete, anatomically detailed, three-dimensional representation of a male and a female cadaver body. The acquisition was done by CT, MR and cryosection images where the male body was sectioned at one millimeter intervals and the female body at one-third of a millimeter intervals (see also National Library of Medicine).

[1] Volume Graphics GmbH

2 Results on Spline Models

In this section we discuss the visual and numerical results of our spline models presented in part II. Therefore, in the first step we consider different test functions often used in visualization papers [ML94] [RZNS03] [NRSZ04] [RZNS04a] to show the reconstruction quality of the data approximation models. In the second step we rewrite some error norms which turned out to be useful to measure the deviation between the original synthetic data values (function values) and the reconstructed spline values. By using the synthetic functions and the error norms we present the numerical results of our splines in the third step. Since we consider six different data approximation schemes, many tables and figures with results would confuse the reader and unnecessarily enlarge the section, therefore, we refer the interested reader to the appendix for more detail. Here we only give the most important outcomes concerning the cubic type-6 spline model. In the fourth and the final pace the visual quality and the performance of the splines is shown, respectively.

2.1 Test Functions

The quality of reconstruction models is often verified concerning some appropriate but difficult tests. However, the Bernstein-Bézier coefficients for the different splines are calculated from the given data samples, which itself are obtained from smooth test functions (synthetic data). Once the coefficients are determined, the splines are evaluated at some prescribed positions. The reconstructed values are compared to the original values obtained from the test functions itself at the same positions by using appropriate error norms (see next section).
The first test function is called the ***M**arschner-**L**obb* [ML94] test function $f_{ML} : [-1,+1]^3 \rightarrow \mathbb{R}$, which is frequently used as a test in volume visualization [MMMY97b] [MMMY97a] [MMK+98] [MPWW00] [TMG01] [MJC01] [SM02] [BMDS02] [KWTM03] [LM05] [KM05] and is defined as

$$f_{ML}(\mathbf{v}) = \frac{1-\sin(\pi v_z/2)+\alpha(1+\cos(2\pi\beta\cos(\pi\sqrt{v_x^2+v_y^2}/2)))}{2(1+\alpha)}, \qquad (2.1)$$

where $\mathbf{v} = (v_x, v_y, v_z) \in [-1,+1]^3$, $\alpha = 1/4$ and $\beta = 6$. There are two major aspects why this function is often used to test a data reconstruction model. First, this function is designed by means that it becomes more and more oscillating from its center to its outer parts, i.e. it contains low as well as high frequencies. Second, usually only very few samples are examined from this function for data as well as derivatives approximation by a model. That means, a data set sampled at the Nyquist rate allows theoretically a reconstruction of that function using the optimal sinc filter. However, the question one usually would like to answer is, how well non optimal filters or models, namely our splines, perform in such cases.

The second test function is called the ***Sq**rt* test function $f_{Sq} : [-0.5, +0.5]^3 \to \mathbb{R}$. This spherical function is defined as

$$f_{Sq}(\mathbf{v}) \quad = \quad \sqrt{v_x^2 + v_y^2 + v_z^2}, \tag{2.2}$$

where $\mathbf{v} = (v_x, v_y, v_z) \in [-0.5, +0.5]^3$. This function was used to shows smoothness behavior of the *Super-Splines* [RZNS04a], hence, for comparison reasons it is a good choice for our cubic type-6 spline model as well.

2.2 Different Types of Errors

The approximation quality or order of the reconstruction models is shown by measuring the error between the reconstructed values and the original values obtained from the splines and the test functions, respectively. Hence, in the following we consider different kinds of errors (cf. [KLS96] [NRSZ04]).
Our first error measure is the the maximal error at the grid data points, i.e.

$$\mathrm{err}^{\xi}_{data} := \max\{|(f_\xi - s_\xi)(h(2i+1, 2j+1, 2k+1)/2)| : i, j, k = 0, \ldots, n-1\}. \tag{2.3}$$

The second measure is the maximal error in the uniform norm, i.e.

$$\mathrm{err}^{\xi}_{max} := \max\{|(f_\xi - s_\xi)(\mathbf{v})| : \mathbf{v} = (v_x, v_y, v_z) \in \Omega\}, \tag{2.4}$$

and is computed as follows. In each cube $Q \in \diamondsuit$ we take 240 uniformly distributed points and measure the error at these points, i.e. the set of all points chosen in this way from Ω is denoted as χ, where the total number of points is given as $|\chi|$. Then, at the points from χ the error err^{ξ}_{max} is found as the maximal error. The third and fourth measure is the average error given by

$$\mathrm{err}^{\xi}_{mean} := \frac{1}{|\chi|} \sum_{\mathbf{v} \in \chi} |(f_\xi - s_\xi)(\mathbf{v})|, \tag{2.5}$$

and the root mean square error defined as

$$\mathrm{err}^{\xi}_{rms} := \sqrt{\frac{1}{|\chi|} \sum_{\mathbf{v} \in \chi} ((f_\xi - s_\xi)(\mathbf{v}))^2}, \tag{2.6}$$

respectively. Here, $\xi \in \{ML, Sq\}$ is used to differentiate between the functions.
For the visual tests as shown in the figures of the next two sections we use a slight different approach. First, we use an 8bit look-up table *LUT* which transforms error values to colors (cf. Fig. 2.1). Here, we define some threshold values a, b, c, and the

Figure 2.1: *Look-up table of colors for appropriate error values. Index values in the ranges of* $[0, 85]$, $[85, 170]$, *and* $[170, 255]$ *are encoded from yellow to green, from green to blue, and from blue to red, respectively.*

interval $[0, a, b, c]$ with corresponding index threshold values of $85, 170, 255$, respectively, to be able to compute a linear map between the error and index values. Then, all error values $> c$ are shown as red and error values in the ranges $[0, a]$, $[a, b]$, and $[b, c]$ are encoded from yellow to green, from green to blue, and from blue to red, respectively. Second, we appropriately redefine the error functions err^ξ and we scale the computed errors appropriately as well, such that, our look-up table can be considered as a function $LUT : [0, 255] \to \mathbf{c}$ with $\mathbf{c} := (r, g, b) \in [0, 255]^3$. The exact scaling factors and threshold values are given in each figure.
Now, for the color-coded errors of the iso-values we proceed as follows. We use a user-specified iso-value δ (the precise values are given in each figure as well) and apply our shear-warp approach to compute the position $\mathbf{v}$ in the three-dimensional domain Ω where the iso-value δ is intersected by a ray. For this computation we apply the considered spline model. Once we have found the position $\mathbf{v}$ in Ω with $s_\xi(\mathbf{v}) = \delta$, we compute the iso-value at the same position using the original function $f_\xi(\mathbf{v})$. The redefined error between the two iso-values is $\text{err}^\xi = |x - y|$ with $x := s_\xi(\mathbf{v})$ and $y := f_\xi(\mathbf{v})$. This error value is scaled and the linear map between the error and the index of the look-up table is defined, such that the table LUT can be applied.
The color-encoded errors of the gradients are obtained similarly. We perform the iso-surface computations as before, but here we use the error formula $\text{err}_1^\xi = (g_{1,x}g_{2,x} + g_{1,y}g_{2,y} + g_{1,z}g_{2,z})/(\|\mathbf{g}_1\|\|\mathbf{g}_2\|)$, where $\mathbf{g}_1 = (g_{1,x}, g_{1,y}, g_{1,z})$ and $\mathbf{g}_2 = (g_{2,x}, g_{2,y}, g_{2,z})$ are the gradients obtained at the same positions $\mathbf{v}$ at a user-defined iso-value δ, i.e. when $s_\xi(\mathbf{v}) = \delta$. The first gradient $\mathbf{g}_1$ is computed by using the considered spline model which represents the original test function[1] and the second gradient $\mathbf{g}_2$ is directly determined from the original test function $f_\xi(\mathbf{v})$ at the same position $\mathbf{v}$. In this way we measure the error of the gradients or the quality of the spline model. Here, we define the norm as $\|\mathbf{g}\| = \sqrt{g_x^2 + g_y^2 + g_z^2}$. The final error value is then obtained from $\text{err}^\xi = (180^\circ \arccos(\text{err}_1^\xi)/\pi)$, hence the error $\text{err}^\xi(\text{err}_1^\xi)$ is in the interval $[180^\circ(-1), 90^\circ(0), 0^\circ(+1)]$, where values $> 90^\circ(< 0)$ are clipped to $90^\circ(0)$ (radiants). Here we define the linear map from $[0^\circ, 90^\circ]$ to $[0, 255]$, hence, we also apply an index of 8bit to find the corresponding color code in the color look-up table. Once more the exact threshold values a, b, c are given in each figure.
Similarly, we proceed for the color-encoded errors of the mean curvatures. After the position of an iso-value is found, we compute the mean curvatures as discussed in part I using the considered spline model, the original function, and the error function $\text{err}^\xi = |x - y|$ (where now x and y represent the mean curvatures). The previously shown look-up table is used again with appropriate threshold values a, b, and c which are given at the figure captions and of course the errors of the curvature values have to be scaled in a careful way as well.

2.3 Numerical Tests

In the following we show some numerical results – partly raised by images – of the spline models discussed in part II and [RZNS03] [SZ05]. Hence, some parts maybe

[1] The original function is sampled first and afterwards the parameters (coefficients) for the considered spline model are determined.

quite similar to the results already presented in the above papers and in [NRSZ04] [RZNS04a] [SHZ+05]. Further, since an overall comparison of the different models we have implemented would heavily increase in size this results part, we shifted some tables, figures, and discussions to the appendix (cf. App. A). Here we give a summary of the numerical results with a focus on the cubic type-6 spline model.
However, the implementation of splines requires at first a verification step which shows the correctness of the implemented data reconstruction models. Hence, a first simple test is to use only the polynomials from the space $\mathcal{P}_n := \text{span}\{x^i y^j z^k : i, j, k \geq 0, i+j+k \leq n\}$ of trivariate polynomials as test functions, which can be reproduced by the considered splines. Once this is done, we can proceed with the other numerical tests by using the test functions and error types mentioned above to illustrate the efficiency and the approximation quality of the reconstruction models (cf. also App. A).

2.3.1 Value Reconstruction

Figure 2.2 depicts a comparison of the trilinear and quadratic type-0 models as well as the quadratic and cubic type-6 spline models based on value reconstruction of the Marschner-Lobb benchmark. That means, the figure shows the errors between the spline values and the original function values located at the same domain positions on the user defined iso-surface.

h	trilinear type-0	quadratic type-0	quadratic type-6	cubic type-6
1/16	0.0740	0.0765	0.0774	0.0770
1/32	0.0643	0.0529	0.0548	0.0538
1/64	0.0312	0.0201	0.0204	0.0202
1/128	0.0095	0.0057	0.0057	0.0057
1/256	0.0025	0.0014	0.0014	0.0014

Table 2.1: *Approximation errors (root mean square) for the data values of the Marschner-Lobb test function f_{ML} using trivariate piecewise trilinear and quadratic type-0 as well as quadratic and cubic type-6 splines for reconstruction, respectively.*

However, the linear type-0 splines in Bernstein-Bézier form defined on Ω have two main difficulties. First, they generate jags in the final images when general function reconstruction is considered (e.g. the Marschner-Lobb function). Second, the gradients obtained from that model are not smooth across neighboring unit cubes or cells. To overcome the disadvantages one often considers the trilinear model in volume visualization algorithms. However, the first difficulty remains even for the trilinear model, because in general it can only reproduce piecewise linear functions. Therefore, even the values of general functions (e.g. the Marschner-Lobb test) can be reconstructed for increasing volume data sizes quite well (cf. Tab. 2.1 and App. A) compared to the more sophisticated models, these trilinear splines generate jags in the final images (as can be see in the top left image in Fig. 2.2) when more general function reconstruction is considered. The second difficulty of the linear type-0 splines can be omitted by considering different gradient estimation techniques in the trilinear model, which is discussed below.
Piecewise quadratic type-0 splines in Bernstein-Bézier form defined on a volumetric domain Ω lead to better reconstruction results (cf. Tab. 2.1) compared to the previous

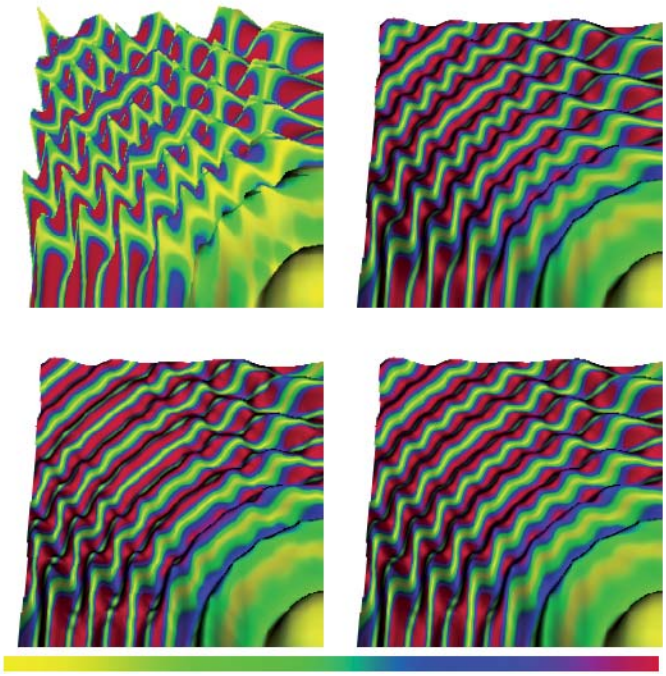


Figure 2.2: *The value reconstruction of the Marschner-Lobb test function at points* $\mathbf{v}$*, where the iso value becomes* 127.5/255.0 *(i.e. where* $s_{ML}(\mathbf{v}) = 0.5$*) by using type-0 piecewise trilinear (top left) and quadratic (top right) splines and type-6 quadratic (bottom left) and cubic (bottom right) models, respectively. The images show the error* $|(f_{ML} - s_{ML})(\mathbf{v})|$ *between the values obtained from the Marschner-Lobb test function* $f_{ML}(\mathbf{v})$ *and values reconstructed by the different piecewise spline models* $s_{ML}(\mathbf{v})$*. Here, the error threshold values are* $a = 0.025, b = 0.05$*, and* $c = 0.075$*. The grid spacing is* $h = 1/32$*. The bottom most image shows the lookup table (cf. 2.1).*

trilinear model. The main advantages of that quadratic model (and also the models below) are: First, it can reproduce higher degree polynomials, so that for non-linear functions no jags occur in the final images (cf. top right image in Fig. 2.2). Second, the gradients can be directly computed from that model and are smooth (see below). Third, noisy data (e.g. arising from MRI) is smoothed automatically, because of the approximating behavior of that splines. However, this can sometimes also be a disadvantage. For example, interpolating a piecewise linear function is not possible.

The main goal for the type-6 models is to develop approximating splines which satisfy smoothness properties needed for volume visualization. However, both approaches can be seen as a compromise between fast visualization and accurate approximation of the data. In the bottom left and right images of figure 2.2 one can observe that the quadratic as well as the cubic type-6 spline models are superior compared to the trilinear model (top left image). They do not produce any jags and the errors of the data values remain low as for the quadratic type-0 splines (cf. also Tab. 2.1). Nevertheless, the *Super-Splines* have more difficulties to reconstruct the iso-surface of the Marschner-Lobb test function because of the low total degree of its piecewise polynomials. That means, because only few samples are taken from the test function, the high frequencies at the outer parts of

the benchmark are not visualized as good as in the reconstruction by cubic type-6 or quadratic type-0 splines. These splines tend to oscillate more because of their higher total degree polynomials used for reconstruction which in this case reflect the hight frequencies at the outer parts of the benchmark in a more natural way.

2.3.2 Gradient Reconstruction

Figure 2.3 depicts a comparison of the trilinear and quadratic type-0 models as well as the quadratic and cubic type-6 spline based on gradient reconstruction of the Marschner-Lobb benchmark. In this case, the figure shows the errors between the spline derivatives and the original function derivatives located at the same domain positions on the user defined iso-surface.

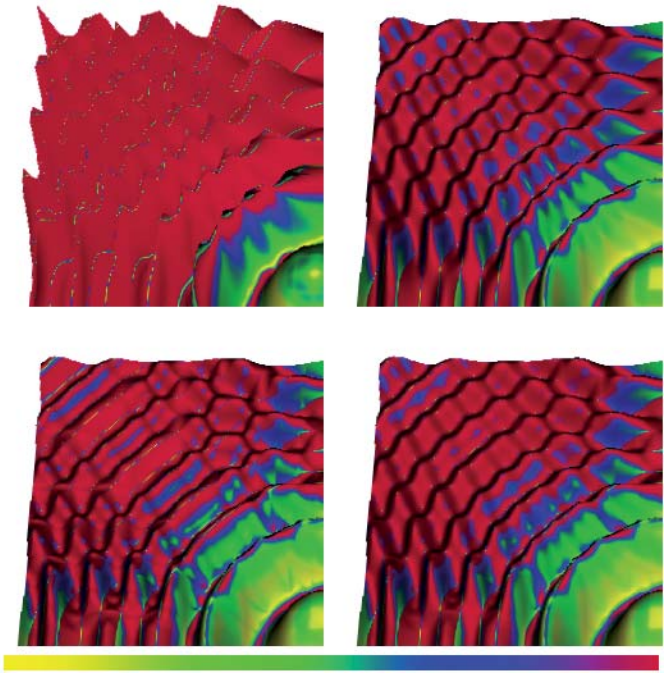


Figure 2.3: *The gradient reconstruction of the Marschner-Lobb test function at points* $\mathbf{v}$*, where the iso value becomes* 127.5/255.0 *(i.e. where* $s_{ML}(\mathbf{v}) = 0.5$*) by using type-0 piecewise trilinear (top left) and quadratic (top right) splines and type-6 quadratic (bottom left) and cubic (bottom right) models, respectively. The images show the error* $|\nabla(f_{ML} - s_{ML})(\mathbf{v})|$ *between the gradients obtained from the Marschner-Lobb test function* $f_{ML}(\mathbf{v})$ *and gradients reconstructed by the different piecewise spline models* $s_{ML}(\mathbf{v})$*. Here, the angle (measured in degrees) between the two gradients defines the error, i.e. the angle thresholds are* $0°, a = 10°, b = 20°$*, and* $c = 30°$*. The grid spacing is* $h = 1/32$*. The bottom most image shows the lookup table (cf. 2.1).*

As already discussed above, linear type-0 splines are not the preferred choice for volume visualization. An additional reason for this is, because the gradients computed from that model are not smooth and, hence, not a good choice for evaluating a lightning model (e.g. Phong lightning). The piecewise constant behavior of the gradients would result in low quality images. However, this is the main consideration why the piecewise trilinear

h	trilinear type-0	quadratic type-0	quadratic type-6	cubic type-6
1/16	2.2893	2.4504	2.4905	2.4757
1/32	2.4044	2.0222	2.1083	2.0595
1/64	1.4338	0.8054	0.8250	0.8141
1/128	0.4754	0.2301	0.2336	0.2318
1/256	0.2534	0.0596	0.0604	0.0600

Table 2.2: *Approximation errors (root mean square) for the gradients along the x direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise trilinear and quadratic type-0 as well as quadratic and cubic type-6 splines for reconstruction, respectively.*

model (not really in Bernstein-Bézier form) is often applied for reconstruction. However, in this model a different gradient estimation technique is applied. That means, the 1st derivatives do not directly result from that spline model, instead they are computed at any location of the volume domain Ω by linear interpolation of preestimated gradients defined at prescribed positions (e.g. grid points) of the volume. The trilinear model is a compromise between the linear and the quadratic type-0 spline models. It allows on one side faster reconstructions of the data compared to the above quadratic tensor product splines in Bernstein-Bézier form. On the other side, a pre-computation of gradients for each grid point location of the volume data set by using central differences or the Sobel operator and a following linear interpolation of that grid point gradients allows the user to reconstruct derivatives anywhere on the volumetric domain Ω in a more accurate way compared to the simple linear type-0 splines. This, however, requires an additional gradient volume data set which needs at least three times as much memory as the data itself.

In contrary, the quadratic C^1 type-0 splines (tensor product splines) do not need any additional gradient data sets to be pre-computed, nor any other models for gradient estimation. The first as well as the second derivatives can be directly obtained from that model. However, the computational complexity of that splines increases, but they also generates more satisfying results (compare the trilinear and the quadratic splines from the top left and right images of Fig. 2.3, respectively, and Tab. 2.2).

Quadratic and cubic type-6 splines behave very similar to the tensor product model. The main goal for the quadratic type-6 splines was to develop approximating splines with the lowest possible polynomial degree which additionally satisfy smoothness properties needed for volume visualization. In general appropriate smoothness conditions have to be satisfied to obtain, for example, a C^1 quadratic spline model on Δ. For this quadratic type-6 spline model some of the C^1 smoothness conditions are replaced by other useful conditions, i.e. averages of smoothness conditions, to obtain a compact representation of that splines. In case of the cubic type-6 model non of the C^1 smoothness conditions have been removed, which results in an overall cubic C^1 spline model on Δ. However, the quadratic splines are smooth nearly everywhere on the volumetric partition $\diamondsuit$. Further, regarding the approximation properties, the splines s yield nearly optimal approximation order, while their derivatives yield optimal approximation order of smooth functions f which is a non-standard mathematical phenomenon (see part II and [RZNS03] [NRSZ04]). However, one could expect that the numerical accuracy or the visual quality of the quadratic type-6 model compared to the quadratic type-0 or

cubic type-6 splines will diminish a lot. This is not the case as can be observed in table 2.2, which shows the decrease of the approximation error of the spline models to the Marschner-Lobb test function for decreasing grid spacing h. In figure 2.3 one corresponding iso-surface is shown, where the approximation error of the spline derivatives according to the original function derivatives is color coded. The not overall C^1 consistency of the quadratic type-6 splines will be revealed in the next section.

2.3.3 2nd Derivative Reconstruction

Figure 2.4 depicts a comparison of that models based on 2nd derivative reconstruction of the Marschner-Lobb benchmark. However, it is clear that C^1 models are not able to reconstruct the second derivatives of a general function in a smooth way. Nevertheless, in some cases (e.g. for fast curvature visualization of volume data sets) even piecewise constant 2nd derivatives or curvatures could produce satisfying results (cf. part I). The numerical results are given in table 2.3.

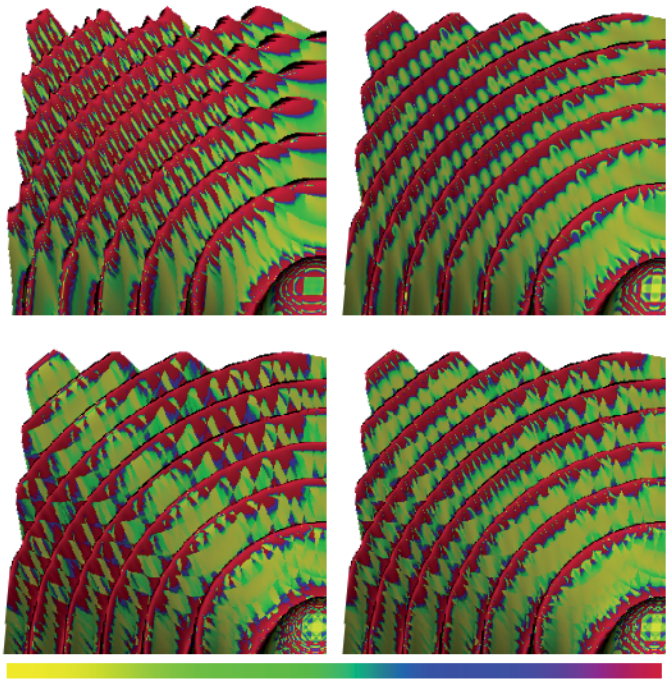


Figure 2.4: *Reconstruction of Hesse matrices of the Marschner-Lobb test function at points* $\mathbf{v}$*, where the iso value becomes* $127.5/255.0$ *(i.e. where* $s_{ML}(\mathbf{v}) = 0.5$*) by using type-0 piecewise trilinear (top left) and quadratic (top right) splines and type-6 quadratic (bottom left) and cubic (bottom right) models, respectively. The images (with* $h = 1/64$*) show the error between the Hesse matrices obtained from the Marschner-Lobb test function* $f_{ML}(\mathbf{v})$ *and Hesse matrices reconstructed by the piecewise quadratic spline model* $s_{ML}(\mathbf{v})$*. Here, the mean curvature computed from the Hesse matrices of the original function and this represented by the spline model is used to define the error. The error thresholds are* $0.0, a = 0.0025, b = 0.005$*, and* $c = 0.0075$*. Once more the bottom most image shows the lookup table (cf. 2.1) used for error encoding.*

h	trilinear type-0	quadratic type-0	quadratic type-6	cubic type-6
1/16	94.89	109.6	109.9	109.5
1/32	107.5	97.43	99.67	97.52
1/64	69.77	53.68	53.27	51.28
1/128	30.64	26.91	26.39	24.91
1/256	13.46	13.46	13.15	12.32

Table 2.3: *Approximation errors (root mean square) for the 2nd derivatives along the x direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise trilinear and quadratic type-0 as well as quadratic and cubic type-6 splines for reconstruction, respectively.*

2.4 Visual Quality

In the previous section we gave numerical results based on the different trivariate spline models. These results allowed us to study the approximation order of the splines. The corresponding images taken at a user-defined iso-surface of the Marschner-Lobb test function gave us further a first impression of the visual quality one could achieve with the appropriate model. In this section we compare the visual quality of the different piecewise polynomials based on the synthetic spherical test function $f_{Sq}(\mathbf{v})$, because the visual quality is more critical in real world volume rendering applications than any numerical tests, and this spherical benchmark allows us to enhance some interesting properties of the considered spline models. We also show how the spline models perform on real world data sets. Therefore we consult the Bonsai data set of 256^3 samples.

2.4.1 Linear Models

The next two figures (i.e. Fig. 2.5 and Fig. 2.6) give results obtained by using different linear spline models. However, it is well known that these spline models are not the

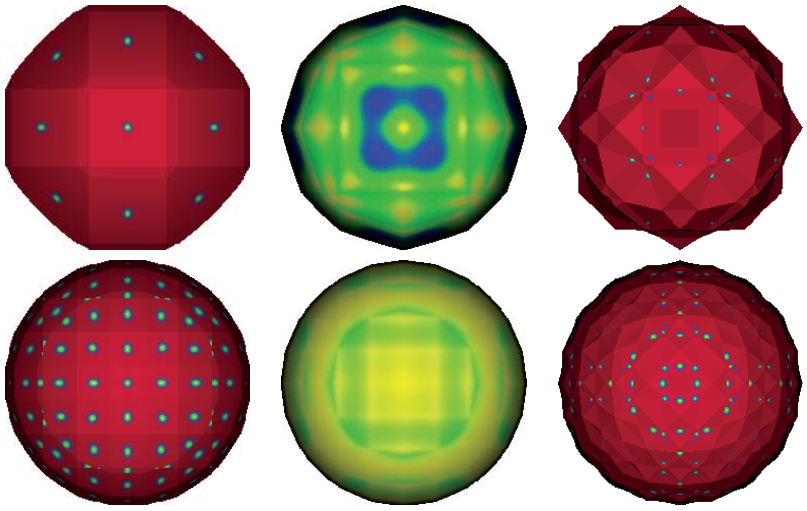


Figure 2.5: *The top and bottom rows show images rendered from* 8^3 *and* 16^3 *volume data sets, respectively, sampled from the spherical test function* f_{Sq}*. All images show the color-coded errors of the gradients obtained from the original spherical function* $f_{Sq}(\mathbf{v})$ *and from the different spline models* $s_{Sq}(\mathbf{v})$ *at the user-defined iso-value* $s_{Sq}(\mathbf{v}) = 60.0/255.0$*. The reconstruction models are (from left to right) the linear tensor product spline model, the trilinear model (two-side differences) on* Ω*, and the the linear spline model on* Δ*. The threshold values are* $0.0°, a = 0.85°, b = 1.7°$*, and* $c = 2.55°$*. Hence, the red color denotes angles (errors* err*) between the two gradients which are bigger than* $2.55°$*.*

appropriate choice for volume visualization, because the models generate piecewise constant gradients. This leads mostly to images where the volume grid structure is heavily visible. Gradients reconstructed by one of these methods are only at few positions on the sphere suitably accurate. At all other location on the sphere the errors, i.e. the difference between the exact and reconstructed gradients, are rather inaccurate. The trilinear model is considered as the standard method for the reconstruction of volume data and is

often applied in visualization approaches. We can say that this trilinear model combines different techniques to overcome the difficulties of the simple linear models. First, one usually chooses, for example, the linear tensor product spline model to perform the reconstruction of some values. Then, at each volume grid point a gradient is pre-computed by using central-differences, the Sobel operator or other schemes from the data values at nearby grid points. Once this is done, two models are constructed. The data model is determined from the data samples itself located on the volume grid points by appropriately setting the parameters of the chosen spline model by using the local neighborhood (see part II). Then, instead of computing the gradients directly from that data model by applying Bernstein-Bézier techniques (see also part II), i.e. the de Casteljau algorithm, one usually defines a different gradient model. For that model the same linear tensor product splines are used. But now the parameters are determined from the x, y and z components of the pre-computed gradient samples and the local neighborhood. This is done separately for each component of the gradient. A very similar model could be defined using linear type-6 splines. However, it is clear that both models are able to

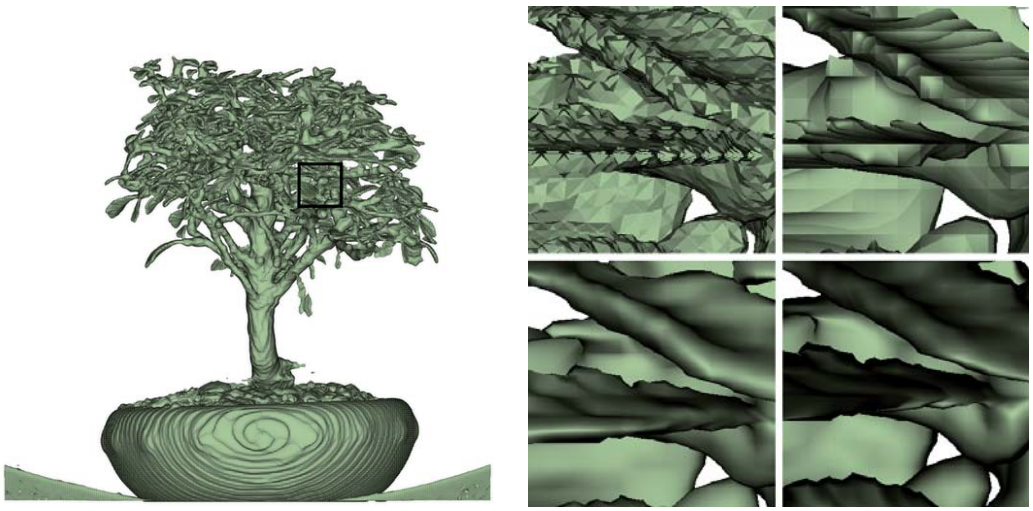


Figure 2.6: *Left: Iso-surface ($s(\mathbf{v}) = 40.0/255.0$) of Bonsai data set using linear type-6 splines. Right: A zoom into the black marked area (cf. left image) using linear type-6 (top-left) and type-0 (top-right) spline models and the trilinear model with one-side (bottom-right) and two-side (bottom-left) differences.*

generate more accurate gradients due to, for example, the central-difference technique used for derivative estimations, which in fact is derived from a second degree polynomial model.

A real world example (cf. Fig. 2.6) makes it more evident that linear splines are a bad choice for volume visualization. Even the grid structure of the volume data set becomes not so apparent in the rendered images when the eye location is far away from the object, the visualization appears a little bit rough or coarse. Once a zoom into the object is performed, i.e. the eye position is transformed near to the place of interest in world space, the regular volumetric partitions $\triangle$ and $\diamondsuit$ become clearly visible. In this sense the trilinear model generates more satisfying results.

2.4.2 Quadratic Models

Higher order splines, as discussed next, have the advantage that smooth gradients are directly available from the considered data model. That means, once all the parameters of the appropriate splines are computed from data samples of the volume grid by considering the local neighborhood, the data values as well as smooth gradients can be reconstructed anywhere in the volumetric domains Δ or Ω. There is no need to define two separate models for data and derivatives, respectively. The visual performance of the trilinear model (using the Sobel operator for gradient estimation), the quadratic tensor product splines, the quadratic, and cubic type-6 splines is shown in Fig. 2.7, 2.8, and 2.9. In the first figure we used the spherical function again and show the quality

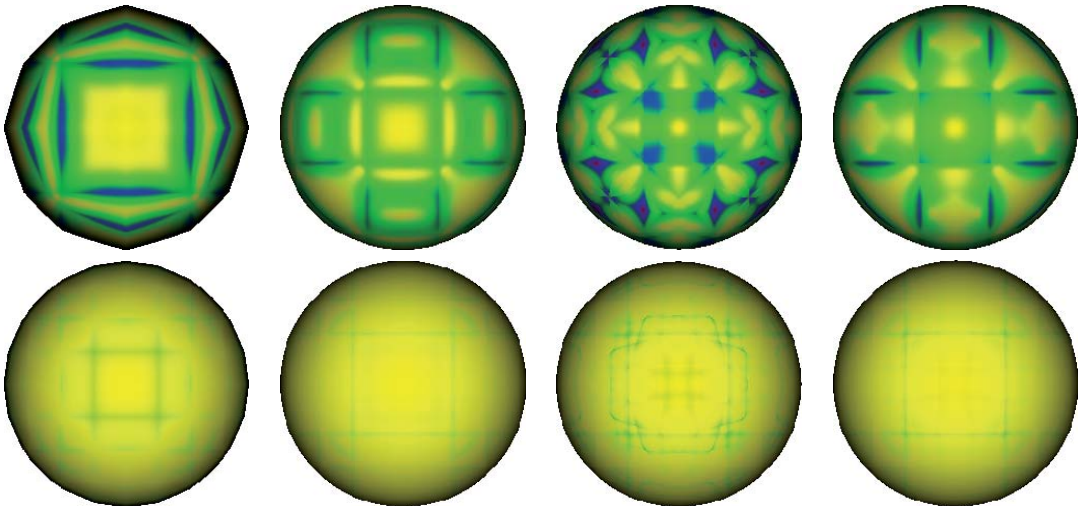


Figure 2.7: *The top and bottom rows show images rendered from* 8^3 *and* 16^3 *volume data sets, respectively, sampled from the spherical test function* f_{Sq}*. All images show the color-coded errors of the gradients obtained from the original spherical function* $f_{Sq}(\mathbf{v})$ *and from the different spline models* $s_{Sq}(\mathbf{v})$ *at the user-defined iso-value* $s_{Sq}(\mathbf{v}) = 60.0/255.0$*. The reconstruction models are (from left to right) the trilinear interpolation model (Sobel operator), the quadratic spline model on* Ω*, and the quadratic and cubic spline models on* Δ*. The threshold values are* $0.0°, a = 0.85°, b = 1.7°$*, and* $c = 2.55°$*. Hence, the red color denotes angles (errors* err*) between the two gradients which are bigger than* $2.55°$*.*

of the trilinear model (with Sobel operator) and higher order spline models. In the top left most image of Fig. 2.7 where we use a data set of 8^3 samples obtained from the spherical test function the projection or visualization of the sphere data set do not look like a circle as should be the case. Instead we clearly see an approximation of a circle by 11 line segments connecting the 12 corners. If the number of data samples increases, as in the bottom row in the considered figure, the approximation becomes more accurate. This example shows the limitations of the trilinear model. For high zoom-in factors into data sets (see also Fig. 2.9) or for difficult test functions where only a few data samples are considered for reconstruction, the trilinear model generates such angular structures at the boundaries of the object's surface as, for example, the sphere or the leaves of the Bonsai data set. Quadratic or cubic spline models on type-0 or type-6 partitions do not have the difficulty, they generate smooth boundaries because of the higher degree

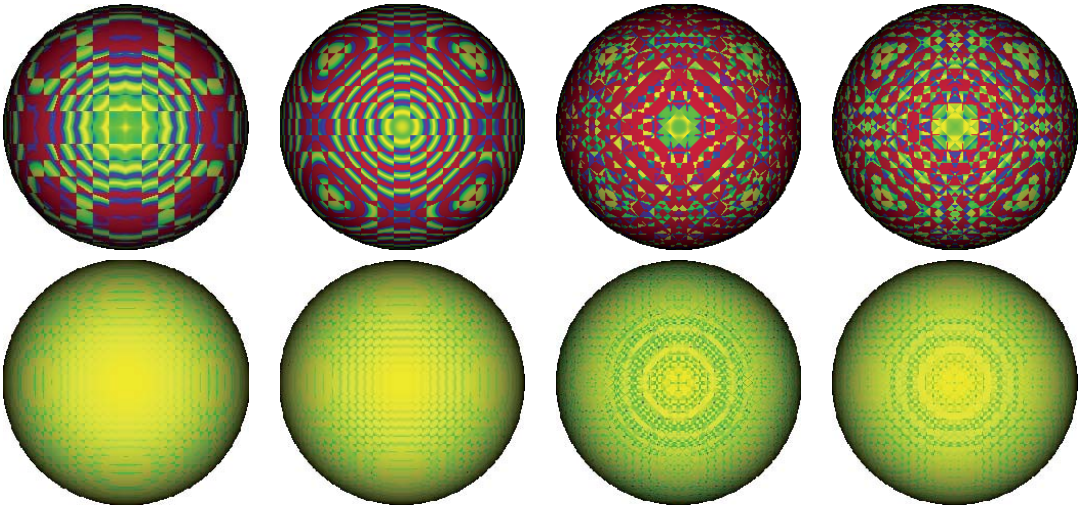


Figure 2.8: *The top and bottom rows show images rendered from* 8^3 *and* 16^3 *volume data sets, respectively, sampled from the spherical test function* f_{Sq}*. All images show the color-coded errors of the mean curvatures obtained from the original spherical function* $f_{Sq}(\mathbf{v})$ *and from the different spline models* $s_{Sq}(\mathbf{v})$ *at the user-defined iso-value* $s_{Sq}(\mathbf{v}) = 60.0/255.0$*. The reconstruction models are (from left to right) the trilinear interpolation model (Sobel operator), the quadratic spline model on* Ω*, and the quadratic and cubic spline models on* Δ*. The error thresholds are* $0, a = 0.0002125, b = 0.000425$*, and* $c = 0.0006375$*.*

polynomials used for reconstruction.

Meanwhile, the gradients are another main issue for volume visualization. They are used as input for the considered illumination model and have to be chosen carefully. We have seen in case of linear splines what happens when the first derivatives are not continuous. In case of using the trilinear model with the Sobel operator, the gradients can be reconstructed quite well. It is clear because this operator results from a quadratic model. Therefore, at the neighborhood in the volume grid the pre-computed gradients behave like a piecewise linear function, in other words, they are continuous over the whole domain. Even more, the gradients are further smoothed across their orthogonal directions by a discrete gaussian kernel. However, the gradients obtained by trilinear interpolation of the Sobel operator do not perform as well as the gradients directly computed from the quadratic model on type-0 partitions or the cubic model on type-6 partitions. This can be observed in Fig. 2.7. One can see from the image corresponding to the trilinear model, that it contains more blue coded areas compared to the images corresponding to the type-0 quadratic and type-6 cubic models, respectively. Further, we have already discussed in part II that the quadratic type-6 spline model has some minor drawbacks. The gradients are continuous across faces of the unit cubes Q in $\diamondsuit$ (i.e. over the planes defined in Equ. 2.3), but are not always across the faces of the tetrahedra T in $\triangle$ (i.e. over the planes defined in Equ. 3.1). For smooth functions the gradients are continuous across all the planes defining the tetrahedral partition. However, this property of that quadratic type-6 splines reveals in a non continuous variations of colors used to encode the error of gradients as can be seen in Fig. 2.7 (on the top row the second image from the right side). At some location on the sphere, where the colors

become red, the errors are moderately high.
In a practical example, that means in Fig. 2.9 the iso-surface of the leaves of the Bonsai tree looks a little bit wavy or jagged, which comes from the discontinuous gradients as well. This phenomenon can not be observed on the other images, what means, that the corresponding data reconstruction models perform quite well here (even the trilinear model with the Sobel operator). A comparison of the quadratic type-6 model with the trilinear model shows that the boundary of the iso-surface of the different leafs is more smooth due to the piecewise quadratic polynomials used for data reconstruction (a similar observation has been done above for the sphere test data set with 8^3 samples). Hence, from this point of view the quadratic type-6 model is very similar to the quadratic type-0 and cubic type-6 splines. However, the difference between all the models becomes most visible for high frequency areas (as e.g. the leaves of the Bonsai tree) where a small number of data samples is used for feature representation only. No matter of the applied reconstruction model, the underlying grid structures or partitions $\triangle$ or $\diamondsuit$ of the corresponding domains Δ or Ω, respectively, are always clearly visible in the final images in Fig. 2.7 and Fig. 2.8.
Finally, the visual results of the second derivatives are given in Fig. 2.8. Here we have displayed the errors of the mean curvatures computed from the Hesse matrices and the first derivatives obtained from the considered model and the corresponding original function. However, it is clear that C^1 reconstruction models can not represent second derivatives of arbitrary functions in a smooth fashion. Hence, usually they can not be applied in non-photorealistic volume rendering as, for example, curvature or silhouette enhancements. Nevertheless, for increasing data sizes the errors of the mean curvature become even smaller (see bottom row in Fig. 2.8 and the numerical results). That means, we can apply these models in real world volume rendering applications, where we often deal with data sets of size more than 256^3 samples, to approximately display regions of high curvature or the silhouettes of an object (examples are given in part I). Note, the look-up table used to determine the corresponding color for a computed mean (or gaussian) curvature(s) should contain smooth transitions only.

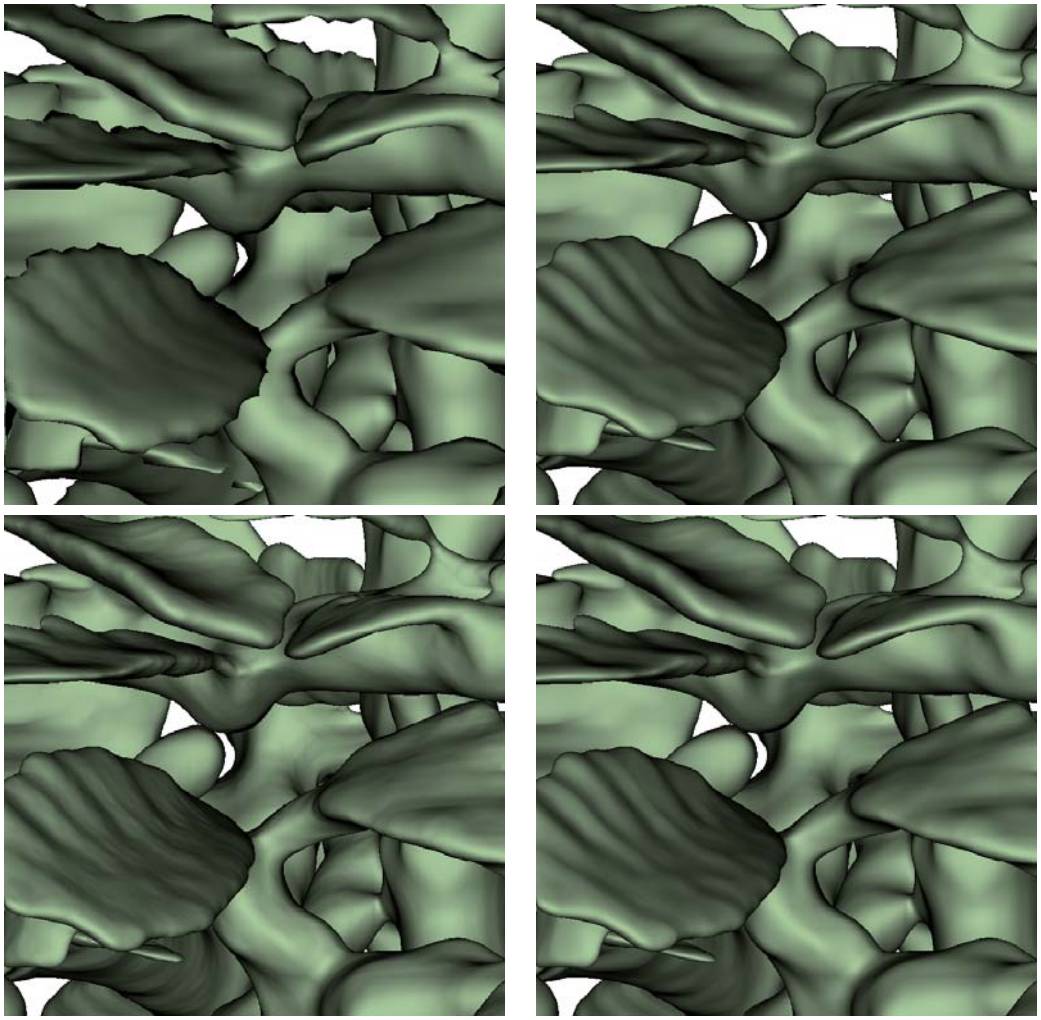


Figure 2.9: *Iso-surface ($s(\mathbf{v}) = 40.0/255.0$) of Bonsai data set using (from top-left to bottom-right) type-0 trilinear (with Sobel operator), type-0 quadratic, type-6 quadratic, and type-6 cubic spline models, respectively.*

2.5 Performance

In the following we discuss the performance of the spline models. These Bernstein-Bézier methods can be considered as two-step techniques. That means, in the first stage the Bernstein-Bézier coefficients are computed to completely determine the considered spline model. In the second stage the coefficients are usually used to reconstruct data values and derivatives at arbitrary positions in the volumetric domain. These Bernstein-Bézier methods can also be implemented as one-step techniques. Instead of determining the coefficients first, the values and derivatives are directly computed from the local data neighborhood. Hence, for the measurement of the performance of the different spline models we consider a unit cube Q and two different tests.
More specifically, in the first test we measure the average time on how fast we can determine the coefficients for a unit cube using the appropriate averaging formula of the different splines. This allows us to estimate lower bounds of the time necessary during the preprocessing stage, e.g. the time needed to pre-compute all Bernstein-Bézier coefficients for a considered data set. For this test we consider again data sets of 16^3, 32^3, 64^3 and 128^3 samples taken from the Marschner-Lobb test function and we compute the coefficients for all the unit cubes. For each cube we measure the computation time of the coefficients and determine from that the average time as well the variance for one unit cube. In the second stage of the first test we use the same data sets and about 240 randomly distributed positions according to a unit cube (see also the numerical tests). We reconstruct values as well as derivatives at these positions and determine in a similar way the average time and variance for the reconstruction of a data value and its derivatives at one position only (cf. Tab. 2.4 and Tab. 2.5). This allows us to give lower bounds for the time necessary during the reconstruction or visualization stage, e.g. the time needed to render an image from an arbitrary viewing position.

Model	BBC	Values	Gradients	Hesse matrices
type-0 linear	3.661[±0.486]	0.052[±0.002]	0.092[±0.001]	0.144[±0.003]
type-0 quadratic	4.217[±0.421]	20.03[±0.169]	20.05[±0.173]	20.06[±0.191]
type-0 trilinear	4.246[±0.389]	0.052[±0.002]	0.103[±0.002]	0.202[±0.003]
type-6 linear	3.671[±0.491]	0.075[±0.001]	0.098[±0.002]	0.119[±0.003]
type-6 quadratic	3.715[±0.446]	0.098[±0.004]	0.122[±0.003]	0.258[±0.007]
type-6 cubic	4.509[±0.315]	0.156[±0.005]	0.174[±0.004]	0.309[±0.008]

Table 2.4: *For each model the performance in microseconds (μs) is given. In the first column denoted as BBC the times needed to pre-compute all Bernstein-Bézier coefficients for a unit cube are given. In the adjacent columns denoted as Values, Gradients, and Hesse matrices the times needed to reconstruct the value, the gradient, and the Hesse matrix at a specified location are shown.*

In our second test we measure the copy time, i.e. the time needed to copy a 27-neighborhood of the volume grid into a local buffer. Then, we use the values stored in the local buffer to determine the on-the-fly reconstruction time, i.e. we directly evaluate the model (determine the values, gradients, and the Hesse matrices) at a specified location (cf. Tab. 2.6 and Tab. 2.7). Here, we do not pre-compute any Bernstein-Bézier coefficients, hence it is a good test to estimate the time necessary for on-line recon-

Model	BBC	Values	Gradients	Hesse matrices
type-0 linear	$44+, 8*$	$7+, 14*$	$18+, 30*$	$39+, 54*$
type-0 quadratic	$108+, 54*$	$270+, 1053*, 324/$	$270+, 1053*, 324/$	$270+, 1053*, 324/$
type-0 trilinear	$68+, 32*$	$7+, 14*$	$28+, 56*$	$70+, 92*$
type-6 linear	$38+, 14*$	$3+, 4*$	$12+, 16*$	$12+, 16*$
type-6 quadratic	$154+, 66*$	$15+, 20*$	$24+, 32*$	$77+, 104*$
type-6 cubic	$776+, 342*$	$45+, 60*$	$54+, 72*$	$107+, 144*$

Table 2.5: *For each model the arithmetic operations (the number of additions $+$, multiplications $*$, and exponentials $/$) are given. In the first column denoted as BBC the arithmetic operations needed to pre-compute all Bernstein-Bézier coefficients for a unit cube are given. In the adjacent columns denoted as Values, Gradients, and Hesse matrices the arithmetic operations needed to reconstruct the value, the gradient, and the Hesse matrix at a specified location are shown.*

struction and visualization as well as to compare with two-step techniques. It is clear that such one-step techniques perform unnecessary and repeated computations for local data reconstruction, i.e. when we consider several locations within the same unit cube or tetrahedron. This can be thought as, repeatedly computing only the necessary Bernstein-Bézier coefficients within the unit cube or better the tetrahedron for the reconstruction of values at a specified location. This has to be performed for several times of course. The two-step techniques pre-compute all necessary coefficients for a cube or tetrahedron first, and then reuse them for the reconstruction of the data at several specified locations. Hence, some arithmetic operations can be saved using the later technique.

Model	Copy	Values	Gradients	Hesse matrices
type-6 linear	$3.647[\pm 0.486]$	$0.103[\pm 0.002]$	$0.122[\pm 0.002]$	$0.143[\pm 0.003]$
type-6 quadratic	$3.651[\pm 0.484]$	$0.226[\pm 0.002]$	$0.243[\pm 0.003]$	$0.386[\pm 0.005]$
type-6 cubic	$3.656[\pm 0.480]$	$0.434[\pm 0.005]$	$0.450[\pm 0.005]$	$0.567[\pm 0.007]$

Table 2.6: *For each type-6 model the performance in microseconds (μs) is given. In the first column denoted as Copy the time needed to copy the 27-neighborhood associated with a unit cube are given. In the adjacent columns denoted as Values, Gradients, and Hesse matrices the time needed to reconstruct the value, the gradient, and the Hesse matrix at a specified location are shown.*

Before we are going to discuss our performance results of the different spline models given in four different tables, let us make some notes first. Since our focus lies on type-6 splines, we have only applied the first performance test for type-0 splines, i.e. we have measured the pre-computation and the reconstruction time only. The number of operations and the time shown in the last columns of each table denoted as *Hesse matrices* include the amount of operations and time needed to reconstruct the values, first and second derivatives, where the columns denoted as *Gradients* include the amount of operations and time needed to reconstruct the values and the first derivatives and so on. For the trilinear method we have applied central differences to compute the gradients (or the Bernstein-Bézier coefficients). The number of operations (or the performance) needed

Model	Copy	Values	Gradients	Hesse matrices
type-6 linear	27-N	$18+, 7*$	$27+, 19*$	$27+, 19*$
type-6 quadratic	27-N	$108+, 44*$	$117+, 56*$	$170+, 128*$
type-6 cubic	27-N	$270+, 135*$	$279+, 147*$	$332+, 219*$

Table 2.7: *For each type-6 model the on-the-fly arithmetic operations (the number of additions + and multiplications ∗) are given. In the first column denoted as Copy the 27-neighborhood of data values corresponding to a unit cube is copied into a local buffer only. In the adjacent columns denoted as Values, Gradients, and Hesse matrices the arithmetic operations needed to reconstruct the value, the gradient, and the Hesse matrix at a specified location are shown.*

for gradient estimation using the Sobel operator is be about $452+, 464*$ ($5.236[\pm 0.582]$). Whereas the number of operations during the reconstruction phase remains constant no matter of the gradient-estimation operator used. For a better comparison of the different models we determine the eight coefficients of the linear type-0 splines by means that they become approximating splines (see part II), i.e. by using a 27-neighborhood in the volume grid. For example, to compute one Bernstein-Bézier coefficient located at a corner of the unit cube 7 additions and 1 multiplication are needed. Hence, in total, i.e. to compute all eight coefficients, 56 additions and 8 multiplications would be necessary. A simple optimization of the number of arithmetic operations results in 44 additions and 8 multiplications. This kind of optimizations we have performed for each model except the type-0 quadratic splines, hence this quadratic spline model is not well optimized yet and lead to a bad performance. The trilinear model is basically the same as the type-0 linear splines. That means, instead of using 8 Bernstein-Bézier coefficients for a unit cube only, 32 coefficients are generated (8 data coefficients for the data model and 24 gradient coefficients for the gradient model, i.e. 8 for each component x, y, and z of the gradient). For linear, quadratic, and cubic type-6 splines we pre-compute 15, 65, 175 coefficients for a unit cube for the first test, respectively, and later evaluate the spline models at several locations within the cube using the pre-computed Bernstein-Bézier coefficients. For the second test we directly evaluate the appropriate spline model from the given data values of a 27-neighborhood in the volume grid.

However, from the second columns (denoted as *BBC* and *Copy*) of Tab. 2.4 and Tab. 2.6 we can observe that most of the time is spent for copying the data of a 27 neighborhood of the volume grid into a local buffer. The time needed to pre-compute the necessary Bernstein-Bézier coefficients for a unit cube is quite small for all spline models. Only the cubic type-6 splines need approximately the same time for their arithmetic operations (i.e. to compute all coefficients for a unit cube) as the time necessary to copy the 27-neighborhood into a local buffer. In the third, fourth, and fifth column of the above tables we can further observe that once the coefficients are pre-computed for a unit cube the reconstruction time of data values, gradients and Hesse matrices can be approximately halved for type-6 splines. This becomes immediately clear looking at Tab. 2.5 and Tab. 2.7, where we give the number of corresponding arithmetic operations. It can be observed that each model needs about twice as many multiplications when the Bernstein-Bézier coefficients are not pre-computed. So the doubled reconstruction time is not surprising. We can not pre-compute the coefficients for the whole data set because we would have to store too much data. However, considering one cube only, a small array

of size 15, 65, or 175 floating point cells have to be used to be able to pre-compute and store the necessary Bernstein-Bézier coefficients using the two-step technique. In case of the one-step technique always a buffer of size 27 floating point cells is required. The question now is, if that one-step technique is applicable and efficient? We can say, no, since the sum of pre-computation time (shown in the column denoted as *BBC*) and the corresponding reconstruction time (shown in the columns denoted as *Values*, *Gradients*, and *Hesse matrices*) of Tab. 2.4 is mostly smaller than the sum of copy time (shown in the column denoted as *Copy*) and the corresponding reconstruction time of Tab. 2.6.
Finally, Tab. 2.4 allows us to estimate the pre-processing time for the pre-computation of all coefficients for the whole data set, i.e. when one likes to do this. However, in volume rendering often only about 10% of all cubes of the original data are classified by opacity tables to be visualized. Hence, the pre-processing of a N^3 volume would take about $0.10N^3x$ microseconds (μs), where x represents the appropriate time values from the above table needed to compute the coefficients for one cube only. More specifically, if $N = 256$ then the linear, quadratic, and cubic type-6 splines would need at least about 6.14, 6.22, and 7.56 seconds according to the above performance test to generate a pre-classified data set with pre-computed Bernstein-Bézier coefficients. Neglecting the overhead for organizing the coefficients in an appropriate way and classification of the data itself. The time to generate an image, of e.g. an iso-surface of a volume, can be estimated as well. For this, a final image (or viewport) of size $M \times N$ pixels with associated rays is considered, where about 50% of all rays will intersect the iso-surface of the function only. Further, linear, quadratic and cubic type-6 spline models are used which require to reconstruct 2, 3 and 4 data values inside a considered tetrahedron to further generate local piecewise polynomials of total degree one, two, and three, respectively. These polynomials can be used in root-finding algorithms to compute the iso-surface. However, the lower bound for the rendering time of an image containing the iso-surface of a function would be about $0.5(2MNx+1MNy)$, $0.5(3MNx+1MNy)$, and $0.5(4MNx + 1MNy)$ for linear, quadratic, and cubic splines, respectively. Once more, x and y represent the appropriate time values needed for one data value or gradient reconstruction from the above table. Hence, the lower time bound for rendering a 512×512 image of the iso-surface of a volume is about 32.50, 54.53, and 104.60 milliseconds for linear, quadratic and cubic splines, respectively. This results in a maximum of 30, 18, and 9 frames per second. Note that since we do not know in advance if there occurs an iso-surface intersection of a ray and the data, we often have to perform these computations until one iso-surface is found. Then, the early-ray termination flag can be set, such that all adjacent tetrahedra visited by the considered ray can be skipped and hence, the iso-surface computations as well. Also note that the estimation of the frame rates does not include the time needed for the root-finding process itself, it contains only the reconstruction time of necessary values and gradients by the considered model, which are further used in the root-finding algorithm and for the Phong illumination algorithm. From this point of view it is very difficult to achieve the above estimated frame rates. However, we have applied the shear-warp approach for volume visualization and we will see in the following sections how near we will come to that theoretical frame rates.

3 Results on Shear-Warp

In this section we give the results on our new shear-warp algorithm. We first consider the parallel projection case and compare our new method with the original algorithm [LL94] [Lac95] as well as the well known shear-warp deluxe approach using intermediate slices [SM02] for rendering. Therefore we have re-implemented both algorithms. Then, we show results of our new method by using different spline models for data reconstruction. Finally, the perspective case is considered and the efficiency of the new data structures used to realize a similar approach as for the parallel projection case is discussed.

3.1 Parallel Projection Case

3.1.1 Equidistant Sampling

The quality of the original shear-warp approach heavily depends on the viewing direction (see part III). For an oblique view onto the volume staircasing effects are clearly visible (see Fig. 3.1) due to the increased sampling distance in object space. As discussed in

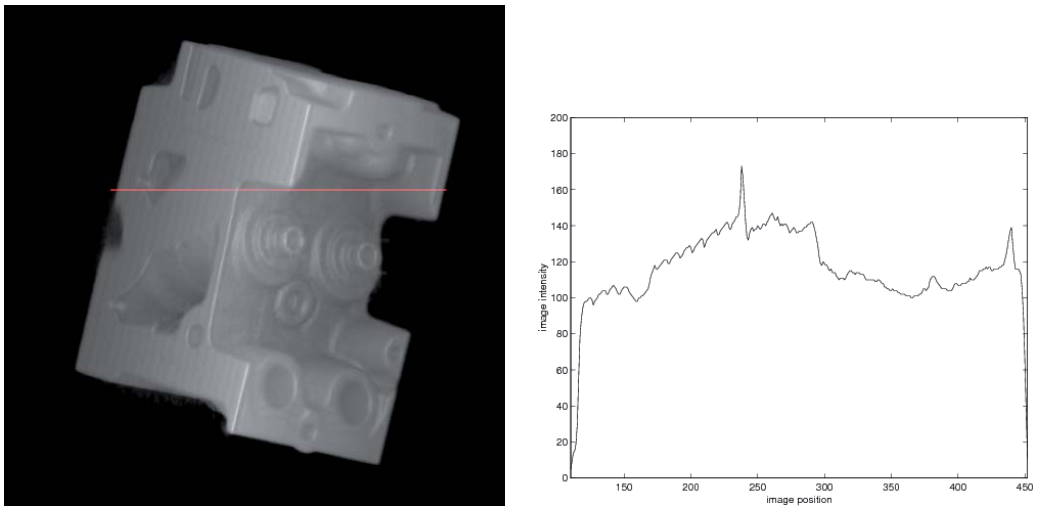


Figure 3.1: *Left: An oblique view onto the Engine volume data set, where staircasing artifacts are clearly visible due to the increased sampling distance. Right: The intensity profile corresponding to the horizontal red line in the left image. Here, the staircasing artifacts appear as jagged structures, visible in the left area in this profile.*

part III as well, a partial solution to the problem of the varying sampling distance is to introduce intermediate slices. This reduces the maximal sampling distance to $\sqrt{3}/(N+1)$ with N the number of intermediate slices and allows us to stay within the Nyquist limit of 0.5 (for rectilinear, volumetric domains with unit cube size of 1^3). However, this approach

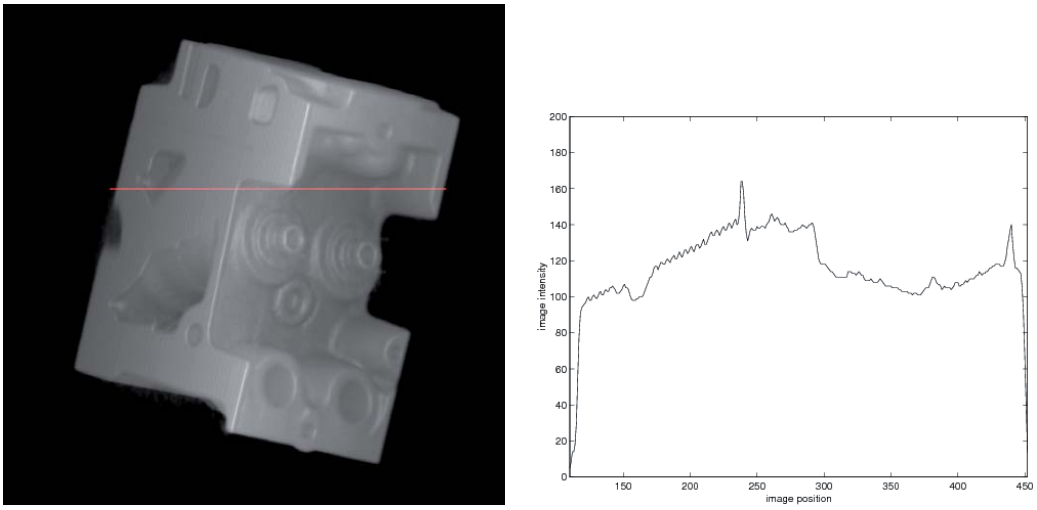


Figure 3.2: *Left: An oblique view onto the Engine volume data set (the same configuration as in Fig. 3.1), where staircasing artifacts are still visible even using intermediate slices. Right: The intensity profile corresponding to the horizontal red line in the left image. Here, the staircasing artifacts appear as jagged structures, visible in the left area in this profile. The jagged structures are less suspicious than in Fig. 3.1.*

generates less suspicious staircasing artifacts. In general arbitrary many intermediate slices can be rendered to decrease the sampling distance and to improve the quality of the resulting image. Note that the opacity has to be corrected according to the varying sampling distance as well. However, once we have decided to render one intermediate slice, e.g. at the object space location $t = 0.5$ between two original (encoded) slices located at position $t = 0.0$ and $t = 1.0$, respectively, the Nyquist limit can be satisfied. The number of intermediate slices depends on the underlying data reconstruction model used. That means, for the trilinear model which generates piecewise polynomials of total degree 3 for oblique views onto the volume, one should in principle apply two intermediate slices for a correct reconstruction of the data (even not considering the volume rendering integral correctly). Hence, one intermediate slice is not enough.

In our new approach we solve the problem discussed above in a different way. Instead of using intermediate slices, we have developed a new data structure called *column* template. This (see part III) has two main advantages. The first benefit is that an equidistant spacing of the sampling points in object space can be defined, so that we do not need to correct the opacity during rendering as in the approach discussed above. Then, since a parallel projection is considered here, a coherency of the rays in object space can be observed. Thus, we can pre-compute all sampling positions within the volume domain along all the different rays and store the sampling locations in our template. The second main profit is that, using this data structure two of the three run-length encoded volume data sets have never to be used during rendering and thus, have not to be encoded nor stored anymore.

However, a cell or *cube* template which is a part of the *column* template and represents a unit cube needs exactly 208 Bytes to store the necessary information for rendering (i.e. the position of the sampling points located inside a unit cube and their projection

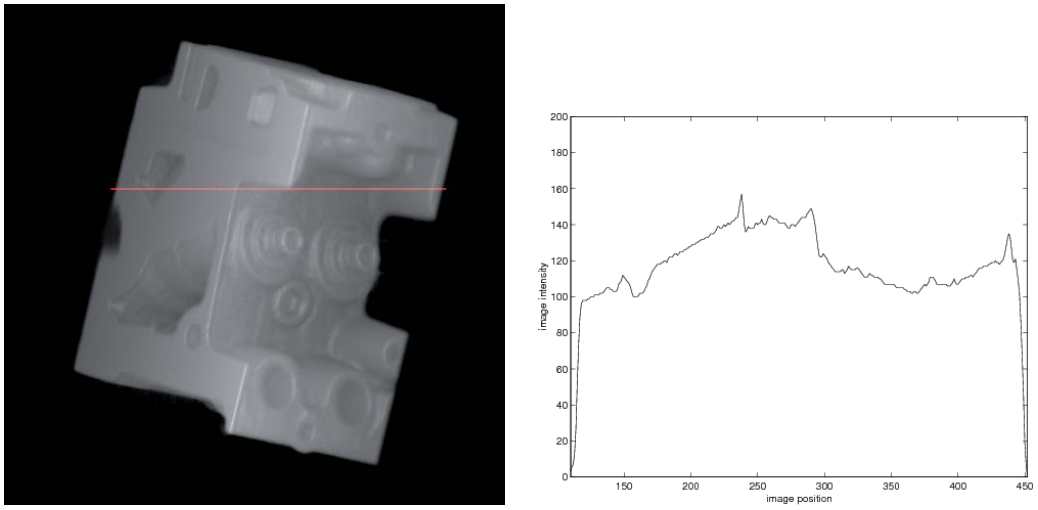


Figure 3.3: *Left: An oblique view onto the Engine volume data set (the same configuration as in Fig. 3.1 and Fig. 3.2), where staircasing artifacts are still visible even using trilinear interpolation and correct sampling (i.e. the sampling distance is fixed to* $1/2$ *and does not depend on the viewing direction anymore). Right: The intensity profile corresponding to the horizontal red line in the left image. Here, the staircasing artifacts appear as jagged structures as well, visible in the left area in this profile. The jagged structures are less suspicious than in Fig. 3.1 and Fig. 3.2.*

information). Hence, for a volume with a maximal size of K unit cubes along any of its three main axes the size of the whole template becomes exactly $208K$ bytes huge. More specifically, for a data set of size $(L \times M \times N) = (256 \times 128 \times 64)$ voxels the template needs a maximum of $208K = 208 \max(L, M, N) = 52$ kilobytes of memory for the pre-computed information. This data structure fits easily into the second level cache, so that a loss of performance of our new shear-warp approach due to this template is unlikely. We

Size	Time
64	77.07[±35.25]
128	85.68[±11.02]
256	165.52[±19.64]
512	333.51[±45.28]
1024	652.18[±77.24]

Table 3.1: *Pre-processing times in microseconds (μs) of the different sized* column *templates for equidistant sampling.*

have verified this by considering 1000 randomly generated local ray start positions and directions according to different sized unit columns. In other words, for each random ray a *column* template representing a unit column of pre-defined size has been recomputed and the time for this process has been measured. In this way we have obtained 1000 measurements from which we have calculated the average time (and variance) for the generation of one *column* template only. We have done this test for different sized

templates. In Tab. 3.1 we show the results. However, it is obvious that if the size of the template is doubled, the time to create the template grows by approximately the same factor. The average time to compute the information of one unit cube represented by a *cube* template is approximately 0.6 microseconds. Hence, the pre-computation of the *column* template in our new shear-warp approach does not affect the speed of the total rendering algorithm.
In the following (see Tab. 3.2) we give the performance of our shear-warp approach compared to the original method and the shear-warp deluxe algorithm which uses intermediate slices (all methods are our own implementations). We render the data sets from 100 randomly generated viewing directions and determine the average time for one frame or the number of frames per second the considered method can achieve. The per-

Data Set	Size	SWO	SWI	SWN
Engine	$256^2 \times 128$	24.0	16.5	5.1
Teapot	$256^2 \times 178$	20.5	14.6	4.0
Bonsai	256^3	19.5	14.3	3.9

Table 3.2: *Average frame rates of the different rendering methods in frames per second. SWO: Shear-Warp without intermediate slices. SWI: Shear-Warp with intermediate slices. SWN: Our new rendering method with trilinear interpolation using the* column *template. In each method we have applied a linear opacity transfer function.*

formance loss of the shear-warp deluxe algorithm by a factor of approximately one and a half can be explained by the increased number of intermediate slices which are rendered (a factor of two according to the number of slices used in the original method). Another reason is that we consider four run-length encoded volume scan-lines to generate one intermediate scan-line on-the-fly which is not stored but directly rendered to the appropriate intermediate image scan-line. This leads to suboptimal space leaping considering the original method. Of course, we could use only two volume scan-lines at a time as in the original approach. This would result in another intermediate image buffer which had to be used to store intermediate results which would be further composited into original intermediate image (see part III and the original paper [SM02]). However, that is the reason why we decided to apply the former approach using four volume scan-lines. Our new algorithm is about a factor of five slower than the original method. The reasons are very similar as before. First, we always consider four volume scan-lines during rendering (not only for the rendering of the intermediate slices) to generate one intermediate image scan-line. Second, we always apply the real trilinear data reconstruction model (in the deluxe approach we switch between a bilinear reconstruction within the original encoded slices and a trilinear reconstruction within the intermediate slices). Third, in two cases, i.e. for two of the three main viewing directions, two intermediate image scan-lines are considered at a time for correct early-ray termination and compositing. This is suboptimal according to the original as well as the deluxe method where only one scan-line has to be taken into account. In case of the third viewing direction we also have to apply early-ray termination in an oblique fashion within at most three dynamically encoded (oblique) intermediate image scan-lines (cf. discussion part III). Fourth, the number of sampling points to reconstruct the data and to evaluate the line integral (i.e. the volume rendering integral) along the different viewing rays varies according to the

viewing direction. In other words, if the ray direction is parallel to one main viewing axis of the volume we need as many sampling points as the original method. Otherwise the number of samples varies with the ray direction by a factor in the interval $[1, \sqrt{3}]$. However, the original method has a varying sampling distance where the number of samples is constant, in our new method the distance is fixed and the number of samples varies therefore. As mentioned before, we do not have to correct the opacity because of the fixed sampling distance.

3.1.2 Accurate Sampling

In the previous section we have discussed the results of our new shear-warp approach based on using a parallel projection matrix and an equidistant sampling of the data along different rays. For this we have applied a pre-computed *column* template which represents the information of all equidistant samples taken along all rays going through the object. The main drawback of such an equidistant sampling is that no matter of the data reconstruction model the sampling distance remains constant. Of course, one could decrease the distance between two samples to increase the accuracy and thus the visual result of the final image. But the question is, how can we do this in an optimal way? The answer in the context of our new shear-warp approach is a further development of the *column* template. That means, instead of storing the equidistant sampling positions along the rays we pre-compute and store the intersection positions of the rays with the different planes of the rectangular $\diamondsuit$ or tetrahedral partitions $\triangle$. The spacing between the intersection positions is then not necessarily equidistant. However, this allows us to correctly define and to compute univariate piecewise polynomials along the rays with respect of the unit cubes or the tetrahedra using the appropriate data reconstruction models. In other words, considering for example the trilinear model which generates univariate polynomials of total degree 3 along a ray according to a unit cube, we need to take at least two samples in the interior of an unit cube to be able to define a third degree polynomial and to correctly evaluate the iso-surface problem or the full volume rendering equation. Having the intersections of all rays with the unit cubes (represented in our template) this can be done easily and quickly. With the further development of the *column* template where we store the intersection positions of the rays and the unit cubes or the tetrahedra, we are able to implement fast as well as more accurate volume rendering methods, because only as many sample positions are generated and stored as are necessary for a considered data reconstruction model and volume partition.
However, as before a *cube* template which is a part of the *column* template and represents a unit cube (or all tetrahedra within a cell) needs exactly 516 Bytes to store the necessary information for rendering (i.e. the intersection positions of the rays and the unit cubes or tetrahedra and their projection information). Hence, considering our example from above about twice as much memory is needed now, i.e. $512 \max(L, M, N) = 128$ kilobytes of memory for the pre-computed information is necessary. Note that the size of the *column* data structure can be further reduced if parallel projections are considered only. At this time we use the same data structure for perspective rendering as well. That means, its dimensions are taken such that we can deal with both projection types. Nevertheless, even this data structure fits easily into the second level cache. We have applied the same tests as above, i.e. by considering 1000 randomly generated local ray start positions and directions according to different sized unit columns. In Tab. 3.3 we

Size	Time ($\diamondsuit$)	Time ($\triangle$)
64	56.82[±24.45]	193.35[±86.39]
128	64.93[±8.33]	240.03[±46.00]
256	125.13[±15.06]	437.34[±48.65]
512	246.72[±30.08]	872.83[±97.84]
1024	494.34[±62.94]	1741.78[±193.60]

Table 3.3: *Pre-processing times in microseconds (μs) of the different sized* column *templates using accurate sampling for rectangular partitions $\diamondsuit$ (left) and for tetrahedral partitions $\triangle$ (right).*

show the results. As before, it is obvious that if the size of the template is doubled, the time to create the template grows by approximately the same factor. The average time to compute the information of one unit cube represented by a *cube* template considering the rectangular $\diamondsuit$ or tetrahedral partitions $\triangle$ is approximately 0.5 or 1.7 microseconds, respectively. However, due to the additional six intermediate planes which are used to split the rectangular partition into the tetrahedral partition we have an increase of computation time for a *cube* or *column* template by a factor of approximately 3.5. However, it is not surprising that the pre-computation time does not affect the speed of the total rendering algorithm as before. But one may be astound that it takes more time to generate the *column* data structure for equidistant sampling (discussed above) than for the accurate sampling where intersection computations have to be carried out. The reason is simple, we have derived the algorithm for equidistant sampling from that method discussed here, hence there is a post processing required in case of equidistant sampling and thus the computation of the information for one cell takes about 0.1 microseconds more.
Next, we give the performance (see Tab. 3.4 and Tab. 3.6) of our new shear-warp implementation using this template allowing an accurate sampling of the volume data. The mentioned tables show the results obtained for two of the three main viewing directions considered in the shear-warp method where the volume scan-lines are processed in a parallel manner to the intermediate image scan-lines. In Tab. 3.5 and Tab. 3.7 we give the rendering times obtained for the 3rd main viewing direction, i.e. when the volume scan-lines have to be processed in a perpendicular manner according to the intermediate image. In this case the dynamically encoded image scan-lines pass obliquely through the intermediate image. This processing order results from the projected volume scan-lines which go off obliquely through the image. For the reconstruction of the data we apply our spline models as discussed in part II. It is clear from the results that we obtained for the splines (see Sec. 2) which models perform best in terms of time or reconstruction. Nevertheless, we didn't give the rendering performance of our shear-warp implementation using this accurate sampling with conjunction to our spline models up to now. For this we proceed as above. We render the different data sets from 1000 randomly generated viewing directions and measure the average time for the rendering of one frame. This can be recomputed to the number of frames per second our algorithm is able to achieve considering a specified data model. We decide to apply iso-surface rendering for this test because of two reasons. First, iso-surface rendering can be considered as full volume rendering with a threshold opacity transfer function. Second, the conditions of an iso-surface test are often easier to reproduce than the conditions of a full volume ren-

dering test. Even we have decided to render surfaces of objects this performance test can be compared to the results given before (i.e. to these obtained for the shear-warp deluxe approach and our method using trilinear interpolation at equidistant spaced sampling points along rays, in both methods a linear opacity transfer function is applied).

Data Set	Size	SW L	SW Q	SW T
Engine	$256^2 \times 128$	7.29	1.07	7.18
Teapot	$256^2 \times 178$	5.77	1.07	5.78
Bonsai	256^3	3.46	0.65	3.33

Table 3.4: *The iso-surface rendering performance (in frames per second) of our new shear-warp approach along the first two main viewing directions by considering accurate sampling for linear, quadratic, and trilinear spline models on $\diamondsuit$ (from left to right). The iso-values for the different data sets (Engine, Teapot, and Bonsai) are* 80*,* 50*, and* 40*, respectively.*

Data Set	Size	SW L	SW Q	SW T
Engine	$256^2 \times 128$	4.95	1.02	4.90
Teapot	$256^2 \times 178$	4.26	1.00	4.26
Bonsai	256^3	2.73	0.61	2.65

Table 3.5: *The iso-surface rendering performance (in frames per second) of our new shear-warp approach along the 3rd main viewing direction by considering accurate sampling for linear, quadratic, and trilinear spline models on $\diamondsuit$ (from left to right). The iso-values for the different data sets (Engine, Teapot, and Bonsai) are* 80*,* 50*, and* 40*, respectively.*

Data Set	Size	SW L	SW Q	SW C
Engine	$256^2 \times 128$	6.36 (4.29)	4.00 (1.93)	1.83 (0.85)
Teapot	$256^2 \times 178$	5.05 (3.20)	3.13 (1.34)	1.27 (0.58)
Bonsai	256^3	3.04 (1.80)	1.75 (0.71)	0.66 (0.30)

Table 3.6: *The iso-surface rendering performance (in frames per second) of our new shear-warp approach along the first two main viewing directions by considering accurate sampling for linear, quadratic, and cubic splines on $\triangle$ (from left to right). The iso-values for the different data sets (Engine, Teapot, and Bonsai) are* 80*,* 50*, and* 40*, respectively.*

However, for this test we also decide not to pre-compute any spline coefficients. We simply organize the volume data in a linear fashion according to the linearly run-length encoded data sets in the shear-warp approach. The pre-processing of the data takes about 9.8, 13.8, and 19.8 seconds for Engine, Teapot, and the Bonsai, respectively. During rendering the local neighborhood corresponding to a unit cube is read from the linearly organized data array or directly accessed from the volume grid (then the pre-processing time can be omitted, but the read operation will take a little more time), copied into a local buffer, and the necessary coefficients for the considered spline model are computed on-the-fly. There are two possible solutions as mentioned in Sec. 2. First, for linear, quadratic, and cubic splines on $\triangle$ we determine the 15, 65, and 175 coefficients

Data Set	Size	SW L	SW Q	SW C
Engine	$256^2 \times 128$	4.15 (2.82)	2.43 (1.18)	0.96 (0.45)
Teapot	$256^2 \times 178$	3.62 (2.25)	2.08 (0.94)	0.81 (0.38)
Bonsai	256^3	2.35 (1.36)	1.28 (0.59)	0.40 (0.17)

Table 3.7: *The iso-surface rendering performance (in frames per second) of our new shear-warp approach along the 3rd main viewing direction by considering accurate sampling for linear, quadratic, and cubic splines on $\triangle$ (from left to right). The iso-values for the different data sets (Engine, Teapot, and Bonsai) are* 80, 50, *and* 40, *respectively.*

within a unit cube from the 3^3 local neighborhood of data values, respectively, and later on evaluate the considered spline model (i.e. determine the values and derivatives) for all ray segments going through that considered unit cube. Second, we do not expand the local neighborhood as before into all coefficients which are necessary for the considered unit cube. Instead we use the data from the appropriate local neighborhood to compute only the coefficients needed for the current considered tetrahedron, i.e. we compute only 4, 10, and 20 coefficients and use them directly for the evaluation of the values and derivatives along the considered ray segment. A similar test is done in Sec. 2 and it has turned out that the performance gain due to the pre-computed coefficients for a considered unit cube is about a factor of two. A similar result is obtained in this test as well (see the rendering times given in brackets in the above tables). Hence, in the following we consider only the former test. From tables 3.4, 3.6, 3.5, and 3.7 one can observe rendering time speed-offs of 1.7 and 2.4 between choosing a linear or a quadratic and a quadratic or a cubic type-6 spline model, respectively. Similar factors can be observed in Sec. 2. There is almost no speed difference between renderings obtained with the linear type-0 splines and the trilinear model. The computation of derivatives for the trilinear model is delegated to the preprocessing stage (with almost the same preprocessing times as mentioned above), hence during rendering we have to approximate (or interpolate) the data and the derivatives only. Where in the linear type-0 model we approximate (or interpolate) the data and compute the gradients using the de Casteljau algorithm. Hence, both methods need a similar number of operations (cf. Tab. 2.5). The quadratic type-0 spline model is about a factor of 5.0 slower than both the linear and trilinear model. The visualization of an arbitrary data set from the 3rd main viewing direction (i.e. when the volume scan-lines are considered perpendicularly to the intermediate image) results in a performance reduction of the algorithm as well. No matter which splines are used, the ratios here go from about 1.3 to 1.7. However, since all parts of the algorithm are the same for all viewing directions except the implementation of early-ray termination, we obtain a $25 - 40\%$ slower algorithm for the 3rd viewing direction where oblique (dynamically run-length encoded) intermediate image scan-lines have to be considered. Without early-ray termination the factor would be about 2.5. Finally, we have computed in Sec. 2 upper bounds on the number of frames one could achieve with an algorithm using our spline models, i.e. with some assumptions 30, 18, or 9 frames per second could be theoretically obtained for an iso-surface rendering algorithm using our linear, quadratic, or cubic splines, respectively. However, we have measured frame rates of at most 7, 5, and 2 for the respective spline models. That means, a factor of about 4 is spent for the root computations of polynomials (e.g. using Cardano's

formula) and our shear-warp algorithm (i.e. for implementation of space leaping, early-ray termination, convex hull tests to omit root computations, compositing, and the final warp operation).

3.2 Perspective Projection Case

For the perspective projection case we do not differentiate between the equidistant and accurate sampling. We first give the performance results of our implementation of the original method discussed in [Lac95] and [SNL01]. And the results obtained for the accurate sampling approach where the memory consumption of some additional data structures is given and the performance of our algorithm is presented.

Data Set	Size	SWO	SWI	SWN
Engine	$256^2 \times 128$	10.8	7.73	-
Teapot	$256^2 \times 178$	9.69	6.65	-
Bonsai	256^3	9.06	6.39	-

Table 3.8: *Average frame rates of the perspective rendering method in frames per second. SWO: Shear-Warp without intermediate slices. SWI: Shear-Warp with intermediate slices. SWN: Our new rendering method with trilinear interpolation. In each method we have applied a linear opacity transfer function.*

The original perspective shear-warp approach is quite efficient. As reported in the above mentioned papers this algorithm needs approximately twice as much time as that one considering parallel rays to render a volume data set. In our implementation we have observed a similar behavior (cf. Tab. 3.8). On the one side this cut-off in performance is due to the overhead one has to investigate for synchronization between the run-length encoded volume and image scan-lines. They are both processed in a parallel manner to each other as before due to an appropriate factorization of the total viewing transformation, but additionally the volume slices have to be scaled now, hence the volume scan-lines have to be processed by an averaging filter as well as an interpolation filter to generate the result for the intermediate image. In other words, since there is a scaling between volume slices and the intermediate image, mostly more than four adjacent data values (therefore more than two volume scan-lines at once as well) have to be considered to compute the result for one image pixel. That means, now we often have to deal with a footprint of $N \times M \geq 2 \times 2$ voxels (within the slices), which have to be filtered first and interpolated afterward to obtain the correct result for the corresponding intermediate image pixel. On the other side opacity correction becomes more complex for the perspective rendering algorithm. In the parallel projection method the spacing between two sampling positions (considered in intermediate image space) remains constant along any ray. Hence, one pre-computed opacity correction table have to be generated and applied during rendering. For the perspective shear-warp the spacing (the ratio) is constant along one ray only which originates from the corresponding intermediate image pixel. Hence, for each pixel a different ratio have to be pre-computed first and applied appropriately during rendering. In a brute force method this would result in as many per-computed opacity correction tables as there are pixels in the intermediate image.

First, this is not practicable. Second, since the changes of the rations corresponding to adjacent rays are small, only a few tables are sufficient to interpolate other values not represented by that tables. However, this is essentially another interpolation step which has to be performed during rendering. The same is true applying the intermediate slice approach to the perspective shear-warp algorithm and a similar rendering performance cut-off is observed as in the parallel projection case (compare Tab. 3.8 and Tab. 3.2).

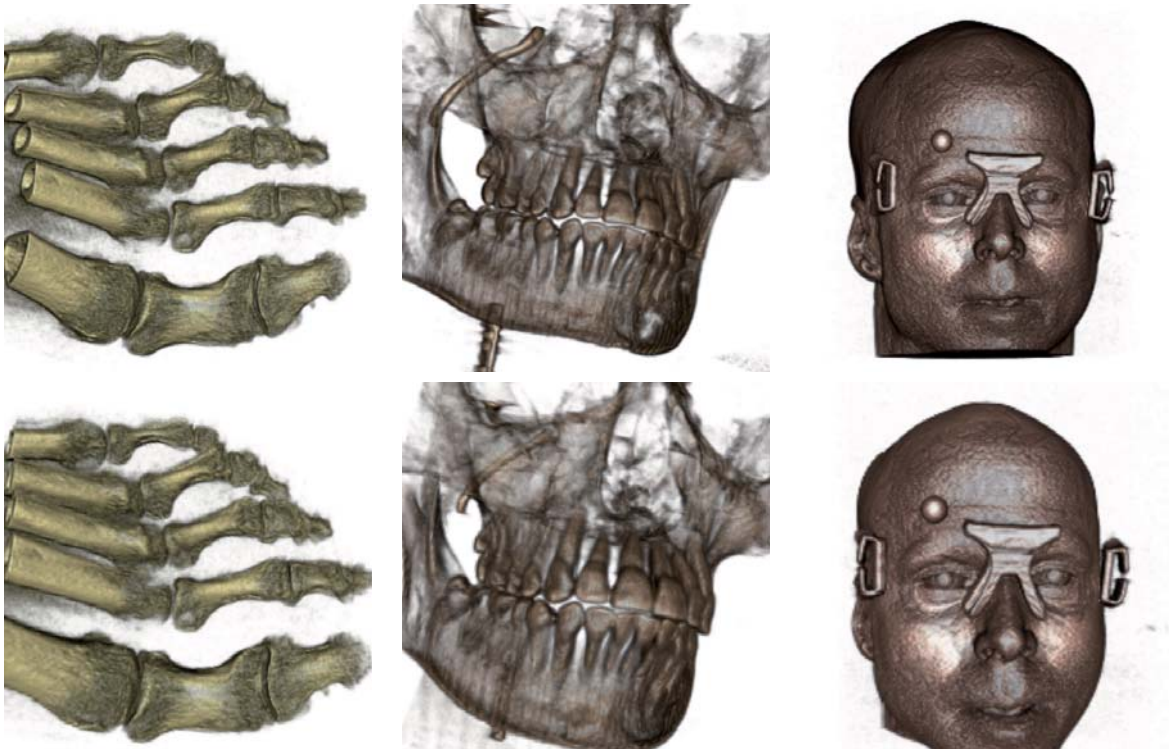


Figure 3.4: *Full volume rendering of the Foot (left), Skull (middle), and Head (MRI, right) data sets by our shear-warp approach using new data structures which allow accurate sampling. The top and bottom rows show images obtained with an orthographic and perspective camera, respectively. The opacity transfer table is set to a piecewise linear function where low valued densities (indicating noise) are mapped to zero.*

For our new perspective shear-warp approach we have also developed some data structures which allow an accurate sampling of the volume data along rays by using the previously discussed reconstruction models. These data structures are also necessary to be able to apply only one run-length encoded data set during rendering (as for the parallel case). However, the implementation of the perspective approach in not as coherent as for the simpler parallel shear-warp method. That means, for the first two main viewing directions we relate to the original algorithm which considers filtering and bilinear interpolation within slices (as discussed above). For the spacial case of the 3rd viewing direction we consult our new approach. Our new data structure used for ray propagation consists of two scan-lines of size equal to the width W of the intermediate image which itself depends on the width L and depth N of the volume data set with size $L \times M \times N$ and the ray direction (or the angel $\alpha \in [0°, 45°]$ measured between the ray direction and the main viewing axis, α represents one pole coordinate). The width of the intermediate

Data Set	Size	SW L (L)	SW Q (Q)	SW C (T)
Engine	$256^2 \times 128$	0.13 (0.13)	0.13 (0.09)	0.10 (0.13)
Teapot	$256^2 \times 178$	0.11 (0.11)	0.10 (0.09)	0.09 (0.11)
Bonsai	256^3	0.07 (0.07)	0.07 (0.06)	0.06 (0.07)

Table 3.9: *The iso-surface rendering performance (in frames per second) of our new perspective shear-warp approach along the all main viewing directions by considering accurate sampling for linear (linear), quadratic (quadratic), and cubic (trilinear) splines on $\triangle$ ($\diamondsuit$) (from left to right). The iso-values for the different data sets (Engine, Teapot, and Bonsai) are* 80, 50, *and* 40, *respectively.*

image[1] (hence of our data structure) is $W = L + N\tan(\alpha)$ and becomes maximal if $\alpha = 45°$, i.e. $W = L + N$. Each cell of this data structure represents a corresponding unit column of the volume. Basically, each cell contains a pointer into an ordered list of ray segments going through a corresponding column of the volume. For each ray segment we store the enter and exit planes according to the unit column which are intersected by the corresponding ray, the appropriate intermediate image pixel coordinates the ray segment contributes to, the ray segments start position and direction according to the column and a parameter which can be used to calculate the ray segments exit position (located on the exit plane). Hence, to store the information of one ray segment within a column we need exactly 36 Bytes. The number of segments within a unit column is variable, this can be implemented by an array data structure which allocates as much memory as required for all ray segment within a considered column. This has to be done dynamically during rendering, hence we have decided to fix the size of that array to a maximum of 512 ray segments which can be stored for a unit column. This is enough for usual perspective projections, makes the algorithm faster since there is no dynamic memory allocation procedure during rendering, but requires more space than often needed. The total memory necessary for that data structure of size of two intermediate image scan-lines which are needed to correctly propagate the ray information and represent information within unit columns of the volume is about $2 * (W + 1) * 512 * 36$ bytes, i.e. for a volume of size 256^3 about 18 megabytes are required. Additionally, we recompute the ray information from the above data structure into the *column* template so that the changes to parallel rendering algorithm can be minimized. However, as shown in Tab. 3.9 this data structure – even empty space skipping and early-ray termination can be realized – is not the right choice to implement a perspective projection within the shear-warp algorithm. No matter which of our reconstruction models we use for the visualization of a data set in our experiment, the rendering time takes about $7 - 10$ seconds, i.e. the most limiting factor in our algorithm is the application of our data structure. The perspective shear-warp method is about one order of magnitude slower than the parallel approach. However, a further development of our method is necessary. A possible way is discussed in part III.

[1]The height is $H = M + N\tan(\beta)$ with $\beta \in [0°, 45°]$ where β is another pole coordinate.

4 Results on Wavelet Hierarchy

Our new interactive volume rendering approach based on hierarchically organized data aims at achieving high-quality results efficiently. Therefore, it is compared to a recent, fast shear-warp type implementation that was developed in order to reduce artifacts as well [SM02] (see above). We omit a comparison with ray-casting and splatting as alternatives since their implementation is substantially different to shear-warp and would therefore raise questions of fair comparison which is not the focus of this document. Nevertheless, for these volume rendering methods quadratic Super-Splines may show advantageous properties as well. Motivated by the opacity functions in use, we refer to our implementation of the algorithm in [SM02] [SHM04] as *TL* (*tri*linear) and denote our approach by *QSS* (*q*uadratic *s*uper-*s*plines). Two different classifications are selected as prototypes for translucent (linear opacity transfer function) and surface-enhanced images (where the threshold function provides results similar to iso-surface rendering and therefore being a prototype for comparison with [RZNS03] [RZNS04a]).

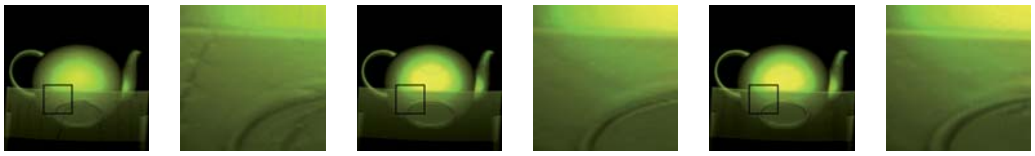


Figure 4.1: *CT scan of the SIGGRAPH 1989 Teapot. Data courtesy: Terarecon Inc, MERL, Brigham and Women's Hospital. Fast visualization of shear-warp by using wavelet encoded data. The close-up views are taken from marked areas. Zooming close to the local regions of interest the improved visual quality becomes increasingly evident. Left: Standard model (trilinear interpolation). Center: Approximation by quadratic Super-Splines. Right: Approximation by quadratic Super-Splines with decimated data (*14.8% *of the given data). The quadratic Super-Splines have the potential to reduce noise and leads to almost artifact-free visualizations.*

We measured the error between original data and the spline approximation by the peak-signal-to-noise ratio as shown in Fig. 4.4 ($PSNR_{rms} = 20log_{10}(255/RMSE)$, where $RMSE$ is the root-mean-square error as defined in [GW02]). It shows that the overhead of spline coding can be well compensated by the hierarchical representation without affecting the resulting image quality significantly. Due to the current construction this approach is, however, not as efficient as the method in [BIP01].

In our experiments, TL requires 1.2, 1.2, and 3.6 seconds to encode the engine, Teapot, and Head data set, respectively, while QSS needs 2.7, 2.6 and 8.7 seconds for the same data sets. Hence, we observe that QSS only requires about twice the time, although the pre-computed spline-coding of QSS increases by a factor of about 10.5. Visual comparisons showing the reduction of artifacts and the improved visual quality of QSS are given in Fig. 4.2 and 4.1, where we use the full (pre-classified) Engine and Teapot[1] data set, respectively. However, using the wavelet decompositions followed by data reduction of the size can be cut by up to a factor of 5 relative to the pre-classified

[1] In Fig. 4.1 we use a data decimation based on 78% of the pre-classified data.

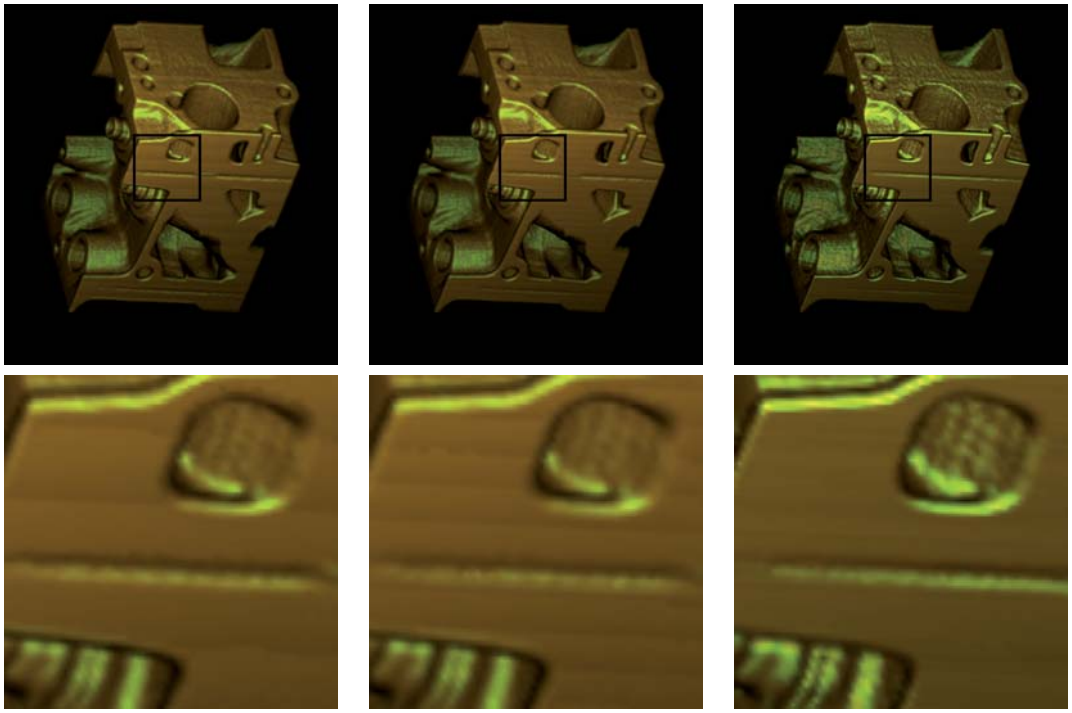


Figure 4.2: *Volume rendering results for the original Engine data set original shear-warp (left), applying TL (middle), and QSS (right), with no decimation. Top: overview, bottom: close-up views. Both, the original shear-warp and TL show typical stripe artifacts on the planar side of the machine part which disappear almost for the QSS model.*

volume without significant losses in image quality (see Fig. 4.3). The preprocessing for encoding the data hierarchy requires 22.3, 20.5, and 27.1 seconds for the engine, Teapot, and Head data set, respectively.

For comparing the rendering time measured in milliseconds and to obtain a robust statistics, all data sets are rendered from 100 random viewing directions. Although the frame-rate of TL outperforms QSS by a factor of 24 in case of the linear classification function and a factor of 5 in the case of the threshold function (see Tab. 4.1 and previous sections) one should mention that the QSS implementation (a) still yields interactive rates at up to 2 frames/s, (b) suppresses noise in the image more efficient, (c) leads to almost artifact-free and natural visualizations, and (d) outperforms the frame rate of the simple iso-surface renderer [RZNS03] [RZNS04a] by nearly one order of magnitude. With increasing performance of computers in the near future, we can therefore expect

Data Set	TL [ms]	QSS [ms]	TL [ms]	QSS [ms]
Engine	132 ± 3	2396 ± 315	92 ± 2	469 ± 87
Teapot	152 ± 5	5699 ± 957	94 ± 2	419 ± 42
Head	378 ± 13	5887 ± 804	130 ± 4	693 ± 101

Table 4.1: *Comparison of average rendering times and their deviation measured in milliseconds (ms) using a linear (left) and threshold (right) opacity transfer function. TL is an implementation of [SM02] [SHM04] and QSS is our new approach.*

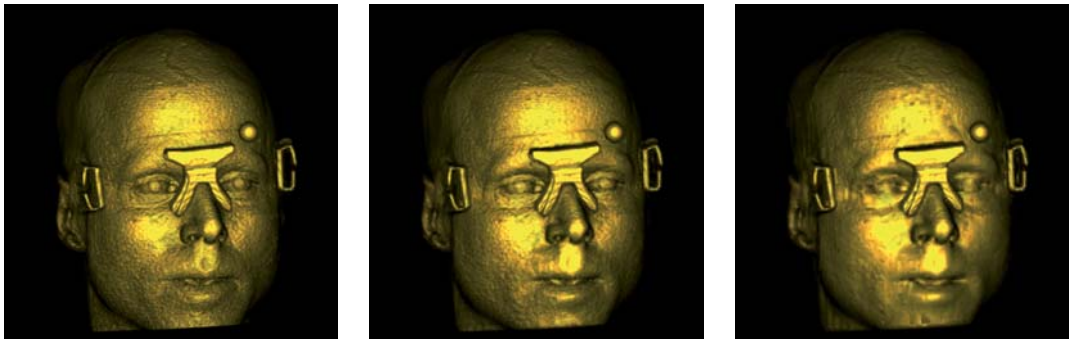
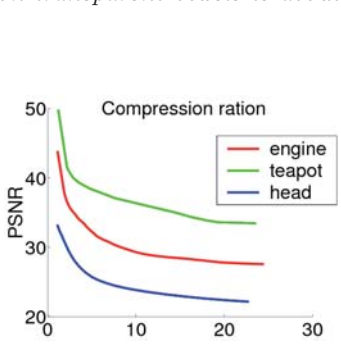


Figure 4.3: *Volume rendering of the Head by using QSS for different tolerated errors. From the left to the right the accepted error of the data is 0, 6.5, and 13.6, where the number of non-transparent voxels is about* $1.1 \cdot 10^6$, $0.4 \cdot 10^6$ *and* $0.2 \cdot 10^6$.



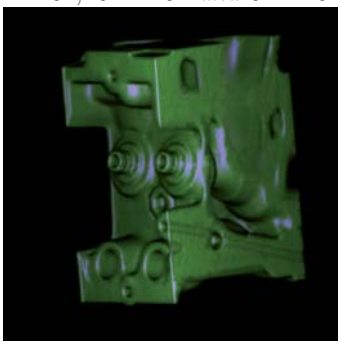


Figure 4.4: *Left: Peak-signal-to-noise ratio (PSNR) as a function of the compression factor between original data set and the hierarchically spline coded data. Middle and right: Volume rendering by using QSS of the Engine and the Teapot based on a compressed data set using about* 32% *and* 46%, *respectively, of the pre-classified data.*

this renderer type being a good candidate for displaying delicate data sets in medicine and quality control, where artifact-free results are at premium, although even for current systems the performance and the memory demands are already acceptable for given data sets which are not too large. More sophisticated types of data hierarchies are currently under investigation, in particular usage of averaged samples obtained from the quadratic Super-Splines on finer hierarchy levels might yield essential improvements.

Discussion

The work presented herein dealt with the fast visualization of three-dimensional scalar data sets by using a software-based rendering algorithm, named *shear-warp*. This approach generates projections from classified run-length encoded scalar-valued data sets which are, on the other hand, pre-computed from the original volume according to some user-specified classification functions. The rendering effectiveness of this algorithm is very high and is due to several facts. First, the two main acceleration techniques, namely early-ray termination and empty-space skipping, are profitably realized by a (dynamic) run-length encoding of the image pixels and volume voxels into scanlines, respectively. Both types of scanlines are processed in a parallel manner to each other possible due to the factorization of the overall transformation matrix into a shear and a warp component and a three-fold data redundancy. Second, after encoding one of the three volume data sets is processed in scanline by scanline manner according to the main viewing axis. This leads to an optimal cache performance considering today's personal computes. And third, information necessary for rendering is pre-computed and discretized extensively (e.g. interpolation weights and data gradients) such that the required calculations during the visualization step are kept to a minimum. In this work several extensions were realized and discussed.

The first main contribution in that thesis is the reduction of the required run-length encoded data sets to a minimum by strict change of the processing order of one run-length encoded volume depending on the view direction, i.e. only one data set is encoded and used for visualization instead of three. This has several advantages. Obvious benefits are that the memory and pre-processing requirements to generate the encoded data are reduced by a factor of 3 where the overhead for new data structures necessary for storing and processing the remaining run-length data set in a correct way can be neglected for the parallel projection case[2]. This issue becomes very significant considering the huge sizes of nowadays real CT or MRI data sets. For example, an unclassified CT data set obtained from a small animal imaging device is about 4GBytes. In that view a three-fold data redundancy (even after classification) is most inconvenient, when not all of the three run-length encoded data sets would fit into main memory at once which is on the other hand a demand for fast rendering[3]. Another advantage is that there are no visualization artifacts in the final image arising due to a switch between two different data sets according to a change of the main viewing axis. However, as supposed the data reduction comes not for free. The rendering speed of the new algorithm using a parallel projection matrix is diminished by factor 5 because of several reasons. First, four instead of two volume scanlines are processed at once for trilinear instead of bilinear reconstruction of the volume data, respectively, which also leads to a suboptimal empty-space skip-

[2] Note, the data is still processed in object-order, i.e. in a scanline by scanline manner.

[3] Note, one could still consider loading scanlines from harddisk on demand if the only one run-length encoded data set will still not fit into main memory.

ping process of the algorithm. Second, for reasons of correctly compositing data along ray segments within the four volume scanlines into the image, up to three dynamically encoded oblique image scanlines are processed at once which also affects the early-ray termination process of the algorithm[4]. And third, to stay within the Nyquist limit (i.e. for more precise data reconstruction) the number of sampling positions along a ray is increased by factor $\sqrt{3}$ (considering regular grids). However, the algorithm presented herein has the right proportion regarding accuracy, speed and data requirements. Nevertheless, one could further optimize the algorithm for accuracy or speed by still using only one encoded data set. An optimization for speed would guide a step back to the original method[5]. Further, the perspective shear-warp has been investigated too, where similar results have been shown as already discussed above for the parallel method, i.e. the reduction of run-length encoded data sets to one volume only, application of intermediate slices or trivariate spline models to obtain better visual quality, as well as performance issues with regard to the original method. Anyhow, the main issues with respect to the perspective rendering algorithm are as follows. The chosen new data structures are not the best possible for realizing a ray propagation scheme to implement early-ray termination and to omit the data redundancy of the original shear-warp method. That means, especially the template data structure, which has a thickness of two volume slices and stores the required projection and sampling information for rendering, has a size of several megabytes. This, of course, does not fit into the cache memory and is one reason why the algorithm is not as promising as for the parallel case. Instead, as proposed, another scheme could be investigated, which does not require any template data structures, but directly projects the not empty parts of the volume columns into the image in a correct sequence (to satisfy compositing order) to examine the appropriate non opaque pixels and thus also the rays which are not terminated. Afterwards only along these rays data could be reconstructed and the volume rendering integral evaluated. This scheme seems to be more promising. Since, first, there is no need for any additional templates and thus no extra memory space is required. And, second, the number of samples taken along rays is reduced to a minimum (i.e. samples along ray segments corresponding to non opaque pixels going through non empty data columns). Nevertheless, the scheme as such may introduce other computational cost, hence, finally only after investigation one can judge if that proposed technique would speed up the perspective shear-warp method.

The second main contribution of this work is the improvement of accuracy of the above discussed rendering method. First, a Hermite-Spline technique has been developed that allows interpolating the gray value change between slices in a more accurate way than typical ad hoc solutions like intermediate slices, where the computation performance of both techniques have been shown to be comparable. Second, a further development of the algorithm also allows applying real trivariate data reconstruction models which lead to a more accurate solution of the volume rendering integral and thus further increase the visual result of the final image. For the solution of the volume rendering integral different integration rules have been considered, i.e. the Newton-Cotes formula and Gaussian

[4] In the new algorithm the volume and image scanlines are only processed in a parallel manner to each other considering two of the three main viewing axes, for one main axis the image scanlines have to be processed in an oblique fashion

[5] That means, by applying bilinear reconstruction within the volume slices and still using the new data structures presented in this thesis with one run-length encoded data set only.

quadrature, where it turned out that the a special case of the Newton-Cotes formula, namely the Simpson integration rule, is best regarding accuracy and speed for most of the considered splines. However, for volume reconstruction the well known linear and quadratic tensor product splines in Bernstein-Bezier form (of total degree three and six, respectively) as well as the often utilized trilinear interpolation model has been realized. These models were mainly implemented for comparison reasons. Further, a new type of (linear,) quadratic, and cubic Bernstein-Bezier splines defined on tetrahedral partitions has been developed. These approximating piecewise polynomials have lowest possible total degree - i.e. (one,) two, and three, respectively - and deliver appropriate smoothness properties necessary for the visualization process. That means, the derivatives of the splines yield optimal approximation order for smooth data, while the theoretical error of the values is nearly optimal because of the averaging. The smoothness properties of the splines have been confirmed from some known test functions by evaluating the trivariate spline values and their gradients using efficient Bernstein-Bezier techniques well known in Computer Aided Geometric Design. The main outcome in this regard is twofold. As expected, the choice of a data model is a trade-off between accuracy and speed and thus depends on the task at hand. Piecewise linear splines are very fast to evaluate but are not suitable for volume visualization when first derivatives of the data are taken into account (e.g. for the lightning model). The next higher trilinear model[6] produces satisfying visual results even though it generates small jag-like artifacts which are mostly visible when zooming into the data or when difficult data with many high frequency components is considered for visualization. The corresponding quadratic tensor product model is currently not optimized for speed. Nevertheless, on one hand, because of its high total polynomial degree, it is more computation intensive for iso-surface and full volume rendering than the counterpart tetrahedral quadratic splines, also called Super Splines, but on the other hand it is also more accurate because of its smooth gradients. Further, it is less accurate as the counterpart tetrahedral cubic splines because of the required numerical root finding algorithms for evaluation of the six degree polynomials in iso-surface rendering and the higher degree integration rules necessary for full volume rendering. In this respect, with the trivariate quadratic and cubic splines defined on tetrahedral partitions one is not only able to apply simple explicit formulas, e.g. Cardano's formula, for the root calculations of quadratic and cubic polynomials and to apply more simple quadrature formulas, e.g. Simpson rule, for the evaluation of the volume rendering integral. Both spline models have been shown to generate smooth, accurate and natural looking images. However, the Super Splines are better with respect to the visual quality and are comparable in speed to the trilinear model[7], but they need twice as much memory for pre-computation of the Bernstein-Bezier coefficients compared to the number of coefficients required for the trilinear model. Additionally, the Super Splines are only smooth across the faces of the unit cubes of the volume partition, but in general not across the faces of the tetrahedra within a unit cube. This reveals in stripe artifacs when zooming into high frequency components of a volume data set. Thus, in such cases, the corresponding cubic model is necessary where it uncovers its capability to generate artifact-free visualizations of the data when considering first derivatives for shading only.

[6]The trilinear model consists in fact of two separate models, one for data and the other for gradient reconstruction, where both are interpolated trilinearly.

[7]This model also delivers piecewise linear gradients but from an inconsistent model

In this matter also the quadratic tensor product model can be gathered, where when optimized for speed it should show also similar timing results as the tetrahedral cubic model. As before, the main disadvantage of the cubic splines defined on tetrahedral partitions is the number of Bernstein-Bezier coefficients required for their evaluation. Hence, with the concern of fast volume rendering using the above Shear-Warp approach one has to pre-compute the coefficients of the splines and carefully organize them into a linear data structure according to the run-length encoded volume. This transfers in a one order of magnitude faster algorithm compared to the current available software-based approaches using these kind of reconstruction models with pre-computed spline coefficients. However, the number of Bernstein-Bezier coefficients required for evaluation of the splines does blow up the volume size considerably, i.e. by factor 20-175, thus only volumes of usual size (e.g. 512^3) can be handled by that method, especially for full volume rendering. A straight forward improvement by pre-computing only a part of the required Bernstein-Bezier coefficients has been shown to decrease the number of coefficients by a factor of 3 with an increase of computation cost during rendering by approximately 0.5. Another modification of the given full volume rendering approach could be to compute and to store the coefficients on the fly for visible voxels only which would slow down the visualization of the first image from a considered viewing direction. A further attempt could be to evaluate the splines using directly the data samples, i.e. without any computation of Bernstein-Bezier coefficients. This, as has been shown, would result in an at least by a factor of 2 to 3 slower Shear-Warp rendering algorithm.

The final main part dealt with the reduction of volume data particularly with regard to the number of pre-computed Bernstein-Bezier coefficients. Therefore an octree data structure has been setup guided by constant, linear, and quadratic wavelet decompositions followed by volume data reduction. This hierarchy was represented by appropriately defined Super Splines within the considered scale-space. From that hierarchy run-length encoded data were pre-computed for the Shear-Warp rendering algorithm. However, it turned out that the data size and thus the number of coefficients can be cut by up to a factor of 5 relative to the pre-classified volume without significant losses in image quality. The pre-processing for encoding the data hierarchy requires on the other hand up to one order of magnitude more time than a direct computation of run-length encoded data from the volume. However, a question is, as if it is legal to apply accurate spline models with data reduction schemes afflicted with truncation and approximation errors. A general answer can not be given here, but at one hand more sophisticated types of data hierarchies may be investigated, in particular the usage of averaged samples obtained directly from e.g. the quadratic Super-Splines on finer hierarchy levels which might yield essential improvements. On the other hand, data reduction schemes could be further developed for increasing the data compression ratio. In particular with regard to the shear-warp approach another shear-warp like algorithm for hierarchical data structures using a deep first search algorithm (well known in graph theory) has been discussed. This object-order method promises a faster visualization algorithm in connection with hierarchical data structures, as e.g. octrees, than traditional algorithms where expansive non object-order traversals of hierarchical data structures have to be performed. In addition, only the required spline coefficients of the visible and non-empty nodes or voxels could be computed for fast and accurate visualization. For that a visible

and non-empty narrow band within the classified volume is proposed, which also stores only a minimum set of required coefficients for volume rendering.

Summary and Outlook

This thesis has given a substantial insight into state of the art in volume rendering, and additionally, in the introductory chapters, techniques necessary to setup such a rendering framework were discussed. Thereby it has turned out that most software-based as well as hardware-based algorithms utilizing special techniques and data structures to fulfill the contradictory requirements of obtaining, first, high-quality visual rendering results and, second, interactive frame rates. However, it is obvious that there is no *golden standard* algorithm which performs best in each situation. Hence, the appliance and design of a specific rendering method depends on the task at hand, e.g. whether a real-time or a high-quality and accurate system is needed, if structured or highly irregular grids are going to be rendered, or if special purpose hardware is available for accelerated rendering or software-based solutions should be preferred only.

Under the above considerations the fast software-based rendering algorithm called *shear-warp* has been investigated and a further development has been presented in this thesis, where new techniques and data structures allow combining the fast shear-warp with the accurate ray-casting approach.
The first aspect considered thereby was the reduction of the three-fold data redundancy of the shear-warp approach while increasing the image quality by a correct sampling approach. Hereby, a Hermite-Spline technique has been developed that allows interpolating the gray value change between slices in a more accurate way than typical ad hoc solutions like intermediate slice techniques. Next, it was observed that the three-fold data redundancy can be reduced by a strict change of the processing order of one run-length encoded volume depending on the view direction. For removing redundancy completely, new data structures had to be developed. The main contribution in this view was the assembling of the so called *column* data structure, where all necessary information pre-computed for rendering can be stored (as e.g. sampling positions within the volume voxels along rays, projection offsets into the appropriate image pixels, etc.). Since uniform rays have been considered at first, only one such column data structure was required for representing the information for the whole encoded volume, where the space consumption of that template is negligible. It has been shown, that due to this template, a combination of shear-warp and ray-casting is not only possible but also results in a method which deduces the benefits of both techniques, that means, object-order traversal of the data with fast empty space skipping as well as fast early-ray termination can be simultaneously applied. Additionally, the data structures in use easily admit for adjustments to trade-off between rendering speed and precision of the algorithm. In addition, real trivariate data reconstruction models have been applied which allow a more accurate solution of the volume rendering integral and thus further increase the precision of the new volume rendering method. Therefore, the column template had to be extended in a straight forward way, that means, now the exact intersection locations of rays with volume voxels had to be pre-computed and stored in that data structure.

A further aspect considered in this work was the development of new data structures designed for the perspective shear-warp approach with the same targets as discussed above. Therefore a scheme was developed, which propagates the divergent rays from a so called *divergence point* through the whole volume in the correct order such that the volume integral can be computed in the proper way. For that a matrix of column templates has been designed which stores the propagation information (also e.g. sampling positions and projection offsets) of all rays going through the current considered volume columns used during rendering. Two such matrices are required for a clean propagation of the information. Nevertheless, using this data structure allows overcoming the data redundancy known from the original method. Additionally, it allows applying fast space-skipping because still possible object-order traversal of the volume, and also early-ray termination for accelerated volume rendering, since propagation information of not terminated rays is stored temporary in that data structure only. The trivariate reconstruction models as discussed can also be used since the propagated information can be both, equidistant sampling positions along rays as well as the intersection locations with the volume voxels. However, this data structure of matrix-type increases the data emergence, which also results in moderate rendering speed. Therefore, possible model-specific options have been discussed for a further improvement of that method.

This work also presented trivariate (linear,) quadratic and cubic spline models defined on symmetric tetrahedral partitions $\triangle$ as well as the counterpart linear and quadratic tensor product spline models specified on rectilinear volumetric partitions $\diamondsuit$ of a three-dimensional domain. These former spline models define piecewise polynomials of total degree (one,) two and three with respect to a tetrahedron, i.e. the local splines have the lowest possible total degree. They are still adequate for efficient and accurate volume visualization because their gradients are continuous and also deliver optimal approximation order for smooth data. The later splines define piecewise polynomials of total degree three and six, respectively, but only the quadratic splines yield satisfying visual quality. However, with the often used trilinear model one gets satisfying visual results, even though they generate small jag-like artifacts. Where the trivariate quadratic splines defined on tetrahedral partitions $\triangle$ are able to generate smooth, more accurate and more natural looking images. The corresponding cubic model is only necessary when zooming into a volume data set. In this case it uncovers its capability to generate artifact-free visualizations of the data when considering 1st derivatives for shading only. All of these models have been applied in conjunction with the previously discussed rendering approach which also leading to a one order of magnitude faster algorithm compared to traditional approaches using similar reconstruction models.

Finally, a hierarchy-based rendering method has been developed in this work. In a pre-computation step a wavelet decomposition of the volume data for data reduction was utilized, an octree data structure for its representation was applied, and trivariate splines for data reconstruction were claimed. In the visualization step a new shear-warp algorithm has been presented, which uses run-length encoded scanlines directly generated form the hierarchical data representation. However, there is still space for improving the method. Additionally, it came into notice that even after data reduction still a huge number of spline coefficients have to be pre-computed for all the non-empty octree nodes

yet to allow fast volume visualization. During rendering, though, many pixels become opaque and, thus, because of early-ray termination the respective voxels with the corresponding coefficients will not be used[8] at all.

Hence, an outlook has been given. Another shear-warp like algorithm for hierarchical data structures using a deep first search algorithm (well known in graph theory) has been discussed. This object-order method promises a faster visualization algorithm in connection with hierarchical data structures, as e.g. octrees, than traditional algorithms where expansive non object-order traversals of hierarchical data structures have to be performed. In addition, only the required spline coefficients of the visible and non-empty nodes or voxels need to be computed for fast and accurate visualization. For that a visible and non-empty narrow band within the classified volume is proposed, which also stores only a minimum set of required coefficients for volume rendering.

Moreover, other possible further development or research could include higher order trivariate polynomials for data reconstruction in conjunction with the shear-warp method. This would allow computing elaborate second-order differential structures of the volume field (e.g. curvature information) usable for more accurate non-photorealistic rendering, as e.g for silhouette enhancement or to display ridges and valleys within an iso-surface of a data set, in still quite fast visualizations.

A possible follow up shear-warp framework could also include on-the-fly data processing techniques as, for example, data smoothing, edge enhancement of volume data, or (an-)isotropic diffusion filtering. This data processing could be performed during rendering for non-empty and visible data voxels only (i.e. within a narrow band of the volume data). In that way one would obtain the same final visual result as by, first, filtering the whole data set in a pre-processing step (which is very time consuming) and, second, visualizing it afterwards.

Finally, one could also include other lightning models based on first and second derivatives of the volume data. Clipping planes and volumes could be applied to reveal more information of data form its interior while still preserving context in the finally rendered image. Similarly, classified data sets could be used to visualize objects with variant properties within a volume with different materials or transfer functions. And, distance or area measures could be integrated and displayed while interactively changing the viewing position to allow the user a better analysis of the data in use.

[8] This depends of course on the choice of the classification functions.

Appendix

A Additional Results on Spline Models

In this appendix section we show additional tables and figures with numerical results that have been already discussed in most instances in part V. The interested reader may use the information below for comparison issues, therefore the numerical results are given in tables instead of in diagrams. However, the sub-sections are divided by the considered spline models, where the table columns and rows have the following meanings. The first column in each of the following tables contains the grid spacing constant $h = 1/N$ with N the number of data samples considered in each of the three directions x, y and z of the data set. The remaining columns contain different types of errors as defined above. In other words, the second column in each table depicts the (approximate) root mean square error err^{ξ}_{rms}, the third column shows the (approximate) mean average error $\mathrm{err}^{\xi}_{mean}$, in the fourth column the maximal error err^{ξ}_{max} of the spline in the uniform norm on Ω is presented, and in the last column we give the maximal error $\mathrm{err}^{\xi}_{data}$ of the function values at the grid points. The former error is computed approximately on each unit cube Q of the domain Ω by choosing a fixed (high) number of uniformly distributed points in each unit cube of $\diamondsuit$. All numerical tests have been computed in double floating point precision.

A.1 Linear Splines on Ω

Piecewise linear splines in Bernstein-Bézier defined on Ω are not very suitable for volume visualization. Even the values of the Marschner-Lobb test function can be reconstructed for increasing volume data sizes quit well (cf. Tab. A.1), the derivatives are not continuous across faces belonging to two adjacent unit cubes in the volume domain Ω. The derivatives of this model can be thought as varying bi-linearly within the considered unit cube only. This leads to stripe artifacts which become visible in the resulting images. Hence, the underlying partition type (grid structure) becomes visible as well (cf. Fig. A.1). The second derivatives of the Marschner-Lobb test function along the x, y and z direction, namely $D^2_x ML, D^2_y ML$, and $D^2_z ML$, can not be reconstructed using this linear tensor product splines and are not shown here.

h	err^{ML}_{rms}	err^{ML}_{mean}	err^{ML}_{max}	err^{ML}_{data}
1/16	0.0740	0.0643	0.1764	0.1058
1/32	0.0643	0.0561	0.1225	0.1009
1/64	0.0312	0.0265	0.0629	0.0629
1/128	0.0095	0.0079	0.0197	0.0197
1/256	0.0025	0.0020	0.0052	0.0052

Table A.1: *Approximation errors for data values of the Marschner-Lobb test function f_{ML} using trivariate piecewise linear splines s_{ML} on Ω for reconstruction.*

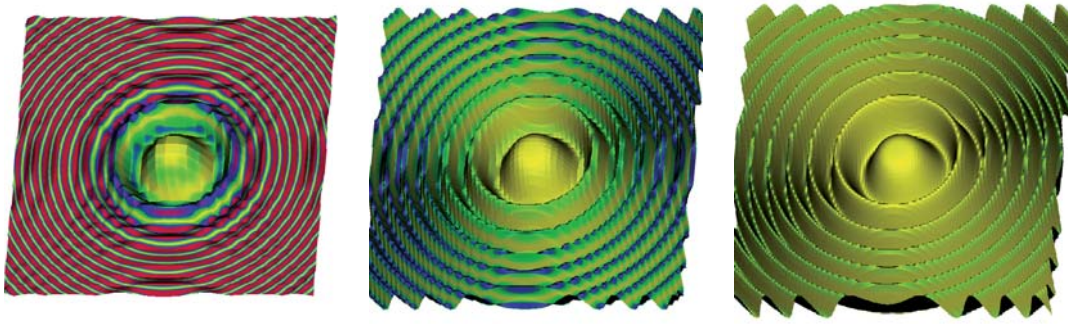


Figure A.1: *Reconstruction of values of the Marschner-Lobb test function at points* $\mathbf{v}$*, where the iso value becomes* $127.5/255.0$ *(i.e. where* $s_{ML}(\mathbf{v}) = 0.5$*) by using trivariate piecewise linear splines. The images (from left to right with* $h = 1/32, h = 1/64$*, and* $h = 1/128$*) show the error* $|(f_{ML} - s_{ML})(\mathbf{v})|$ *between the values obtained from the Marschner-Lobb test function* $f_{ML}(\mathbf{v})$ *and values reconstructed by the piecewise linear spline model* $s_{ML}(\mathbf{v})$*. Here, the error threshold values are* $a = 0.025, b = 0.05$*, and* $c = 0.075$*.*

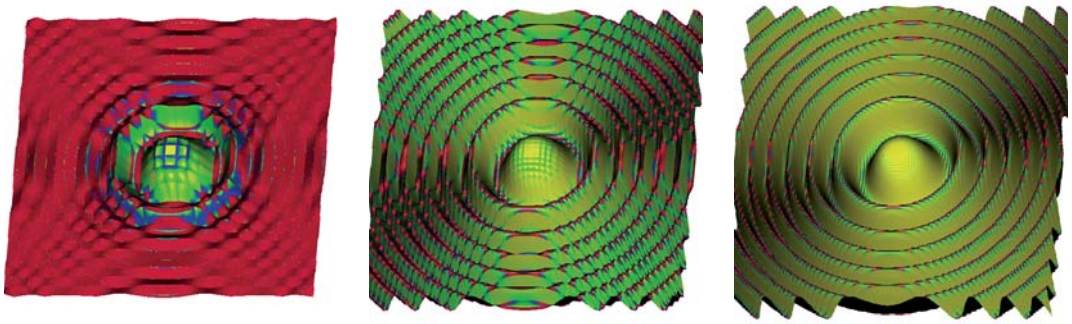


Figure A.2: *Reconstruction of gradients of the Marschner-Lobb test function at points* $\mathbf{v}$*, where the iso value becomes* $127.5/255.0$ *(i.e. where* $s_{ML}(\mathbf{v}) = 0.5$*) by using trivariate piecewise linear splines. The images (from left to right with* $h = 1/32, h = 1/64$*, and* $h = 1/128$*) show the error between the gradients obtained from the Marschner-Lobb test function* $f_{ML}(\mathbf{v})$ *and gradients reconstructed by the piecewise linear spline model* $s_{ML}(\mathbf{v})$*. Here, the angle (measured in degrees) between the two gradients defines the error, i.e. the angle thresholds are* $0°, a = 10°, b = 20°$*, and* $c = 30°$*.*

h	$\mathrm{err}_{rms}^{D_x^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 ML}$	$\mathrm{err}_{max}^{D_x^1 ML}$	$\mathrm{err}_{data}^{D_x^1 ML}$
1/16	2.4764	1.9702	6.2338	5.1192
1/32	2.4448	1.9008	6.4878	6.4818
1/64	1.3579	1.0229	4.6767	2.8051
1/128	0.5982	0.4232	2.6196	0.7998
1/256	0.2787	0.1841	1.3410	0.2066

Table A.2: *Approximation errors for* 1*st derivatives along the* x *direction of the Marschner-Lobb test function* f_{ML} *using trivariate piecewise linear splines on* Ω *for reconstruction. The errors show similar behavior for the* y *and* z *directions.*

h	$\mathrm{err}_{rms}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{max}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{data}^{D_x^1 D_y^1 ML}$
1/16	73.14	55.30	179.7	168.0
1/32	73.39	55.21	180.4	165.5
1/64	42.63	31.00	169.4	78.46
1/128	19.98	13.75	102.1	23.47
1/256	9.641	6.325	54.00	6.125

Table A.3: *Approximation errors for* 2*nd derivatives along the* xy *direction of the Marschner-Lobb test function* f_{ML} *using trivariate piecewise linear splines on* Ω *for reconstruction. Note, second derivatives along* xz *and* yz *directions lead to almost no error due to the considered function and reconstruction model.*

A.2 Quadratic Splines on Ω

As already discussed above in the results and one could expect, piecewise quadratic splines in Bernstein-Bézier form defined on a volumetric domain Ω lead to better reconstruction results (see Tab. A.4) compared to the previous linear model. The more important issue here is that of course the first derivatives are continuous across faces belonging to two adjacent unit cubes in the volumetric domain Ω. Hence, applying this model for volume visualization has the advantage of generating high quality smooth images (see Fig. A.3).

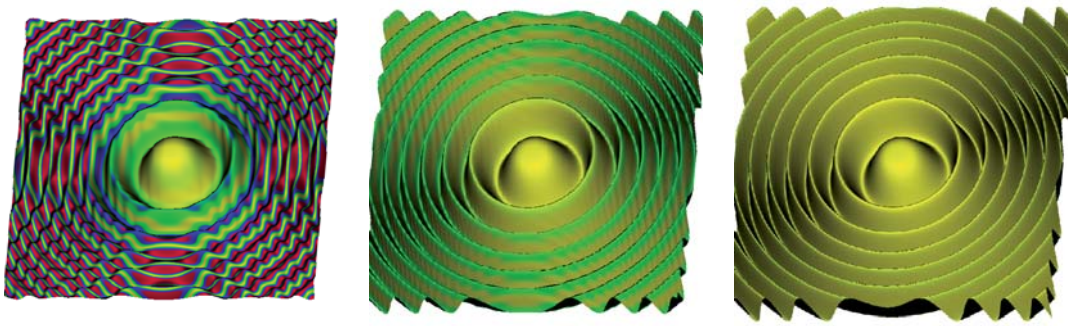


Figure A.3: *Reconstruction of values of the Marschner-Lobb test function. The same configuration as in Fig. A.1 using trivariate piecewise quadratic splines.*

h	err_{rms}^{ML}	err_{mean}^{ML}	err_{max}^{ML}	err_{data}^{ML}
1/16	0.0765	0.0636	0.1794	0.0719
1/32	0.0529	0.0451	0.1223	0.0710
1/64	0.0201	0.0170	0.0392	0.0333
1/128	0.0057	0.0048	0.0104	0.0099
1/256	0.0014	0.0012	0.0026	0.0026

Table A.4: *Approximation errors for data values of the Marschner-Lobb test function f_{ML} using trivariate piecewise quadratic splines on Ω for reconstruction.*

h	$\mathrm{err}_{rms}^{D_x^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 ML}$	$\mathrm{err}_{max}^{D_x^1 ML}$	$\mathrm{err}_{data}^{D_x^1 ML}$
1/16	2.4504	1.9169	6.2106	5.0753
1/32	2.0222	1.5181	6.4923	6.4855
1/64	0.8054	0.5953	2.7928	2.7930
1/128	0.2301	0.1696	0.7945	0.7942
1/256	0.0596	0.0439	0.2064	0.2064

Table A.5: *Approximation errors for 1st derivatives along the x direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise quadratic splines on Ω for reconstruction. Note, the errors have a similar behavior along the y and z directions.*

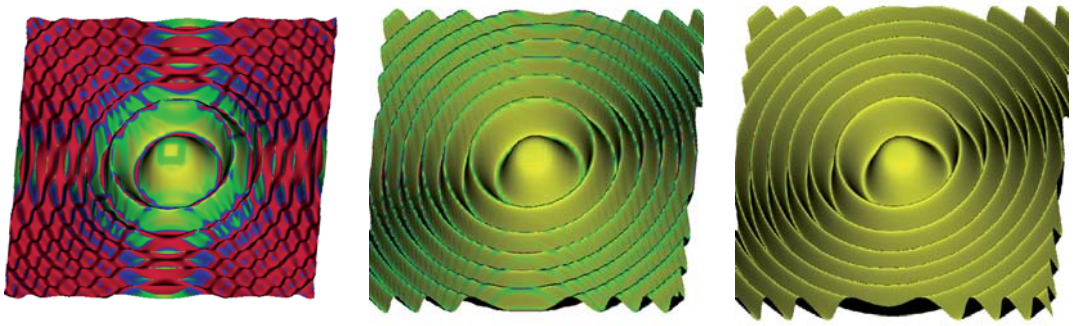


Figure A.4: *Reconstruction of gradients of the Marschner-Lobb test function. The same configuration as in Fig. A.2 using trivariate piecewise quadratic splines.*

h	$\mathrm{err}_{rms}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{max}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{data}^{D_x^1 D_y^1 ML}$
1/16	73.03	54.77	179.0	168.0
1/32	62.12	45.31	167.7	165.5
1/64	25.63	18.41	78.43	78.46
1/128	7.357	5.265	23.47	23.47
1/256	1.905	1.364	6.124	6.125

Table A.6: *Approximation errors for 2nd derivatives along the xy direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise quadratic splines on Ω for reconstruction. Again the there is almost no error considering the xz and yz directional derivatives (due to the considered function and reconstruction model).*

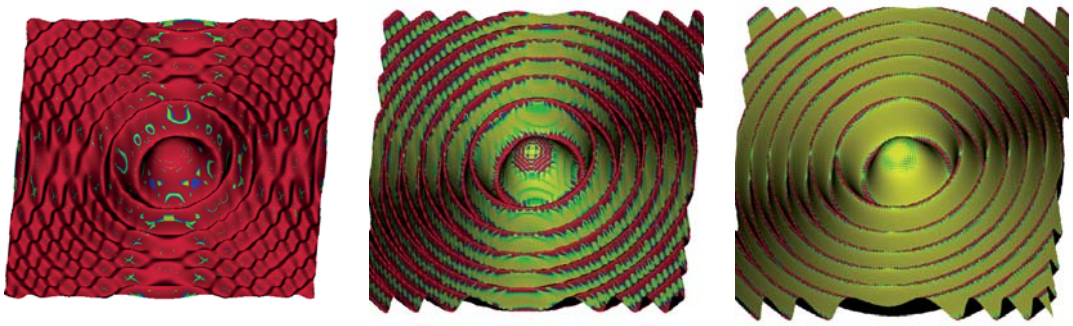


Figure A.5: *Reconstruction of Hesse matrices of the Marschner-Lobb test function at points $\mathbf{v}$, where the iso value becomes 127.5/255.0 (i.e. where $s_{ML}(\mathbf{v}) = 0.5$) by using trivariate piecewise quadratic splines. The images (from left to right with $h = 1/32, h = 1/64$, and $h = 1/128$) show the error between the Hesse matrices obtained from the Marschner-Lobb test function $f_{ML}(\mathbf{v})$ and Hesse matrices reconstructed by the piecewise quadratic spline model $s_{ML}(\mathbf{v})$. Here, the mean curvature computed from the Hesse matrices of the original function and this represented by the spline model is used to define the error. The error thresholds are $0.0, a = 0.0025, b = 0.005$, and $c = 0.0075$. Note, the second derivatives are piecewise constant only, this is very well visualized by the piecewise constant colors used to encode the errors.*

h	$\mathrm{err}_{rms}^{D_x^2 ML}$	$\mathrm{err}_{mean}^{D_x^2 ML}$	$\mathrm{err}_{max}^{D_x^2 ML}$	$\mathrm{err}_{data}^{D_x^2 ML}$
1/16	109.6	77.43	342.1	291.1
1/32	97.43	66.52	368.3	247.3
1/64	53.68	34.30	279.8	86.49
1/128	26.91	16.03	153.9	23.74
1/256	13.46	7.697	80.01	6.120

Table A.7: *Approximation errors for 2nd derivatives along the x direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise quadratic splines on Ω for reconstruction. The errors for the 2nd derivatives along the y and z direction are very similar to the errors shown here.*

A.3 Trilinear Model on Ω

The piecewise trilinear model (not in Bernstein-Bézier form, i.e. here we deal with the well known trilinear model often used in volume reconstruction and visualization) defined on Ω show a very similar behavior as the piecewise linear splines above. However, the trilinear model is a compromise between the linear and the quadratic spline models above. It allows at the one side faster reconstructions of the data compared to the above quadratic tensor product spline model in Bernstein-Bézier form. On the other side, a pre-computation of gradients for each grid point location of the volume data set by using central differences or the Sobel operator and a following linear interpolation of that grid point gradients allows us to reconstruct more accurate derivatives anywhere on the volumetric domain Ω compared to the pure linear tensor product splines. (compare figures A.6,A.1 and A.7,A.2). However, table A.8 shows the errors between the values obtained from the original function and the reconstructed values using this trilinear model. That errors are the same as already obtained for the linear spline model in Bernstein-Bézier form above. The errors ($\text{err}_{\star}^{D_x^1 ML}$) between the first derivatives obtained from the

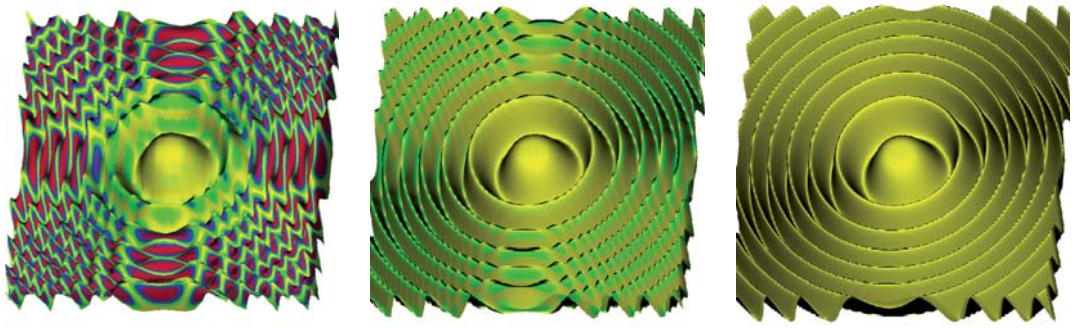


Figure A.6: *Reconstruction of values of the Marschner-Lobb test function. The same configuration as in Fig. A.1 using piecewise trilinear model.*

h	err_{rms}^{ML}	err_{mean}^{ML}	err_{max}^{ML}	err_{data}^{ML}
1/16	0.0765	0.0610	0.1985	0.1503
1/32	0.0474	0.0378	0.1200	0.1200
1/64	0.0152	0.0120	0.0385	0.0385
1/128	0.0041	0.0032	0.0103	0.0103
1/256	0.0022	0.0019	0.0049	0.0051

Table A.8: *Approximation errors for data values of the Marschner-Lobb test function f_{ML} using piecewise trilinear model on Ω for reconstruction.*

Marschner-Lobb test function and the trilinear model (reconstructed values) along the x direction show a similar behavior as the errors along y and z direction (cf. Tab. A.9). The approximation of the gradients along the z direction is as before more accurate than along the other two direction. Further, visual results encoding the error between the original and reconstructed derivatives are shown in Fig. A.7, where the same color

encoding is used as before. One can clearly observe in this figure that due to the smooth derivatives (which are in fact obtained from another model) the grid structure is less visible compared to the linear tensor product spline in Fig. A.2. Specifically, the colors vary smoothly across the whole image similar as for the quadratic spline model. The main difference is that the gradients obtained from the quadratic spline model approximate the real gradients more accurately and hence, in Fig. A.4, there are less red colored areas than in Fig. A.7. This becomes especially visible in the center images of these two figures. The trilinear model even allows us to reconstruct second derivatives such as

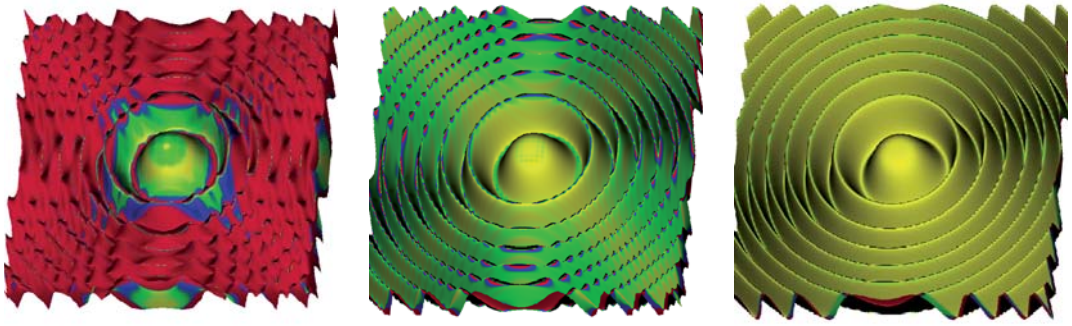


Figure A.7: *Reconstruction of gradients of the Marschner-Lobb test function. The same configuration as in Fig. A.2 using piecewise trilinear model.*

h	$\mathrm{err}_{rms}^{D_x^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 ML}$	$\mathrm{err}_{max}^{D_x^1 ML}$	$\mathrm{err}_{data}^{D_x^1 ML}$
1/16	2.2893	1.8151	5.3808	5.1454
1/32	2.4044	1.8733	6.4396	5.4683
1/64	1.4338	1.0993	3.9583	3.9505
1/128	0.4754	0.3628	1.3212	1.3211
1/256	0.2534	0.1634	0.9832	0.2656

Table A.9: *Approximation errors for* 1*st derivatives along the* x *direction of the Marschner-Lobb test function* f_{ML} *using trivariate piecewise trilinear model on* Ω *for reconstruction. The errors for* 1*st derivatives along the* y *and* z *direction are again similar to the errors shown here.*

for example $D_x^1 D_z^1 ML$, $D_y^1 D_z^1 ML$, or even D_x^2, D_y^2, and D_z^2. This is possible because of the pre-computed first derivatives at each grid point location which can be further used to compute higher order derivatives on the fly by applying the de Casteljau algorithm. Hence, in figure A.8 we show the errors between the original second derivatives computed directly from the Marschner-Lobb function and the reconstructed second derivatives using the pre-computed first derivatives. Note that the results here are quit similar to that ones obtained for the second derivatives of the quadratic tensor spline model (compare figures A.8 and A.5). Once more tables A.10 and A.11 show the errors for the second derivatives which are comparable with the errors obtained for the quadratic model in tables A.6 and A.7.

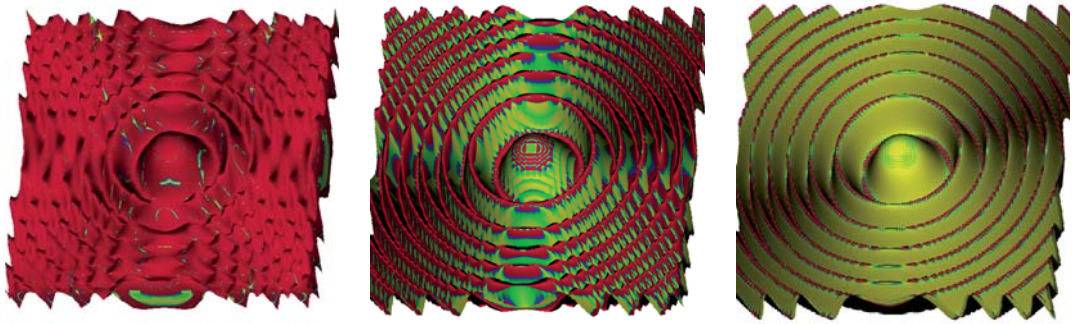


Figure A.8: *Reconstruction of Hesse matrices of the Marschner-Lobb test function. The same configuration as in Fig. A.5 using piecewise trilinear model.*

h	$\mathrm{err}_{rms}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{max}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{data}^{D_x^1 D_y^1 ML}$
1/16	65.56	48.22	177.0	163.3
1/32	72.26	53.76	175.9	170.3
1/64	47.51	34.77	134.4	112.9
1/128	19.12	13.75	66.32	39.03
1/256	8.943	4.667	31.43	14.43

Table A.10: *Approximation errors for 2nd derivatives along the xy direction of the Marschner-Lobb test function f_{ML} using piecewise trilinear model on Ω for reconstruction.*

h	$\mathrm{err}_{rms}^{D_x^2 ML}$	$\mathrm{err}_{mean}^{D_x^2 ML}$	$\mathrm{err}_{max}^{D_x^2 ML}$	$\mathrm{err}_{data}^{D_x^2 ML}$
1/16	94.89	67.36	286.1	214.9
1/32	107.5	75.23	348.4	343.1
1/64	69.77	48.07	276.5	182.4
1/128	30.64	20.12	153.5	57.45
1/256	13.46	8.697	81.41	15.12

Table A.11: *Approximation errors for 2nd derivatives along the x direction of the Marschner-Lobb test function f_{ML} using piecewise trilinear model on Ω for reconstruction.*

A.4 Linear Splines on Δ

Piecewise linear splines in Bernstein-Bézier defined on the tetrahedral domain Δ have a very similar behavior for volume visualization as the linear tensor product splines discussed above (do not confuse with trilinear tensor product model). According to the resulting images in figure A.9 and the error values in table A.12 they are able to approximate the synthetic Marschner-Lobb test function quit well for increasing data sizes. Hence, they are suitable for simple scaling of data sets similar as the above linear splines defined on the volumetric domain Ω. However, even the first derivatives

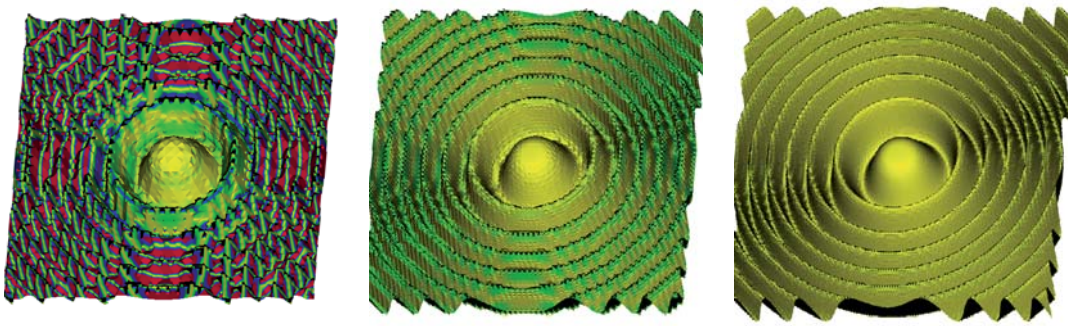


Figure A.9: *Reconstruction of values of the Marschner-Lobb test function. The same configuration as in Fig. A.1 using trivariate piecewise linear splines.*

h	err_{rms}^{ML}	err_{mean}^{ML}	err_{max}^{ML}	err_{data}^{ML}
1/16	0.0762	0.0619	0.1906	0.0000
1/32	0.0505	0.0415	0.1224	0.0000
1/64	0.0182	0.0148	0.0551	0.0000
1/128	0.0051	0.0041	0.0170	0.0000
1/256	0.0013	0.0010	0.0044	0.0000

Table A.12: *Approximation errors for data values of the Marschner-Lobb test function f_{ML} using trivariate piecewise linear splines on Δ for reconstruction.*

(gradients) reconstructed by that model approximate the real function's gradients for decreasing grid spacing quite well (see Tab. A.13), even thought they are piecewise constant and visibility of the tetrahedral structure here is even more inconvenient than the visibility of the cubic structure (compare Fig. A.9 and A.10 with Fig. A.1 and A.2). It is obvious that second derivatives can not be reconstructed by this model. In contrast to the linear tensor product splines even the mixed terms, i.e. $D_x^1 D_y^1 ML$, $D_x^1 D_z^1 ML$ and $D_y^1 D_z^1 ML$, can not be obtained directly from that model without additional effort.

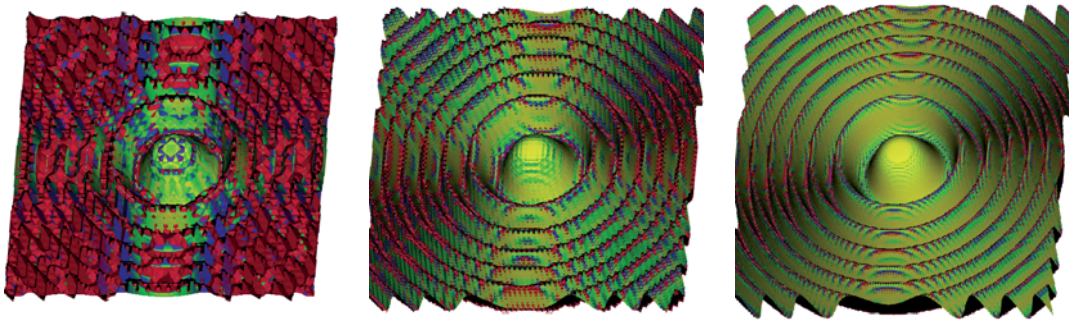


Figure A.10: *Reconstruction of gradients of the Marschner-Lobb test function. The same configuration as in Fig. A.2 using piecewise linear splines.*

h	$\mathrm{err}_{rms}^{D_x^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 ML}$	$\mathrm{err}_{max}^{D_x^1 ML}$	$\mathrm{err}_{data}^{D_x^1 ML}$
1/16	2.3994	1.8737	6.4491	5.3061
1/32	2.0634	1.5477	6.9157	7.1542
1/64	1.1628	0.8286	4.8474	4.8411
1/128	0.5874	0.3923	2.9033	2.6373
1/256	0.2937	0.1882	1.5221	1.3434

Table A.13: *Approximation errors for* 1*st derivatives along the* x *direction of the Marschner-Lobb test function* f_{ML} *using trivariate piecewise linear splines on* Δ *for reconstruction.*

A.5 Quadratic Splines on Δ

The next natural choice for a data reconstruction model defined on a tetrahedral domain Δ are piecewise quadratic splines in Bernstein-Bézier. However, the main goal for this approach was to develop a reconstruction model which results in approximating, quadratic splines s defined on Δ and additionally satisfies smoothness properties needed for volume visualization. The basic idea was to give up some C^1 smoothness conditions which would lead to an overall C^1 quadratic spline on Δ (see cubic splines in the following section) and introduce other useful conditions, i.e. averages of smoothness conditions. The coefficients of the splines can be computed efficiently applying a local data stencil of size 3^3 and some averaging rules which are chosen carefully such that many smoothness conditions are automatically satisfied. However, this approach can be seen as a compromise between fast visualization and accurate approximation of the data by using trivariate, quadratic splines. It can be proofed that the splines s are smooth not only at the vertices of the volumetric partition $\diamondsuit$. Further, regarding the approximation properties, the splines s yield nearly optimal approximation order, while its derivatives yield optimal approximation order of smooth functions f which is a non-standard mathematical phenomenon (see part II and [RZNS03] [NRSZ04]). The accuracy is given in Tab. A.14, which shows the decrease of the approximation error of the quadratic spline to the Marschner-Lobb test function for decreasing grid spacing h. In Fig. A.11 again one corresponding iso-surface is shown, where the approximation error of the splines according to the original function is color coded as before. However,

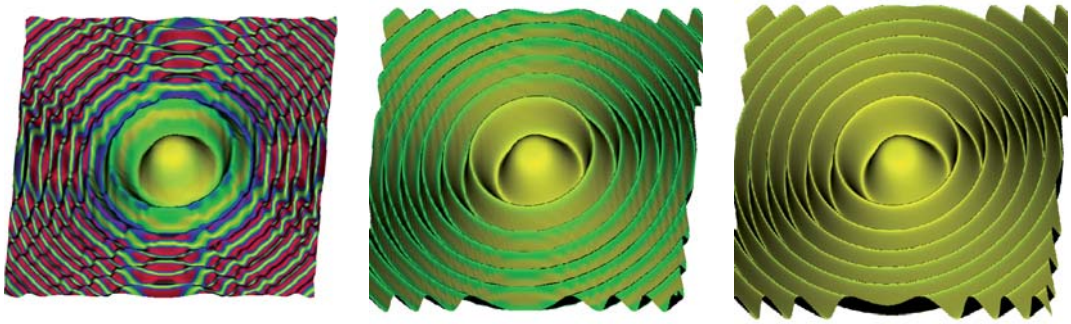


Figure A.11: *Reconstruction of values of the Marschner-Lobb test function. The same configuration as in Fig. A.1 using trivariate piecewise quadratic splines.*

a comparison with previous methods [UAE93a] [UAE93b] [ML94] ([PSL$^+$98]) [MJC01] and [BMDS02] shows that this model has the same theoretical approximation order for the error of the reconstructed values. Further, the piecewise polynomials defined by this model have a lower total degree (i.e. two) compared to for example linear or quadratic tensor product splines (which has total degree three or six, respectively). This is the case because the polynomials here are considered on the tetrahedral partition, i.e. according to a tetrahedron. However, for a fair comparison one should mention, that in fact only polynomials according to a unit cube as considered for tensor product splines [MJC01] [BMDS02] should be investigated. That means, once several tetrahedra have to be taken into account in a unit cube, as is mostly the case in volume visualization algorithms,

h	err_{rms}^{ML}	err_{mean}^{ML}	err_{max}^{ML}	err_{data}^{ML}
1/16	0.0774	0.0647	0.1803	0.0925
1/32	0.0548	0.0467	0.1222	0.0918
1/64	0.0204	0.0173	0.0392	0.0365
1/128	0.0057	0.0048	0.0104	0.0102
1/256	0.0014	0.0012	0.0026	0.0026

Table A.14: *Approximation errors for data values of the Marschner-Lobb test function f_{ML} using trivariate piecewise quadratic splines on Δ for reconstruction.*

one would obtain polynomials of higher total degree (according to a unit cube). Hence, considering for example iso-surface rendering where one has to solve for the roots of arbitrary polynomials. That quadratic tensor product splines need more computational power to solve for the roots of the corresponding six degree polynomials defined on a unit cube in a numerical accurate and stable way. Where this model spends more time for searching the appropriate tetrahedra in a unit cube which possible contain the iso value we are looking for and less computational power is needed to solve the second degree polynomial pieces defined according to a tetrahedron. However, the results for the data

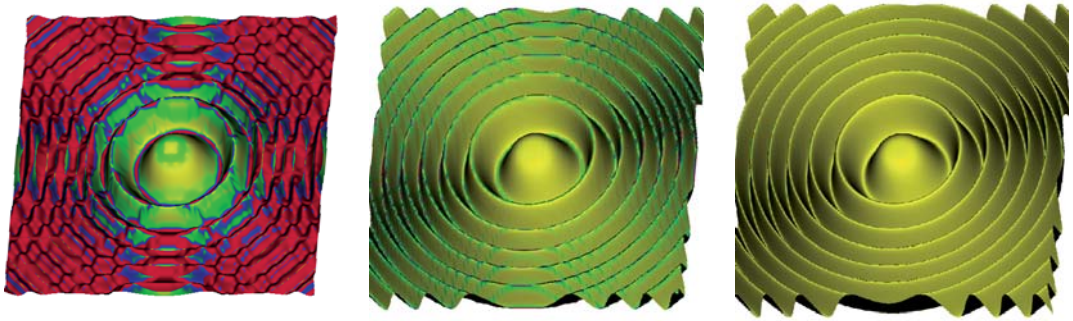


Figure A.12: *Reconstruction of gradients of the Marschner-Lobb test function. The same configuration as in Fig. A.2 using trivariate piecewise quadratic splines.*

h	$\text{err}_{rms}^{D_x^1 ML}$	$\text{err}_{mean}^{D_x^1 ML}$	$\text{err}_{max}^{D_x^1 ML}$	$\text{err}_{data}^{D_x^1 ML}$
1/16	2.4905	1.9642	6.1650	5.0314
1/32	2.1083	1.5953	6.5002	6.4892
1/64	0.8250	0.6121	2.7994	2.7827
1/128	0.2336	0.1726	0.7954	0.7899
1/256	0.0604	0.0446	0.2063	0.2062

Table A.15: *Approximation errors for 1st derivatives along the x direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise quadratic splines on Δ for reconstruction.*

approximation error of this quadratic splines, the quadratic tensor product splines, and even higher order piecewise polynomial are up to a constant value very similar, as can

be see in tables A.14 and A.4. Further, the derivatives of this splines yield an optimal approximation order for smooth functions f, but it is not always clear if the methods mentioned above provide a similar error bound for the derivatives. The visual result of the approximation quality of the gradients obtained from the Marschner-Lobb function and this quadratic splines defined on a tetrahedral partition $\triangle$ is given in figure A.12. The corresponding table A.15 depicts the numerical values. As before some results on the quality of the second derivatives obtained from that model are presented as well. It is clear that C^1 splines do not allow a reconstruction of smooth second derivatives except for quadratic functions defined over the whole domain Ω. For arbitrary functions as for example the Marschner-Lobb benchmark this model generates piecewise constant second derivatives. Hence, the color coded error measured between the original derivatives computed directly from the test function and the derivatives obtained from that model leads to piecewise constant colors as shown in figure A.13. Note that this figure does not show the error of the derivatives directly. Instead the error of the mean curvatures, which are computed using the local Hesse matrices on the considered iso-surface. In Tab. A.16 and Tab. A.17 the errors of the derivatives $D_x^1 D_y^1 ML$ and $D_x^2 ML$ along directions xy and x are given, respectively.

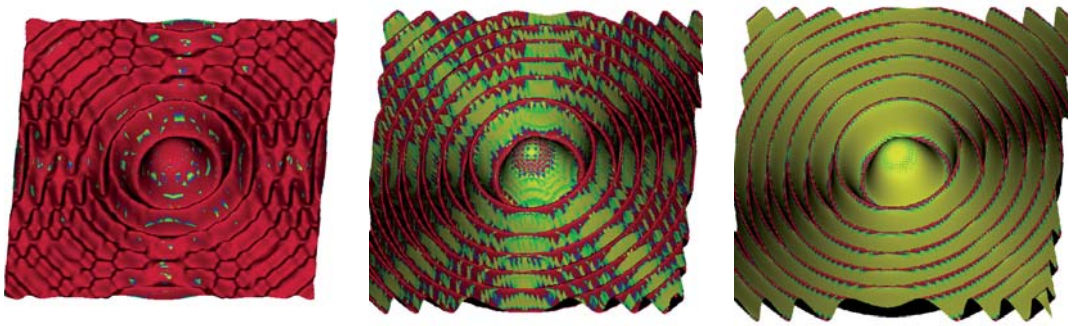


Figure A.13: *Reconstruction of Hesse matrices of the Marschner-Lobb test function. The same configuration as in Fig. A.5 using trivariate piecewise quadratic splines.*

h	$\mathrm{err}_{rms}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{max}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{data}^{D_x^1 D_y^1 ML}$
1/16	73.81	55.54	179.4	169.7
1/32	65.09	47.66	169.4	170.8
1/64	31.54	22.63	112.9	114.4
1/128	14.19	9.759	60.57	61.09
1/256	6.794	4.500	30.72	30.96

Table A.16: *Approximation errors for 2nd derivatives along the xy direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise quadratic splines on Δ for reconstruction.*

h	$\mathrm{err}_{rms}^{D_x^2 ML}$	$\mathrm{err}_{mean}^{D_x^2 ML}$	$\mathrm{err}_{max}^{D_x^2 ML}$	$\mathrm{err}_{data}^{D_x^2 ML}$
1/16	109.9	78.23	345.3	292.2
1/32	99.67	69.60	374.0	249.5
1/64	53.27	36.32	295.1	108.5
1/128	26.39	17.20	163.0	59.85
1/256	13.15	8.317	84.18	30.77

Table A.17: *Approximation errors for* 2*nd derivatives along the* x *direction of the Marschner-Lobb test function* f_{ML} *using trivariate piecewise quadratic splines on* Δ *for reconstruction.*

A.6 Cubic Splines on $\triangle$

A further development of the above quadratic spline model (also called *super splines*) is an approximating scheme based on cubic C^1 splines define on type-6 tetrahedral partitions $\triangle$ as well. However, the piecewise polynomials are directly determined by setting their Bernstein-Bézier coefficients to appropriate combinations of the data values. The exact repeated averaging schemes are discussed in part II. A more detailed discussion especially from the mathematical point of view can be found in [SZ05]. However, each polynomial piece of the approximating spline is immediately available from local portions of the data and there is no need to use prescribed derivatives at any point of the domain. It can be shown that the locality of the method and the uniform boundedness of the operator result in an error bound. Further, that the approach can be well applied for data approximation and the reconstruction of trivariate functions (cf. the numerical test in Tab. A.18 and a visual improvement in Fig. A.14). As before the results in table A.18 confirm that the quasi-interpolating splines yield approximation order two, since in each row the error decreases by about the factor of four while the grid spacing h goes down to $h/2$. It is well-known in spline theory, that spline operators possess the

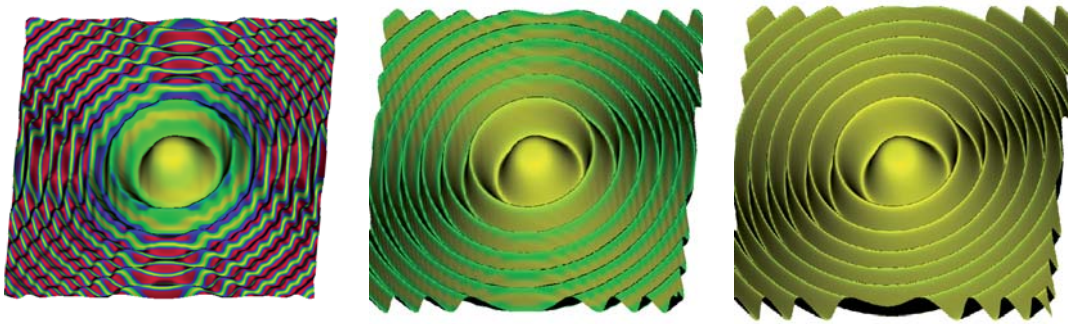


Figure A.14: *Reconstruction of values of the Marschner-Lobb test function. The same configuration as in Fig. A.1 using trivariate piecewise quadratic splines.*

h	err^{ML}_{rms}	err^{ML}_{mean}	err^{ML}_{max}	err^{ML}_{data}
1/16	0.0770	0.0642	0.1803	0.0787
1/32	0.0538	0.0458	0.1222	0.0780
1/64	0.0202	0.0171	0.0392	0.0343
1/128	0.0057	0.0048	0.0104	0.0100
1/256	0.0014	0.0012	0.0026	0.0026

Table A.18: *Approximation errors for data values of the Marschner-Lobb test function f_{ML} using trivariate piecewise cubic splines on Δ for reconstruction.*

advantageous property to simultaneously approximate derivatives of a smooth function, even if only the values of this function are used. However, for smooth functions the first derivatives of the quasi-interpolating cubic spline provide the same order of accuracy as the spline itself, which is a non-standard phenomenon considering general spline theory.

The derivatives of the quasi-interpolating splines yield nearly optimal approximation order, similar to the quadratic splines. However, one of the main differences between the quadratic and cubic splines on type-6 partitions $\triangle$ is that obviously the cubic splines can reproduce polynomial functions of the 19-dimensional space of cubic polynomials without any error, namely all functions f with $f \in \mathcal{P}_3$. However, a more important property of the cubic splines is that they are C^1 everywhere on $\Delta \subseteq \Omega$, where the quadratic splines are only almost everywhere C^1. The errors $\mathrm{err}_{rms}^{D_x^1 ML}$ for the first derivative $D_x^1 ML$ of the Marschner-Lobb function ML are given in table A.19. The results in that table indicate that the corresponding errors behave in the same way as the errors of the values. Therefore, the error of the first derivative D_x^1 is nearly optimal. Once more figure A.15 depicts the error between two gradients, i.e. the original and the reconstructed. The original gradient is obtained directly from the test function and the second gradient is reconstructed by the splines at the user-defined iso surface.

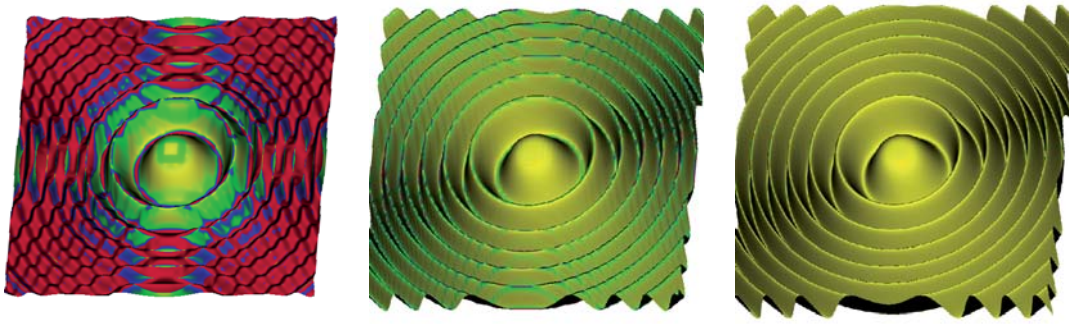


Figure A.15: *Reconstruction of gradients of the Marschner-Lobb test function. The same configuration as in Fig. A.2 using trivariate piecewise cubic splines.*

h	$\mathrm{err}_{rms}^{D_x^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 ML}$	$\mathrm{err}_{max}^{D_x^1 ML}$	$\mathrm{err}_{data}^{D_x^1 ML}$
1/16	2.4757	1.9469	6.1787	5.0533
1/32	2.0595	1.5546	6.4952	6.4874
1/64	0.8141	0.6045	2.7943	2.7879
1/128	0.2318	0.1716	0.7945	0.7921
1/256	0.0600	0.0444	0.2064	0.2063

Table A.19: *Approximation errors for* 1*st derivatives along the x direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise cubic splines on Δ for reconstruction.*

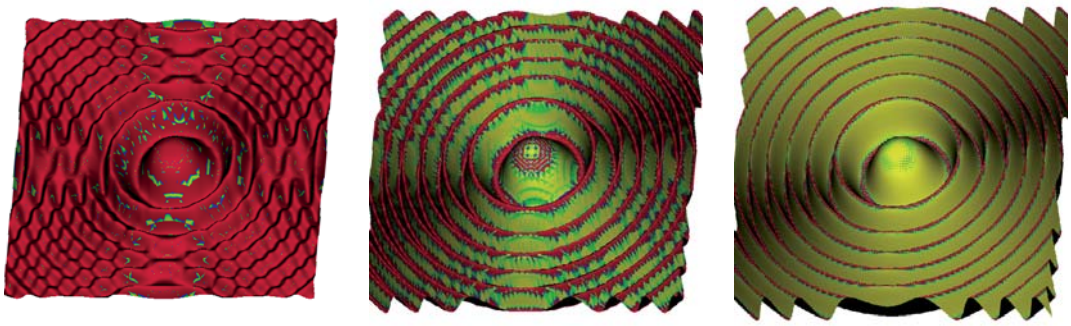


Figure A.16: *Reconstruction of Hesse matrices of the Marschner-Lobb test function. The same configuration as in Fig. A.5 using trivariate piecewise cubic splines.*

h	$\mathrm{err}_{rms}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{mean}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{max}^{D_x^1 D_y^1 ML}$	$\mathrm{err}_{data}^{D_x^1 D_y^1 ML}$
1/16	73.05	54.87	179.2	168.8
1/32	62.58	45.82	170.3	167.2
1/64	26.89	19.51	87.35	88.86
1/128	9.465	6.793	35.80	36.31
1/256	3.782	2.613	16.00	16.33

Table A.20: *Approximation errors for 2nd derivatives along the xy direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise cubic splines on Δ for reconstruction.*

h	$\mathrm{err}_{rms}^{D_x^2 ML}$	$\mathrm{err}_{mean}^{D_x^2 ML}$	$\mathrm{err}_{max}^{D_x^2 ML}$	$\mathrm{err}_{data}^{D_x^2 ML}$
1/16	109.5	77.68	344.4	291.1
1/32	97.52	66.94	367.0	246.9
1/64	51.28	33.28	283.6	87.96
1/128	24.91	15.04	156.1	30.81
1/256	12.32	7.101	79.92	15.53

Table A.21: *Approximation errors for 2nd derivatives along the x direction of the Marschner-Lobb test function f_{ML} using trivariate piecewise cubic splines on Δ for reconstruction.*

List of Figures

List of Tables

Bibliography

[Ake93] K. Akeley. Realityengine graphics. In *Computer Graphics*, pages 109–116, 1993.

[AW87] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics'87*, pages 3–10. Elsevier Science Publishers, 1987.

[Bew04] J. Bewersdorff. *Algebra für Einsteiger: Von der Gleichungsauflösung zur Galois-Theorie*. 2 edition, 2004.

[BG05] S. Bruckner and M. E. Gröller. Volumeshop: An interactive system for direct volume illustration. In *Proceedings of IEEE Visualization 2005*, pages 671–678, 2005.

[BGKG05] S. Bruckner, S. Grimm, A. Kanitsar, and M. E. Gröller. Illustrative context-preserving volume rendering. In *Proceedings of EuroVis 2005*, pages 69–76, 2005.

[BIP01] C. Bajaj, I. Ihm, and S. Park. 3d rgb image compression for interactive applications. In *ACM Transactions on Graphics*, volume 20, pages 10–38. ACM Press, 2001.

[BIPS00] C. Bajaj, I. Ihm, S. Park, and D. Song. Compression-based ray casting of very large volume data in distributed environments. In *High Performance Computing in Asia-Pacific Region*, pages 720–725, 2000.

[Bli82] J.F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Conference on Computer Graphics and Interactive Techniques*, pages 21–29, 1982.

[BLM97] M.J. Bentum, B.B.A. Lichtenbelt, and T. Malzbender. Frequency analysis of gradient estimators in volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):242–254, 1997.

[BMDS02] L. Barthe, B. Mora, N. Dodgson, and M. Sabin. Triquadratic reconstruction for interactive modelling of potential fields. In *Shape Modeling International*, pages 145–153, 2002.

[BPI01] C. Bajaj, S. Park, and I. Ihm. Visualization-specific compression of large volume data. In *Pacific Conference on Computer Graphics and Applications*, pages 212–220, 2001.

[BSMM97] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Harri Deutsch, 3 edition, 1997.

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

[CDSB03] R. Claypoole, G.M. Davis, W. Sweldens, and R. Baraniuk. Nonlinear wavelet transforms for image coding via lifting. In *IEEE Transactions on Image Processing*, pages 1449–1459, 2003.

[CDSY97] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. Lossless image compression using integer to integer wavelet transforms. In *International Conference on Image Processing (ICIP), Vol. I*, pages 596–599, 1997.

[Che95] E.V. Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. Technical report, CERN Report, 1995.

[Che01] H. Chen. *Fast Volume Rendering and Deformation Algorithms*. PhD thesis, University Mannheim, 2001.

[CHH+03] C.S. Co, B. Heckel, H. Hagen, B. Hamann, and K.I. Joy. Hierarchical clustering for unstructured volumetric scalar fields. In *IEEE Visualization*, pages 43–51, 2003.

[CHM01] H. Chen, J. Hesser, and R. Männer. Fast free-form volume deformation using inverse-ray-deformation. In *VIIP*, pages 163–168, 2001.

[Chr05] M. Christen. Ray tracing on gpu. Master's thesis, Diploma Thesis, University of Applied Sciences Basel, 2005.

[Chu88] C. Chui. *Multivariate Splines*. SIAM, 1988.

[CICS05] S.P. Callahan, M. Ikits, J.L.D. Comba, and C.T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[CKG99] B. Csébfalvi, A. König, and E. Gröller. Fast maximum intensity projection using binary shear-warp factorization. In *WSCG '99*, pages 47–54, 1999.

[CLP02] G. Caumon, B. Lévy, and J-C. Paul. Combinatorial data structures for volume rendering unstructured grids. *Submitted: IEEE Transactions on Visualization and Computer Graphics*, 2002.

[CMH+01] B. Csébfalvi, L. Mroz, H. Hauser, A. König, and E. Gröller. Fast visualization of object contours by non-photorealistic volume rendering. *Comput. Graph. Forum*, 20(3), 2001.

[CMM+97] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[CMS01] H. Carr, T. Möller, and J. Snoeyink. Simplicial subdivisions and sampling artifacts. In *IEEE Visualization*, volume 37, pages 99–106, 2001.

[CN94] T.J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical report, University of North Carolina at Chapel Hill, 1994.

[CPC84] R.L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *Conference on Computer Graphics and Interactive Techniques*, pages 137–145, 1984.

[Cro84] F.C. Crow. Summed-area tables for texture mapping. In *Computer Graphics and Interactive Techniques*, pages 207–212, 1984.

[CT82] R.L. Cook and K.E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24, 1982.

[Dau88] I. Daubechies. Orthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, pages 909–996, 1988.

[Dau92] I. Daubechies. *Ten Lectures on Wavelets*. SIAM, 1992.

[DCH88] R.A. Derbin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4):65–74, 1988.

[Dod97] N.A. Dodgson. Quadratic interpolation for image resampling. *IEEE Transactions on Image Processing*, 6(9):1322–1326, September 1997.

[EHK+05] K. Engel, M. Hadwiger, J.M. Kniss, A.E. Lefohn, C. Rezk-Salama, and D. Weiskopf. Course optnotes 28: Real-time volume graphics. *IEEE Visualization*, 2005.

[EHKRS02] K. Engel, M. Hadwiger, J. Kniss, and C. Rezk-Salama. High-quality volume graphics on consumer pc hardware. *Course OPTnotes for Course 42 at SIGGRAPH 2002*, 2002.

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, 2001.

[ER00] D. Ebert and P. Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings Visualization 2000*, pages 195–202, 2000.

[Far86] G. Farin. Triangular bernstein-bézier patches. *Computer Aided Geometric Design*, 3:83–127, 1986.

[Far02] G. Farin. *History of Curves and Surfaces in CAGD*, chapter 1, pages 1–22. Handbook of CAGD, Elsevier, 2002.

[FRD06] C. Fox, H.E. Romeijn, and J.F. Dempsey. Fast voxel and polygon ray-tracing algorithms in intensity modulated radiation therapy treatment planning. *Med. Phys.*, 33(5):1364–1371, 2006.

[FS97] J. Freund and K. Sloan. Accelerated volume rendering using homogeneous region encoding. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 191–197, 1997.

[FvDFH97] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles And Practice, 2nd Edition In C.* Addison-Wesley, 2 edition, 1997.

[Gal91] D. Le Gall. Mpeg: a video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, 1991.

[Gar90] M.P. Garrity. Raytracing irregular volume data. In *Symposium on Volume Visualization*, pages 35–40, 1990.

[GBKG04a] S. Grimm, S. Bruckner, A. Kanitsar, and M. E. Gröller. Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In *Proceedings IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics*, pages 1–8, 2004.

[GBKG04b] S. Grimm, S. Bruckner, A. Kanitsar, and M.E. Gröller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computer & Graphics*, 28(5):719–729, 2004.

[Geo03] A.S. Georghiades. Recovering 3-d shape and reflectance from a small number of photographs. *Eurographics Symposium on Rendering 2003*, 2003.

[GG98] B. Gooch and A. Gooch. *Computer Vision.* Springer, 1998.

[GG01] B. Gooch and A. Gooch. *Non-Photorealistic Rendering.* AK Peters, Ltd., 2001.

[GGSC98] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452, 1998.

[Gla84] A.S. Glassner. Space subdivision for fast ray tracing. In *IEEE Computer Graphics and Applications*, volume 4, pages 15–22, 1984.

[Gla90] A.S. Glassner. Multidimensional sum tables. *Graphics Gems*, pages 376–381, 1990.

[GRS+02] S. Guthe, S. Roettger, A. Schieber, W. Straßer, and T. Ertl. High-quality unstructured volume rendering on the pc platform. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 119–125, 2002.

[GS01] S. Guthe and W. Straßer. Real-time decompression and visualization of animated volume data. In *IEEE Visualization*, pages 349–356, 2001.

[GSG+99] B. Gooch, P.-P.J. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 31–38, 1999.

[GTGB84] C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. In *Conference on Computer Graphics and Interactive Techniques*, pages 213–222, 1984.

[GW02] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Prentice-Hall, Inc., 2 edition, 2002.

[GWGS02] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive rendering of large volume data sets. In *IEEE Visualization*, pages 53–60, 2002.

[HBH03] M. Hadwiger, C. Berger, and H. Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, 2003.

[HCSM00] J. Huang, R. Crawfis, N. Shareef, and K. Mueller. Fastsplats: optimized splatting on rectilinear grids. In *IEEE Visualization*, pages 219–226, 2000.

[HHS93] H.C. Hege, T. Höllerer, and D. Stalling. Volume rendering - mathematical models and algorithmic aspects. Technical report, Konrad-Zuse-Zentrum für Informationstechnik, 1993.

[HKG00] J. Hladůvka, A. König, and M.E. Gröller. Curvature-based transfer functions for direct volume rendering. In *Spring Conference on Computer Graphics*, volume 16, pages 58–65, 2000.

[HL93] J. Hoschek and D. Lasser. *Fundamentals of Computer Aided Geometric Design*. A K Peters, Ltd., 1993.

[HMBG00] H. Hauser, L. Mroz, G.-I. Bischi, and E. Gröller. Two-level volume rendering – fusing mip and dvr. In *Proceedings of IEEE Visualization 2000*, pages 211–218, 2000.

[HMK+95] J. Hesser, R. Männer, G. Knittel, W. Straßer, H.-P. Pfister, and A. Kaufman. Three architectures for volume rendering. *Computer Graphics Forum*, 14(3):111–122, 1995.

[HN00] D. Holliday and G. Nielson. Progressive volume models for rectilinear data using tetrahedral coons volumes. *Data Visualization*, pages 83–92, 2000.

[HQK05] W. Hong, F. Qiu, and A. Kaufman. Gpu-based object-order ray-casting for large datasets. In *Volume Graphics*, pages 177–185, 2005.

[HTHG01] M. Hadwiger, T. Theul, H. Hauser, and E. Gröller. Hardware-accelerated hiqh-quality filtering on pc graphics hardware. In *Proceedings of Vision, Modeling, and Visualization 2001*, 2001.

[HVTH02] M. Hadwiger, I. Viola, T. Theußl, and H. Hauser. Fast and flexible high-quality texture filtering with tiled high-resolution filters. In *Proceedings of Vision, Modeling, and Visualization 2002*, 2002.

[HZ03] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2 edition, 2003.

[IFP95] V. Interrante, H. Fuchs, and S. Pizer. Enhancing transparent skin surfaces with ridge and valley lines. In *VIS '95: Proceedings of the 6th conference on Visualization '95*, page 52, 1995.

[IFP96] V. Interrante, H. Fuchs, and S. Pizer. Illustrating transparent surfaces with curvature-directed strokes. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 211–ff, 1996.

[IFP97] V. Interrante, H. Fuchs, and S.M. Pizer. Conveying the 3d shape of smoothly curving transparent surfaces via texture. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):98–117, 1997.

[Int97] V. Interrante. Illustrating surface shape in volume data via principal direction-driven 3d line integral convolution. In *Conference on Computer Graphics and Interactive Techniques*, pages 109–116, 1997.

[IP99] I. Ihm and S. Park. Wavelet-based 3d compression scheme for interactive visualization of very large volume data. In *Computer Graphics Forum*, volume 18, pages 3–15, 1999.

[JWH+04] Y. Jang, M. Weiler, M. Hopf, J. Huang, D.S. Ebert, K.P. Gaither, and T. Ertl. Interactively visualizing procedurally encoded scalar fields. In *Symposium on Visualization*, pages 35–44, 2004.

[Kaj86] J.T. Kajiya. The rendering equation. *International Conference on Computer Graphics and Interactive Techniques*, 20(4):143–150, 1986.

[Kat01] F. Katscher. *Die kubischen Gleichungen bei Nicolo Tartaglia*. Verlag der österreichischen Akademie der Wissenschaften – Austrian Academy of Sciences Press, 2001.

[Kau91] A.E. Kaufman. 3d volume visualization. *Advances in Computer Graphics VI, Images: Synthesis, Analysis, and Interaction*, pages 175–203, 1991.

[KD98] G. Kindlmann and J.W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proceedings of IEEE Volume Visualization '98*, pages 79–86, 1998.

[KE02] M. Kraus and T. Ertl. Adaptive texture maps. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 7–15, 2002.

[Kin98] G. Kindlmann. *Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering*. PhD thesis, Cornell University, 1998.

[KKH01] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proceedings of IEEE Visualization 2001*, pages 255–262, 2001.

[KL04] F. Karlsson and C.J. Ljungstedt. Ray tracing fully implemented on programmable graphics hardware. Master's thesis, Diploma Thesis, Chalmers Technical University, 2004.

[KLS96] R. Klein, G. Liebich, and W. Strasser. Mesh reduction with error control. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 311–318, 1996.

[KM05] A. Kaufman and K. Mueller. *Overview of Volume Rendering*, chapter Chapter for The Visualization Handbook. Academic Press, 2005.

[KPHE02] J. Kniss, S. Premoze, C. Hansen, and D. Ebert. Interactive translucent volume rendering and procedural modeling. In *IEEE Visualization*, pages 109–116, 2002.

[Kru90] W. Krueger. The application of transport theory to visualization of 3d scalar data fields. *IEEE Visualization, Proceedings of the 1st conference on Visualization '90*, pages 273–280, 1990.

[KTH+05] A. Krüger, C. Tietjen, J. Hintze, B. Preim, I. Hertel, and G. Strauss. Interactive visualization for neck dissection planning. In *Proceedings of EuroVis 2005*, pages 295–302, 2005.

[KvH84] J.T. Kajiya and B.P. von Herzen. Ray tracing volume densities. *International Conference on Computer Graphics and Interactive Techniques*, 18:165–174, 1984.

[KW03] J. Krüger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *IEEE Visualization*, pages 38–44, 2003.

[KWTM03] G.L. Kindlmann, R.T. Whitaker, T. Tasdizen, and T. Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *IEEE Visualization*, pages 513–520, 2003.

[Lac95] P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation.* PhD thesis, Stanford University, 1995.

[LC87] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *In Proc. SIGGRAPH, Computer Graphics*, 21(4):163–169, 1987.

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

[Lev90] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[Lev92] M. Levoy. Volume rendering using the fourier projection-slice theorem. In *Graphics interface '92*, pages 61–69, 1992.

[LGS99] T.M. Lehmann, C. Gönner, and K. Spitzer. Survey: Interpolation methods in medical image processing. *IEEE Transactions on Medical Imaging*, 18(11):1049–1075, November 1999.

[LH99] D. Laur and P. Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. *SIGGRAPH Computer and Graphics*, 25(4):285–288, 1999.

[LHJ99a] E. LaMar, B. Hamann, and K.I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *IEEE Visualization*, pages 355–361, 1999.

[LHJ99b] E.C. LaMar, B. Hamann, and K.I. Joy. High-quality rendering of smooth isosurfaces. *Journal of Visualization and Computer Animation*, 10:79–90, 1999.

[LHSW03] F. Losasso, H. Hoppe, S. Schaefer, and J. Warren. Smooth geometry images. In *Eurographics Symposium on Geometry Processing 2003*, pages 138–145, 2003.

[LK02] W. Li and A. Kaufman. Accelerating volume rendering with texture hulls. In *IEEE Symposium on Volume Visualization and Graphics*, pages 115–122, 2002.

[LL94] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH Computer Graphics and Interactive Techniques*, pages 451–458, 1994.

[LLVT03] T. Lewiner, H. Lopes, A.W. Vieira, and G. Tavares. Efficient implementation of marching cubes' cases with topological guarantees. *Journal of Graphics Tools*, 8(2):1–15, 2003.

[LM02] E.B. Lum and K.-L. Ma. Hardware-accelerated parallel non-photorealistic volume rendering. In *International Symposium on Non-photorealistic Animation and Rendering*, pages 67–ff, 2002.

[LM05] S. Li and K. Mueller. Spline-based gradient filters for high-quality refraction computations in discrete datasets. In *Eurographics / IEEE VGTC Symposium on Visualization*, pages 217–222, 2005.

[LMC01] E.B. Lum, K.L. Ma, and J. Clyne. Texture hardware assisted rendering of time-varying volume data. In *IEEE Visualization*, pages 263–270, 2001.

[LME+02] A. Lu, C.J. Morris, D.S. Ebert, P. Rheingans, and C. Hansen. Non-photorealistic volume rendering using stippling techniques. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 211–218, 2002.

[LMK03] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *IEEE Visualization*, pages 42–50, 2003.

[LSM03] E.B. Lum, A. Stompel, and K.-L. Ma. Using motion to illustrate static 3d shape- kinetic visualization. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):115–126, 2003.

[Mal89] S.G. Mallat. Multiresolution approximations and wavelet orthogonal bases of $l^2(r)$. *Trans. Amer. Math. Soc.*, 315(1):69–87, 1989.

[Mal93] T. Malzbender. Fourier volume rendering. *ACM Trans. Graph.*, 12(3):233–250, 1993.

[Mal99] S. Mallat. *A Wavelet Tour of Signal Processing*, volume Second Edition. Elsevier, 1999.

[Max95] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, pages 99–108, 1995.

[MB95] O. Monga and S. Benayoun. Using partial derivatives of 3d images to extract typical surface features. *Comput. Vis. Image Underst.*, 61(2):171–189, 1995.

[MBC93] N. Max, B. Becker, and R. Crawfis. Flow volumes for interactive vector field visualization. In *IEEE Visualization*, pages 19–24, 1993.

[MBF92] O. Monga, S. Benayoun, and O. Faugeras. From partial derivatives of 3D volumic images to ridge lines. In *IEEE Conference on Vision and Pattern Recognition (CVPR)*, 1992.

[MC98] K. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. In *IEEE Visualization*, pages 239–245, 1998.

[MC01] W. Martin and E. Cohen. Representation and extraction of volumetric attributes using trivariate splines: a mathematical framework. *Solid Modelling and Applications*, pages 234–240, 2001.

[MD95] S. Mann and T. DeRose. Computing values and derivatives of bézier and b-spline tensor products. In *Computer Aided Geometric Design*, volume 12, pages 107–110, 1995.

[ME05] B. Mora and D.S. Ebert. Low-complexity maximum intensity projection. *ACM Trans. Graph.*, 24(4):1392–1416, 2005.

[MG02] M. Meissner and S. Guthe. Interactive lighting models and pre-integration for volume rendering on pc graphics accelerators. In *Graphics Interface*, pages x–x, 2002.

[MGK99] L. Mroz, E. Gröller, and A. König. Real-time maximum intensity projection. In *Data Visualization '99*, pages 135–144. Springer-Verlag Wien, 1999.

[MH92] D. Mitchell and P. Hanrahan. Illumination from curved reflectors. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 283–291, 1992.

[MHC90] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. In *Workshop on Volume Visualization*, pages 27–33, 1990.

[MHG00] L. Mroz, H. Hauser, and E. Gröller. Interactive high-quality maximum intensity projection. In *EUROGRAPHICS '2000*, pages 341–350, 2000.

[MHIL02] K.-L. Ma, A. Hertzmann, V. Interrante, and E.B. Lum. Siggraph 2002 course 23: Recent advances in non-photorealistic rendering for art and visualization. In *Course OPTnotes SIGGRAPH 2002*, 2002.

[MJC01] B. Mora, J.-P. Jessel, and R. Caubet. Visualization of isosurfaces with parametric cubes. In *Eurographics*, pages 377–384, 2001.

[MKG00] L. Mroz, A. König, and E. Gröller. Maximum intensity projection at warp speed. *Computers and Graphics*, 24(3):343–352, 2000.

[MKS98] M. Meiner, U. Kanus, and W. Straßer. Vizard ii: A pcicard for real-time volume rendering. In *Siggraph/Eurographics Workshop on Graphics Hardware*, pages 61–67, 1998.

[MKW+04] G. Marmitt, A. Kleer, I. Wald, H. Friedrich, and P. Slusallek. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *Vision, Modelling, and Visualization 2003 (VMV)*, 2004.

[ML94] S.R. Marschner and R.J. Lobb. An evaluation of reconstruction filters for volume rendering. In *IEEE Visualization*, pages 100–107, 1994.

[MMC99] K. Mueller, T. Möller, and R. Crawfis. Splatting without the blur. In *IEEE Visualization*, pages 363–370, 1999.

[MMK+98] T. Möller, K. Mueller, Y. Kurzion, R. Machiraju, and R. Yagel. Design of accurate and smooth filters for function and derivative reconstruction. In *IEEE Symposium on Volume Visualization*, pages 143–151, 1998.

[MMMY97a] T. Möller, R. Machiraju, K. Mueller, and R. Yagel. A comparison of normal estimation schemes. In *IEEE Visualization*, pages 19–27, 1997.

[MMMY97b] T. Möller, R. Machiraju, K. Mueller, and R. Yagel. Evaluation and design of filters using a taylor series expansion. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):184–199, 1997.

[MN88] D.P. Mitchell and A.N. Netravali. Reconstruction filters in computer-graphics. In *Conference on Computer Graphics and Interactive Techniques*, pages 221–228, 1988.

[MPWW00] M. Meiner, H. Pfister, R. Westermann, and C.M. Wittenbrink. Volume visualization and volume rendering techniques. *Eurographics Tutorial, 2000*, 2000.

[MSHC99] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999.

[Mur93] S. Muraki. Volume data and wavelet transforms. *IEEE Computer and Graphics Applications*, 13(4):50–56, 1993.

[NCKG00] L. Neumann, B. Csébfalvi, A. König, and E. Gröller. Gradient estimation in volume data using 4D linear regression. In *Computer Graphics Forum (Eurographics 2000)*, volume 19(3), pages 351–358, 2000.

[NH93] P. Ning and L. Hesselink. Fast volume rendering of compressed data. In *IEEE Visualization*, pages 11–18, 1993.

[NM03] N. Neophytou and K. Mueller. Post-convolved splatting. In *Symposium on Data Visualisation*, pages 223–230, 2003.

[NM05] N. Neophytou and K. Mueller. Gpu accelerated image aligned splatting. In *Workshop on Volume Graphics*, pages 247–254, 2005.

[NRSZ04] G. Nürnberger, C. Rössl, H.P. Seidel, and F. Zeilfelder. Quasi-interpolation by quadratic piecewise polynomials in three variables. In *Computer Aided Geometric Design*, volume 22, pages 221–249, 2004.

[NS01] K.G. Nguyen and D. Saupe. Rapid high quality compression of volume data for visualization. *Compututer Graphics Forum*, 20(3), 2001.

[NSW02] Z. Nagy, J. Schneider, and R. Westermann. Interactive volume illustration. In *In Proceedings of Vision, Modeling and Visualization Workshop '02*, 2002.

[OLG+05] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, 2005.

[OM01] J. Orchard and T. Möller. Accelerated splatting using a 3d adjacency data structure. In *GRIN'01: No description on Graphics interface 2001*, pages 191–200, 2001.

[PBMH02] T.J. Purcell, I. Buck, W.R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[PH04] M. Pharr and G. Humphrey. *Physically Based Rendering: from theory to implementation.* Morgan Kaufmann, 1 edition, 2004.

[PHK+99] H.-P. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *Annual Conference on Computer Graphics and Interactive Techniques*, pages 251–260, 1999.

[PHK+03] V. Pekar, D. Hempel, G. Kiefer, M. Busch, and J. Weese. Efficient visualization of large medical image datasets on standard pc hardware. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, pages 135–140, 2003.

[Pho75] B.T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[PSL+98] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization*, pages 233–238, 1998.

[PTVF99] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C – The Art of Scientific Computing*. Cambridge University Press, 2 edition, 1999.

[Pur04] T.J. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004.

[RE02] S. Roettger and T. Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *IEEE Symposium on Volume Visualization and Graphics*, pages 23–28, 2002.

[RGW+03] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering. In *Symposium on Data Visualisation*, pages 231–238, 2003.

[RKE00] S. Roettger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *IEEE Visualization*, pages 109–116, 2000.

[Rod99] F.F. Rodler. Wavelet based 3d compression with fast random access for very large volume data. In *Pacific Conference on Computer Graphics and Applications*, pages 108–116, 1999.

[RSEB+00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware*, pages 109–118, 2000.

[RSEH05] C. Rezk-Salama, K. Engel, and F.V. Higuera. The openqvis project. ., 2005.

[RUL00] J. Revelles, C. Urena, and M. Lastra. An efficient parametric algorithm for octree traversal. In *International Conference on Computer Graphics and Visualization*, 2000.

[RZNS03] C. Rössl, F. Zeilfelder, G. Nürnberger, and H.-P. Seidel. Visualization of volume data with quadratic super splines. In *IEEE Visualization*, pages 52–60, 2003.

[RZNS04a] C. Rössl, F. Zeilfelder, G. Nürnberger, and H.-P. Seidel. Reconstruction of volume data with quadratic super splines. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):397–409, 2004.

[RZNS04b] C. Rössl, F. Zeilfelder, G. Nürnberger, and H.-P. Seidel. Spline approximation of general volumetric data. In *SM '04: Proceedings of the ninth ACM symposium on Solid modeling and applications*, pages 71–82, 2004.

[Sab88] P. Sabella. A rendering algorithm for visualizing 3d scalar fields. *In Proc. SIGGRAPH, Computer Graphics*, 22(4):51–58, 1988.

[Sam90] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 2 edition, 1990.

[SBM94] C.M. Stein, B.G. Becker, and N.L. Max. Sorting and hardware assisted rendering for volume visualization. In *Symposium on Volume Visualization*, pages 83–89, 1994.

[SBS02] B.-S. Sohn, C. Bajaj, and V. Siddavanahalli. Feature based volumetric video compression for interactive playback. In *IEEE Symposium on Volume Visualization and Graphics*, pages 89–96, 2002.

[SCBC05] C.T. Silva, J.L.D. Comba, F.F. Bernardon, and S.P. Callahan. A survey of gpu-based volume rendering of unstructured grids. *Revista de Informtica Terica e Aplicada*, 12(2):9–29, 2005.

[SCC+04] M. Straka, M. Cervenanský, A. La Cruz, A. Köchl, M. Srámek, M. E. Gröller, and D. Fleischmann. The vesselglyph: Focus and context visualization in ct-angiography. In *IEEE Visualization 2004*, pages 392–385, 2004.

[SFYC43] R. Shekhar, E. Fayyad, R. Yagel, and J.F. Cornhill. Octree-based decimation of marching cubes surfaces. In *IEEE Visualization*, page 1996, 335–343.

[SH94] B.T. Stander and J.C. Hart. A lipschitz method for accelerated volume rendering. *Proceedings of the 1994 Symposium on Volume Visualization*, 8(2):107–114, 1994.

[SH05] C. Sigg and M. Hadwiger. Fast third-order texture filtering. In *In GPU Gems 2, Matt Pharr (ed.)*, pages 313–329. Addison-Wesley, 2005.

[Sha49] C.E. Shannon. Communications in the presence of noise. In *Proc. of the IRE*, volume 37, pages 10–21, January 1949.

[SHM04] G. Schlosser, J. Hesser, and R. Männer. Volume rendering on one rle compressed data set by a new combination of ray-casting and shear-warp.

In *SIGGRAPH '04: Computer Graphics and Interactive Techniques*, page Full Conference DVD ROM, 2004.

[SHZ+05] G. Schlosser, J. Hesser, F. Zeilfelder, C. Rössl, R. Männer, G. Nürnberger, and H.-P. Seidel. Fast visualization by shear-warp on quadratic super-spline models using wavelet data decompositions. In *IEEE Visualization*, pages 351–358, 2005.

[Sid85] L. Siddon. Fast calculation of the exact radiological path for a threedimensional ct array. *Med. Phys.*, pages 252–258, 1985.

[SKLE03] J.P. Schulze, M. Kraus, U. Lang, and T. Ertl. Integrating pre-integration into the shear-warp algorithm. In *Eurographics/IEEE TVCG Workshop on Volume Graphics*, pages 109–118, 2003.

[SL02] J.P. Schulze and U. Lang. The parallelization of the perspective shear-warp volume rendering algorithm. In *Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 61–69, 2002.

[SM02] J. Sweeney and K. Mueller. Shear-warp deluxe: the shear-warp algorithm revisited. In *Symposium on Data Visualisation*, pages 95–103, 2002.

[SMM+97] J.E. Swan, K. Mueller, T. Möller, N. Shareef, R. Crawfis, and R. Yagel. An anti-aliasing technique for splatting. In *IEEE Visualization*, pages 197–205, 1997.

[SNL01] J.P. Schulze, R. Niemeier, and U. Lang. The perspective shear-warp algorithm in a virtual environment. In *IEEE Visualization*, pages 207–214, 2001.

[SS96] P. Schröder and W. Sweldens. Building your own wavelets at home. in: Wavelets in computer graphics. *Siggraph, Course OPTnotes*, 1996.

[SSKE05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Volume Graphics*, pages 187–195, 2005.

[ST90] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Symposium on Volume Visualization*, volume 24, pages 63–70, 1990.

[SW91] J. Spackman and P. Willis. The smart navigation of a ray through an oct-tree. *Computer and Graphics*, 15(2):185–194, 1991.

[SW03] J. Schneider and R. Westermann. Compression domain volume rendering. In *IEEE Visualization*, pages 39–47, 2003.

[SZ05] T. Sorokina and F. Zeilfelder. Local quasi-interpolation by cubic c^1 splines on type-6 tetrahedral partitions. *IMA Journal of Numerical Analysis*, pages 1–28, 2005.

[TC00] S.M.F. Treavett and M. Chen. Pen-and-ink rendering in volume visualisation. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 203–210, 2000.

[TMG01] T. Theußl, T. Möller, and M.E. Gröller. Optimal regular volume sampling. In *IEEE Visualization*, pages 91–98, 2001.

[UAE93a] M. Unser, A. Aldroubi, and M. Eden. B-Spline signal processing: Part I—Theory. *IEEE Transactions on Signal Processing*, 41(2):821–833, February 1993.

[UAE93b] M. Unser, A. Aldroubi, and M. Eden. B-Spline signal processing: Part II—Efficient design and applications. *IEEE Transactions on Signal Processing*, 41(2):834–848, February 1993.

[UH99] J. K. Udupa and G. T. Herman. *3D Imaging in Medicine*, volume 2 edition. CRC Press, 1999.

[Vet86] M. Vetterli. Filter banks allowing perfect reconstruction. *Signal Processing*, 10(3):219–244, 1986.

[VG05] I. Viola and M. E. Gröller. Smart visibility in visualization. In *Proceedings of EG Workshop on Computational Aesthetics Computational Aesthetics in Graphics, Visualization and Imaging*, pages 209–216, 2005.

[VHMK99] B. Vettermann, J. Hesser, R. Männer, and A. Kugel. Implementation of algorithmically optimized volume rendering on fpga-hardware. In *Proc. Late Breaking Hot Topics, IEEE Visualization '99.*, pages ff–ff, 1999.

[VK95] M. Vetterli and J. Kovacevic. *Wavelets and Suband Coding*. Prentice Hall, 1995.

[VKG04] I. Viola, A. Kanitsar, and M.E. Gröller. Importance-driven volume rendering. In *Proceedings of IEEE Visualization'04*, pages 139–145, 2004.

[Š94] M. Šrámek. Fast surface rendering from raster data by voxel traversal using chessboard distance. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 188–195, 1994.

[WBH+05] M. Weiler, R.P. Botchen, J. Huang, Y. Jang, S. Stegmeier, K.P. Gaither, D.S. Ebert, and T. Ertl. Hardware-assisted feature analysis and visualization of procedurally encoded multifield volumetric data. In *IEEE Computer Graphics and Applications, to appear*, 2005.

[Wes89] L. Westover. Interactive volume rendering. In *VVS '89: Proceedings of the 1989 Chapel Hill workshop on Volume visualization*, pages 9–16, 1989.

[Wes90] L. Westover. Footprint evaluation for volume rendering. In *Computer Graphics and Interactive Techniques*, pages 367–376, 1990.

[Wes94] R. Westermann. A multiresolution framework for volume rendering. In *Symposium on Volume Visualization*, pages 51–58, 1994.

[Wes95] R. Westermann. Compression domain rendering of time-resolved volume data. In *IEEE Visualization*, pages 168–174, 1995.

[WG90] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. In *Symposium on Volume Visualization*, pages 57–62, 1990.

[WG92] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[WKME03a] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *IEEE Visualization*, pages 44–52, 2003.

[WKME03b] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):163–175, 2003.

[WMFC02] B. Wylie, K. Moreland, L.A. Fisk, and P. Crossno. Tetrahedral projection using vertex shaders. In *IEEE Symposium on Volume Visualization and Graphics*, pages 7–12, 2002.

[WMK04] A. Wood, B. McCane, and S.A. King. Ray tracing arbitrary objects on the gpu. In *Proceedings of Image and Vision Computing New Zealand (IVCNZ 2004)*, pages 21–23, 2004.

[WMS98] P.L. Williams, N.L. Max, and C.M. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, 1998.

[WNDS99] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The official guide to learning OpenGL, Version 1.2.* Addison-Wesley, 3 edition, 1999.

[WS01] I. Wald and P. Slusallek. State of the art in interactive ray tracing. *EUROGRAPHICS, State of the Art Reports*, pages 21–42, 2001.

[WWH+00] M. Weiler, R. Westermann, C. Hansen, K. Zimmermann, and T. Ertl. Level-of-detail volume rendering via 3d textures. In *IEEE Symposium on Volume Visualization*, pages 7–13, 2000.

[XC02] D. Xue and R. Crawfis. Efficient splatting using modern graphics hardware. *Graphics Tools*, 8(3):1–21, 2002.

[XZC05] D. Xue, C. Zhang, and R. Crawfis. isbvr: Isosurface-aided hardware acceleration techniques for slice-based volume rendering. In *Volume Graphics*, pages 207–215, 2005.

[YK92] R. Yagel and A. Kaufman. Template-based volume viewing. In *Computer Graphics Forum (EUROGRAPHICS '92 Proceedings)*, volume 11, pages 153–167, 1992.

[YS93] R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *IEEE Visualization*, pages 62–69, 1993.

[ZKV92] K.J. Zuiderveld, A.H. Koning, and M.A. Viergever. Acceleration of ray-casting using 3-d distance transforms. In *Proc. SPIE, Visualization in Biomedical Computing '92*, volume 1808, pages 324–335, 1992.

[ZPvBG02] M. Zwicker, H.P. Pfister, J. van Baar, and M. Gross. Ewa splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.

[ZRB+04] M. Zwicker, J. Räsänen, M. Botsch, C. Dachsbacher, and M. Pauly. Perspective accurate splatting. In *Graphics Interface*, pages 247–254, 2004.