Direct Volume Visualization of Geometrically Unpleasant Meshes

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Martin Kraus

aus Erlangen

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. R. Westermann
	Prof. Dr. N. Max

Tag der mündlichen Prüfung: 24. April 2003

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2003

David: *You're O.K.?*
George: *Yeah. Not fair, you know?*
You get used to one thing and then ...
David: *I know, it's not.*

Dialog from the movie *Pleasantville*

Contents

List of Abbreviations and Acronyms **5**

Abstract and Chapter Summaries (in English and German) **7**

1 Introduction **31**
1.1 Outline of This Thesis 32
1.2 Acknowledgments 33

2 Direct Volume Visualization **35**
2.1 Applications 36
2.2 Visualization Pipeline 37
2.3 Volume Rendering Algorithms 39
2.4 Ray Integration 41
2.4.1 Volume Rendering Integral 41
2.4.2 Pre- and Post-Classification 42
2.4.3 Numerical Integration 43
2.5 Pre-Integrated Classification 48
2.5.1 Ray Integration with Pre-Integrated Classification 48
2.5.2 Accelerated Approximative Pre-Integration 51
2.5.3 Applications to Volume Rendering Algorithms 52
2.6 Classification of Meshes 53
2.6.1 Structured Meshes 53
2.6.2 Unstructured Meshes 54
2.6.3 Hybrid and Hierarchical Meshes 54
2.7 Interpolation in Meshes 55
2.7.1 Nearest-Neighbor Interpolation 55
2.7.2 Linear, Bilinear, and Trilinear Interpolation 55
2.7.3 Linear Interpolation in Simplices 57
2.7.4 Higher-Order Interpolation 58
2.8 Geometrically Unpleasant Meshes 59

3 Non-Uniform Meshes **61**
3.1 Pre-Integrated Cell Projection . . . 61
3.1.1 Projected Tetrahedra Algorithms . . . 62
3.1.2 Pre-Integrated Classification for Projected Tetrahedra . . . 63
3.2 Hardware-Assisted Resampling . . . 77
3.3 Hardware-Assisted Ray Casting . . . 79

4 Non-Convex and Cyclic Meshes **85**
4.1 Convexification of Non-Convex Meshes . . . 85
4.2 Edge Collapses in Non-Convex Meshes . . . 88
4.2.1 Edge Collapses in Convex Meshes . . . 88
4.2.2 Edge Collapses in Convexified Meshes . . . 89
4.3 Cell Sorting for Non-Convex and Cyclic Meshes . . . 95
4.3.1 Cell Sorting for Convex, Acyclic Meshes . . . 96
4.3.2 Cell Sorting for Convex, Cyclic Meshes . . . 98
4.3.3 Cell Sorting for Non-Convex, Cyclic Meshes . . . 101
4.4 Cell Projection for Cyclic Meshes . . . 104
4.4.1 Rendering of Cyclic Occlusions of Polygons . . . 104
4.4.2 Rendering of Cyclic, Convex Polyhedral Cells . . . 109

5 Non-Simplicial and Non-Adaptive Meshes **111**
5.1 Texture-Based Pre-Integrated Volume Rendering . . . 112
5.2 Topology-Guided Downsampling . . . 113
5.2.1 Algorithm . . . 115
5.2.2 Examples . . . 121
5.3 Adaptive Volume Textures . . . 124
5.3.1 Adaptive Texture Mapping in Two Dimensions . . . 124
5.3.2 Volume Rendering with Adaptive Volume Textures . . . 133

6 Geometrically Unpleasant Meshes in General **135**
6.1 Proposed Solutions . . . 136
6.2 Problem-Solving Strategies . . . 138
6.2.1 Understanding the Problem . . . 138
6.2.2 Simplifying the Problem . . . 140
6.2.3 Solving the Problem . . . 141
6.3 Further Problems . . . 143

Color Plates **145**

Bibliography **151**

Peggy Jane: *Hey, M.S., how you doin'?*
Jennifer: *Cool, P.J., how you doin'?*
Peggy Jane: *Cool. Cool.*
Jennifer: *Cool.*

Dialog from the movie *Pleasantville*

List of Abbreviations and Acronyms

△	triangle
⊠	tetrahedron
2D	two-dimensional
3D	three-dimensional
a.	answer
BFS	breadth-first search
BSP	binary space partitioning
CPU	central processing unit
CT	computer tomography
CTA	CT angiography
def.	definition
depend.	dependent
DFS	depth-first search
d.h.	das heißt (that is)
Dr. rer. nat.	Doctor rerum naturalium (Doctor of Science)
e.g.	exempli gratia (for example)
et al.	et alii, et aliae, et alia (and others)
etc.	et cetera (and so forth)
HIAC	high accuracy
i.e.	id est (that is)
KB	kilobyte
MAD	multiply & add
MB	megabyte
MHz	megahertz
MPVO	meshed polyhedra visibility ordering
MPVOC	MPVO cyclic
MPVONC	MPVO non-convex
MPVONCC	MPVO non-convex cyclic
MR	magnetic resonance
M.S.	Mary Sue
O.K.	"oll korrect"
orig. impl.	original implementation
pixel	picture element
P.J.	Peggy Jane
Prof. Dr.	Professor Doctor
PT	projected tetrahedra
q.	question
RGB	red, green, and blue
RGBA	red, green, blue, and alpha
texel	texture element
voxel	volume element
XMPVO	extended MPVO
z.B.	zum Beispiel (for example)

David: *I thought the books were blank?*
Will: *They were.*
Jennifer: *O.K. This was not my fault.*
When they asked me what it was about,
I didn't remember
because I read it like back in tenth grade.
When I told them what I did remember,
that's when the pages filled in.
David: *The pages filled in?*
Jennifer: *Uh-huh, but only up until the part with the raft,*
'cause that's as far as I read.
Tommy: *Do you know how it ends?*
David: *Yeah, I do.*
Margaret: *So how does it end?*

Dialog from the movie *Pleasantville*

Abstract and Chapter Summaries (in English and German)

Abstract 7
Chapter Summaries 8
Zusammenfassung (Abstract in German) 18
Kapitelzusammenfassungen (Chapter Summaries in German) 19

Abstract

Interactive volume visualization (i.e., the visualization of scalar data defined on volumetric meshes in real time) is not only difficult to achieve for large meshes but it is also complicated by particular geometric features of volumetric meshes, e.g., non-uniform cells, non-convex boundaries, or visibility cycles.

This thesis addresses several of these geometric features and their unpleasant consequences with respect to direct volume visualization, which is one of the most successful techniques for interactive volume visualization. In order to overcome, or at least alleviate, these difficulties, several new algorithmic solutions are presented: pre-integrated cell projection and hardware-assisted ray casting for non-

uniform meshes, edge collapses in non-convex meshes, cell sorting and cell projection for non-convex and cyclic meshes, as well as texture-based pre-integrated volume rendering, topology-guided downsampling, and adaptive volume textures for non-simplicial volumetric meshes (i.e., non-tetrahedral meshes).

As this work cannot cover all geometrically unpleasant features of volumetric meshes, particular emphasis is put on a description of the development of the proposed algorithms. In fact, most of the presented techniques are (or may be interpreted as) generalizations, adaptations, or extensions of existing methods. The intention of explaining these origins is to motivate new solutions for those geometrically unpleasant features of meshes that were out of the scope of this work.

Chapter Summaries

The following summaries give a rather detailed overview of the contents of each chapter. The form of these summaries is as compact as possible in order to facilitate a quick orientation. Note that section headings are set in *italics*.

Chapter 1: Introduction

The primary goal of Chapter 1 is to introduce and motivate the subject of this thesis.

Direct volume visualization is often complicated by large data sets, high resolution output images, or the need for real-time interactive frame rates. Because of these difficulties, research on direct volume visualization usually focuses on "appropriate" meshes, in contrast to "geometrically unpleasant" meshes, which include any mesh with an exceptional or (for any reason) "difficult" geometric shape of its cells or boundary. Two reasons for this approach are, for example:

- It is sensible to solve the simple cases before addressing the more challenging cases.
- It is useful to address the most relevant problems (i.e., the most popular meshes) first.

However, there are also several disadvantages:

- geometrically unpleasant meshes occur in real-life applications;
- all meshes are geometrically unpleasant in some respects; and
- restricting research to particular meshes hampers progress in direct volume visualization in the long run.

These reasons motivate the subject of this thesis, i.e., algorithms for direct volume visualization of geometrically unpleasant meshes, in particular more efficient, more robust, and more general algorithms.

Section 1.1 (page 32) of the introduction gives a brief *outline of this thesis*, which is more detailed than the abstract and includes references to individual sections.

Chapter 2: Direct Volume Visualization

Chapter 2 introduces direct volume visualization and presents many of the methods and concepts employed in subsequent chapters, in particular

- applications of direct volume visualization (Section 2.1),
- a visualization pipeline for direct volume visualization (Section 2.2),
- volume rendering algorithms (Section 2.3),
- ray integration (Section 2.4),
- pre-integrated classification (Section 2.5),
- a basic classification of meshes (Section 2.6),
- data interpolation in meshes (Section 2.7), and
- an informal definition of geometrically unpleasant meshes (Section 2.8).

Direct volume visualization assigns opacities to all data points; thus, occluded data points are not necessarily invisible but may be visible through the semi-transparent, occluding points in front of them. Advantages of this technique are:

- It offers a continuous trade-off between the opaque occlusion and the visibility of data points.
- The rendering of isosurfaces and slices are special cases of direct volume rendering.
- The performance of direct volume rendering algorithms is often independent of the actual scalar values of the volumetric data.

Disadvantages include:

- The resulting images are often “nebulous” and “fuzzy”.
- The choice of appropriate colors and opacities for data points is not trivial.

In Section 2.1 (page 36), actual and potential *applications* of direct volume visualization are discussed, e.g., visualization of medical scans, data from other physical or chemical measurements, data from numeric simulations, or mathematical functions.

Several additional reasons for the limited popularity of direct volume visualization are mentioned, e.g., insufficient support by standard graphics hardware, a lack of non-commercial tools, uncomfortable and inefficient interfaces of existing tools, or the rare use of volume visualization for publications. Apart from these difficulties, there are also several unsolved problems in direct volume visualization, e.g., the specification of multi-dimensional transfer functions, the automatic generation of transfer functions, the visualization of time-dependent data sets, or real-time interactive rendering of large volumes.

Section 2.2 (page 37) describes the main stages of the *visualization pipeline*, i.e.,

- data acquisition (resulting in raw data),
- filtering (resulting in visualization data),
- mapping (resulting in geometric data), and
- rendering (resulting in image data).

Specializing this pipeline for direct volume visualization results in the following stages:

- data acquisition (resulting in raw data),
- filtering (resulting in volume data),
- classification and shading (resulting in color and opacity data), and
- ray integration (resulting in image data).

Several *volume rendering algorithms* are presented in Section 2.3 (page 39), in particular

- ray casting (see also Section 3.3),
- cell projection (see also Section 3.1),
- splatting,
- texture-based volume rendering (see also Section 5.1), and
- the shear-warp algorithm.

Section 2.4 (page 41) discusses *ray integration*, in particular

- the volume rendering integral for the integration of colors and opacities along viewing rays;
- transfer functions mapping scalar data to colors and opacities;
- the difference between pre- and post-classification, i.e., the difference between applying transfer functions before or after the interpolation of the scalar data;
- the numerical integration of the volume rendering integral by discretization; and
- an estimation of the required sampling rate by comparison with Carson's rule for frequency-modulated signals.

Pre-integrated classification is presented in Section 2.5 (page 48). This technique requires considerably lower sampling rates while still offering an improved accuracy. It was developed together with Stefan Röttger and Klaus Engel and was first published in [56] and [18].

The primary disadvantage of pre-integrated volume rendering is the required computation of a (possibly large) lookup table. Several techniques for the acceleration of this computation are presented:

- constant sampling rate,
- local updates of the lookup table, and
- the approximative computation of lookup tables with integral functions.

Pre-integrated classification may be applied to many volume rendering algorithms, in particular

- cell projection (see Section 3.1),
- texture-based volume rendering (see Section 5.1),
- ray casting (see Section 3.3, [30], and [47]), and
- the shear-warp algorithm ([57]).

A basic *classification of meshes* is presented in Section 2.6 (page 53). It distinguishes between

- structured meshes, e.g., uniform, rectilinear, and curvilinear meshes, and

- unstructured meshes, e.g., simplicial and zoo meshes.

There are meshes that do not fit into this classification, e.g., hybrid and hierarchical meshes. However, these meshes are not discussed in detail in this thesis.

Interpolation in meshes is discussed in Section 2.7 (page 55), in particular

- nearest-neighbor interpolation;
- linear interpolation on line segments, bilinear interpolation within rectangles, and trilinear interpolation in boxes; and
- linear interpolation in simplices, i.e., on line segments and within triangles and tetrahedra.

The computation of the bilinear and trilinear interpolation and of the linear interpolation within triangles and tetrahedra by sequences of linear interpolations on line segments is described in detail.

In Section 2.8 (page 59), *geometrically unpleasant meshes* are discussed with respect to the information provided in the preceding sections. In particular, the following issues are mentioned:

- consequences of a non-convex mesh boundary for volume rendering algorithms,
- the limitation of a continuous, piecewise linear interpolation to simplicial meshes,
- the restriction of hardware-supported interpolation to uniform meshes, and
- other unpleasant geometric features of meshes apart from the shape of cells and the boundary of meshes, e.g., visibility cycles and a non-adaptive mesh resolution.

Chapter 3: Non-Uniform Meshes

One of the basic problems of direct volume visualization of non-uniform meshes is the lack of native hardware support for any non-uniform mesh, i.e., the restriction to trilinear interpolation. Therefore, three hardware-assisted volume rendering algorithms for non-uniform meshes are presented in this chapter:

- pre-integrated cell projection (Section 3.1),
- hardware-assisted resampling (Section 3.2), and
- hardware-assisted ray casting (Section 3.3).

Pre-integrated cell projection is presented in Section 3.1 (page 61). It combines pre-integrated classification (see Section 2.5) with the Projected Tetrahedra algorithm (see [61]) and was first published together with Stefan Röttger and Thomas Ertl in [56]. The main features of pre-integrated cell projection are:

- decomposition of the projections of tetrahedra into triangles and hardware-accelerated rasterization of these triangles (as in the Projected Tetrahedra algorithm),
- hardware-accelerated interpolation of scalar data on the surface of tetrahedral cells and of the thickness of tetrahedra along viewing rays, and
- computation of the resulting colors and opacities by table lookups, which are implemented by hardware-accelerated three-dimensional texture mapping.

As the precomputation of the required three-dimensional textures is not possible at real-time interactive frame rates, an approximation is presented, which requires only two-dimensional texture mapping at the cost of a reduced image quality.

Pre-integrated cell projection is particularly useful for the rendering of a large number of isosurfaces as the rendering performance depends only on the number of cells intersected by isosurfaces (without double-counting). The rendering of isosurfaces requires

- the computation of appropriate pre-integrated lookup tables, in particular for multiple isosurfaces;
- the rendering of smoothly shaded isosurfaces, which is achieved by a new, hardware-assisted implementation of the interpolation between two shadings suggested by Westermann in [73]; and
- the combination of isosurfaces with semi-transparent volumes.

The rendering of shaded isosurfaces requires two rasterization passes; thus, the rendering performance is reduced to about one half of the rendering performance for unshaded isosurfaces. The latter is comparable to previously reported implementations of the Projected Tetrahedra algorithm.

In Section 3.2 (page 77), the basic idea of an algorithm for *hardware-assisted resampling* of tetrahedral meshes published by Westermann in [71] is sketched because it is based on the technique employed for smoothly shaded isosurfaces in Section 3.1.

Section 3.3 (page 79) presents a new algorithm for *hardware-assisted ray casting* in tetrahedral meshes on programmable graphics hardware. The primary advantage of this algorithm is the dramatically reduced communication between the main processor and the graphics board. This is achieved by

- storing the complete tetrahedral mesh encoded in textures on the graphics board and
- tracing all viewing rays in parallel by repeatedly rasterizing screen-filling rectangles with appropriate per-pixel operations.

Chapter 4: Non-Convex and Cyclic Meshes

This chapter's subject is the direct volume visualization of non-convex and cyclic meshes. Several algorithms for tetrahedral meshes are discussed, in particular

- the convexification of non-convex meshes (Section 4.1),
- edge collapses in non-convex meshes (Section 4.2),
- cell sorting for non-convex and cyclic meshes (Section 4.3), and
- cell projection for cyclic meshes (Section 4.4).

Section 4.1 (page 85) describes the *convexification of non-convex meshes*, i.e., the conversion of a non-convex mesh into a convex mesh. The algorithm was proposed by Williams in [80] and consists of the following steps:

- computation of the convex hull of the non-convex mesh,
- computation of the space between the convex hull and the boundary of the mesh, and
- decomposition of this space and of all holes inside the mesh into tetrahedra, which are called "imaginary".

The algorithm is illustrated with several examples; however, a general implementation of it is out of the scope of this work as the efficient decomposition of an arbitrarily shaped polyhedron into tetrahedra is an extremely challenging problem.

Provided that the algorithm can be applied, it permits to apply slightly adapted algorithms for convex meshes to non-convex meshes, e.g., for

- cell sorting (see [80] and Section 4.3),

- point location (see [80]),
- ray casting (see Section 3.3), or
- edge collapses (see Section 4.2).

A new algorithm for the latter problem, *edge collapses in non-convex meshes*, is presented in Section 4.2 (page 88). This solution was first published together with Thomas Ertl in [35]. The goal is to avoid edge collapses that cause self-intersections of the mesh. This is achieved by performing edge collapses in the corresponding convexified mesh (see Section 4.1) and testing the cells in the local neighborhood of the collapsed edge for inversions. Additionally, a method for the preservation of the convex hull is proposed, and local tests for the preservation of the mesh's topology are suggested.

Section 4.3 (page 95) discusses *cell sorting for non-convex and cyclic meshes*; more specifically spoken, a new variant of a topological sorting algorithm for convex tetrahedral meshes is presented that can be applied to meshes with visibility cycles, i.e., cyclic cell occlusions. This algorithm was first published together with Thomas Ertl in [31] and combines the MPVO (Meshed Polyhedra Visibility Ordering) algorithm published by Williams in [80] with the depth-first search algorithm for the computation of strongly connected components of a directed graph published by Tarjan in [66].

Moreover, extensions of this algorithm for non-convex, cyclic meshes are discussed, in particular

- an adaptation for convexified meshes (see Section 4.1) and
- an adaptation of the inexact sorting for non-convex meshes proposed by Williams in [80].

Cell projection for cyclic meshes is discussed in Section 4.4 (page 104). The proposed algorithm was first published together with Thomas Ertl in [31] and is based on an algorithm for cyclic occlusions of triangles published by Snyder and Lengyel in [63]. The basic idea is to decompose the projection of the tetrahedra of one visibility cycle into several passes, namely

- the attenuation of the background by all tetrahedra,
- the attenuation of the emission of each tetrahedron by the occluding tetrahedra, and
- the summation of all contributions.

These steps can be implemented very efficiently on standard graphics hardware as attenuation and summation are commutative image operations.

Chapter 5: Non-Simplicial and Non-Adaptive Meshes

This chapter is primarily about uniform meshes, which are geometrically unpleasant because of some specific geometrical features:

- The cells are not simplicial; therefore, a linear interpolation within the cells is impossible.
- The boundary of the mesh has to be a box, i.e., it cannot be adapted to the data. Similarly, the resolution of the mesh is uniform, i.e., it is also not adaptive to the data.

These features have several unpleasant consequences for direct volume visualization. In order to overcome (or at least alleviate) these problems, three new algorithms are proposed:

- texture-based pre-integrated volume rendering (Section 5.1),
- topology-guided downsampling (Section 5.2), and
- adaptive volume textures (Section 5.3).

In Section 5.1 (page 112), the first algorithm, *texture-based pre-integrated volume rendering*, is presented. It was first published together with Klaus Engel and Thomas Ertl in [18] and combines pre-integrated classification (see Section 2.5) with texture-based volume rendering (see Section 2.3) by exploiting programmable graphics hardware, in particular "dependent texture lookups", which are employed for the lookup in a pre-integrated table of colors and opacities.

Topology-guided downsampling is discussed in Section 5.2 (page 113). This algorithm was first published together with Thomas Ertl in [32] and demonstrates how to apply concepts related to the topology of a volumetric scalar field in the case of non-simplicial meshes.

The goal of this downsampling algorithm is to preserve as many of the critical points (maxima, minima, and saddle points) of a volumetric mesh as possible. This is achieved by virtually decomposing the uniform mesh into tetrahedra, computing critical points for this simplicial mesh, and preferably selecting the scalar values of critical points in the downsampling process. Thereby, more of the topology of the scalar field is preserved than by other downsampling methods and, therefore, the topology of isosurfaces extracted from the downsampled mesh is much closer to the topology of the isosurfaces of the original mesh. This effect is demonstrated with three different examples.

In Section 5.3 (page 124), *adaptive volume textures* are discussed. This technique was first published together with Thomas Ertl in [34]. Its main features are:

- adaptive storage of mesh data in uniform textures ("adaptive" with respect to the boundary and the resolution of the mesh) and
- fast random access by an efficient decoding with the help of per-pixel operations offered by programmable graphics hardware.

Based on the proposed data structures, which are "derived" from some basic requirements (adaptivity, fast random access, and limitations of current graphics hardware), an implementation of the sampling (i.e., decoding) of adaptive textures is discussed first. Then, the generation of adaptive textures from uniform meshes is presented, and the application of this technique to volume rendering is illustrated with an example.

Chapter 6: Geometrically Unpleasant Meshes in General

In this chapter an attempt is made to identify and generalize the methods that have been employed to find the algorithms proposed in this work. Therefore,

- the new algorithms are summarized and categorized (Section 6.1),
- problem-solving strategies and their applications in this work are discussed (Section 6.2), and
- further problems caused by geometrically unpleasant meshes that were out of the scope of this work are proposed (Section 6.3).

Section 6.1 (page 136) summarizes the *proposed solutions* presented in this thesis. Although most of these algorithms are limited to either simplicial or uniform meshes, they can be applied to more meshes by certain conversions, e.g., resampling and cell triangulation. The algorithms proposed in this work are categorized according to

- the kind of mesh they can be applied to (simplicial or uniform) and
- the kind of geometrically unpleasant feature, i.e., whether the feature is related to individual cells, groups of cells, or the mesh's boundary.

In Section 6.2 (page 138) several *problem-solving strategies* from [41] are discussed and their application to the problems addressed in this work are presented. The strategies include techniques to

- understand a problem (especially a problem caused by a geometrically unpleasant mesh);
- simplify a problem by redefining, generalizing, or dividing the problem, and

- actually solving a problem by directly attacking, avoiding, masking, or ignoring the problem.

Several *further problems* that were out of the scope of this work are proposed in Section 6.3 (page 143). They are categorized according to the classification of geometrically unpleasant features of meshes suggested in Section 6.1:

- problems of individual cells, e.g., “badly” shaped cells, polyhedral cells, or cells with non-planar faces;
- problems of groups of cells, e.g., “unfair” intersections; and
- problems that do not fit into this classification, e.g., problems of hierarchical or hybrid meshes, or problems of mesh generation.

Zusammenfassung (Abstract in German)

Interaktive Volumenvisualisierung (also die Visualisierung von auf räumlichen Gittern definierten, skalaren Daten in Echtzeit) ist nicht nur für große Gitter eine schwierige Aufgabe, sondern wird auch durch spezielle geometrische Eigenschaften räumlicher Gitter erschwert, zum Beispiel durch nicht uniforme Zellen, nicht konvexe Gitterränder oder zyklische Verdeckungen von Zellen.

Diese Dissertation behandelt einige dieser geometrischen Eigenschaften und deren unangenehme Konsequenzen für die direkte Volumenvisualisierung, die eine der erfolgreichsten Techniken zur interaktiven Volumenvisualisierung darstellt. Um diese Schwierigkeiten zu beheben oder zumindest zu entschärfen, werden mehrere neue algorithmische Lösungen vorgestellt: vorintegrierte Zellprojektion und Hardware-unterstützte Sehstrahlintegration für nicht uniforme Gitter, Kantenkontraktionen in nicht konvexen Gittern, Zellsortierung und -projektion für nicht konvexe und zyklische Gitter sowie Textur-basiertes vorintegriertes Volumen-Rendering, Topologie-gestütztes Ausdünnen und adaptive Volumentexturen für nicht simpliziale Gitter, d.h. Gitter deren Zellen keine Simplizes (in drei Dimensionen also Tetraeder) sind.

Da in dieser Arbeit nicht alle geometrisch unangenehmen Eigenschaften von räumlichen Gittern behandelt werden können, wurde besonderer Wert auf die Darstellung der Entwicklung der vorgeschlagenen Algorithmen gelegt. Tatsächlich entstanden die meisten der vorgestellten Techniken als Verallgemeinerungen, Adaptionen oder Erweiterungen von bekannten Methoden. Diese Zusammenhänge werden vor allem deswegen beschrieben, um neue Lösungen für diejenigen geometrisch unangenehmen Gittereigenschaften anzuregen, die nicht im Rahmen dieser Arbeit behandelt werden konnten.

Kapitelzusammenfassungen (Chapter Summaries in German)

Die folgenden Zusammenfassungen bieten einen recht detaillierten Überblick zum Inhalt der einzelnen Kapitel. Sie wurden so kompakt wie möglich gehalten, um dem Leser eine schnelle Orientierung zu erlauben. Zu diesem Zweck wurden auch die übersetzten Überschriften von Abschnitten *kursiv* gesetzt.

Kapitel 1: Einführung

Kapitel 1 soll vor allem in das Thema dieser Dissertation einführen und die Motivation des Themas erläutern.

Direkte Volumenvisualisierung wird oft erschwert durch große Datensätze, hochaufgelöste Bildausgaben oder die Notwendigkeit von hohen Bildwiederholraten zur Interaktion in Echtzeit. Aufgrund dieser Schwierigkeiten konzentriert sich die Forschung zur direkten Volumenvisualisierung üblicherweise auf „geeignete“ Gitter, im Gegensatz zu „geometrisch unangenehmen“ Gittern, zu denen alle Gitter mit einer außergewöhnlichen oder (aus welchen Gründen auch immer) „schwierigen“ geometrischen Gestalt ihrer Zellen oder ihrer Ränder zählen. Zwei Gründe für diese Vorgehensweise sind zum Beispiel:

- Es ist sinnvoll, zunächst die einfacheren Fälle zu lösen, bevor man die schwierigeren Fälle behandelt.
- Es ist nützlich, die relevantesten Probleme (d.h. die verbreitesten Gitter) zuerst zu behandeln.

Aber der Ansatz birgt auch einige Nachteile:

- geometrisch unangenehme Gitter treten in praktischen Visualisierungsanwendungen tatsächlich auf;
- alle Gitter sind in gewisser Hinsicht geometrisch unangenehm und
- Forschung auf bestimmte Gitter einzuschränken behindert auf lange Sicht den Fortschritt auf dem Gebiet der direkten Volumenvisualisierung.

Diese Gründe rechtfertigen das zentrale Thema dieser Dissertation, nämlich Algorithmen zur direkten Volumenvisualisierung von geometrisch unangenehmen Gittern, insbesondere effizientere, robustere und allgemeinere Algorithmen.

Abschnitt 1.1 (Seite 32) der Einführung gibt einen kurzen *Überblick zu dieser Dissertation*, der detaillierter als die Zusammenfassung ist und Referenzen auf einzelne Abschnitte enthält.

Kapitel 2: Direkte Volumenvisualisierung

Kapitel 2 führt genauer in die direkte Volumenvisualisierung ein und stellt viele der Methoden und Konzepte vor, die in den folgenden Kapiteln eingesetzt werden, insbesondere

- Anwendungen der direkten Volumenvisualisierung (Abschnitt 2.1),
- eine Spezialisierung der Visualisierungspipeline für die direkte Volumenvisualisierung (Abschnitt 2.2),
- Algorithmen zur Volumendarstellung (Abschnitt 2.3),
- Sichtstrahlintegration (Abschnitt 2.4),
- vorintegrierte Klassifikation (Abschnitt 2.5),
- eine einfache Klassifikation von Gittern (Abschnitt 2.6),
- Dateninterpolation in Gittern (Abschnitt 2.7) und
- eine nicht formale Definition von geometrisch unangenehmen Gittern (Abschnitt 2.8).

Für die direkte Volumenvisualisierung werden allen Datenpunkten Opazitäten zugeordnet, so dass verdeckte Datenpunkte nicht notwendigerweise unsichtbar sind, sondern eventuell durch die halbtransparenten, verdeckenden Punkte vor ihnen sichtbar bleiben. Die Vorteile dieser Technik sind:

- Sie erlaubt einen kontinuierlichen Kompromiss zwischen der opaken Verdeckung und der Sichtbarkeit von Datenpunkten.
- Die Darstellung von Isoflächen und Schnittebenen sind Spezialfälle der direkten Volumenvisualisierung.
- Die Geschwindigkeit von Algorithmen zur Volumendarstellung ist oft unabhängig von den tatsächlichen Skalarwerten in den Volumendaten.

Zu den Nachteilen zählt:

- Die resultierenden Bilder erscheinen oft „neblig“ und „unscharf“.
- Die Wahl geeigneter Farben und Opazitäten für Datenpunkte ist nicht trivial.

In Abschnitt 2.1 (Seite 36) werden existierende und potenzielle *Anwendungen* der direkten Volumenvisualisierung besprochen, z.B. die Visualisierung von medizinischen Aufnahmen, Daten anderer physikalischer oder chemischer Messungen, Daten aus numerischen Simulationen oder mathematischen Funktionen.

Einige weitere Gründe für die begrenzte Popularität der direkten Volumenvisualisierung werden erwähnt, z.B. die diesbezüglich unzureichenden Fähigkeiten von gängiger Graphikhardware, der Mangel an nicht kommerziellen Software-Werkzeugen, unbequeme und ineffiziente Benutzungsschnittstellen existierender Werkzeuge oder der seltene Einsatz von Volumenvisualisierung in Publikationen. Neben diesen Schwierigkeiten gibt es auch eine Reihe ungelöster Probleme in der direkten Volumenvisualisierung, z.B. die Spezifikation von mehrdimensionalen Transferfunktionen, die automatische Erzeugung von Transferfunktionen, die Visualisierung von zeitabhängigen Datensätzen oder die Darstellung von großen Volumen in Echtzeit.

Abschnitt 2.2 (Seite 37) beschreibt die wesentlichen Schritte der *Visualisierungspipeline*, d.h.

- Datenerfassung (die zu den Rohdaten führt),
- Datenaufbereitung (die zu den Visualisierungsdaten führt),
- Erzeugung einer geometrischen Repräsentation (die zu Graphikdaten führt), und
- Darstellung (die zu Bilddaten führt).

Die Spezialisierung dieser Pipeline auf die direkte Volumenvisualisierung ergibt folgende Schritte:

- Datenerfassung (die zu den Rohdaten führt),
- Datenaufbereitung (die zu den Volumendaten führt),
- Klassifikation und Schattierung (die zu Farb- und Opazitätsdaten führt), und
- Sichtstrahlintegration (die zu Bilddaten führt).

Mehrere grundlegende *Algorithmen zur Volumendarstellung* werden in Abschnitt 2.3 (Seite 39) vorgestellt, insbesondere

- Ray-Casting-Algorithmen (siehe auch Abschnitt 3.3),
- Zellprojektion (siehe auch Abschnitt 3.1),
- Splatting,

- Textur-basiertes Volumen-Rendering (siehe auch Abschnitt 5.1), und
- der Shear-Warp-Algorithmus.

Abschnitt 2.4 (Seite 41) behandelt die *Sichtstrahlintegration*, insbesondere

- das Volumen-Rendering-Integral ("volume rendering integral") zur Integration von Farben und Opazitäten entlang von Sichtstrahlen;
- Transferfunktionen, die Skalardaten auf Farben und Opazitäten abbilden;
- den Unterschied zwischen vor- und nachgeschalteter Klassifikation, d.h. der Unterschied zwischen der Anwendung der Transferfunktionen vor und nach der Interpolation der Skalardaten;
- die numerische Integration des Volumen-Rendering-Integrals durch Diskretisierung und
- eine Abschätzung der benötigten Samplingrate über einen Vergleich mit der Regel von Carson für Frequenz-modulierte Signale.

Vorintegrierte Klassifikation wird in Abschnitt 2.5 (Seite 48) vorgestellt. Diese Technik benötigt erheblich geringere Samplingraten und bietet dennoch eine erhöhte Genauigkeit. Sie wurde zusammen mit Stefan Röttger und Klaus Engel entwickelt und zuerst in [56] sowie [18] veröffentlicht.

Der wichtigste Nachteil der vorintegrierten Klassifikation ist die notwendige Berechnung einer (unter Umständen großen) Nachschlagetabelle. Mehrere Techniken zur Beschleunigung dieser Berechnung werden dargestellt:

- konstante Samplingrate,
- lokale Aktualisierungen der Nachschlagetabelle und
- eine näherungsweise Berechnung der Nachschlagetabelle mit Hilfe von Integralfunktionen.

Vorintegrierte Klassifikation kann mit vielen Algorithmen zur Volumendarstellung kombiniert werden, insbesondere

- Zellprojektion (siehe Abschnitt 3.1),
- Textur-basiertes Volumen-Rendering (siehe Abschnitt 5.1),
- Ray-Casting-Algorithmen (siehe Abschnitt 3.3, [30] und [47]) und
- dem Shear-Warp-Algorithmus ([57]).

In Abschnitt 2.6 (Seite 53) wird eine einfache *Klassifikation von Gittern* vorgestellt. Sie unterscheidet zwischen

- strukturierten Gittern, insbesondere uniformen, rectilinearen und curvilinearen Gittern, sowie
- unstrukturierten Gittern, insbesondere simplizialen Gittern und „Zoo-Gittern“ aus verschiedenartigen Polyedern.

Einige Gitter, wie z.B. hybride oder hierarchische Gitter, werden nicht von dieser Klassifikation erfasst, sind aber in dieser Dissertation auch nicht von Bedeutung.

Die *Interpolation in Gittern* wird in Abschnitt 2.7 (Seite 55) besprochen, insbesondere

- Interpolation anhand des nächsten Nachbarn;
- lineare Interpolation auf Linienstücken, bilineare Interpolation in Rechtecken und trilineare Interpolation in Quadern sowie
- lineare Interpolation in Simplizes, d.h. auf Linienstücken, in Dreiecken und Tetraedern.

Die Berechnung der bilinearen und trilinearen Interpolation in Dreiecken und Tetraedern mit Hilfe von mehreren linearen Interpolationen auf Linienstücken wird detailliert dargestellt.

In Abschnitt 2.8 (Seite 59), werden *geometrisch unangenehme Gitter* auf der Grundlage der Darstellungen aus den vorhergehenden Abschnitten beschrieben. Insbesondere werden folgende Aspekte erwähnt:

- Konsequenzen eines nicht konvexen Randes für Algorithmen zur Volumendarstellung,
- die Beschränkung einer kontinuierlichen, stückweise linearen Interpolation auf simpliziale Gitter,
- die Einschränkung der Hardware-unterstützten Interpolation auf uniforme Gitter und
- andere unangenehme geometrische Eigenschaften abgesehen von der Form der Zellen und des Randes von Gittern, z.B. zyklische Verdeckungen von Zellen oder eine nicht adaptive Gitterauflösung.

Kapitel 3: Nicht uniforme Gitter

Eines der grundsätzlichen Probleme der direkten Volumenvisualisierung von nicht uniformen Gittern ist die fehlende Hardware-Unterstützung für nicht uniforme Gitter, d.h. die Beschränkung auf trilineare Interpolation. Wie sich Graphikhardware dennoch zur Volumendarstellung nicht uniformer Gitter einsetzen lässt, wird in diesem Kapitel anhand von drei Algorithmen gezeigt:

- vorintegrierte Zellprojektion (Abschnitt 3.1),
- Hardware-unterstütztes Resampling (Abschnitt 3.2) und
- Hardware-unterstütztes Ray-Casting (Abschnitt 3.3).

Vorintegrierte Zellprojektion wird in Abschnitt 3.1 (Seite 61) beschrieben. Das Verfahren verbindet vorintegrierte Klassifikation (siehe Abschnitt 2.5) mit dem "Projected Tetrahedra"-Algorithmus (siehe [61]) und wurde zuerst zusammen mit Stefan Röttger und Thomas Ertl in [56] veröffentlicht. Vorintegrierte Zellprojektion zeichnet sich durch folgende Schritte aus:

- Zerlegung der projizierten Tetraeder in Dreiecke und Hardware-beschleunigte Rasterisierung dieser Dreiecke (wie im "Projected Tetrahedra"-Algorithmus),
- Hardware-beschleunigte Interpolation der Skalardaten auf der Oberfläche der Tetraeder und der Dicke der Tetraeder entlang der Sichtstrahlen, und
- Berechnung der Farben und Opazitäten durch Tabellennachschläge, die mit Hilfe von dreidimensionalen Texturen implementiert werden können.

Da die Vorberechnung der benötigten dreidimensionalen Texturen nicht in Echtzeit möglich ist, wird auch eine Näherung vorgestellt, die nur zweidimensionale Texturen benötigt, gleichzeitig aber die Bildqualität beeinträchtigt.

Vorintegrierte Zellprojektion ist besonders gut für die gleichzeitige Darstellung vieler Isoflächen geeignet, da die Darstellungsgeschwindigkeit nur von der Anzahl der von Isoflächen geschnittenen Zellen abhängt (ohne Mehrfachzählung). Für die Darstellung von Isoflächen ist es nötig,

- entsprechende Nachschlagetabellen zu berechnen,
- glatt schattierte Isoflächen darzustellen, was mit Hilfe einer Hardware-unterstützten Implementierung der Interpolation zwischen zwei Schattierungen, die von Westermann in [73] vorgeschlagen wurde, erzielt wird, und
- semi-transparente Volumendarstellungen mit Isoflächen zu kombinieren.

Da die Darstellung schattierter Isoflächen zwei Rasterisierungsschritte benötigt, wird die Darstellungsgeschwindigkeit im Vergleich zu unschattierten Isoflächen etwa halbiert. Letztere ist vergleichbar mit der Geschwindigkeit von früheren Implementierungen des "Projected Tetrahedra"-Algorithmus.

Abschnitt 3.2 (Seite 77) skizziert das grundlegende Konzept eines *Hardware-unterstützten Algorithmus zum Resampling* von Tetraedergittern, der von Westermann in [71] veröffentlicht wurde. Der Algorithmus ist vor allem deshalb in dieser Arbeit interessant, da er auf der Technik basiert, die in Abschnitt 3.1 zur Darstellung von schattierten Isoflächen eingesetzt wird.

Abschnitt 3.3 (Seite 79) stellt einen neuen *Hardware-unterstützten Ray-Casting-Algorithmus* für Tetraedergitter vor, der auf den Möglichkeiten von programmierbarer Graphikhardware aufbaut. Der wesentliche Vorteil dieses Algorithmus ist die drastisch reduzierte Kommunikation zwischen dem Hauptprozessor und der Graphikkarte. Dies ist möglich, da

- das vollständige Tetraedergitter in Texturen kodiert auf der Graphikkarte gespeichert wird und
- alle Sichtstrahlen parallel verfolgt werden, indem wiederholt Bildschirm-füllende Rechtecke mit geeigneten Pixel-Operationen rasterisiert werden.

Kapitel 4: Nicht konvexe und zyklische Gitter

Das Thema dieses Kapitels ist die direkte Volumenvisualisierung von nicht konvexen und zyklischen Gittern. Dazu werden mehrere Algorithmen für Tetraedergitter diskutiert, insbesondere

- die Konvexifizierung von nicht konvexen Gittern (Abschnitt 4.1),
- Kantenkontraktionen in nicht konvexen Gittern (Abschnitt 4.2),
- Zellsortierung für nicht konvexe und zyklische Gitter (Abschnitt 4.3) und
- Zellprojektion für zyklische Gitter (Abschnitt 4.4).

Abschnitt 4.1 (Seite 85) beschreibt die *Konvexifizierung von nicht konvexen Gittern*, d.h. die Konvertierung eines nicht konvexen Gitters in ein konvexes Gitter. Dieser Algorithmus wurde von Williams in [80] vorgeschlagen und besteht aus folgenden Schritten:

- die Berechnung der konvexen Hülle des nicht konvexen Gitters,
- die Berechnung des Raums zwischen der konvexen Hülle und des Randes des Gitters, und

- die Zerlegung dieses Raums und aller Leerräume innerhalb des Gitters in Tetraeder, die „imaginär“ genannt werden.

Der Algorithmus wird anhand mehrerer Beispiele veranschaulicht; eine allgemeine Implementierung ist aber im Rahmen dieser Dissertation nicht möglich, da die effiziente Zerlegung eines beliebig geformten Polyeders in Tetraeder ein extrem schwieriges Problem darstellt.

Falls der Algorithmus anwendbar ist, erlaubt er angepasste Algorithmen für konvexe Gitter auch für nicht konvexe Gitter einzusetzen, z.B. zur

- Zellsortierung (siehe [80] und Abschnitt 4.3),
- Punktlokalisierung (siehe [80]),
- Ray-Casting-Algorithmen (siehe Abschnitt 3.3) oder
- Kantenkontraktionen (siehe Abschnitt 4.2).

Ein neuer Algorithmus für das letztere Problem, *Kantenkontraktionen in nicht konvexen Gittern*, wird in Abschnitt 4.2 (Seite 88) vorgestellt. Diese Lösung wurde zuerst zusammen mit Thomas Ertl in [35] veröffentlicht. Ihr Ziel besteht darin Kantenkontraktionen zu vermeiden, die zu Selbstüberdeckungen des Gitters führen. Dies wird erreicht, indem die Kantenkontraktionen in dem entsprechenden konvexifizierten Gitter (siehe Abschnitt 4.1) durchgeführt und die Zellen in der lokalen Nachbarschaft der kontrahierten Kante auf Invertierungen geprüft werden. Darüber hinaus wird eine Methode zur Erhaltung der konvexen Hülle vorgeschlagen und lokale Kriterien zur Erhaltung der Gittertopologie diskutiert.

Abschnitt 4.3 (Seite 95) behandelt *Zellsortierung für nicht konvexe und zyklische Gitter*. Dazu wird eine neue Variante einer topologischen Sortierung für konvexe Tetraedergitter vorgestellt, die auch auf Gitter mit zyklischen Verdeckungen angewendet werden kann. Dieser Algorithmus wurde zuerst zusammen mit Thomas Ertl in [31] veröffentlicht und verbindet den MPVO-Algorithmus (Meshed Polyhedra Visibility Ordering) von Williams [80] mit der Tiefensuche zur Berechnung der stark zusammenhängenden Komponenten eines gerichteten Graphen von Tarjan [66].

Außerdem werden Erweiterungen dieses Algorithmus für nicht konvexe, zyklische Gitter besprochen, insbesondere

- eine Anpassung für konvexifizierte Gitter (siehe Abschnitt 4.1) und
- eine Integration der inexakten Sortierung für nicht konvexe Gitter, die von Williams in [80] vorgeschlagen wurde.

Zellprojektion für zyklische Gitter wird in Abschnitt 4.4 (Seite 104) behandelt. Der vorgeschlagene Algorithmus wurde zuerst zusammen mit Thomas Ertl in [31] veröffentlicht und basiert auf einem Algorithmus für zyklische Verdeckungen von Dreiecken, der von Snyder und Lengyel in [63] veröffentlicht wurde. Die grundlegende Idee ist, die Projektion der sich zyklisch verdeckenden Tetraeder in mehrere Schritte zu unterteilen, nämlich

- die Abschwächung des Hintergrunds durch alle Tetraeder,
- die Abschwächung der Emission jedes Tetraeders durch die verdeckenden Tetraeder und
- die Addition aller resultierenden Beiträge.

Diese Schritte können sehr effizient mit Hilfe von gängiger Graphikhardware implementiert werden, da die Abschwächungen und die Addition kommutative Bildoperationen sind.

Kapitel 5: Nicht simpliziale und nicht adaptive Gitter

In diesem Kapitel werden vor allem uniforme Gitter behandelt, die aufgrund folgender geometrischer Eigenschaften unangenehm sind:

- Die Zellen sind keine Simplizes, daher ist eine lineare Interpolation innerhalb der Zellen nicht möglich.
- Der Rand eines uniformen Gitters muss ein Quader sein, d.h. er kann nicht an die tatsächlichen Daten angepasst werden. Ebenso ist die Auflösung uniform, d.h. sie kann ebenfalls nicht lokal an die Daten angepasst werden.

Diese Eigenschaften haben einige unangenehme Konsequenzen für die direkte Volumenvisualisierung. Um diese Probleme (zumindest ansatzweise) zu lösen, werden drei neue Algorithmen vorgeschlagen:

- Textur-basiertes vorintegriertes Volumen-Rendering (Abschnitt 5.1),
- Topologie-gestütztes Ausdünnen (Abschnitt 5.2) und
- adaptive Volumentexturen (Abschnitt 5.3).

In Abschnitt 5.1 (Seite 112) wird der erste Algorithmus, *Textur-basiertes vorintegriertes Volumen-Rendering*, vorgestellt. Er wurde zuerst mit Klaus Engel und Thomas Ertl in [18] veröffentlicht und verbindet vorintegrierte Klassifikation (siehe Abschnitt 2.5) mit Texture-basiertem Volumen-Rendering (siehe Abschnitt 2.3) unter Zuhilfenahme von programmierbarer Graphikhardware. Insbesondere werden „abhängige Texturnachschläge“ (“dependent texture lookups”)

eingesetzt, um Farben und Opazitäten aus einer vorintegrierten Nachschlagetabelle zu bestimmen.

Topologie-gestütztes Ausdünnen wird in Abschnitt 5.2 (Seite 113) dargestellt. Dieser Algorithmus wurde zuerst zusammen mit Thomas Ertl in [32] veröffentlicht und zeigt, wie Konzepte, die auf der Topologie eines räumlichen Skalarfeldes basieren, auf nicht simpliziale Gitter angewendet werden können.

Das Ziel dieses Algorithmus ist es, trotz des Ausdünnens möglichst viele der kritischen Punkte (Maxima, Minima und Sattelpunkte) eines räumlichen Gitters zu erhalten. Dies wird erreicht durch eine virtuelle Zerlegung des uniformen Gitters in Tetraeder, die Berechnung der kritischen Punkte für dieses simpliziale Gitter und der bevorzugten Erhaltung der Skalarwerte der kritischen Punkte während des Ausdünnens. Dadurch bleibt ein größerer Teil der Topologie des Skalarfeldes erhalten als bei anderen Methoden zum Ausdünnen und entsprechend ist die Topologie der aus einem so ausgedünnten Gitter extrahierten Isoflächen näher an der Topologie der Isoflächen aus dem ursprünglichen Gitter. Dieser Zusammenhang wird anhand von drei verschiedenen Beispielen erläutert.

In Abschnitt 5.3 (Seite 124) werden *adaptive Volumentexturen* vorgestellt. Diese Technik wurde zuerst zusammen mit Thomas Ertl in [34] veröffentlicht. Die wesentlichen Vorteile sind:

- adaptive Speicherung der Gitterdaten in uniformen Texturen („adaptiv“ bezüglich des Randes und der Auflösung des Gitters) und
- schneller, wahlfreier Zugriff durch effiziente Dekodierung mit Hilfe von Pixel-Operationen, die von programmierbarer Graphikhardware zur Verfügung gestellt werden.

Ausgehend von den vorgeschlagenen Datenstrukturen, die aus einigen grundlegenden Anforderungen (Adaptivität, schneller, wahlfreier Zugriff und die Beschränkungen aktueller programmierbarer Graphikhardware) „abgeleitet“ werden, wird eine Implementation des Texturnachschlags (d.h. der Dekodierung) in adaptiven Texturen vorgestellt. Sodann wird die Erzeugung adaptiver Texturen aus uniformen Gittern diskutiert und die Anwendung dieser Technik auf Volumen-Rendering mit Hilfe eines Beispiels verdeutlicht.

Kapitel 6: Geometrisch unangenehme Gitter im Allgemeinen

In diesem Kapitel wird versucht, diejenigen Methoden zu identifizieren und zu verallgemeinern, mit deren Hilfe die in dieser Arbeit vorgeschlagenen Algorithmen er- bzw. gefunden wurden. Dazu werden

- die neuen Algorithmen zusammengefasst und kategorisiert (Abschnitt 6.1),

- allgemeine Strategien zur Problemlösung und ihre Anwendung in dieser Arbeit besprochen (Abschnitt 6.2) und
- weitere Probleme von geometrisch unangenehmen Gittern vorgeschlagen, die nicht in dieser Dissertation behandelt werden konnten (Abschnitt 6.3).

Abschnitt 6.1 (Seite 136) fasst die in dieser Arbeit *vorgestellten Lösungen* zusammen. Obwohl die meisten dieser Algorithmen entweder auf simpliziale oder uniforme Gitter beschränkt sind, können sie mit Hilfe geeigneter Gitterkonvertierungen, z.B. Resampling oder Zelltriangulierung, auf weitere Gitter angewendet werden. Die Kategorisierung der vorgestellten Algorithmen erfolgt anhand

- der Art des Gitters, auf die sie angewendet werden können (simplizial oder uniform), und
- der Art der geometrisch unangenehmen Eigenschaft, d.h. ob sich diese Eigenschaft auf einzelne Zellen, Gruppen von Zellen oder den Rand des Gitters bezieht.

In Abschnitt 6.2 (Seite 138) werden einige *Strategien zur Problemlösung* aus [41] erörtert und ihre Anwendung auf die in dieser Arbeit behandelten Probleme vorgestellt. Diese Strategien beinhalten Techniken, um

- ein Problem zu verstehen (insbesondere ein Problem, das von einem geometrisch unangenehmen Gitter verursacht wird),
- ein Problem zu vereinfachen, indem es neu definiert, verallgemeinert oder aufgeteilt wird, und
- ein Problem tatsächlich zu lösen, indem es direkt gelöst, vermieden, maskiert oder ignoriert wird.

Auf einige *weitere Probleme*, die nicht in dieser Arbeit behandelt werden konnten, wird in Abschnitt 6.3 (Seite 143) hingewiesen. Auch sie werden anhand der Kategorien für geometrisch unangenehme Gittereigenschaften aus Abschnitt 6.1 eingeteilt:

- Probleme von einzelnen Zellen, z.B. „schlecht“ geformte Zellen, Polyeder-Zellen oder Zellen mit nicht flachen Seiten;
- Probleme von Gruppen von Zellen, z.B. „unfaire“ Zellüberschneidungen, und
- Probleme, die in keine der Kategorien fallen, z.B. Probleme von hierarchischen oder hybriden Gittern oder Probleme der Gittererzeugung.

David: *I've got something to say.*
Big Bill: *Very well.*
David: *You don't have a right to do this.*
I mean, I know you want to stay pleasant around here,
but there are so many things that are so much better.
Like silly or sexy or dangerous or brief.
And every one of those things is in you all the time,
if you just have the guts to look for them.
Big Bill: *That's enough.*
David: *I thought I was allowed to defend myself.*
Big Bill: *You're not allowed to lie.*
David: *I am not lying.*

Dialog from the movie *Pleasantville*

Chapter 1

Introduction

In principle, direct volume visualization offers the possibility to visualize all the scalar data given on a volumetric mesh at the same time in a single image. Moreover, recent implementations achieve real-time interactive frame rates, even for useful image dimensions and data volumes. However, many real-life applications require image dimensions of several millions of pixels and data volumes of up to several gigabytes or even terabytes. Thus, even with high-end graphics hardware these requirements are hard to fulfill, if a solution is feasible at all.

Because of these difficulties, research on direct volume visualization usually focuses on "geometrically simple" meshes, i.e., complications are avoided by restricting all considerations to certain kinds of "appropriate" meshes. Thus, any mesh with an exotic, pathologic, exceptional, or simply more difficult geometric shape of its cells or boundary is excluded.

There are good reasons for this practice as there is no need to solve the general problem of handling arbitrary meshes as long as the simpler special cases are not completely solved. Also, it is obviously more important to find algorithms for the most relevant volumetric meshes than to design algorithms without applications.

Nonetheless, the focus of this work is on exactly those exotic, unpopular, non-simple, and "difficult" meshes, which will be called "geometrically unpleasant"

here. It is neither mercy with these neglected meshes, nor pure academic curiosity that motivates this approach but a few basic observations: Firstly, meshes with an exceptional geometry, e.g., numeric degeneracies, occur in real-life applications. Ignoring these cases means to reduce the robustness of an algorithm. Even if these special cases were correctly identified and rejected, the range of applications of an algorithm would be reduced.

Secondly, there is no "geometrically pleasant mesh": From the point of view of direct volume visualization, the geometries of all meshes have some unpleasant features. For example, a uniform mesh of cubic cells is usually considered to be particularly simple. However, cubic cells are not simplicial in the geometric sense and, therefore, give rise to several serious problems in direct volume visualization. For instance, any linear interpolation in cubic cells results in a non-continuous scalar field.

Thirdly, the identification of unpleasant features of meshes helps to improve volume visualization algorithms by precisely specifying problems and possible shortcomings. Also, the comparison between algorithms for meshes with and without a certain unpleasant feature helps to understand the nature of a particular problem and may indicate ways to solve it.

Therefore, research on direct volume visualization of unpleasant meshes is likely to lead to more efficient, more robust, and more general algorithms. This work is an attempt to give an overview of problems associated with geometrically unpleasant meshes, solve some of them, and sketch solutions to others. Of course, a complete treatment of this subject is beyond the scope of this work; thus, only a selection of problems is addressed. However, these problems appear to be diverse enough to let us extract several strategies, which might help to solve similar problems.

1.1 Outline of This Thesis

Chapter 2 offers an introduction to direct volume visualization, briefly discusses applications, and describes the most popular rendering algorithms. In particular, pre-classification, post-classification, and pre-integrated classification are presented. The chapter also introduces a classification of meshes and discusses specific problems of particular kinds of meshes. Thereby, an informal definition of "geometrically unpleasant meshes" is given.

The main part of this work is to be found in Chapters 3 to 5. Each chapter discusses the (unpleasant) consequences of particular geometric features of volumetric meshes and offers solutions to these problems. Some of these solutions are well-known in the literature on volume visualization, some were implemented and published as part of this work, and some have not been published before.

As these chapters are for the most part independent of each other, the reader is free to read them in any order he or she prefers. However, some cross-references were included in order to avoid unnecessary repetitions. Chapters 3 and 4 cover unstructured — in particular tetrahedral — meshes, while structured — especially uniform — meshes are discussed in Chapter 5.

More specifically, Chapter 3 is about algorithms for *non-uniform meshes*, i.e., meshes that cannot easily be stored in a three-dimensional texture, especially tetrahedral meshes. This includes algorithms for cell projection with pre-integrated classification (Section 3.1), hardware-assisted resampling (Section 3.2), and hardware-assisted ray casting (Section 3.3). Chapter 4 covers problems specific to meshes with a non-convex boundary, i.e., *non-convex meshes*, and meshes with visibility cycles, i.e., *cyclic meshes*. It presents algorithms for edge collapses (Section 4.2), cell sorting and cell projection for non-convex and cyclic tetrahedral meshes (Sections 4.3 and 4.4).

Chapter 5 presents three algorithms for *non-simplicial* and *non-adaptive meshes*, in particular uniform meshes. One general problem of non-simplicial cells is a non-linear data interpolation within cells. Therefore, pre-integrated volume rendering and many topology-related compression algorithms cannot easily be applied to these meshes. However, modifications of these algorithms for uniform meshes are feasible as demonstrated with two algorithms presented in Sections 5.1 and 5.2. The third algorithm attempts to overcome the non-adaptivity of three-dimensional textures with the help of programmable graphics hardware (Section 5.3).

As mentioned, Chapters 3 to 5 cover only a selection of specific problems caused by geometrically unpleasant meshes. In Chapter 6 the new algorithms proposed in Chapters 3 to 5 are summarized (Section 6.1) and an attempt is made to find strategies to solve similar problems by generalizing these solutions (Section 6.2). Chapter 6 is concluded with a list of "further problems" (Section 6.3), i.e., geometrically unpleasant features of meshes that are not covered in this work.

1.2 Acknowledgments

The Bluntfin data set visualized in Figures 3.11b, 4.3a, and C.5b is used courtesy of C. M. Hung and P. G. Buning. The Tapered Cylinder data set in Figure 4.3b is used courtesy of D. Jespersen and C. Levit. The CTA scan in Figures 5.9 and C.7 is used courtesy of Bernd Tomandl. The Engine data set in Figures 5.10 and C.8 is used courtesy of General Electric. The CT scan of a bonsai in Figures 5.11 and C.9 is used courtesy of Bernd Tomandl and Stefan Röttger. The CT scan of the Stanford terra-cotta bunny in Figures 5.16 and C.11 is used courtesy of Terry Yoo, Sandy Napel, Geoff Rubin, and Marc Levoy.

This work would not exist without the help and support of a lot of people. Many thanks to my advisor, Thomas Ertl, in particular for his supportive advice and his confidence in my ideas. I am greatly indebted to Nelson Max and Daniel Weiskopf for their thorough reading of earlier drafts of this thesis and their detailed and valuable comments. Particular thanks to Stefan Röttger for his contributions to (what we now call) our pre-integrated cell projection algorithm presented in Section 3.1, to Klaus (Dieter) Engel for his contributions to our pre-integrated texture-based rendering algorithm discussed in Section 5.1, to Günter Knittel and Christianne Leidecker for the fruitful discussions about the appropriate sampling rate for volume rendering with post-classification (see Section 2.4.3), to Thomas Gerstner for his comments on topology-guided downsampling (see Section 5.2), to Manfred Weiler for asking the right question when I only had the answer and for making my idea work (see Section 3.3 and [69]), and to Jürgen Schulze for our work on the pre-integrated shear-warp algorithm. Thanks to Joachim Diepstraten, (my long-term office mate) Matthias Hopf, Marcelo Magallón, (again) Daniel Weiskopf, and Rüdiger Westermann for lots of discussions, suggestions, and criticism related to this work.

Many thanks also to the people I have had the pleasure to work with at the University of Stuttgart and at the Computer Graphics Group of the Friedrich-Alexander-University Erlangen-Nuremberg; in particular (in alphabetic order; without the ones already mentioned) Nicolas Bardili, Maria Baroti, Katrin Bidmon, Michael Braitmaier, Marianne Castro, Volker Diekert, Wilhelm ("Willy") Dilly, Mike Eissele, Norbert Frisch, Rul Gunzenhäuser, Kenji Hanakata, Peter Hastreiter, (my short-term office mate) Kai Hormann, Andreas Hub, Sabine ("I got you, babe") Iserhardt-Bauer, Thomas Klein, Hermann ("there is tea") Kreppein, Andreas Mailänder, Markus Merz, Guido Reina, Matthias ("the Older") Ressel, Christof Rezk-Salama, Ulrike ("U.") Ritzmann, Dirc ("lowering the level") Rose, Alexander Rosiutar, Martin Rotard, Martin Schmid, Martin ("the First") Schulz, Waltraud Schweikhardt, Ove ("Dr.") Sommer, Simon ("Martin is the evil incarnate") Stegmaier, Christian ("Mr. Pro7") Teitzel, and Alfred ("we will see") Werner.

It goes without saying that I am most thankful to my family and friends.

David: *It's an art book.*
[...]
Mr. Johnson: *It's beautiful, Bud ...*
David: *What's wrong?*
Mr. Johnson: *No, it's just I'll never be able to do that.*
David: *Well, you just started.*
I mean, you can't do it now.
Mr. Johnson: *No, no, no, that's not it.*
Just where am I gonna see colors like that?
Must be awful lucky to see colors like that.
I bet they don't know how lucky they are.

Dialog from the movie *Pleasantville*

Chapter 2

Direct Volume Visualization

The visualization of volumetric data is particularly challenging because occlusions are often severely limiting the efficiency of volume visualizations. Therefore, many techniques in volume visualization aim at extracting a subset of a volumetric scalar field (or "volume" for short) in order to minimize occlusions.

A simple example is a single slice through a volume, which avoids any occlusions by reducing the dimensionality of the visualized data set. Similarly, an isosurface, i.e., the set of all points of a volumetric scalar field associated with the same scalar value (the isovalue), is only a two-dimensional subset with less occlusions than the whole volume. Other structures of lower dimensionality extracted from scalar fields are level set surface models (see [77]) and topological information (see [2]). The common disadvantage of these techniques is the limitation to a subset of the data, i.e., the invisibility of a large part of the volume.

Direct volume visualization, on the other hand, aims at a continuous trade-off between occlusions and the visibility of data points by assigning opacities to all data points. Points with high opacities will be more prominent in the final image but usually occlude other points, while points with low opacity will be less prominent but also less occluding. Unfortunately, this smooth transition between transparent and opaque data points frequently leads to rather nebulous and fuzzy images. Moreover, the choice of appropriate opacities and colors is not trivial.

However, direct volume visualization offers important advantages. For example, isosurfaces and slices through a volume may be rendered with the help of direct volume rendering as these techniques are only special cases of direct volume visualization. Furthermore, the performance of rendering algorithms for direct volume visualization is often independent of the complexity of the visualized volume data and independent of user-specified visualization parameters, e.g., transfer functions. Therefore, it is often possible to maintain a constant frame rate in interactive applications.

2.1 Applications

There is no doubt that direct volume visualization has not reached its “mass market” yet. This “mass market” is in fact rather small, namely scientists, researchers, engineers, and other professionals who deal with continuous volumetric fields or discrete approximations to them. Currently, the vast majority of these professionals is still using simple plotting techniques in order to visualize data on axes-parallel slices or opaque isosurfaces. Even semi-transparent isosurfaces or slices that are not axis-aligned are not commonly used.

While volume graphics, i.e., the rendering of volumetric objects, has found its way into movie and video game productions, volume visualization has only found some niches, although it has been successfully applied in almost all sciences, including mathematics, physics, chemistry, biology, geology, medicine, many engineering sciences, etc. Not only does it allow users to visualize measured data, e.g., medical scans, but also data from numeric simulations, e.g., finite-element simulations, or any other computation, e.g., arbitrary mathematical functions in three dimensions.

However, volume visualization is by far less popular than one might expect from the large range of potential applications. In particular, direct volume visualization is frequently rejected for quantitative analyses because of the fuzzy nature of the resulting images. In these cases, isosurfaces are often strongly preferred. Thus, the visualization of medical scans, e.g., CT (computer tomography) or MR (magnetic resonance) scans, is still by far the most important field of application of direct volume visualization.

There are many reasons for this limited popularity of volume visualization apart from the mentioned “fuzzyness”: the insufficient support by standard graphics hardware, a lack of non-commercial tools, uncomfortable and inefficient interfaces of existing tools, the rare use of volume visualization for publications outside of the visualization community, and the lack of education about volume visualization, to name just a few. Unfortunately, some of these reasons reinforce each other.

Besides these problems, there are still many unsolved problems (or "challenges") in direct volume visualization, which have not been solved completely, e.g., the specification of multi-dimensional transfer functions, the automatic generation of transfer functions, the visualization of time-dependent data sets, real-time interactive rendering of large volumes, etc. Thus, in order to exploit the full potential of direct volume visualization and to encourage actual applications, it is necessary to solve at least some of these problems. Therefore, one should not be tempted to believe that all fundamental problems of direct volume visualization have been solved yet.

2.2 Visualization Pipeline

In analogy to the concept of a graphics pipeline in computer graphics, there is a *visualization pipeline* in computer visualization. It is one of the most valuable concepts in visualization, not only for the basic design of visualization applications but also for the discussion of visualization algorithms.

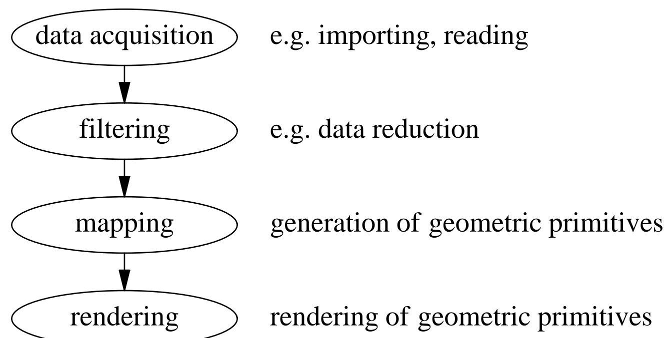


Figure 2.1 Main stages of the visualization pipeline.

There are many variants of the visualization pipeline; however, they all describe the data flow from some source, e.g., measurements or computer simulations, to some kind of image data, e.g., in the form of static pictures or animated movies. The stages of the visualization pipeline sketched in Figure 2.1 are *data acquisition*, *filtering*, *mapping*, and *rendering*. Figure 2.2a shows an extended scheme of this pipeline, which includes the names of intermediate data structures.

Data acquisition comprises all operations necessary to make the *raw data* available to the visualization pipeline, e.g., reading files, requesting and receiving data, or low-level data conversion. As the raw data is often too detailed and/or not all of it should be visualized at the same time, several *filtering* operations are usually performed to reduce and preprocess it. The filtering results in the

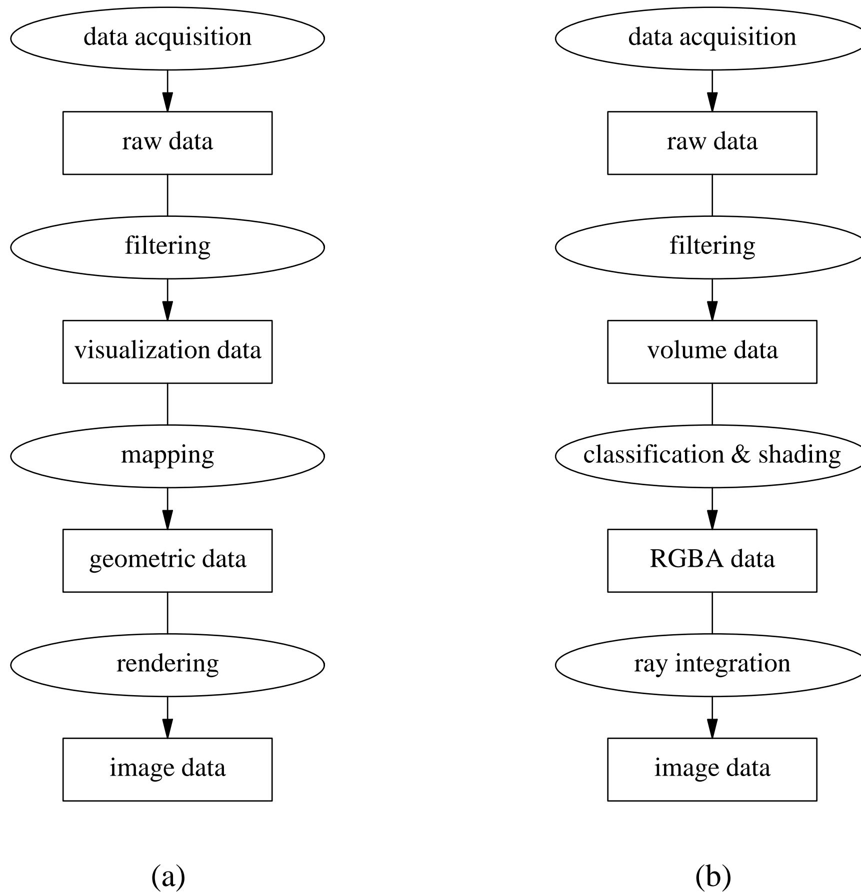


Figure 2.2 **(a)** Main stages and data structures of the visualization pipeline. **(b)** Variant of the visualization pipeline for direct volume visualization.

so-called *visualization data*, i.e., the data for which a graphical representation is computed. This *mapping* step is the next stage of the pipeline, which transforms the visualization data to the *geometric data*, i.e., a set of geometric primitives, which might be structured by a scene graph or similar constructs. By computing pure geometric data it is ensured that the last stage of the visualization pipeline, i.e., the *rendering*, is independent of the particular visualization problem and just renders geometric data by employing appropriate techniques of general computer graphics.

The mapping stage is of particular relevance as the graphical representations employed in a visualization are defined exclusively in this stage. However, the performance of the rendering stage is usually more important for the interactivity of a visualization application. Also, the choice of an appropriate filtering step is often crucial in order to generate a useful visualization.

For volume visualization, the filtering operations might include the sampling or the geometric clipping of a volume data set. Useful mapping techniques for volume visualization include the extraction of slices through the volume data, which can be represented by textured polygons, or the computation of polygonal approximations to isosurfaces. However, this work focuses on direct volume visualization, which does not easily fit into this pipeline.

Figure 2.2b depicts a modification of the general visualization pipeline that is specialized for direct volume visualization. In particular, the mapping and rendering stages were replaced by a *classification & shading* stage, which generates *RGBA data*, i.e., sets of red, green, blue, and alpha components, and a *ray integration* stage, which composites this data in order to generate the image data. Since the visualization data corresponds to volumetric data in this case, the latter two stages are also referred to as *volume rendering*.

2.3 Volume Rendering Algorithms

As there is a large number of published volume rendering algorithms, only a few of the most important concepts are discussed in this section. The common tasks of sampling volumetric data, assigning colors, and integrating them will be described in subsequent sections.

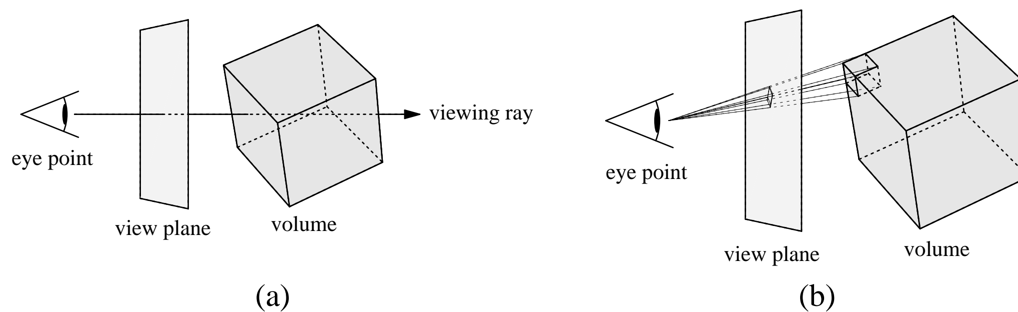


Figure 2.3 **(a)** Direct volume rendering with ray casting: One viewing ray is traced for each pixel. **(b)** Direct volume rendering with cell projection: Cells of the volume mesh are projected to the view plane.

As depicted in Figure 2.3a, the basic idea of *ray casting* algorithms for volume visualization is to trace viewing rays through a volumetric scalar field. Along each ray, colors and opacities are calculated at discrete sampling points and composited in order to compute the total color of each ray. Usually, at least one ray is traced for each pixel and the final image is built pixel by pixel, i.e., in *image-order*. Ray casting algorithms are among the earliest volume rendering algorithms; see for

example [39, 22]. Nonetheless, ray casting is still an active research area; one of today's topics is, for example, hardware support for ray casting, see Section 3.3.

In contrast to image-order algorithms, *object-order* algorithms process part after part of the volumetric data. Each part, usually a cell of a volumetric mesh, can contribute to the color of many pixels; however, there are very efficient ways to compute all contributions of a single cell of the mesh.

One of the most important object-order algorithms in volume rendering is cell projection, in particular the projection of tetrahedra as first suggested by Shirley and Tuchman in [61]. However, cells of other shapes may also be projected to the view plane, e.g., cubes as indicated by Figure 2.3b. Cell projection is discussed in more detail in Section 3.1.

In [76], Westover suggested a different object-order algorithm, which is called *splatting*. This algorithm accelerates the computation of color contributions to different pixels by pre-computing the contributions of voxels in a so-called "footprint table".

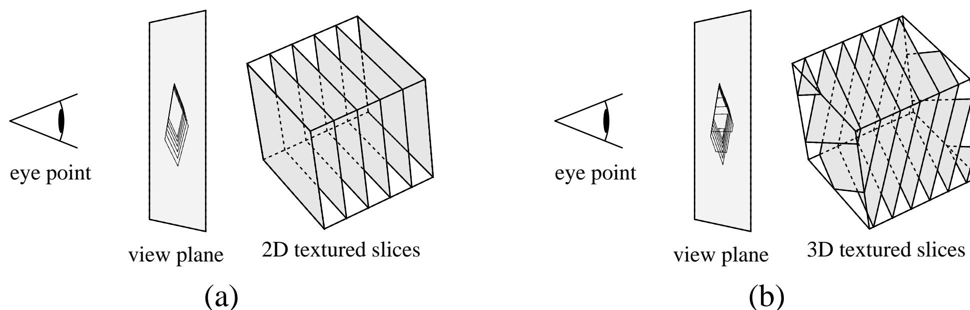


Figure 2.4 **(a)** Direct volume rendering with 2D textured slices: The slices are aligned to the volumetric object. **(b)** Direct volume rendering with 3D textured slices: The slices are usually aligned to the view plane.

The availability of hardware-accelerated texture mapping led to texture-based volume rendering algorithms, first published by Cabral et al. in [7]. Figure 2.4 illustrates the difference between the use of two-dimensional and three-dimensional textures.

As depicted in Figure 2.4a, the 2D texture-based variant renders a set of textured slices that are aligned to the axes of the cubic volumetric mesh. The slices are oriented perpendicular to the axis of the volume that is closest to the viewing direction. Therefore, the volume data has to be encoded in three stacks of two-dimensional textures: one stack for each of the three axes. In contrast, the 3D texture-based variant requires only one three-dimensional texture and renders slices parallel to the view plane, as depicted in Figure 2.4b. Texture-based volume rendering will be discussed in more detail in Section 5.1.

The shear-warp algorithm (see for example [37]) is in some sense related to the 2D texture-based rendering algorithm; however, it is usually implemented in software. Instead of texturing and projecting slices through the volume with the help of graphics hardware, they are efficiently rasterized in software by a clever decomposition of the viewing transformation.

Many more volume rendering algorithms have been proposed; in particular extensions, enhancements, and hybrid combinations of the mentioned techniques. Moreover, these algorithms differ in the way they solve the common operations of interpolating data, assigning colors, and integrating them, which are discussed in the following sections.

2.4 Ray Integration

2.4.1 Volume Rendering Integral

The basic task of any volume renderer is an (approximate) evaluation of the *volume rendering integral* for each pixel, i.e., the integration of attenuated colors and extinction coefficients along each viewing ray. While this is obvious for ray casting algorithms as illustrated in Figure 2.3a, many other volume rendering algorithms do not explicitly represent viewing rays. Instead, viewing rays are often defined implicitly by the positions of the eye point and the view plane. Also, the volume rendering integrals do not always have to be evaluated one by one; rather the integrals for all pixels may be evaluated simultaneously. For example, all object-order algorithms compute the contributions of parts of the volume to all ray integrals and then update the preliminary values of these integrals, usually by updating an intermediate image. In this larger sense, the evaluation of volume rendering integrals is common to all volume rendering algorithms.

We assume that the viewing ray $\mathbf{x}(\lambda)$ is parametrized by the distance λ to the eye point, and that color densities $\mathrm{color}(\mathbf{x})$ together with extinction densities $\mathrm{extinction}(\mathbf{x})$ may be calculated for any point in space $\mathbf{x}$. (The units of color and extinction densities are color intensity per length and extinction strength per length, respectively. However, we will refer to them as colors and extinction coefficients when the precise meaning is clear from the context.) The volume rendering integral is then

$$I = \int_0^D \mathrm{color}\big(\mathbf{x}(\lambda)\big) \exp\left(-\int_0^\lambda \mathrm{extinction}\big(\mathbf{x}(\lambda')\big)\mathrm{d}\lambda'\right) \mathrm{d}\lambda$$

with the maximum distance D, i.e., there is no color density $\mathrm{color}\big(\mathbf{x}(\lambda)\big)$ for λ greater than D. In words, color is emitted at each point $\mathbf{x}$ according to the function

color($\mathbf{x}$), and attenuated by the integrated extinction coefficients extinction($\mathbf{x}$) between the eye point and the point of emission.

Unfortunately, this form of the volume rendering integral is not useful for the visualization of a continuous scalar field $s(\mathbf{x})$, because the calculation of colors and extinction coefficients is not specified yet. We distinguish two steps in the calculation of these colors and extinction coefficients: the *classification* is the assignment of a *primary color* and an extinction coefficient. (The term *primary color* is borrowed from OpenGL terminology [85] in order to denote the color *before* shading.) The classification is achieved by introducing transfer functions for color densities $\tilde{c}(s)$ and extinction densities $\tau(s)$, which map scalar values $s = s(\mathbf{x})$ to colors and extinction coefficients. In general, $\tilde{c}$ is a vector specifying colors in a color space, while τ is a scalar extinction coefficient.

The second step is called *shading* and calculates the color contribution of a point in space, i.e., the function color($\mathbf{x}$). The shading depends, of course, on the primary color, but may also depend on other parameters, e.g., the gradient of the scalar field $\nabla s(\mathbf{x})$, ambient and diffuse lighting parameters, etc. In the remainder of this section we will not be concerned with shading but only with classification; therefore, we choose a trivial shading, i.e., we identify color($\mathbf{x}$) with the primary color $\tilde{c}\big(s(\mathbf{x})\big)$ assigned in the classification. Analogously, extinction($\mathbf{x}$) is identified with $\tau\big(s(\mathbf{x})\big)$. The volume rendering integral is then written as

$$I = \int_0^D \tilde{c}\Big(s\big(\mathbf{x}(\lambda)\big)\Big) \exp\left(-\int_0^\lambda \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big)\,\mathrm{d}\lambda'\right) \mathrm{d}\lambda. \tag{2.1}$$

2.4.2 Pre- and Post-Classification

Direct volume rendering techniques differ considerably in the way they evaluate Equation (2.1). One important and very basic difference is the computation of $\tilde{c}(s(\mathbf{x}))$ and $\tau(s(\mathbf{x}))$. The scalar field $s(\mathbf{x})$ is usually defined by a mesh with scalar values s_i defined at each vertex $\mathbf{v}_i$ of the mesh in combination with an interpolation scheme, e.g., one of the interpolation schemes discussed in Section 2.7.

The ordering of the two operations, interpolation and the application of transfer functions, defines the difference between *pre-* and *post-classification.* Post-classification is characterized by the application of the transfer functions *after* the interpolation of $s(\mathbf{x})$ from the scalar values at several vertices (as suggested by Equation (2.1)); while pre-classification is the application of the transfer functions *before* the interpolation step, i.e., colors $\tilde{c}(s_i)$ and extinction coefficients $\tau(s_i)$ are calculated in a pre-processing step for each vertex $\mathbf{v}_i$ and then used to interpolate $\tilde{c}(s(\mathbf{x}))$ and $\tau(s(\mathbf{x}))$ for the computation of the volume rendering integral. The difference is also illustrated in Figures 2.7a and 2.7b.

Obviously, pre- and post-classification will produce different results whenever the interpolation does not commute with the transfer functions. As the interpolation usually is non-linear (e.g., trilinear in uniform meshes), it will *only* commute with the transfer functions if the transfer functions are constant or the identity. Otherwise, pre-classification will in general result in deviations from post-classification, which is "correct" in the sense of applying the transfer functions to a continuous scalar field defined by a mesh in combination with an interpolation scheme. Nonetheless, pre-classification is useful under certain circumstances; in particular, because it may be used as a simple segmentation technique.

There is some confusion in the literature about the correctness of pre-classification with linear transfer functions. The discussion above shows that this requires a linear interpolation, which is in fact common in tetrahedral (more generally spoken, in simplicial) meshes but not in uniform meshes. Moreover, both transfer functions have to be linear. However, $\tilde{c}(s)$ usually corresponds to a product of a color $c(s)$ and an extinction coefficient $\tau(s)$ (which is usually called a "density" in this context, and an "opacity" in the discretized form, see below):

$$\tilde{c}(s) = \tau(s)c(s).$$

(See [42] for a comparison between a direct specification of the transfer function $\tilde{c}(s)$ and the specification of $c(s)$.)

Therefore, $\tilde{c}(s)$ is only linear if either the transfer function of the color $c(s)$ or of the extinction coefficient $\tau(s)$ is *constant* and the other transfer function is linear. This restriction can be weakened by defining a transfer function for $\tilde{c}(s)$ directly (see for example [78]) or by employing pre-multiplied colors (see for example [84]). However, even in this case the transfer functions for (pre-multiplied) colors and extinction coefficients are both restricted to linear functions.

For non-linear transfer functions (including piecewise linear functions) and/or non-linear interpolation only post-classified rendering will "correctly" visualize a continuous scalar field provided the volume rendering integral is evaluated with sufficient accuracy.

2.4.3 Numerical Integration

An analytic evaluation of the volume rendering integral is possible in some cases, in particular for linear interpolation and piecewise linear transfer functions (see [81] and [82]). However, this approach is not feasible in general; therefore, a numerical integration is usually required.

The most common numerical approximation of the volume rendering integral in Equation (2.1) is the calculation of a Riemann sum for n equal ray segments of length $d := D/n$. (See also Figures 2.5 and 2.6, and Section IV.A in [42].) It

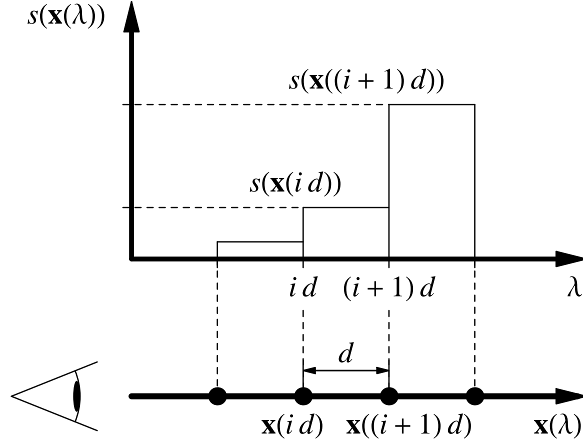


Figure 2.5 Piecewise constant approximation of the function $s(\mathbf{x}(\lambda))$ along a viewing ray.

is straightforward to generalize the following considerations to unequally spaced ray segments.

We will approximate the factor

$$\exp\left(-\int_0^{\lambda} \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big)\,\mathrm{d}\lambda'\right)$$

in Equation (2.1) by

$$\exp\left(-\sum_{i=0}^{\lfloor\lambda/d\rfloor} \tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)\,d\right) =$$

$$= \prod_{i=0}^{\lfloor\lambda/d\rfloor} \exp\left(-\tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)\,d\right) \approx \prod_{i=0}^{\lfloor\lambda/d\rfloor} (1-\alpha_i),$$

where the *opacity* α_i of the i-th ray segment is defined by

$$\alpha_i := 1 - \exp\left(-\int_{i\,d}^{i(d+1)} \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big)\,\mathrm{d}\lambda'\right). \tag{2.2}$$

and approximated by

$$\alpha_i \approx 1 - \exp\left(-\tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)\,d\right).$$

This approximation assumes a piecewise constant value of $s(\mathbf{x}(\lambda))$ as illustrated in Figure 2.5. The result is often further approximated to

$$\alpha_i \approx \tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)\,d.$$

$1-\alpha_i$ will be called the *transparency* of the i-th ray segment.

Similarly, the color $\tilde{C}_i$ emitted in the i-th ray segment is defined by

$$\tilde{C}_i := \int_{i\,d}^{i(d+1)} \tilde{c}\Big(s\big(\mathbf{x}(\lambda)\big)\Big) \exp\left(-\int_{i\,d}^{\lambda} \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big)\mathrm{d}\lambda'\right)\mathrm{d}\lambda. \tag{2.3}$$

$\tilde{C}_i$ may be approximated by

$$\tilde{C}_i \approx \tilde{c}\big(s(\mathbf{x}(i\,d))\big)\,d.$$

Thus, the approximation of the volume rendering integral in Equation (2.1) is

$$I \approx \sum_{i=0}^{\lfloor D/d \rfloor} \tilde{C}_i \prod_{j=0}^{i-1}(1-\alpha_j). \tag{2.4}$$

Therefore, a back-to-front compositing algorithm will implement the equation

$$\tilde{C}'_i = \tilde{C}_i + (1-\alpha_i)\tilde{C}'_{i+1}, \tag{2.5}$$

where $\tilde{C}'_i$ is the color accumulated from all ray segments j with $j \geq i$.

Substituting $\tilde{c}(s)$ by $\tau(s)c(s)$ and employing the approximations

$$C_i \approx c\big(s(\mathbf{x}(i\,d))\big) \qquad \text{and} \qquad \alpha_i \approx \tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)\,d$$

will result in the more common approximation

$$I \approx \sum_{i=0}^{\lfloor D/d \rfloor} \alpha_i C_i \prod_{j=0}^{i-1}(1-\alpha_j)$$

with the corresponding back-to-front compositing equation

$$\tilde{C}'_i = \alpha_i C_i + (1-\alpha_i)\tilde{C}'_{i+1}. \tag{2.6}$$

This compositing equation indicates that $\tilde{C}$ corresponds to a *pre-multiplied color* αC; which is also called *opacity-weighted color* (see [84]) or *associated color*. According to Blinn in [6], associated colors have their opacity associated with them, i.e., they are regular colors composited on black. Blinn also notes that some intensity computations result in associated colors, although they are not

explicitly multiplied by an opacity. In this sense, the transfer function $\tilde{c}(s)$ is in fact a transfer function for an associated color density.

A coherent discretization of viewing rays into equal segments may be interpreted as a discretization of the volume into *slabs*. Each slab emits light and absorbs light from the slabs behind it. However, the light emitted in each slab is not attenuated within the slab itself. (Exactly this approximation is also employed in Section 2.5.2 for pre-integrated classification.)

The discrete approximation of the volume rendering integral will converge to the correct result for $d \to 0$, i.e., for high sampling rates $1/d$. According to the sampling theorem, a correct reconstruction is only possible with sampling rates greater than the Nyquist frequency. However, non-linear features of transfer functions may considerably increase the sampling rate required for a correct evaluation of the volume rendering integral as this sampling rate depends on the product of the Nyquist frequencies of the scalar field $s(\mathbf{x})$ and the maximum of the Nyquist frequencies of the two transfer functions $\tilde{c}(s)$ and $\tau(s)$ (or of the product $c(s)\tau(s)$).

The actual relation may be obtained by comparison with the required sampling rate for a frequency-modulated signal $s_{\mathrm{fm}}(t)$ (see Section 6.4 in [59]):

$$s_{\mathrm{fm}}(t) := A\cos\left(2\pi f_{\mathrm{c}}t + (\Delta f/f_{\mathrm{m}})\sin\left(2\pi f_{\mathrm{m}}t\right)\right),$$

where f_c specifies the carrier frequency, Δf the maximum deviation from f_{c}, and f_{m} the modulation frequency or the maximum frequency of the modulation signal if it is not a single-frequency tone. With the help of the identity

$$\cos(a + x\sin(b)) = \sum_{k=-\infty}^{\infty} \cos(a + k\,b)\, J_k(x)$$

(where J_k denotes the Bessel function of the first kind of order k), the modulated signal may be written as

$$s_{\mathrm{fm}}(t) = A \sum_{k=-\infty}^{\infty} J_k\left(\Delta f/f_{\mathrm{m}}\right)\cos\left(2\pi f_{\mathrm{c}}t + 2\pi f_{\mathrm{m}}k\,t\right).$$

The spectrum of $s_{\mathrm{fm}}(t)$ may be obtained directly from this representation: Apart from the carrier frequency f_{c}, there is an infinite number of sidebands at frequencies $f_{\mathrm{c}} \pm f_{\mathrm{m}}k$ with $k \in \mathbb{N}$. Thus, the modulated signal is not bandwidth-limited and there is no maximum frequency. However, according to an approximation by John Remington Carson (known as "Carson's rule"), the actually required bandwidth (for more than 98 % of the signal power) is $2(\Delta f + f_{\mathrm{m}})$, i.e., contributions of sidebands outside the interval $[f_{\mathrm{c}} - (\Delta f + f_{\mathrm{m}}), f_{\mathrm{c}} + (\Delta f + f_{\mathrm{m}})]$ are negligible. (It might be worth noting that Carson abandoned the concept of frequency modulation in the 1920s because it required a larger bandwith than amplitude modulation.)

In order to apply this result to the problem of determining an appropriate sampling frequency along a viewing ray, some additional symbols have to be introduced. Let $U(S)$ denote a transfer function for scalar values $S \in [0,1]$ with Nyquist frequency f_U. In order to define values $U(S)$ for $S \notin [0,1]$, let $U(S)$ be a symmetric function with period 2, i.e., $U(S) = U(-S)$ and $U(S) = U(S+2k)$ for $k \in \mathbb{N}$. Furthermore, let $S(t)$ denote a scalar field with Nyquist frequency f_S. Thus, the problem is to determine an appropriate sampling frequency for $U(S(t))$. This problem can be simplified with the help of a Fourier cosine series:

$$U(S(t)) = \sum_{k=0}^{\infty} a_k \cos(\pi\, k\, S(t)).$$

As the appropriate sampling rate for this sum corresponds to the maximum of the sampling rates for the individual summands, it is possible to restrict the following considerations to the summand with the maximum k with $a_k \neq 0$. This $k_{\max}$ corresponds to a maximum frequency $k_{\max}/2$, which is given by half the Nyquist frequency f_U, i.e.

$$k_{\max}/2 = f_U/2.$$

Thus, $U(S(t))$ can be specialized to the form

$$A\cos(\pi\, k_{\max}\, S(t)) = A\cos\left((2\pi f_U/2)\, S(t)\right),$$

with $A = a_{k_{\max}}$. This function is already close to a frequency-modulated signal, where $S(t)$ corresponds to the modulation signal. As mentioned above, it is common to replace an arbitrary modulation signal by a single-frequency tone of the maximum frequency for the purpose of estimating an appropriate sampling rate. Thus, $S(t)$ is replaced by $\sin\left((2\pi f_S/2)\, t\right)$. The new form of $U(S(t))$ is:

$$\tilde{U}(\tilde{S}(t)) = A\cos\left((2\pi f_U/2)\sin\left((2\pi f_S/2)\, t\right)\right).$$

In order to apply Carson's rule, $\tilde{U}(\tilde{S}(t))$ has to be matched to $s_{\mathrm{fm}}(t)$, which is defined by

$$s_{\mathrm{fm}}(t) := A\cos\left(2\pi f_c t + (\Delta f/f_m)\sin\left(2\pi f_m t\right)\right).$$

For this purpose, f_m should be identified with half the Nyquist frequency f_S of the scalar field, and f_c has to be 0 as there is no "carrier frequency" for the transfer function. Thus, $\Delta f/f_m$ should be identified with $2\pi f_U/2$.

According to Carson's rule, the required frequencies for this signal ($f_c = 0$) are in the interval $[0, \Delta f + f_m]$ corresponding to $[0, 2\pi f_U f_S/4 + f_S/2]$. For $f_U f_S \gg f_S$ this interval is given by $[0, \pi f_U f_S/2]$, i.e., the required sampling frequency is $\pi f_U f_S$.

Thus, it is by no means sufficient to sample the volume rendering integral with the Nyquist frequency f_S of the scalar field if non-linear transfer functions are employed. Artifacts resulting from this kind of undersampling are frequently observed unless they are avoided by very smooth transfer functions, i.e., transfer functions with a small Nyquist frequency f_U.

2.5 Pre-Integrated Classification

In order to overcome the limitations discussed above, the approximation of the volume rendering integral has to be improved. In fact, many improvements have been proposed, e.g., higher-order integration schemes, adaptive sampling, etc. However, these methods do not explicitly address the problem of high sampling frequencies required for non-linear transfer functions. With *pre-integrated classification* these high sampling frequencies are avoided by reconstructing a piecewise linear, continuous scalar function along the viewing ray, and evaluating the volume rendering integral between each pair of successive samples of the scalar field by table lookups. This allows us to avoid the problematic product of Nyquist frequencies mentioned in the previous section since the sampling rate for the reconstruction of the scalar function along the viewing ray does not dependent on the transfer functions.

Pre-integrated classification was developed together with Stefan Röttger and Klaus Engel as part of this work; therefore, it will be described in some detail here. Applications of this technique to particular volume rendering algorithms are discussed in Sections 3.1, 3.3, and 5.1. This section was first published in a similar form in [18].

2.5.1 Ray Integration with Pre-Integrated Classification

Ray integration requires the sampling of a continuous scalar field $s(\mathbf{x})$ along a viewing ray. Note that the Nyquist frequency for this sampling is not affected by the transfer functions, in contrast to the sampling of the volume rendering integral, which was discussed in detail in the previous section. For the purpose of pre-integrated classification, the sampled values of the three-dimensional scalar field define a one-dimensional, piecewise linear scalar field, which approximates the original scalar field along the viewing ray. The volume rendering integral for this piecewise linear scalar field is efficiently computed by one table lookup for each ray segment. The three arguments of this table lookup for the i-th ray segment from $\mathbf{x}(i)$ to $\mathbf{x}(i\,d)$ are the scalar value at the start (front) of the segment $s_f := s\big(\mathbf{x}(i\,d)\big)$, the scalar value at the end (back) of the segment $s_b := s\big(\mathbf{x}((i+1)d)\big)$, and the length of the segment d; see Figure 2.6. For the purpose of illustration,

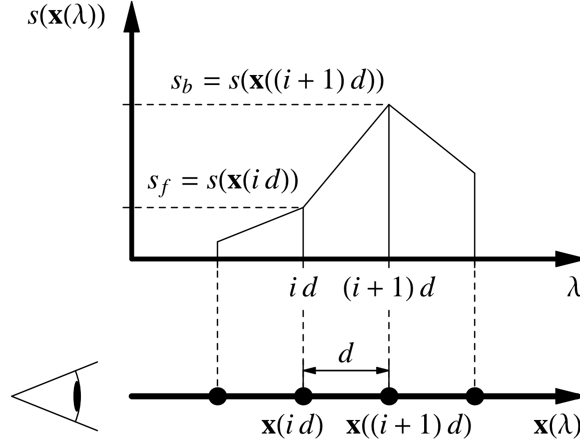


Figure 2.6 Piecewise linear approximation of the function $s(\mathbf{x}(\lambda))$ along a viewing ray.

we assume that the lengths of the segments are all equal to a constant d. In this case, the table lookup is independent of d.

More precisely spoken, the opacity α_i of the i-th segment defined in Equation (2.2) is approximated by

$$\alpha_i \approx 1 - \exp\left(-\int_0^1 \tau\big((1-\omega)s_f + \omega s_b\big) d\,\mathrm{d}\omega\right). \tag{2.7}$$

Thus, α_i is a function of s_f, s_b, and d. (Or of s_f and s_b, if the lengths of the segments are equal.)

The (associated) color $\tilde{C}_i$ defined in Equation (2.3) is approximated correspondingly:

$$\begin{aligned} \tilde{C}_i \approx & \int_0^1 \tilde{c}\big((1-\omega)s_f + \omega s_b\big) \\ & \times \exp\left(-\int_0^\omega \tau\big((1-\omega')s_f + \omega' s_b\big) d\,\mathrm{d}\omega'\right) d\,\mathrm{d}\omega. \end{aligned} \tag{2.8}$$

Analogously to α_i, $\tilde{C}_i$ is a function of s_f, s_b, and d.

Thus, pre-integrated classification will approximate the volume rendering integral by evaluating Equation (2.4):

$$I \approx \sum_{i=0}^{\lfloor D/d \rfloor} \tilde{C}_i \prod_{j=0}^{i-1} (1-\alpha_j)$$

with colors $\tilde{C}_i$ pre-computed according to Equation (2.8) and opacities α_i pre-computed according to Equation (2.7).

For non-associated color transfer function, i.e., if $\tilde{c}(s)$ is substituted by $\tau(s)$ $c(s)$, we will also employ Equation (2.7) for the approximation of α_i and the following approximation of the associated color $\tilde{C}_i^\tau$:

$$\begin{aligned} \tilde{C}_i^\tau \approx & \int_0^1 \tau\big((1-\omega)s_f + \omega s_b\big) c\big((1-\omega)s_f + \omega s_b\big) \\ & \times \exp\Big(-\int_0^\omega \tau\big((1-\omega')s_f + \omega' s_b\big) d\, \mathrm{d}\omega'\Big) d\, \mathrm{d}\omega. \end{aligned} \tag{2.9}$$

Note that pre-integrated classification always computes associated colors, whether a transfer function for associated colors $\tilde{c}(s)$ or for non-associated colors $c(s)$ is employed.

In both cases, pre-integrated classification allows us to sample a continuous scalar field $s(\mathbf{x})$ without the need to increase the sampling rate for any non-linear transfer function. Therefore, pre-integrated classification has the potential to improve the accuracy (by less undersampling) and the performance (by fewer sampling operations) of a volume renderer at the same time.

Figure 2.7c summarizes the basic steps of pre-integrated classification and compares them with pre-classification (Figure 2.7a) and post-classification (Figure 2.7b).

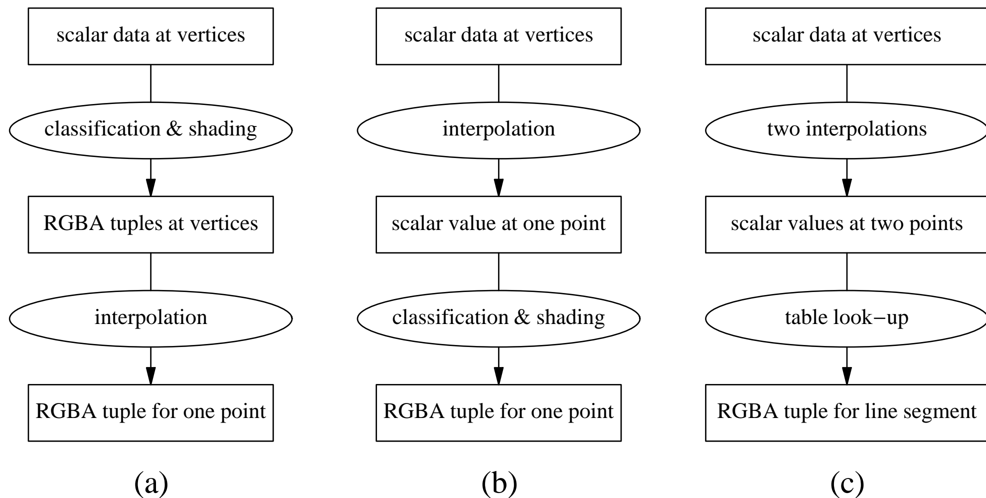


Figure 2.7 Data flow schemes for **(a)** pre-classification, **(b)** post-classification, and **(c)** pre-integrated classification.

2.5.2 Accelerated Approximative Pre-Integration

The primary drawback of pre-integrated classification in general is the required pre-computation of the lookup tables that map the three integration parameters (scalar value at the front s_f, scalar value at the back s_b, and length of the segment d) to pre-integrated colors $\tilde{C} = \tilde{C}(s_f, s_b, d)$ and opacities $\alpha = \alpha(s_f, s_b, d)$. As these tables depend on the transfer functions, any modification of the transfer functions requires an update of the lookup tables. This might be no concern for games and entertainment applications, but it strongly limits the interactivity of applications in the domain of scientific volume visualization, which often depend on user-specified transfer functions. Therefore, we will suggest three approaches to accelerating the pre-integration step.

Firstly, it is sometimes possible to reduce the dimensionality of the tables from three to two (only s_f and s_b) by assuming a constant length of the segments. Obviously, this applies to ray casting with equidistant samples. It also applies to 3D texture-based volume visualization with orthographic projection and is a good approximation for most perspective projections. It is less appropriate for object-aligned 2D texture-based volume rendering as discussed in Section 5.1. Even if very different lengths occur, the complicated dependency on the segment length might be approximated by a linear dependency as suggested in [56]; thus, the lookup tables may be calculated for a single segment length.

Secondly, a local modification of the transfer functions for a particular scalar value s does not require to update the whole lookup table. In fact, only the values $\tilde{C}(s_f, s_b, d)$ and $\alpha(s_f, s_b, d)$ with $s_f \leq s \leq s_b$ or $s_f \geq s \geq s_b$ have to be updated; i.e., in the worst case about half of the lookup table has to be updated.

Finally, the pre-integration may be greatly accelerated by evaluating the integrals in Equations (2.7), (2.8), and (2.9) by employing integral functions for $\tau(s)$, $\tilde{c}(s)$, and $\tau(s)c(s)$, respectively. A very similar technique has been first published by Max et al. in [44]. More specifically, Equation (2.7) for $\alpha_i = \alpha(s_f, s_b, d)$ can be rewritten as

$$\begin{aligned} \alpha(s_f, s_b, d) &\approx 1 - \exp\left(-\int_0^1 \tau\big((1-\omega)s_f + \omega s_b\big) d \, \mathrm{d}\omega\right) \\ &= 1 - \exp\left(-\frac{d}{s_b - s_f}\int_{s_f}^{s_b} \tau(s)\,\mathrm{d}s\right) \\ &= 1 - \exp\left(-\frac{d}{s_b - s_f}\big(T(s_b) - T(s_f)\big)\right) \end{aligned} \tag{2.10}$$

with the integral function $T(s) := \int_0^s \tau(s')\mathrm{d}s'$, which is easily computed in practice as the scalar values s are usually quantized.

Equation (2.8) for $\tilde{C}_i = \tilde{C}(s_f, s_b, d)$ may be approximated analogously; however, this requires to neglect the attenuation within a ray segment. As mentioned above, this is a common approximation for post-classified volume rendering and is well justified for small products $\tau(s)d$.

$$\begin{aligned}
\tilde{C}(s_f, s_b, d) &\approx \int_0^1 \tilde{c}\big((1-\omega)s_f + \omega s_b\big) d\mathrm{d}\omega \\
&= \frac{d}{s_b - s_f} \int_{s_f}^{s_b} \tilde{c}(s)\mathrm{d}s \\
&= \frac{d}{s_b - s_f}\big(K(s_b) - K(s_f)\big)
\end{aligned} \tag{2.11}$$

with the integral function $K(s) := \int_0^s \tilde{c}(s')\mathrm{d}s'$. For the non-associated color transfer function $c(s)$ we approximate Equation (2.9) by

$$\begin{aligned}
\tilde{C}^{\tau}(s_f, s_b, d) &\approx \int_0^1 \tau\big((1-\omega)s_f + \omega s_b\big) \\
&\quad \times c\big((1-\omega)s_f + \omega s_b\big) d\mathrm{d}\omega \\
&= \frac{d}{s_b - s_f} \int_{s_f}^{s_b} \tau(s)c(s)\mathrm{d}s \\
&= \frac{d}{s_b - s_f}\big(K^{\tau}(s_b) - K^{\tau}(s_f)\big).
\end{aligned} \tag{2.12}$$

with $K^{\tau}(s) := \int_0^s \tau(s')c(s')\mathrm{d}s'$.

Thus, instead of numerically computing the integrals in Equations (2.7), (2.8), and (2.9) for each combination of s_f, s_b, and d, it is possible to compute the integral functions $T(s)$, $K(s)$, or $K^{\tau}(s)$ only once and employ these to evaluate colors and opacities according to Equations (2.10), (2.11), or (2.12) without any further integration.

2.5.3 Applications to Volume Rendering Algorithms

Pre-integrated classification is not restricted to a particular volume rendering algorithm, rather it may replace the post-classification step of various algorithms. The application to a cell projection algorithm is discussed in Section 3.1, while texture-based rendering algorithms with this kind of classification are discussed in Section 5.1. Furthermore, pre-integrated classification has been combined with the shear-warp algorithm [57] and has been employed for ray casting in software [30] and in hardware [47]. Additional improvements of various aspects of pre-integrated volume rendering are discussed in [24], [46], and [55].

2.6 Classification of Meshes

As mentioned above, the volume rendering integral is defined for a scalar function in three spatial dimensions, which is usually specified by a volumetric mesh in combination with an interpolation scheme. However, a rich variety of mesh data structures has been suggested in order to meet different requirements of particular applications. Unfortunately, the requirements of the visualization process are usually not taken into consideration; therefore, volume visualization is required to cope with all kinds of meshes.

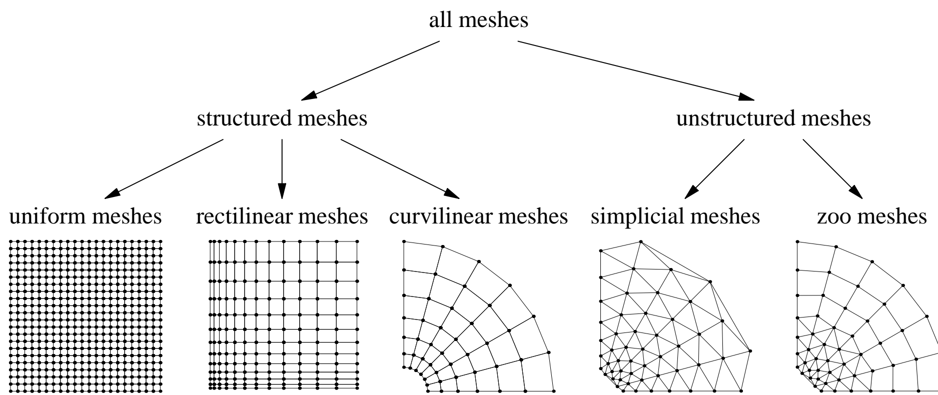


Figure 2.8 Basic classification of meshes with two-dimensional examples.

2.6.1 Structured Meshes

Figure 2.8 depicts are rough classification of meshes with two-dimensional examples. The most general distinction is between *structured* and *unstructured meshes*. While the connectivity of cells in a structured mesh is implicit, this connectivity is not fixed for an unstructured mesh.

Structured meshes may be further classified into *uniform*, *rectilinear*, and *curvilinear meshes*. The cells of uniform meshes are of uniform shape and size. Often, and in particular in this thesis, only uniform meshes with orthogonal axes are considered, i.e., the cells are cuboids or “boxes”. In this case, the position of a vertex $\mathbf{v}^{\text{uni}}_{i,j,k}$ of a uniform volume mesh of dimensions $n_x \times n_y \times n_z$ with cells of size $\Delta x \times \Delta y \times \Delta z$ may be written as

$$\mathbf{v}^{\text{uni}}_{i,j,k} = \mathbf{t} + O \begin{pmatrix} i\Delta x \\ j\Delta y \\ k\Delta z \end{pmatrix}, \qquad i = 1, \dots, n_x; j = 1, \dots, n_y; k = 1, \dots, n_z,$$

with a (translation) vector $\mathbf{t}$ and a rotation matrix O.

Rectilinear meshes are slightly more general as the sizes Δx, Δy, and Δz of the cells may depend on the indices i, j, and k. However, Δx may only depend on i, Δy on j, and Δz on k. Therefore, the position of a vertex $\mathbf{v}_{i,j,k}^{\text{rect}}$ can be specified as

$$\mathbf{v}_{i,j,k}^{\text{rect}} = \mathbf{t} + O \begin{pmatrix} x(i) \\ y(j) \\ z(k) \end{pmatrix}, \qquad i = 1, \ldots, n_x; j = 1, \ldots, n_y; k = 1, \ldots, n_z,$$

with coordinate functions $x(i)$, $y(j)$, and $z(k)$.

While the connectivity of vertices in a curvilinear mesh is the same as that of a uniform or rectilinear mesh, the position of a vertex $\mathbf{v}_{i,j,k}^{\text{curv}}$ of a curvilinear mesh is not restricted in a similar way; therefore, a three-dimensional vector has to be stored for each vertex specifying its position. However, the cells of a curvilinear mesh are often restricted to be non-intersecting or even non-degenerate.

2.6.2 Unstructured Meshes

Unstructured meshes have no predefined connectivity; therefore, the connectivity has to be stored explicitly. While this requires larger data structures, it also leads to greater flexibility, i.e., more adaptivity. Thus, unstructured meshes can discretize domains of any shape and topology. Also, the resolution of an unstructured mesh may be adapted locally.

In particular in the context of finite-element computation, many variants of unstructured meshes have been suggested. Among the most important are *simplicial meshes*, i.e., meshes with cells that are simplices. An n-dimensional simplex is the simplest polytope in n dimensions, i.e., the n-dimensional generalization of a tetrahedron. Therefore, a simplicial mesh in three dimensions is a tetrahedral mesh and a triangular mesh in two dimensions. An important feature of simplicial meshes is the possibility of linear interpolation within each simplicial cell as discussed in the next section.

Non-simplicial unstructured meshes may be further classified by the shapes of their cells. For example, an unstructured mesh with several different cell shapes (e.g., tetrahedra, octahedra, cubes, and prisms) is sometimes called *zoo mesh*.

2.6.3 Hybrid and Hierarchical Meshes

There are also several kinds of meshes that do not fit into this basic classification of structured and unstructured meshes. For example, *hybrid meshes*, i.e., combinations of different kinds of meshes, and *hierarchical meshes*, i.e., meshes that are defined for different levels of detail. As the particular problems associated with these meshes are usually not of a geometric nature, these meshes are not discussed in detail in this work.

2.7 Interpolation in Meshes

As mentioned in Section 2.4, almost all volume rendering algorithms sample scalar fields that are defined by the data values associated with the vertices of a discrete volumetric mesh. This sampling requires an interpolation scheme that computes an interpolated value from the vertices' scalar values and the position of the sampling point.

2.7.1 Nearest-Neighbor Interpolation

The simplest interpolation scheme is a *nearest-neighbor interpolation*, i.e., the scalar value of a sampling point is determined by the scalar value associated with the nearest vertex of the mesh. For an appropriately transformed mesh (the dual mesh), this is equivalent with a constant scalar value per volumetric cell or voxel. As a nearest-neighbor interpolation results in discontinuous "jumps" of the interpolated value, more elaborated interpolation schemes have to be employed in order to guarantee interpolated values that are continuous at boundaries of cells.

2.7.2 Linear, Bilinear, and Trilinear Interpolation

Trilinear interpolation is a continuous interpolation scheme for uniform and rectilinear volumetric meshes. As illustrated in Figure 2.9 it reduces to *bilinear interpolation* in two dimensions and *linear interpolation* in one dimension.

For the linear interpolation in one dimension, two scalar values $s(A)$ and $s(B)$ at points A and B are required (Figure 2.9a). The linearly interpolated value $s(X)$ at any point X between A and B is then defined by

$$s(X) = (1-\alpha)\, s(A) + \alpha\, s(B) \quad \text{with} \quad \alpha = \frac{|X-A|}{|B-A|}.$$

Bilinear interpolation in a rectangle requires four points with associated scalar values, see Figure 2.9b. The interpolated value $s(X)$ at a point X is then given by

$$s(X) = (1-\alpha)\,(1-\beta)\, s(A) + \alpha\,(1-\beta)\, s(B) + \alpha\,\beta\, s(C) + (1-\alpha)\,\beta\, s(D)$$

with

$$\alpha = \frac{|X_{AB}-A|}{|B-A|} \quad \text{and} \quad \beta = \frac{|X_{AD}-A|}{|D-A|},$$

where X_{AB} and X_{AD} are the projections of X onto the lines $\overline{AB}$ and $\overline{AD}$, respectively.

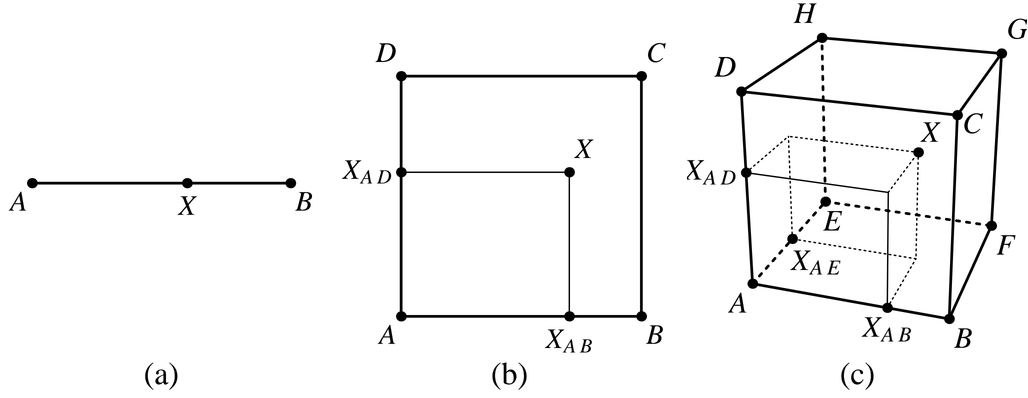


Figure 2.9 Interpolation in n-dimensional cubic cells: **(a)** linear on a line segment, **(b)** bilinear in a square, and **(c)** trilinear in a cube.

Analogously, trilinear interpolation in a cuboid requires eight points and their scalar values, as illustrated in Figure 2.9c. $s(X)$ at X is then interpolated as

$$\begin{aligned} s(X) &= (1-\alpha)\,(1-\beta)\,(1-\gamma)\,s(A) + \alpha\,(1-\beta)\,(1-\gamma)\,s(B) \\ &\quad + \alpha\,\beta\,(1-\gamma)\,s(C) + (1-\alpha)\,\beta\,(1-\gamma)\,s(D) \\ &\quad + (1-\alpha)\,(1-\beta)\,\gamma\,s(E) + \alpha\,(1-\beta)\,\gamma\,s(F) \\ &\quad + \alpha\,\beta\,\gamma\,s(G) + (1-\alpha)\,\beta\,\gamma\,s(H) \end{aligned}$$

with

$$\alpha = \frac{|X_{AB}-A|}{|B-A|}, \quad \beta = \frac{|X_{AD}-A|}{|D-A|}, \quad \text{and} \quad \gamma = \frac{|X_{AE}-A|}{|E-A|},$$

where X_{AB}, X_{AD}, and X_{AE} are projections of X as indicated in Figure 2.9c.

It should be noted that any bilinear and trilinear interpolation may be computed by a sequence of linear interpolations. For example, the bilinear interpolation of $s(X)$ in Figure 2.9b may be written as the three linear interpolations

$$\begin{aligned} s(X_{AB}) &= (1-\alpha)\,s(A) + \alpha\,s(B), \\ s(X_{CD}) &= (1-\alpha)\,s(D) + \alpha\,s(C), \\ s(X) &= (1-\beta)\,s(X_{AB}) + \beta\,s(X_{CD}), \end{aligned}$$

where X_{CD} is the projection of X onto the line $\overline{CD}$, and the weights α and β are defined as above. Analogously, a trilinear interpolation may be computed by a sequence of seven linear interpolations.

Another important feature of these interpolation schemes is the restriction to the vertices of a cell, i.e., in order to interpolate the value of any point within a cell only the values of the vertices of this cell are required.

2.7.3 Linear Interpolation in Simplices

Linear interpolation is not only useful in one dimension but may be employed for a continuous interpolation in any simplicial mesh, e.g., in triangular and tetrahedral meshes. The one-dimensional case depicted in Figure 2.10a is equivalent to the linear interpolation in a one-dimensional cube, which is covered by the discussion of Figure 2.9.

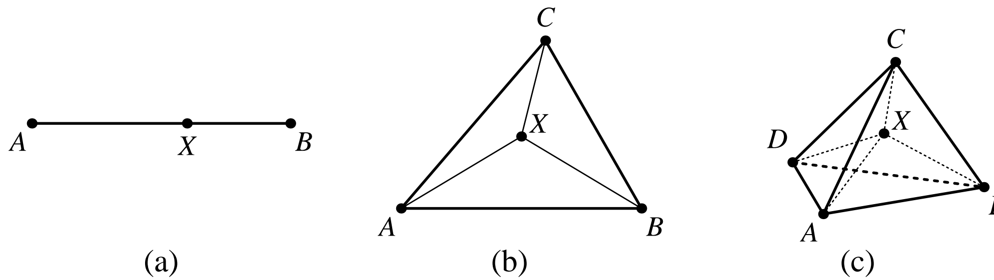


Figure 2.10 Linear interpolation in n-dimensional simplices: **(a)** on a line segment, **(b)** in a triangle, and **(c)** in a tetrahedron.

In two dimensions, a linear interpolation of the scalar value $s(X)$ at a point X within a triangle $\triangle ABC$ (see Figure 2.10b) may be written as

$$s(X) = \alpha\, s(A) + \beta\, s(B) + \gamma\, s(C),$$

where the weights α, β, and γ are the barycentric coordinates of X in $\triangle ABC$, i.e., α, β, and γ are the areas of the triangles $\triangle XBC$, $\triangle AXC$, and $\triangle ABX$, respectively, each divided by the area of $\triangle ABC$ for normalization.

Analogously, a linear interpolation for $s(X)$ at X within a tetrahedron $\boxtimes ABCD$ (see Figure 2.10c) may be computed as

$$s(X) = \alpha\, s(A) + \beta\, s(B) + \gamma\, s(C) + \delta\, s(D)$$

with the weights α, β, γ, and δ given by the volumes of the tetrahedra $\boxtimes XBCD$, $\boxtimes AXCD$, $\boxtimes ABXD$, and $\boxtimes ABCX$, respectively, again each divided by the volume of the whole tetrahedron $\boxtimes ABCD$.

Similarly to the decomposition of any bilinear or trilinear interpolation into linear interpolations, any linear interpolation in triangles or tetrahedra can be decomposed into a sequence of linear interpolations on lines (and triangles). For the case of the linear interpolation in the triangle of Figure 2.11a, let X_{AC} and X_{BC} be any two points on the line segments $\overline{AC}$ and $\overline{BC}$, respectively, such that X is on the

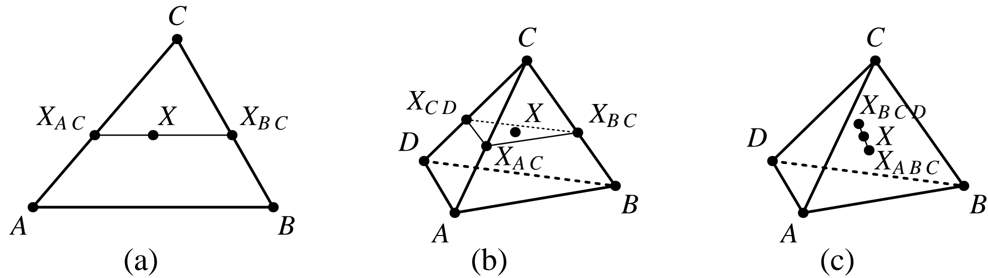


Figure 2.11 Decomposition of linear interpolations: **(a)** in a triangle, **(b)** in a tetrahedron, and **(c)** alternative decomposition in a tetrahedron.

line $\overline{X_{AC}X_{BC}}$. Then $s(X)$ can be calculated by these three linear interpolations on lines:

$$\begin{aligned} s(X_{AC}) &= (1-\alpha)\, s(A) + \alpha\, s(C), \\ s(X_{BC}) &= (1-\beta)\, s(B) + \beta\, s(C), \\ s(X) &= (1-\gamma)\, s(X_{AC}) + \gamma\, s(X_{BC}) \end{aligned}$$

with

$$\alpha = \frac{|X_{AC} - A|}{|C - A|}, \quad \beta = \frac{|X_{BC} - B|}{|C - B|}, \quad \text{and} \quad \gamma = \frac{|X - X_{AC}|}{|X_{BC} - X_{AC}|}.$$

For the case of the linear interpolation in a tetrahedron there are several possible decompositions. One approach is to interpolate the value s at three points on edges of the tetrahedron, which form a triangle that contains X, and then to interpolate $s(X)$ in this triangle as illustrated in Figure 2.11b. Of course, the interpolation in the triangle $\triangle X_{AC}X_{BC}X_{CD}$ may be further decomposed as explained above.

Another approach starts by interpolating the value s at two points of two faces of the tetrahedron. This is illustrated in Figure 2.11c for a point X_{ABC}, which is part of the triangle $\triangle ABC$, and a point X_{BCD}, which is part of $\triangle BCD$. The line between these two points has to contain X; thus, $s(X)$ can be computed by a linear interpolation along this line.

2.7.4 Higher-Order Interpolation

Although there are many more elaborated interpolation schemes, in particular of higher order, we will not discuss any of them as the choice between particular interpolation schemes usually does not depend on the geometry of the cells of a

mesh. (An important exception is that a continuous linear interpolation is only possible in simplicial meshes.) Moreover, real-time interactive volume visualization requires interpolation schemes that can be evaluated extremely efficiently; a requirement that is hard to fulfill with any higher-order interpolation scheme.

2.8 Geometrically Unpleasant Meshes

The discussion of different volume rendering algorithms, meshes, and interpolation schemes in the previous sections of this chapter leads to several important questions that are related to geometric features of meshes. Partial answers to these questions can be given based on the previous sections and the literature on volume visualization:

Q.: How do geometric features of the boundary of a volumetric mesh affect volume rendering algorithms?

A.: The volume rendering integral is defined on the intersection of a viewing ray with the spatial domain of a volume mesh. This intersection is empty or exactly one line segment for convex meshes, i.e., meshes with a convex boundary, and it is empty or a set of disconnected line segments for non-convex meshes. Therefore, volume visualization of non-convex meshes is in general more difficult than volume visualization of convex meshes.

Q.: The possibility of a simple, continuous, linear interpolation within volumetric meshes is limited to simplicial meshes. What are the consequences of this geometric feature and/or of its absence?

A.: As discussed in Section 2.5, pre-integrated classification implicitly assumes a linear interpolation between samples. Moreover, almost all algorithms that compute single features or the complete topology of a scalar field are restricted to piecewise linear scalar fields as provided by simplicial meshes with a linear interpolation scheme. Adaptations of pre-integrated classification and topology-related algorithms to non-simplicial meshes would considerably increase the range of application of these concepts.

Q.: The accurate evaluation of the volume rendering integral requires a large number of interpolations. How can these interpolations be computed efficiently with the help of graphics hardware for cells of different geometric shapes?

A.: With standard graphics hardware, only trilinear interpolation in uniform volumetric meshes is supported natively by means of three-dimensional texture mapping. There is little to no native support for non-uniform meshes.

Q.: Apart from the shape of cells and the boundary of a mesh, which other geometric features of a volumetric mesh affect volume rendering algorithms?

A.: Previous research in volume visualization has shown that the acyclicity of volumetric meshes is a crucial requirement of some algorithms. Only very few publications were concerned with problems caused by cyclic meshes. Mesh simplification is necessary in many applications of volume visualization in order to achieve a sufficient visualization performance. However, mesh simplification usually exploits the possibility to adapt the boundary and the local resolution of a mesh to the data defined on it; thus, the simplification of non-adaptive meshes poses particular challenges.

These answers identify specific geometric features of volume meshes that cause particular problems. These features are called "geometrically unpleasant" in this work and lead to its overall structure, i.e., each of the Chapters 3 to 5 is about algorithms for the direct volume visualization of particular classes of unpleasant meshes. More specifically, Chapter 3 discusses non-uniform meshes, Chapter 4 non-convex and cyclic meshes, and Chapter 5 non-simplicial and non-adaptive meshes.

Of course, there are — even from the limited perspective of volume visualization — more geometrically unpleasant meshes than mentioned above (see Section 6.3 for a list of examples). Therefore, the results of Chapters 3 to 5 are generalized in Chapter 6 in order to find more abstract strategies for the direct volume visualization of geometrically unpleasant meshes.

Big Bill: *People, please. Please.*
I think we all know what's been going on here.
Up until now everything around here has always been,
well, pleasant.
Recently, certain things have become unpleasant.
Now, it seems to me that the first thing we have to do
is to separate out the things that are pleasant
from the things that are unpleasant.

Monolog from the movie *Pleasantville*

Chapter 3

Non-Uniform Meshes

While volume rendering of uniform meshes may be implemented with the help of three-dimensional texture mapping as described in Section 2.3, there is no comparable hardware support for any non-uniform mesh. This lack of hardware support is particularly unpleasant since non-parallel, pure software solutions are usually considerably slower than hardware-assisted implementations. Therefore, this chapter presents three hardware-assisted approaches to volume rendering of non-uniform meshes. However, the discussion will be restricted to simplicial (i.e., tetrahedral) meshes. In order to apply the presented methods to other non-uniform volumetric meshes, they have to be converted to tetrahedral meshes by decomposing all non-tetrahedral cells into tetrahedra.

3.1 Pre-Integrated Cell Projection

Cell projection probably is the most common method of exploiting graphics hardware for the rendering of tetrahedral meshes and unstructured meshes in general. As mentioned in Section 2.5, pre-integrated classification can be applied to many volume rendering algorithms in order to improve image quality and reduce the number of sampling operations. The goal of this section is to combine pre-integrated classification with a cell projection algorithm without sacrificing the use of graphics hardware. This section presents the results of joint work with Stefan Röttger, which were first published in [56].

3.1.1 Projected Tetrahedra Algorithms

The first cell projection algorithm that exploited graphics hardware efficiently was the Projected Tetrahedra (PT) algorithm by Shirley and Tuchman, which was published in [61]. As this algorithm accelerated direct volume rendering of tetrahedral meshes considerably, many research groups started to use and improve the algorithm. In fact, several aspects of the PT algorithm are still subject to research, e.g., the sorting of tetrahedral cells (see also Section 4.3). In this section, however, only the rendering of projected tetrahedra is discussed.

The basic idea of the PT algorithm is to visualize a scalar function $s(\mathbf{x})$ defined for any point $\mathbf{x}$ in a region of three-dimensional space by rendering partially transparent polygons, which can be processed extremely fast by specialized graphics hardware.

The original PT algorithm can be summarized as follows (see also [61]):

1. Decompose the volume into tetrahedral cells. Scalar values are defined at each vertex of the mesh. Inside each tetrahedral cell, $s(\mathbf{x})$ is assumed to be a linear combination of the vertex values as described in Section 2.7.3.

2. Sort the cells according to their visibility.

3. Classify each tetrahedron according to its projected profile and decompose the projected tetrahedron into smaller triangles (see Figure 3.1).

4. Find color and opacity values for the triangle vertices using ray integration.

5. Render the triangles.

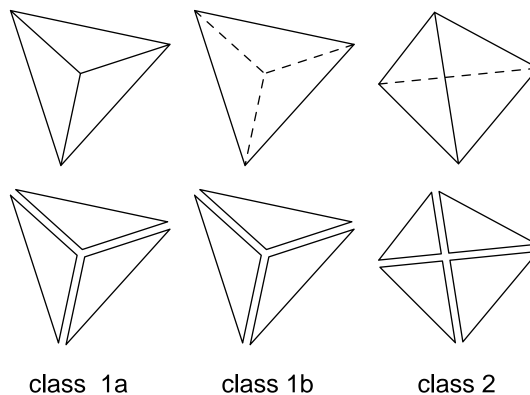


Figure 3.1 Classification of non-degenerate projected tetrahedra (top row) and the corresponding decompositions (bottom row) according to [61].

As mentioned, the methods presented in this section only improve on the latter two points: ray integration and rendering of the decomposed triangles with an emphasis on hardware-accelerated rendering. Visibility sorting is discussed in Section 4.3 as it is particularly complicated for non-convex and cyclic meshes, which are the topic of Chapter 4.

The original PT algorithm interpolates color and opacity linearly between the triangle vertices. This, however, is an approximation which leads to rendering artifacts as demonstrated in [43, 65].

In order to avoid these artifacts Stein et al. suggested in [65] to use a two-dimensional texture with the texture coordinates being the thickness l of the projected cell and the average τ of the two extinction coefficients on the entry and the exit face (see also Figure 3.2). These texture coordinates are computed only at the triangle vertices. The texture specifies an α-component that is set to $\alpha = 1 - \exp(-\tau\, l)$. This α-component is used to modulate the (unattenuated) vertex colors; thus, the exponential attenuation is computed for each pixel instead of each vertex. In between the vertices of each triangle the texture coordinates and, therefore, τ and l are interpolated linearly; thus, this approach is restricted to a linearly varying extinction coefficient τ, i.e., a linear transfer function $\tau = \tau(s)$. Moreover, the color is still linearly interpolated in between vertices. Williams et al. extended these ideas to piecewise linear transfer functions in [82].

3.1.2 Pre-Integrated Classification for Projected Tetrahedra

In the context of projected tetrahedra, pre-integrated classification is just a generalization of the method of Stein et al. [65], as it works for color and opacity, and places no restrictions on the transfer functions. These benefits are achieved by employing three-dimensional instead of two-dimensional texture mapping as explained next.

Figure 3.2 depicts the intersection of a viewing ray with a tetrahedral cell. More precisely spoken, this particular tetrahedron corresponds to a triangle generated by the PT decomposition. The goal is to render this tetrahedron by rasterizing its front face. Thus, the ray integration within the tetrahedron has to be performed by appropriate texture mapping.

As texture coordinates are interpolated linearly, only those variables should be used that vary vary linearly with screen coordinates. For orthographic projections, l varies linearly for geometric reasons; s_f and s_b vary linearly because they are interpolated linearly between vertices as mentioned above. Therefore, s_f, s_b, and l should be the three texture coordinates. Fortunately, all other values, e.g., color, opacity, etc., can be calculated from l, s_f, and s_b. Thus, a three-dimensional texture can be computed that specifies the color and opacity characterizing the intersection of a ray and a cell in dependency of l, s_f, and s_b.

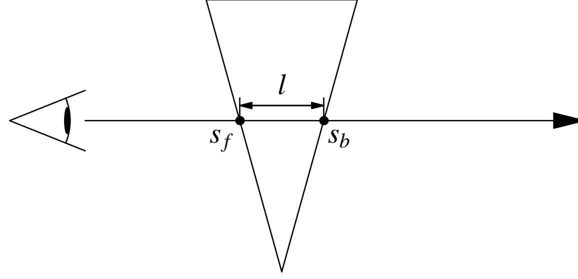


Figure 3.2 Intersecting a tetrahedral cell with a viewing ray. s_f and s_b are the scalar values on the entry (front) and exit (back) face, respectively; l denotes the thickness of the cell for this ray.

For many applications the calculation of the texture is a preprocessing step and, therefore, not time-critical. Usually it includes a numerical integration of a ray for each texel in the three-dimensional texture. In the notation of Section 2.5, the transfer functions are specified by an associated color $\tilde{c} = \tilde{c}(s)$ and an opacity $\tau = \tau(s)$.

Assuming that cells are processed back to front, the addition of the projection of a cell changes an existing pixel color $\tilde{C}$ to a new pixel color $\tilde{C}'$ by the formula (compare Section 2.4 in this work and Equation (4) in [81])

$$\tilde{C}' = \underbrace{\int_0^l \exp\left(-\int_0^t \tau(s_l(u))\mathrm{d}u\right) \tilde{c}(s_l(t))\mathrm{d}t}_{\mathrm{RGB_{t3D}}} + \underbrace{\exp\left(-\int_0^l \tau(s_l(t))\mathrm{d}t\right)}_{1-\alpha_{\mathrm{t3D}}} \tilde{C} \tag{3.1}$$

with the abbreviation

$$s_l(t) = s_f + \frac{t}{l}(s_b - s_f).$$

$\mathrm{RGB_{t3D}}$ denotes the color components (note that $\tilde{c}(s)$ is a vector), and α_{t3D} the opacity of an entry in the three-dimensional texture. $\mathrm{RGB_{t3D}}$ and α_{t3D} depend on the texture coordinates l, s_f, s_b, and the transfer functions $\tilde{c}(s)$ and $\tau(s)$. Thus, the texture has to be updated whenever the transfer functions are modified.

It is an intrinsic limitation of the pre-integrated classification that $\tilde{c}(s)$ and $\tau(s)$ have to depend on the same scalar field. Note that this approach is not limited to associated colors. For a non-associated color transfer function $c(s)$, Equation (3.1) reads

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

After the calculation of the texture in a preprocessing step, all tetrahedra are projected from back to front. Before rendering the triangles of one projected tetrahedron, the three texture coordinates l, s_f, and s_b are set for each vertex of the triangles. The required blending operation corresponds to

$$\tilde{C}' = \mathrm{RGB}_{\mathrm{t3D}} + (1 - \alpha_{\mathrm{t3D}})\, \tilde{C},$$

and is performed very efficiently by today's graphics hardware.

A synthetic example generated with this rendering method is depicted in Figure 3.3a. The scalar values at the vertices of the visualized tetrahedral mesh are determined by the (unsigned) distance of each vertex to the surface of a sphere. The transfer function of the opacity is 0 except for a small interval, which results in the two partially opaque rings in Figure 3.3a.

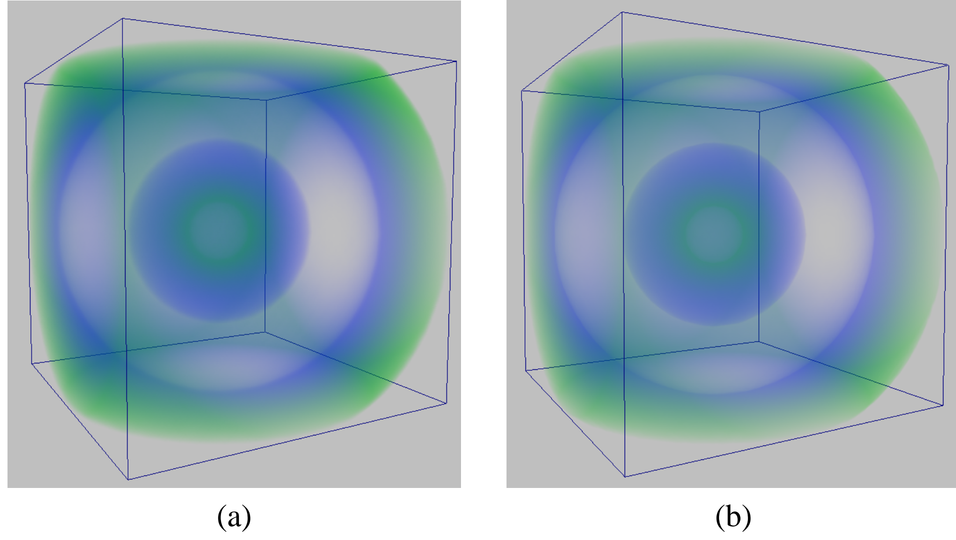
In summary, pre-integrated classification allows us to exploit hardware-supported three-dimensional texture mapping in order to render projected tetrahedra without the need to do any time-consuming calculations for each pixel. The approach is not as accurate as ray casting algorithms or the high accuracy (HIAC) volume rendering system described in [82] because of limited texture memory and non-linear transformations in the case of perspective projections. Especially limited texture memory will limit the size of the three-dimensional texture resulting in a less accurate resampling of the transfer functions. Within this limited accuracy, however, arbitrary transfer functions may be used without affecting the rendering times whereas the performance of the HIAC system crucially depends on the chosen transfer functions. In particular, arbitrarily thin peaks are possible resulting in unshaded isosurfaces as discussed below.

Compared to other volume rendering techniques using three-dimensional texture mapping, the projection of tetrahedra with pre-integrated classification has several specific advantages; for example, it is neither necessary to resample the mesh nor to store the data in texture memory. Moreover, existing implementations of the PT algorithm can easily be extended with this method in order to exploit three-dimensional texture mapping if available.

The most important disadvantages are the need for hardware-supported three-dimensional texture mapping and rather large textures. Moreover, pre-integration is computationally expensive; thus, modifications of the transfer functions cannot be performed in real time.

Approximative Pre-Integrated Classification Using 2D Texture Mapping

In order to overcome these disadvantages, an approximation can be employed that requires only two-dimensional texture mapping. The computation of the corresponding two-dimensional textures is considerably faster than the computation



(a) (b)

Figure 3.3 Visualization of a synthetic data set with non-linear transfer functions. **(a)** Pre-integrated classification with a three-dimensional texture of dimensions $64 \times 64 \times 64$ (1 MB). **(b)** Approximative pre-integrated classification with a two-dimensional texture of dimensions 256×256 (256 KB). *(See also Figure C.1 on page 145 in the color plate section.)*

of the full three-dimensional texture described above because of the reduced dimensionality. This approximative method interpolates the opacity linearly and therefore shows visible artifacts. However, it allows us to use arbitrary transfer functions while existing hardware-accelerated solutions are limited to linear transfer functions within each cell (e.g., [65]).

The basic idea is to approximate the dependencies of the integrals in Equation (3.1) on l by linear terms, and to implement these terms by a modulation of the vertex colors. The remaining integrals depend only on s_f and s_b; thus, they can be tabulated in a two-dimensional texture.

The dependencies on l in Equation (3.1) become more explicit with the variable substitutions $t' = t/l$ and $u' = u/l$:

$$\tilde{C}' = l \int_0^1 \exp\left(-l \int_0^{t'} \tau(s_1(u'))\mathrm{d}u'\right) \tilde{c}(s_1(t'))\mathrm{d}t' + \exp\left(-l \int_0^1 \tau(s_1(t'))\mathrm{d}t'\right) \tilde{C}$$

with $s_1(t) = s_f + t(s_b - s_f)$, which is consistent with the previous definition of $s_l(t)$. For $l = 0$ this equation reduces to $\tilde{C}' = \tilde{C}$. For given $\tau(s)$, $\tilde{c}(s)$, s_f and s_b the integrals are evaluated for another value $l = \bar{l} = \text{const.}$ and extrapolated linearly in l as illustrated in Figure 3.4 for the opacity $1 - \exp(-l \int_0^1 \tau(s_1(t'))\mathrm{d}t')$. The optimal choice of $\bar{l}$ depends on the particular application, but choosing $\bar{l}$ equal

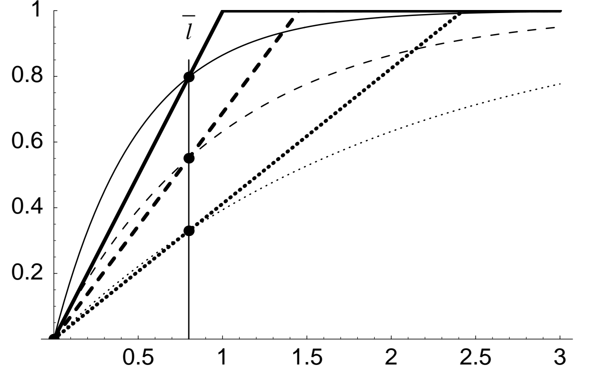


Figure 3.4 Linear approximation of the functions $1 - \exp(-\chi\, l)$ for $\chi = 2$ (*solid thin line*), $\chi = 1$ (*dashed thin line*), and $\chi = \frac{1}{2}$ (*dotted thin line*) by linear functions (*corresponding thick lines*) through $(0,0)$ and $(\bar{l}, \exp(-\chi\bar{l}))$. All function values are clamped to the range $[0,1]$. χ corresponds to integrals that do not depend on l.

to the average cell thickness has proven to be a good approximation. Note that this approximation is improved for large l by clamping the linear term to values between 0 and 1, which is done automatically by many graphics systems.

The two-dimensional texture is defined by

$$\begin{aligned} \mathrm{RGB}_{\mathrm{t2D}} &= \bar{l} \int_0^1 \exp\left(-\bar{l} \int_0^{t'} \tau(s_1(u'))\mathrm{d}u'\right) \tilde{c}(s_1(t'))\mathrm{d}t', \\ \alpha_{\mathrm{t2D}} &= 1 - \exp\left(-\bar{l} \int_0^1 \tau(s_1(t'))\mathrm{d}t'\right) \end{aligned} \tag{3.2}$$

and is modulated by colors at the vertices with the RGBA components set equal to $(l/\bar{l}, l/\bar{l}, l/\bar{l}, l/\bar{l})$. In practice, these colors should be scaled by the maximum opacity value in the texture in order to avoid clamping for values $l > \bar{l}$. This scaling is compensated by multiplying the entries in the texture with the reciprocal value. The combined effect of texturing and blending with appropriate blending coefficients (e.g., in OpenGL `GL_ONE` for the source blend factor and `GL_ONE_MINUS_SRC_ALPHA` for the destination blend factor) is

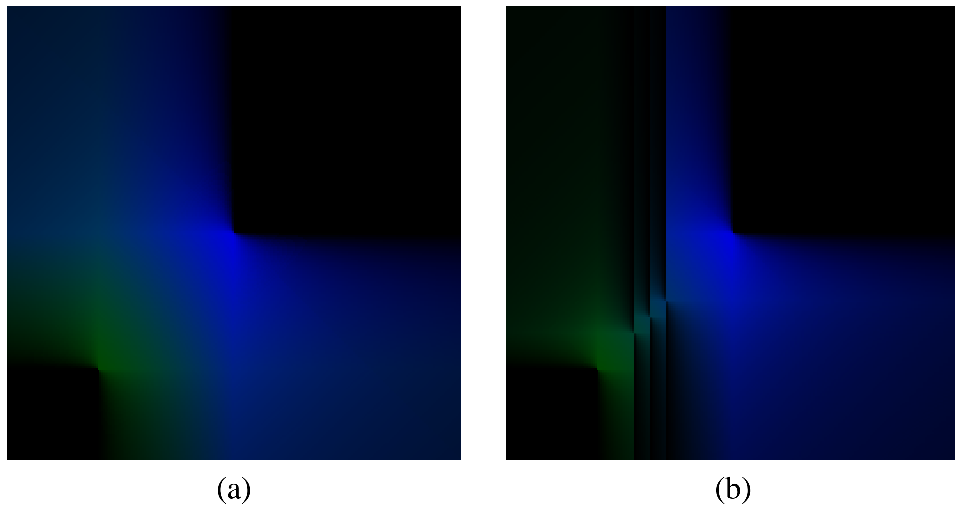
$$\tilde{C}' = (l/\bar{l})\, \mathrm{RGB}_{\mathrm{t2D}} + \left(1 - (l/\bar{l})\, \alpha_{\mathrm{t2D}}\right) \tilde{C},$$

which is our new approximation of Equation (3.1).

On the one hand, this approximation results in artifacts because of the linear interpolation (see [65]); on the other hand, the use of two-dimensional texture mapping enables us to utilize larger textures compared to the three-dimensional textures described above, which results in an improved resampling of the transfer

functions. Moreover, this technique allows us to employ graphics hardware that supports only two-dimensional texture mapping. Another way of achieving the latter goal was published by Röttger and Ertl in [55].

Figure 3.3b shows the synthetic example from Figure 3.3a rendered with two-dimensional instead of three-dimensional texture mapping. The linear approximation leads to slightly smaller opacities resulting in lighter colors while the improved resampling results in sharper edges of the structures generated by the transfer functions. An example of a two-dimensional texture computed with Equation (3.2) is depicted in Figure 3.5a.



(a) (b)

Figure 3.5 Textures for projected tetrahedra with pre-integrated classification. The horizontal (texture) coordinate is s_f, the vertical coordinate is s_b. Black pixels in these images correspond to completely transparent texels. **(a)** The texture employed for the semi-transparent volume in Figure 3.11b. **(b)** In this variant the integration is stopped at the isovalues, which correspond to opaque isosurfaces. *(See also Figure C.2 on page 146 in the color plate section.)*

Rendering of Isosurfaces

As mentioned in Chapter 2, the rendering of isosurfaces is only a special case of direct volume visualization with appropriate transfer functions. For pre-integrated volume rendering these transfer functions correspond to particularly simple pre-integrated textures, which can be derived as follows.

In order to render the isosurface for an isovalue s_{iso}, the opacity transfer function $\tau(s)$ should be defined by $\tau(s) = 0$ for $s \neq s_{\text{iso}}$ and "$\tau(s_{\text{iso}}) = \infty$". Formally, we set $\tau(s) = \xi\delta(s - s_{\text{iso}})$ with a constant ξ and Dirac's delta function δ(x) (see [70]); multiple isosurfaces correspond to a sum of delta functions. As $\tilde{c}(s_{\text{iso}})$ (and

$c(s_{\text{iso}}))$ is constant, we are only interested in the value of α as defined in Equation (3.1):

$$\begin{aligned}
1-\alpha &= \exp\left(-\int_0^l \tau(s_l(t))\mathrm{d}t\right) \\
&= \exp\left(-\int_0^l \xi\delta(s_l(t)-s_{\text{iso}})\mathrm{d}t\right) \\
&= \exp\left(-\int_0^l \xi\delta(s_f+\frac{t}{l}(s_b-s_f)-s_{\text{iso}})\mathrm{d}t\right) \\
&= \exp\left(-\int_0^l \xi\left|\frac{l}{s_b-s_f}\right|\delta\left(t-l\frac{s_{\text{iso}}-s_f}{s_b-s_f}\right)\mathrm{d}t\right) \\
&= \exp\left(-\xi' H\left(\frac{s_{\text{iso}}-s_f}{s_b-s_f}\right)H\left(1-\frac{s_{\text{iso}}-s_f}{s_b-s_f}\right)\right) \\
&= \exp\left(-\xi' H\left(\frac{s_{\text{iso}}-s_f}{s_b-s_f}\right)H\left(\frac{s_b-s_{\text{iso}}}{s_b-s_f}\right)\right)
\end{aligned}$$

with $\xi' = \xi\left|\frac{l}{s_b-s_f}\right|$ and the Heaviside step function $H(x)$ (see [70]). Thus, for $\xi \to \infty$ we obtain

$$\alpha = H\left(\frac{s_{\text{iso}}-s_f}{s_b-s_f}\right)H\left(\frac{s_b-s_{\text{iso}}}{s_b-s_f}\right),$$

which is independent of l. The dependency on s_f and s_b results in a checkerboard-like texture, which is visualized in the right-hand column of Figure 3.6. These two-dimensional textures are in fact special cases of the three-dimensional texture defined in Equation (3.1). An alternative derivation of this result, which is closer to the previous work in [73], is given in [56].

The resulting texture may also be described as follows: Its α-component, i.e., the opacity, has to be 1 for opaque isosurfaces if either the first or the second texture coordinate (but not both) is less than the isovalue, and 0 otherwise (see the right-hand column of Figure 3.6). For flat shading, the RGB-components of the textures and of the vertex colors of the triangles are constant.

Unfortunately, edges of isosurface patches within triangles (see the middle column of Figure 3.6 for some examples) will cause rendering artifacts as there is no mechanism which aligns them exactly to the corresponding edges in the projected tetrahedra in front or behind. Gaps can be avoided by slightly modifying the texture, effectively 'thickening' the isosurface. This eliminates artifacts for opaque isosurfaces; for partially transparent isosurfaces, however, this will visually enhance edges of the tetrahedral mesh by rendering pixels twice. In fact, these

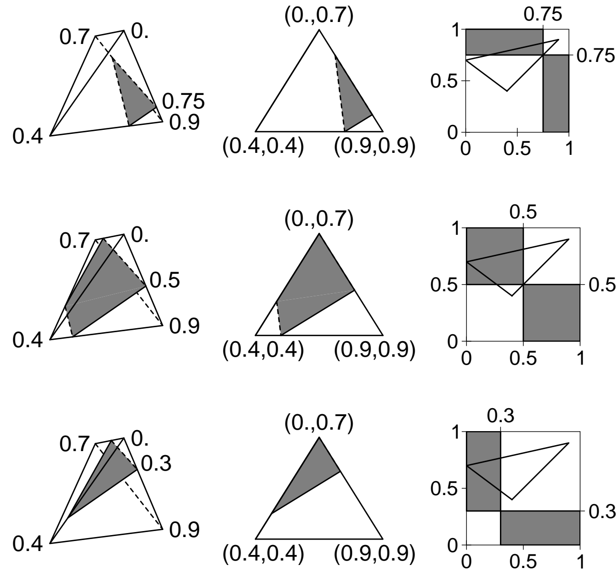


Figure 3.6 Textured triangles for one part of a decomposed projected tetrahedron (middle column) with isosurfaces for isovalues 0.75 (top row), 0.5 (middle row), and 0.3 (bottom row). The left-hand column shows the corresponding part of the tetrahedron slightly rotated with scalar data at the vertices. These values define the texture coordinates included in the images of the actual projections in the middle column. The right-hand column shows the corresponding textures including the triangles in the space of texture coordinates.

edges help to comprehend the three-dimensional structure of flat-shaded isosurfaces. Nonetheless, removing these artifacts for partially transparent isosurfaces is an open problem.

For the special case of a single, semi-transparent isosurface in a tetrahedral mesh these artifacts can be avoided by appropriate fragment tests and a second rasterization of each tetrahedron. This is achieved by clearing the stencil buffer when the polygons of a projected tetrahedron are rendered for the first time. Right afterwards, the same polygons are rendered a second time and the stencil buffer is set where the isosurface cuts the front face of the projected tetrahedron, i.e., for $s_f \approx s_{\text{iso}}$. These pixels can be selected with the help of an alpha test and an additional texture (with texture coordinate s_f) that specifies a particular alpha value for texels with $s_f \approx s_{\text{iso}}$. If only fragments that satisfy the mentioned condition pass the alpha test, the stencil buffer is set only at those pixels. After these two rasterizations of a tetrahedron, those pixels of the isosurface are set in the stencil buffer that are in danger of being rasterized multiple times. In order to avoid a second rasterization of these pixels, it is necessary to activate a stencil test in the

first rendering pass of each tetrahedron that is only passed if the stencil buffer was not set before.

For smoothly shaded isosurfaces the concepts of Westermann's algorithm for shaded isosurfaces in unstructured meshes [73] can be adapted; however, there are several crucial differences. For each triangle of the decomposed projection of a tetrahedron the steps of the algorithm presented here are:

1. Render the shaded back face triangle restricted to the projection of an isosurface patch as explained above.

2. Repeat the preceding step for the front face triangle.

3. Form the weighted sum of the two images.

The shading and lighting of triangles may employ standard OpenGL shading and lighting with the normals determined by the gradient of the scalar field at the vertices of the tetrahedron. (In a tetrahedral mesh with piecewise linear interpolation, the gradient is not defined at the vertices but it may be interpolated from the gradients of the incident tetrahedra for the purpose of shading and lighting.) Note that the vertices of a back face triangle are the same as those of the corresponding front face triangle except for one, which is the "thick" vertex in the decomposition step of the PT algorithm.

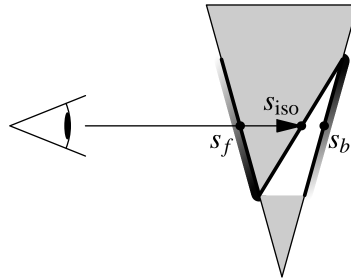


Figure 3.7 Rendering smoothly shaded isosurfaces by shading the back and front face triangle, and forming the weighted sum. Weights are symbolized by gray scales and are determined by the relative distances of the front and back faces to the isosurface given by $(s_{\text{iso}} - s_b)/(s_f - s_b)$ and $(s_{\text{iso}} - s_f)/(s_b - s_f)$, respectively.

The weights differ for each pixel as they depend on the relative distances of the isosurface to the front and back face, respectively (see Figure 3.7). For reasons which will become clear in the next paragraph, let α denote the weight of a pixel of the front triangle. According to Figure 3.7 the weight α is

$$\alpha = \frac{s_{\text{iso}} - s_b}{s_f - s_b} \quad \text{for} \quad s_f < s_{\text{iso}} < s_b \quad \text{or} \quad s_f > s_{\text{iso}} > s_b$$

with the isovalue s_{iso}; s_f and s_b are the scalar values at the front and back face triangle, see Figures 3.2 and 3.7. The weight of a corresponding pixel on the back face triangle is $1-\alpha$. While weights for all pixels were calculated in software in [73], the weighted sum can also be computed completely in hardware by appropriate texture mapping.

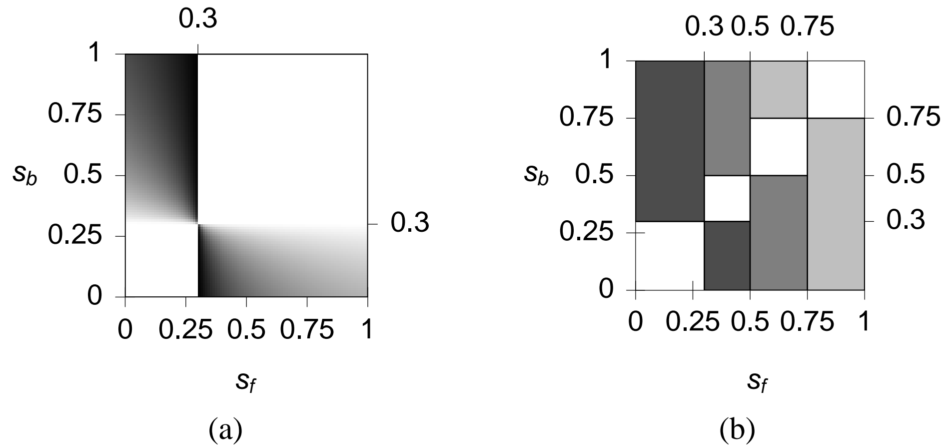
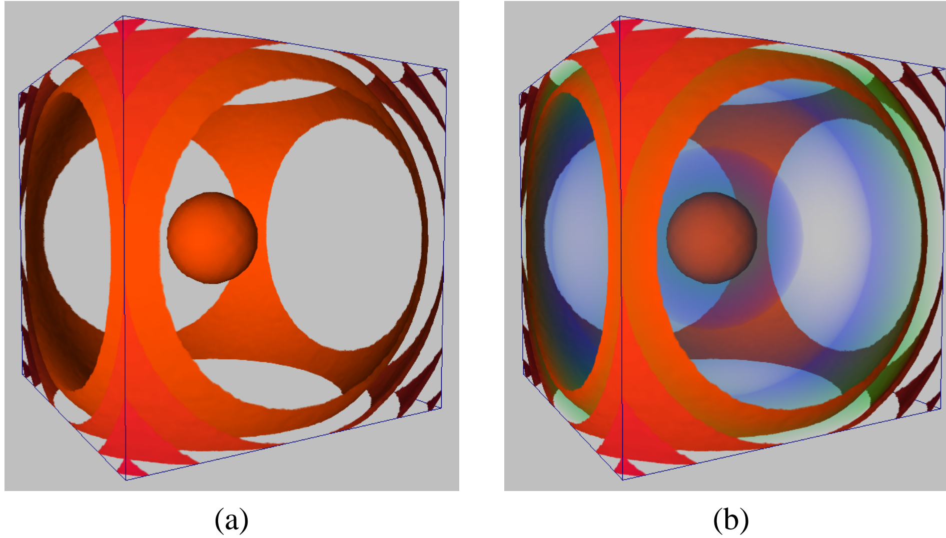


Figure 3.8 **(a)** A two-dimensional texture used for a front face triangle; black corresponds to opacity 1 (opaque), white to opacity 0 (transparent). It is a modulation of the lower texture in Figure 3.6 with the weights $\alpha = \frac{s_{\text{iso}}-s_b}{s_f-s_b}$ and $s_{\text{iso}} = 0.3$. **(b)** The correct combination of the textures from Figure 3.6 into a single texture for multiple isosurfaces.

For the back face triangle the usual two-dimensional texture for unshaded isosurfaces is used, but a modified version of this texture is employed for the front face triangle. This new texture (see Figure 3.8a for an example) is modulated with the weights α. As the original texture contains only opacity values 0 and 1, this modulated texture in fact stores the weights $\alpha = \frac{s_{\text{iso}}-s_b}{s_f-s_b}$ for the front face triangle. (Remember that s_f and s_b are the texture coordinates and that the texture already depends on s_{iso}.) Thus, the weights α in fact specify opacities. Applying this texture when rendering the front face triangle and blending it appropriately onto the opaque back face triangle generates, therefore, the correct weighted sum of both triangles. Thus, the algorithm for smoothly shaded isosurfaces can be reformulated in two passes for each tetrahedron:

1. Render the shaded back face triangle restricted to the projection of an isosurface patch.

2. Blend the shaded front face triangle modulated with a texture containing the correct weights onto the back face triangle.



(a) (b)

Figure 3.9 **(a)** Several isosurfaces extracted from the data set shown in Figures 3.3a and 3.3b. **(b)** Smooth combination of (a) with Figure 3.3b. *(See also Figure C.3 on page 146 in the color plate section.)*

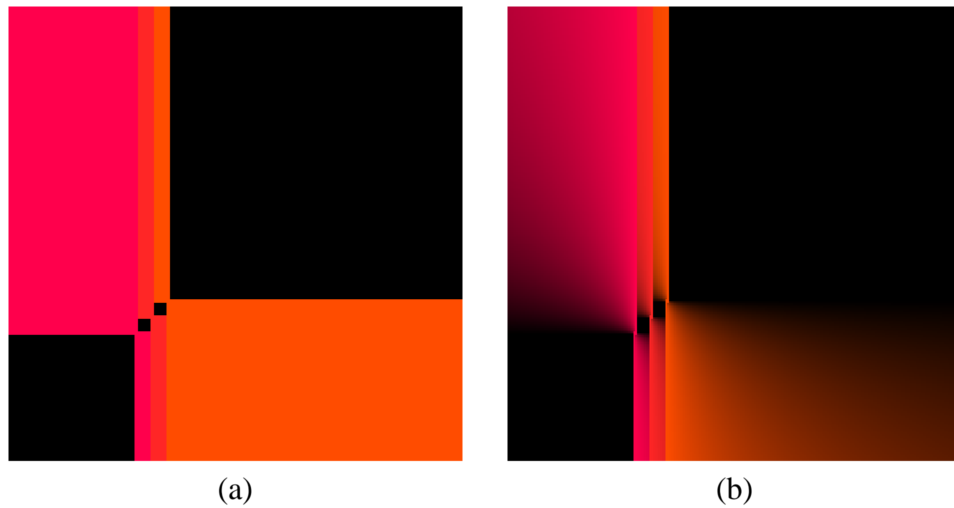
For examples of smoothly shaded isosurfaces rendered with this algorithm see Figure 3.9a.

Special care has to be taken with vertices from the decomposition of projected tetrahedra because they can result in artifacts similar to those induced by hanging nodes. In order to avoid these, the color of a vertex inserted between two vertices of the mesh has to be equal to the color generated by the graphics hardware interpolating between these vertices.

Isosurfaces rendered by the proposed two-dimensional texture mapping may be colored by setting the vertex colors to white and modulating them with colored RGBA textures. Note that the two faces of an isosurface can be colored independently by choosing different colors for texels with $s_f > s_b$ and $s_f < s_b$ respectively.

When rendering multiple isosurfaces, the two-dimensional textures of the individual isosurfaces have to be combined. An example of a combined texture is sketched in Figure 3.8b, which shows the combination of the (colored) textures from Figure 3.6. The 'visibility ordering' is easy to understand: For $s_f < s_b$ we view along the gradient of the scalar field, thus isosurfaces for smaller isovalues occlude those for greater isovalues, and vice versa for $s_f > s_b$. A more realistic example of textures for multiple, smoothly shaded isosurfaces is depicted in Figure 3.10.

Assuming that all cells are rendered, the number of isosurfaces does not affect the rendering time. In fact, the presented method shares this feature with Westermann's algorithm for multiple isosurfaces [74].



(a) (b)

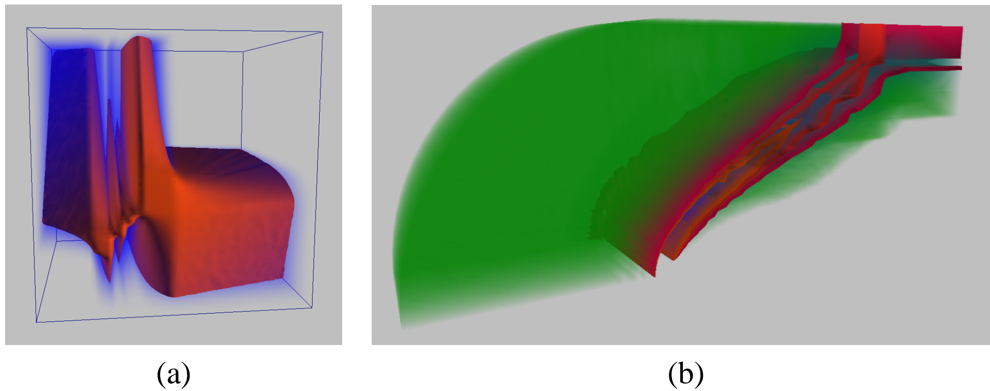
Figure 3.10 These two-dimensional textures of dimensions 256×256 were used to render the Bluntfin data set depicted in Figure 3.11b. The horizontal (texture) coordinate is s_f, the vertical coordinate is s_b. Back face triangles were textured with the image in **(a)**, while the texture in **(b)** was employed for the front face triangles. *(See also Figure C.4 on page 147 in the color plate section.)*

Mixing Isosurfaces with Semi-Transparent Volumes

It was claimed that rendering mixtures of opaque polygons and volumetric data is straightforward, e.g., in [36]. This claim, however, does not apply to cell projecting approaches including the PT algorithm since special attention has to be paid to partially occluded cells.

In [82], Williams et al. suggest slicing each cell at user-specified isovalues. The time complexity of this method, however, depends linearly on the number of isosurfaces. As the time complexity of the algorithm discussed above does not depend on the number of isosurfaces, two alternative methods of mixing isosurfaces and semi-transparent tetrahedra are discussed next, which are more appropriate in this context.

The algorithm for smoothly shaded isosurfaces allows us to smoothly include semi-transparent tetrahedra by rendering them after the corresponding back face triangle and before the front face triangle. This order ensures that the semi-transparent volume is completely occluded where the front face triangle is opaque, i.e., where the isosurface is in front of the volume at the front face, and that the volume is not affected where the front face is transparent, i.e., where the isosurface is behind the volume at the back face. Figure 3.7 illustrates this correlation: The relative thickness of the occluded part of the tetrahedron (white) corresponds to the weight of the front face (left gray scale).



(a) (b)

Figure 3.11 **(a)** Visualization of the opacity of the three-dimensional texture that corresponds to the two-dimensional texture in Figure 3.5b. The additional dimension parameterizes the length of the viewing ray within a tetrahedral cell. The isosurface represents an opacity value of 0.25. **(b)** Visualization of the Bluntfin data set with three isosurfaces mixed with projected tetrahedra. *(See also Figure C.5 on page 147 in the color plate section.)*

Examples employing this method are given in Figure 3.9b, which mixes the isosurfaces of Figure 3.9a with the projected tetrahedra of Figure 3.3b, and in Figure 3.11b, which visualizes the NASA Bluntfin data set. The isosurfaces in the latter figure were rendered with the textures shown in Figure 3.10, while the semi-transparent volume was rendered with the texture in Figure 3.5a.

Although this approach avoids discontinuities, it is not completely accurate with respect to correct ray integration. Therefore, it might be necessary to employ a more rigorous method. For opaque isosurfaces the ray integration in Equation (3.2), respectively Equation (3.1) if three-dimensional texture mapping is employed, has to be stopped as soon as one of the isovalues is reached, i.e., for $s_l(t) = s_{\mathrm{iso}}$ (see Figure 3.7). By rendering the isosurfaces for each triangle first (either in one pass for flat-shaded isosurfaces or two passes for smoothly shaded isosurfaces), followed by the semi-transparent volume with the modified two-dimensional or three-dimensional texture, it is possible to generate an accurate image.

An example of a two-dimensional texture generated this way is given in Figure 3.5b, which corresponds to the texture in Figure 3.5a. The isosurfaces manifest themselves in transparent vertical stripes which correspond to a scalar value s_f on the front face of a tetrahedron slightly greater than one of the isovalues. In Figure 3.11a this technique is used to visualize the opacity of the corresponding three-dimensional texture.

Both methods for mixing isosurfaces and semi-transparent volumes can be generalized to partially transparent isosurfaces.

Performance Comparison

Provided that hardware-accelerated texture mapping is available, pre-integrated cell projection is essentially as fast as existing implementations of the PT algorithm. It should be emphasized that the rendering times for the discussed methods are not affected by the form of the transfer functions.

It is difficult to compare these extensions of the PT algorithm with "non-PT" algorithms for direct volume rendering because another time critical step of the PT algorithm is the sorting of the tetrahedral cells, which is not affected by the extensions presented in this section. The algorithms for the rendering of isosurfaces depend on the correct sorting and decomposition of the tetrahedral cells while most of the previously published algorithms for the extraction and rendering of isosurfaces do not require any sorting or decomposition of tetrahedra. On the other hand, the rendering times for pre-integrated cell projection are independent of the number of isosurfaces (see Table 3.1).

Moreover, the presented rendering algorithms greatly benefit from a combination with projected volume cells because the sorting and decomposition of tetrahedra can be reused in this scenario. As the rendering includes the "extraction" and "triangulation" of the isosurfaces, the rendering time (without sorting and decomposition of tetrahedra) should be compared to the sum of the extraction, triangulation, and rendering times of other algorithms. Additional efforts required by other algorithms for partially transparent isosurfaces and mixing with volume cells should also be considered in a fair comparison.

The rendering times in Table 3.1 were obtained on an Octane MXE with an MIPS R10K 250 MHz CPU. The isosurfaces were extracted from the NASA Bluntfin data set, which was converted into 187,395 tetrahedra. An image with

Table 3.1 Rendering times (including "extraction" and "triangulation") for isosurfaces from the NASA Bluntfin data set. The number of cells refers to the number of tetrahedra intersected by at least one isosurface. Timings for the sorting and decomposition of tetrahedra are not included as these steps are already part of the original PT algorithm.

		rendering times (in seconds)	
number of isosurfaces	number of cells	flat-shaded	smoothly shaded
1	14,729	0.09	0.22
2	25,361	0.20	0.41
10	25,361	0.20	0.41

three isosurfaces is depicted in Figure 3.11b. Obviously, the rendering times for flat-shaded isosurfaces depend on the number of intersected tetrahedra (no double-counting) instead of the number of isosurfaces. Smoothly shaded isosurfaces require about twice as much time because the back and front faces have to be rendered separately. For a single, smoothly shaded isosurface the rendering time is close to the 0.2 seconds reported by Westermann in [73].

3.2 Hardware-Assisted Resampling

As mentioned above, graphics hardware usually does not support the rendering of tetrahedral meshes. In contrast, volume rendering of uniform meshes is well supported by means of three-dimensional texture mapping; thus, non-uniform meshes, in particular tetrahedral meshes, are often resampled to uniform meshes in order to visualize them. As this resampling requires a large number of interpolations, several hardware-assisted algorithms have been suggested, e.g., [68] or [71]. As the algorithm published by Westermann in [71] is based on the rendering technique for smoothly shaded isosurfaces described in Section 3.1, it is briefly sketched in this section.

For the purpose of resampling a tetrahedral mesh to a uniform mesh, the scalar field defined by the tetrahedral mesh has to be interpolated on a stack of slices corresponding to the uniform mesh. Most hardware-assisted approaches to this problem compute the values for each slice separately by assembling the contributions of all intersected tetrahedra in the frame buffer. Note that this setting corresponds to slices that are orthogonal to the viewing direction. The problem of determining the set of intersected tetrahedra will not be addressed here; thus, the remaining problem is to interpolate the scalar values on a patch of a slice within an intersected tetrahedron.

As indicated by Figure 3.12, a slice that is orthogonal to the viewing direction is geometrically equivalent to an isosurface of depth in an orthogonal projection. Thus, the projection of a patch of a slice within a tetrahedron can be rendered by the same technique that was employed to render patches of flat-shaded isosurfaces in Section 3.1 if the scalar data at vertices are replaced by the depths of these vertices and the isovalue (s_{iso} in Figure 3.7) is identified with the depth of the slice d_{slice} (see Figure 3.7). Analogously, the scalar values s_f on a front face and s_b on a back face are identified with the depths d_f at a front face and d_b at a back face. Thereby, uniformly colored patches of the slice can be efficiently rendered. In order to color these patches with the interpolated scalar data, the rendering technique for smoothly shaded isosurfaces may be employed as described next.

The problem of rendering smoothly shaded isosurfaces in Section 3.1 is to linearly interpolate between two different shadings, i.e., the shading on the front

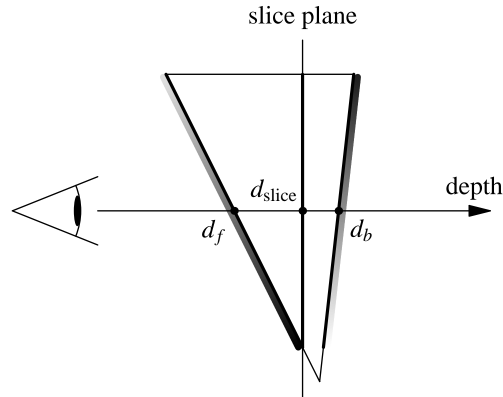


Figure 3.12 A slice orthogonal to the viewing direction through a tetrahedron.

face of a projected tetrahedron and on its back face. Similarly, the scalar value at a point within a tetrahedron can be obtained by a linear interpolation between the scalar value on the front face and the scalar value on the back face. This idea is discussed in more detail in Section 2.7.3 and depicted in Figure 2.11c. While the last linear interpolation along a line may be performed with the help of hardware-accelerated blending, the linear interpolations of the scalar data on the front and back faces may be accomplished by setting the colors of all vertices to their scalar values and rendering the faces without lighting.

Thus, a patch of a slice within a tetrahedron that is colored with the interpolated scalar data can be rendered by these three steps:

1. Render the back face triangle restricted to the projection of a slice patch as explained above.
2. Repeat the preceding step for the front face triangle.
3. Form the weighted sum of the two images.

According to Figure 3.12 (and in analogy with the discussion in Section 3.1), the weight of the scalar value at the front face is $\alpha = (d_b - d_{\text{slice}})/(d_b - d_f)$ and the weight for the back face is $1 - \alpha = (d_{\text{slice}} - d_f)/(d_b - d_f)$. Note that the depth may be transformed by any linear function $f(d) = c_0 + c_1 d$ without affecting α because the constants c_0 cancel due to the differences and the constants c_1 cancel due to the division. In particular, all depths may be mapped to the range 0 to 1, which is appropriate for texture coordinates, by applying the function

$$f(d) = \frac{d - d_{\text{slice}}}{2d_{\max}} + \frac{1}{2},$$

where $d_{\max}$ is the maximum thickness of the tetrahedra.

As discussed in Section 3.1, the algorithm may be implemented in two rendering passes per tetrahedron. In [71], Westermann describes a completely elaborated implementation of these ideas and gives further details about employing programmable pixel shading to implement them in one rendering pass.

3.3 Hardware-Assisted Ray Casting

There are, of course, more approaches to hardware-assisted volume rendering of non-uniform meshes than cell projection and resampling. For example, Westermann and Ertl suggested a hardware-assisted technique for ray casting in tetrahedral meshes in [72]. In this section, a different hardware-assisted implementation of the ray casting algorithm is sketched. It is closer to ray casting in unstructured meshes as proposed by Garrity in [22] and employs programmable pixel shading to perform basically all computations in graphics hardware. Thereby it avoids any significant data transfer between the graphics subsystem and the main memory. This approach was not implemented yet, and it is not likely to perform as well as the previously described methods on today's graphics hardware. However, it has the potential of overcoming bandwith limitations on future graphics hardware. Another disadvantage of this approach is the restriction to convex meshes. However, a feasible extension for non-convex meshes is discussed in Section 4.1. The algorithm was first described in a very similar form in [33]. A related approach to ray tracing was published independently by Purcell et al. in [53].

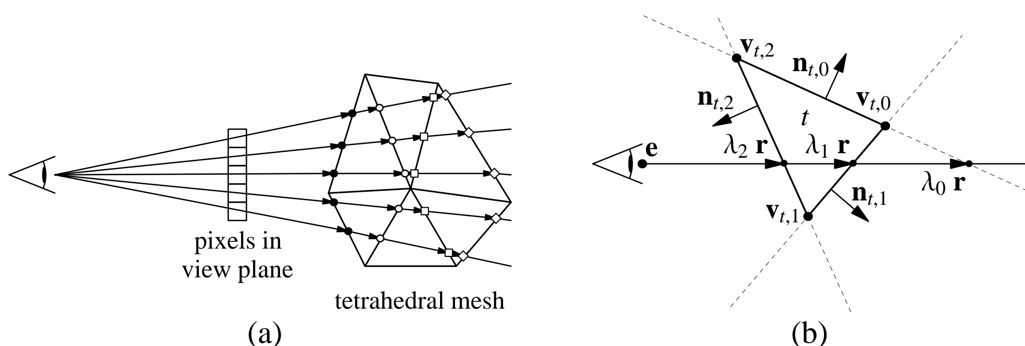


Figure 3.13 Hardware-assisted ray casting: **(a)** For each pixel one viewing ray is traced, which stops at all intersected cell boundaries. The initial intersections are marked with dots (•), further intersections with circles (○), squares (□), and diamonds (◇). **(b)** Intersections of a viewing ray with the boundary of a cell.

The basic principle of a ray casting implementation based on programmable pixel shading is to trace all viewing rays from front to back at the same time; see Figure 3.13a. Each of these viewing rays corresponds to one pixel of the frame

buffer. In an initialization step, the first intersection of each ray with the mesh is computed. After this, the rays are propagated one cell in each of the following passes. Note that one of these passes might require several actual rendering passes because of limitations of the pixel shading hardware. Several two-dimensional RGBA textures of the dimensions of the frame buffer are necessary to store the intermediate information on intersections of rays with cells of the mesh. Each texel of these textures corresponds to exactly one pixel of the frame buffer and, therefore, to exactly one of the viewing rays. Thus, the algorithm will work as follows:

1. Clear the frame buffer and initialize the textures that contain the information on intersections. The initial values are determined by the first intersection of the viewing ray associated with each texel and the cells of the mesh.

2. Update each of the mentioned textures by rendering one screen-filling rectangle into the texture memory. In the rasterization of each texel, compute the next intersection point of the ray corresponding to the texel with the boundary of a cell. Duplicate the previous intersection point if there are no further intersections.

3. Render another screen-filling rectangle into the frame buffer. This time, compute the volume rendering integral between the previous intersection point and the point computed in step 2 and blend the resulting color into the frame buffer.

4. Continue with step 2 unless a specified time limit (or a maximum number of iterations) has been reached.

Step 1 may be implemented using a rasterization of the visible boundary faces of the mesh. As there are usually far less boundary faces than cells in a mesh, this step is not time critical and may also be performed in software.

The computation of the volume rendering integral in step 3 may be performed as discussed for pre-integrated classification in Section 2.5. However, the blending has to be adapted from back-to-front to front-to-back blending. In the notation of Chapter 2 the accumulated associated color $\tilde{C}'_i$ and opacity α'_i after i passes with front-to-back blending is given by

$$\tilde{C}'_i = \tilde{C}'_{i-1} + (1 - \alpha'_{i-1})\tilde{C}_i \qquad \text{and} \qquad \alpha'_i = \alpha'_{i-1} + (1 - \alpha'_{i-1})\alpha_i,$$

where $\tilde{C}_i$ and α_i denote the associated color and the opacity of the segment of the volume rendering integral in the i-th pass.

The stop condition in step 4 guarantees an almost constant frame rate and is easy to evaluate; however, the algorithm might not render the whole mesh. In

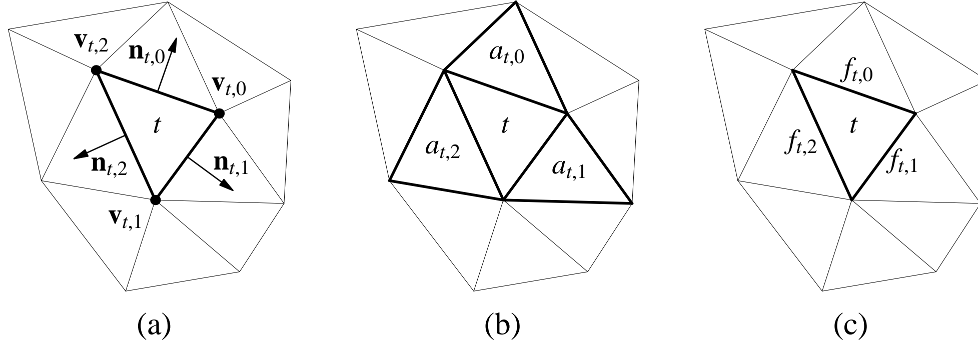


Figure 3.14 Nomenclature in this section: **(a)** The vertex $\mathbf{v}_{t,i}$ is in and the face normal $\mathbf{n}_{t,i}$ is perpendicular to the i-th face of cell t. For tetrahedral cells i is 0, 1, 2, or 3. **(b)** The neighbor $a_{t,i}$ of cell t shares the i-th face. **(c)** Face indices $f_{t,i}$ of t's neighbors: The i-th face of t corresponds to the $f_{t,i}$-th face of t's neighbor $a_{t,i}$.

order to guarantee the rendering of the complete mesh, one has to test whether any new intersection points have been computed in step 2.

In order to describe the computations of step 2, a few notations have to be introduced; see also Figure 3.14. Tetrahedral cells of a mesh consisting of n cells will be identified by an integer index from 0 to $n-1$; often this index will be called t. Each tetrahedron t has four faces, the normal vectors of which are denoted by $\mathbf{n}_{t,i}$, where i specifies the face and is 0, 1, 2, or 3. Note that normal vectors will always point to the outside of their cell. Each tetrahedron t also defines four vertices $\mathbf{v}_{t,i}$; see Figure 3.14a. In contrast to the standard convention, vertex $\mathbf{v}_{t,i}$ is part of the i-th face. As indicated in Figure 3.14b, the neighbor of a tetrahedron t that shares the i-th face is denoted by $a_{t,i}$. The index of the face of $a_{t,i}$ that corresponds to the i-th face of t is denoted by $f_{t,i}$; see Figure 3.14c.

As explained in Chapter 2, the scalar field value $s(\mathbf{x})$ at a point $\mathbf{x}$ usually is linearly interpolated from the scalar values at the vertices of the mesh; thus, $s(\mathbf{x})$ within one tetrahedral cell t is a linear function and can be computed as

$$s_t(\mathbf{x}) = \mathbf{g}_t \cdot (\mathbf{x} - \mathbf{x}_0) + s_t(\mathbf{x}_0) = \mathbf{g}_t \cdot \mathbf{x} + (-\mathbf{g}_t \cdot \mathbf{x}_0 + s_t(\mathbf{x}_0)),$$

where $\mathbf{g}_t$ is the gradient of $s_t(\mathbf{x})$ and $\mathbf{x}_0$ is any point, e.g., one of the vertices. Therefore, $s_t(\mathbf{x})$ may be specified by the vector $\mathbf{g}_t$ and the scalar $\hat{g}_t = -\mathbf{g}_t \cdot \mathbf{x}_0 + s_t(\mathbf{x}_0)$.

As indicated in Figure 3.13b, $\mathbf{e}$ denotes the eye point and the direction of a viewing ray is specified by a normalized vector $\mathbf{r}$. Any point $\mathbf{x}$ on a viewing ray will be identified by the factor λ such that $\mathbf{x} = \mathbf{e} + \lambda\mathbf{r}$.

All data have to be stored in textures in order to access them in a pixel shading program. Table 3.2 summarizes one possibility to organize these textures. Note

Table 3.2 Summary of the textures described in the main text.

	texture coordinates			texture data			
data in texture	u	v	w	r	g	b	α
vertices	t		i	$\mathbf{v}_{t,i}$			—
face normals	t		i	$\mathbf{n}_{t,i}$			—
neighbor data	t		i	$a_{t,i}$		$f_{t,i}$	—
$s_t(\mathbf{x})$	t		—	$\mathbf{g}_t$			$\hat{g}_t$
ray directions	raster position		—	$\mathbf{r}$ for this ray			—
entered cells	raster position		—	t		i	—
intersection points	raster position		—	—	—	λ	$s(\mathbf{e}+\lambda\mathbf{r})$

that the cell indices are encoded in two texture coordinates and two color components as their range might exceed the range of a single texture coordinate or color component. Apart from the five constant textures for vertices, face normals, neighbor data, linear functions, and normalized directions of the viewing rays associated with the pixels of the frame buffer, there are two textures specifying the intersections of viewing rays with the boundaries of cells: One texture specifies the most recently entered cell for each viewing ray indexed by the raster position of the corresponding pixel, and similarly another texture specifies the position of and the scalar value at the last intersection point. This information is necessary in order to compute the segment of the volume rendering integral between two successive intersection points in step 3 of the algorithm. In fact, the latter texture has to be duplicated as the algorithm needs to access the data of the new and the previous intersection point and because it is usually impossible to read and write to a texture at the same time.

In step 2 of the algorithm the next intersection of each viewing ray has to be determined; thus, the two textures specifying the entered cells and the intersection points have to be recomputed. The next intersection point is the exit point of the cell entered at the previous intersection point. This point may be determined by computing all intersection points of the ray with the four faces of the entered cell and choosing the intersection point that is closest to the eye point but not on a face that is visible from the eye point.

With the eye point $\mathbf{e}$ and the normalized direction $\mathbf{r}$ of the viewing ray (see Figure 3.13b), the four intersection points with the faces of cell t are $\mathbf{e}+\lambda_i\mathbf{r}$ with $0 \le i < 4$ and

$$\lambda_i = \frac{(\mathbf{v}_{t,i}-\mathbf{e})\cdot\mathbf{n}_{t,i}}{\mathbf{r}\cdot\mathbf{n}_{t,i}}.$$

This equation is easily implemented with pixel shading operations as three-dimensional vector operations are usually well supported. A face is visible if the denominator in the previous equation is negative; thus, this test comes almost for free. If λ_i is set to an appropriately large number for all visible faces, $\min\{\lambda_i | 0 \leq i < 4\}$ will identify the exit point. Determining the minimum of four numbers is usually less well supported by pixel shading hardware, but it may be implemented with the help of a sequence of conditional per-pixel operations. (An alternative solution based on texture mapping is discussed in [69].)

Once the minimum λ_i and its face i are identified, the intersection point $\mathbf{x}$ may be computed as $\mathbf{x} = \mathbf{e} + \lambda_i \mathbf{r}$ and the value of the scalar field at that point is $s(\mathbf{x}) = \mathbf{g}_t \cdot \mathbf{x} + \hat{g}_t$. For each intersection point, λ_i and $s(\mathbf{x})$ is stored as this information is required in step 3 of the algorithm to compute the volume rendering integral between two successive intersection points.

The cell entered at $\mathbf{x}$ is given by $a_{t,i}$ and the new face index in this cell is $f_{t,i}$. The values of $a_{t,i}$ and $f_{t,i}$ will also indicate whether $\mathbf{x}$ is on the boundary of the mesh as there is no neighbor in this case. Once a viewing ray leaves the mesh, the intersection point should be kept constant because constant intersection points will result in segments of length zero, which will not contribute to the volume rendering integral. However, this procedure is only useful for convex meshes as there may be re-entries if the mesh is non-convex. This is one of the problems associated with non-convex meshes that is addressed in Section 4.1.

As each pass of the algorithm requires the update of two textures and one update of the frame buffer, there are at least three rendering passes per iteration of the algorithm unless a pixel shading program may have multiple outputs (see for example [29]). Further rendering passes and additional textures for intermediate results may be necessary because of restrictions on the maximum number of operations of a pixel shading program. Another disadvantage of this method is the need to rasterize texels and pixels although their viewing rays have already left the mesh. However, it might be possible to use early depth tests, stencil tests, or pixel "kill" operations (see for example [29]) in order to minimize the associated performance penalty.

The main advantage of this approach is that the complete mesh data is stored in the texture memory. Therefore, there is almost no data transfer between the main memory and the graphics subsystem provided that there is enough texture memory. Although this algorithm has not been implemented yet, there is a first spin-off product, which is the hardware-based, view-independent cell projection published by Weiler et al. in [69].

Miss Peters: *Mary Sue.*
Jennifer: *Yeah. What's outside of Pleasantville?*
Miss Peters: *I don't understand ...*
Jennifer: *Outside of Pleasantville ...*
What's at the end of Main Street?
Miss Peters: *Oh, Mary Sue. You should know the answer to that.*
The end of Main Street is just the beginning again.

Dialog from the movie *Pleasantville*

Chapter 4

Non-Convex and Cyclic Meshes

If the boundary of an unstructured mesh is non-convex, additional algorithmic efforts are necessary compared to the case of a convex boundary. This chapter discusses a general approach to the problems caused by non-convex meshes (Section 4.1) and applies it to edge collapses in non-convex meshes (Section 4.2) and the visibility sorting of cells in non-convex meshes (Section 4.3).

Another possibly unpleasant feature of the geometry of general unstructured meshes are visibility cycles as defined in Section 4.3. Since a general solution of this problem (e.g., by splitting enough cells in order to break all cycles) is hard to achieve, only specific solutions to the two most important problems are discussed here, which are cell sorting (Section 4.3) and cell projection (Section 4.4).

Similarly to Chapter 3, the discussion in this chapter is restricted to tetrahedral meshes. Most of the results of this chapter have been published in different forms in [31] and [35].

4.1 Convexification of Non-Convex Meshes

In [80], Williams proposed to convert non-convex meshes to convex meshes by triangulating (i.e., tetrahedralizing) all voids and cavities and marking the cells generated by this triangulation as *imaginary*. (*Virtual* is today's more fashionable word for the same idea.) This conversion is usually a preprocessing step, which is called a *convexification* of a non-convex mesh in this work.

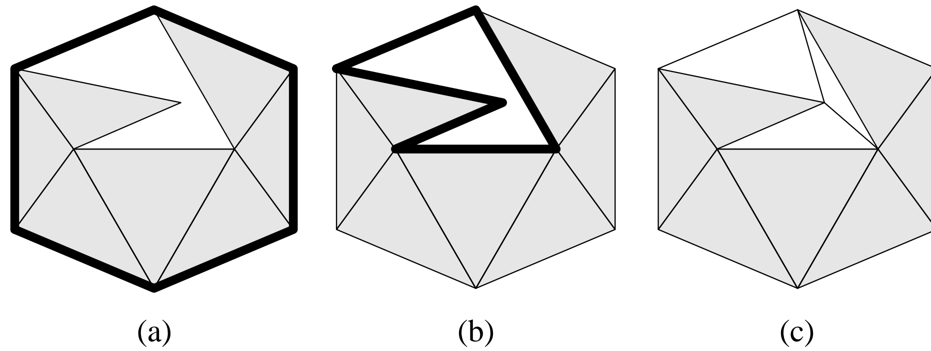


Figure 4.1 A step-by-step convexification of a triangular, two-dimensional mesh. **(a)** The convex hull (*thick line*), **(b)** the non-convex polygon (*thick line*) between the convex hull and the boundary of the mesh, and **(c)** the mesh together with imaginary cells (*white*) generated by the triangulation of the non-convex polygon.

Figures 4.1 and 4.2 summarize the basic steps of the convexification of a triangular and a tetrahedral mesh, respectively. Firstly, the convex hull of the mesh is computed; then all voids and cavities are identified and triangulated; finally, the new imaginary cells are attached to the existing mesh. In two dimensions, these steps are straightforward and the number of imaginary triangles is linear in the number of vertices on the boundary.

However, the problem of tetrahedralizing the space between the convex hull and the mesh's boundary is considerably more difficult in three dimensions. More-

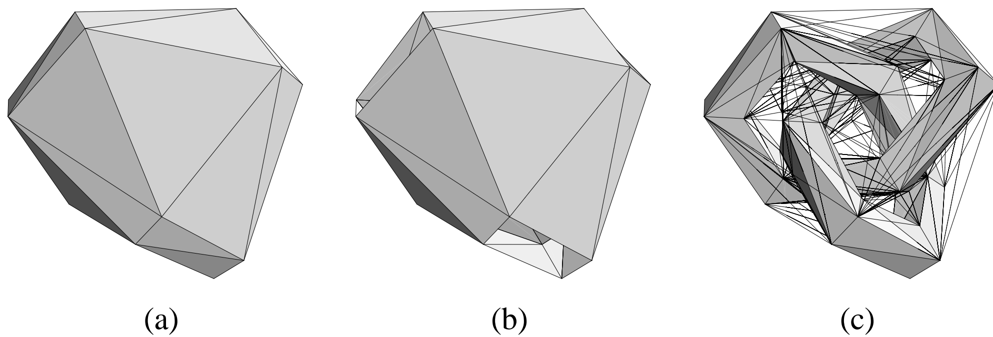
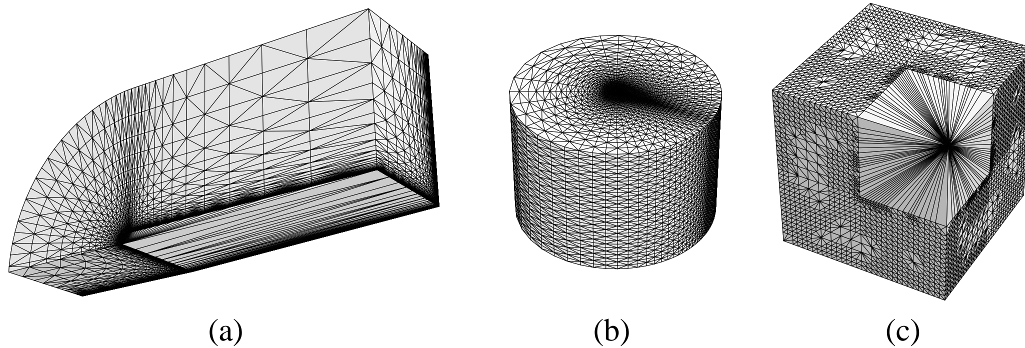


Figure 4.2 (Intermediate) results of the convexification of a tetrahedral mesh (see also Figure 4.7a): **(a)** the convex hull of the mesh, **(b)** the non-convex polyhedron between the convex hull and the boundary of the mesh, which has to be tetrahedralized, and **(c)** the original mesh (a trefoil knot) together with the edges of the imaginary tetrahedra generated by the tetrahedralization.



(a) (b) (c)

Figure 4.3 **(a)** Boundary of the convexified Bluntfin data set. The tetrahedralized cavity is at the bottom of the original mesh. **(b)** Boundary of the Tapered Cylinder data set. The thin cavity is in the center of the cylinder. **(c)** Boundary of the convexified HeatSink data set. The cubic cavity in front has been tetrahedralized with 2,484 imaginary tetrahedra.

over, the number of imaginary tetrahedra is quadratic in the number of vertices of the mesh's boundary in the worst case. Although a general algorithm for this problem exists [9], its implementation is beyond the scope of this work. Thus, only a strongly simplified variant of this algorithm and very specific solutions are employed in this work to achieve the convexifications depicted in Figures 4.2 and 4.3. Figure 4.3a shows the boundary of the convexified Bluntfin data set. This convexification added 3,534 imaginary tetrahedra to the 187,318 tetrahedra of the tetrahedralized Bluntfin data set (without degenerate tetrahedra). Analogously, Figure 4.3b depicts the Tapered Cylinder data set with 624,960 actual and 5,766 imaginary tetrahedra. The HeatSink data set in Figure 4.3c consists of 121,668 actual and 2,484 imaginary tetrahedra.

The convexification of a non-convex mesh allows us to apply slightly modified algorithms for convex meshes to the convexified versions of non-convex meshes. Thus, convexification may be called a *meta-algorithm* because it acts as a tool for the generation of algorithms; for example, Williams proposed in [80] to employ the convexification step for cell sorting of non-convex meshes, as will be discussed in Section 4.3. Moreover, Williams discussed the spatial point location problem, which will not be described in this work. Instead, Section 4.2.2 shows how to overcome problems of edge collapses introduced by non-convexities (including non-trivial topologies of the boundary) with the help of the convexification step. Another algorithm that could benefit from this preprocessing step is the ray casting algorithm discussed in Section 3.3, which is restricted to convex meshes since it does not compute re-entries of rays that have exited the mesh. With a small extension for imaginary cells, the algorithm could be applied to any convexified non-convex mesh.

4.2 Edge Collapses in Non-Convex Meshes

As proposed in [35], edge collapses in non-convex meshes may be performed by convexifying a non-convex mesh and performing the edge collapses in the convexified mesh. In order to set the stage for the discussion of these edge collapses in Section 4.2.2, the well-known case of edge collapses in convex meshes is briefly summarized first.

4.2.1 Edge Collapses in Convex Meshes

In the following, edge collapses in fair tetrahedral meshes are discussed, i.e., each face of a tetrahedral cell is either part of the boundary of the mesh or shared by (at most) two cells. Note that the term *edge collapse* in computer graphics corresponds to the term *edge contraction* in computational geometry. In this work, the former is used in order to be consistent with the majority of the references.

Intersections of cells are not allowed; in fact, avoiding them is the primary concern. As edge collapses in triangular meshes in two dimensions are very similar to the three-dimensional case of tetrahedral meshes, most concepts will be illustrated with triangular meshes.

It is useful for the discussion in Section 4.2.2 to define some particular terms. A tetrahedron is called a *vertex neighbor* of a vertex if the vertex is shared by the tetrahedron. A tetrahedron is an *edge neighbor* of an edge if the edge is shared, and a *vertex neighbor* of an edge if one of the vertices of the edge is shared. Finally, a tetrahedron is a *face neighbor* of another tetrahedron if the tetrahedra share one face. The definitions of *vertex* and *edge neighbors* can also be applied to triangles in triangular meshes.

An edge collapse (see Figure 4.4 for a two-dimensional example) is performed by removing all edge neighbors of the collapsing edge and by joining the two vertices of the collapsing edge in a new vertex. Figure 4.4 also indicates the inverse operation, which is called a *vertex split*.

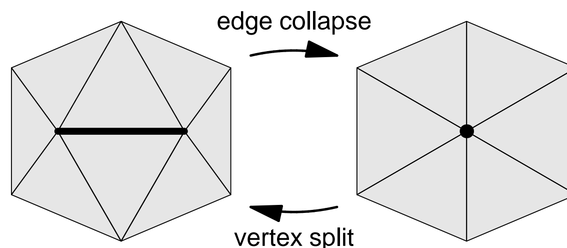


Figure 4.4 The collapse (contraction) of an edge (*thick line*) to a vertex (*dot*) and the inverse vertex split.

Edge collapses are one of the most powerful tools to simplify triangular or tetrahedral meshes. They can be employed to remove vertices or edges [64] and also to remove triangles or tetrahedra by successive edge collapses [11, 67]. Moreover, a sequence of edge collapses can be applied in order to produce hierarchical representations of triangular and tetrahedral meshes as demonstrated in many publications, for example [11, 52, 64, 67].

As illustrated in Figure 4.5, an edge collapse can cause an intersection of cells in a triangular, convex mesh. In order to avoid such self-intersections of a mesh, edge collapses are tested before they are performed [11, 64, 67].

In convex meshes, i.e., meshes the boundary of which are convex polytopes, the test for intersections is particularly simple because any intersection of cells is accompanied by an *inversion* of at least one cell, i.e., a sign flip of the signed volume of a cell. (The inverted cell is marked gray in Figure 4.5.) Therefore, it is sufficient to test all vertex neighbors of a collapsing edge for inversions in order to avoid self-intersections. This test is *local* as only vertex neighbors are involved.

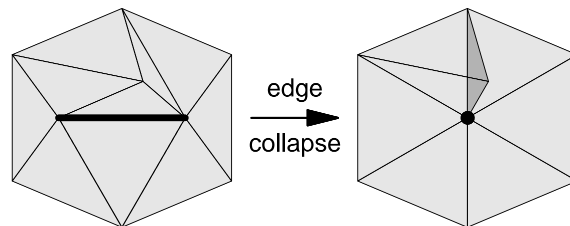


Figure 4.5 An edge collapse that causes several intersections of cells and one inversion of a cell (*dark gray*).

4.2.2 Edge Collapses in Convexified Meshes

In contrast to edge collapses in convex meshes, there is no local test for intersections of cells caused by edge collapses in non-convex meshes. More precisely spoken, if a collapsing edge in a non-convex mesh has vertex neighbors that are cells at the boundary (i.e., one of the faces of the cell is part of the boundary of the mesh), then the edge collapse can cause self-intersections of the mesh without causing an inversion of a cell as demonstrated in Figures 4.6 and 4.7.

A naive procedure to avoid such intersections is to test the vertex neighbors of the collapsing edge for intersections with all boundary cells of the mesh. As the worst-case time complexity of this *global* test depends linearly on the number of cells of the whole mesh, it is usually too expensive to be performed without additional auxiliary data structures. A more elaborated implementation of this test

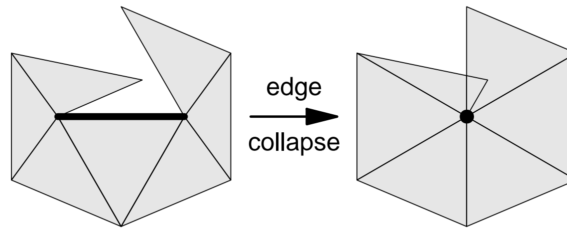


Figure 4.6 An edge collapse that causes an intersection of two cells without causing an inversion of any cell.

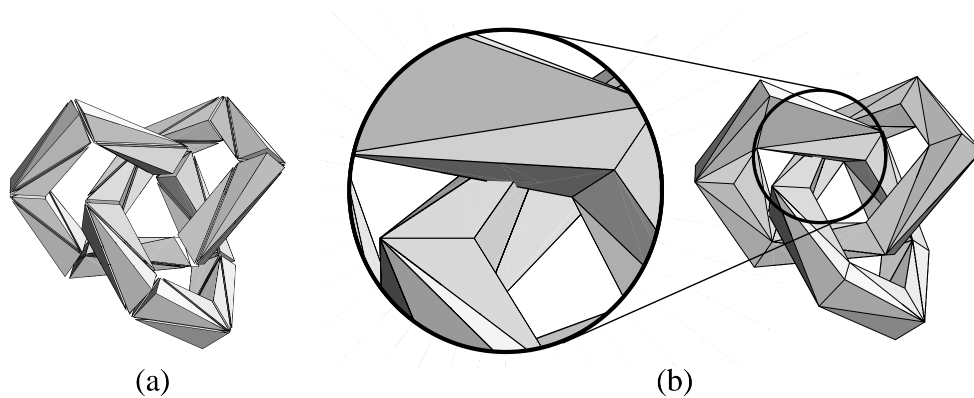


Figure 4.7 **(a)** A tetrahedral mesh with the shape of a trefoil knot. **(b)** The same mesh after one particular edge collapse.

is discussed in [11] and [64] while the system described in [67] tries to preserve the boundary of the tetrahedral mesh.

However, performance issues are not the only problem of edge collapses in non-convex meshes. Additional problems occur in disconnected meshes as edge collapses are obviously not able to join clusters of disconnected meshes in order to simplify them. More generally spoken, it is desirable to modify the topology of the mesh's boundary in a controlled way when performing edge collapses.

In order to overcome these problems of non-convex meshes, a convexified version of the same mesh (as described in Section 4.1) may be employed. The remainder of this section will present the details of this approach.

Geometric Tests

As mentioned previously, edge collapses in non-convex meshes can cause intersections of cells without causing an inversion of any cell. In convexified meshes, however, such edge collapses will always cause an inversion of at least one imaginary cell as shown in Figure 4.8 for the same edge collapse as in Figure 4.6 in a convexified version of the same triangular mesh.

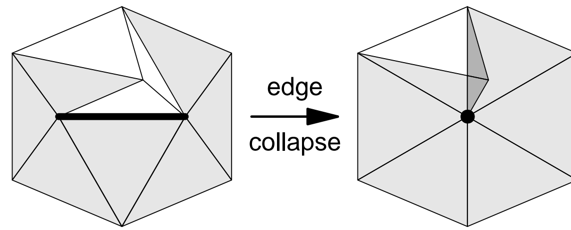
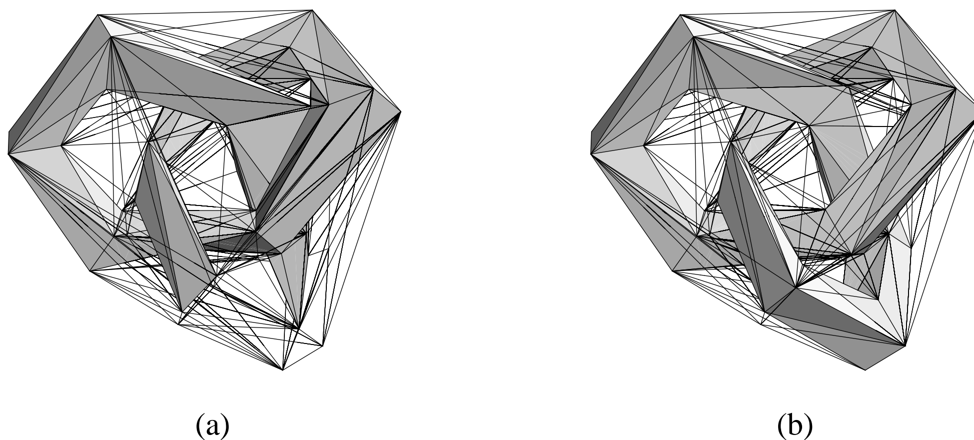


Figure 4.8 The edge collapse of Figure 4.6 in the convexified mesh of Figure 4.1 causes an inversion of an imaginary cell (*dark gray*). (Compare also with Figure 4.5.)

Therefore, convexified meshes allow us to test for self-intersections of cells by simply testing all vertex neighbors (including imaginary cells) of the collapsing edge for sign flips of the signed cell volume, which is a local, geometric test as in the case of convex meshes. Thus, the total number of new imaginary cells generated by the convexification is not relevant for the efficiency of this test.

An example in three dimensions is depicted in Figure 4.9a. The tetrahedral mesh from Figure 4.7a was simplified by a series of edge collapses. The employed geometric tests for cell inversions guarantee that there are no self-intersections and hamper further edge collapses. Note that more edge collapses are possible if the new vertex is not positioned at the center of the collapsed edge.



(a) (b)

Figure 4.9 **(a)** The result of a simplification of the convexified mesh depicted in Figure 4.2 without topological tests. **(b)** Same as (a) with additional topological tests.

Preservation of the Convex Hull

Not only are edge collapses in non-convex meshes more complicated than in convex meshes, they can also transform a convex mesh into a non-convex mesh.

An example is depicted in Figure 4.10, which also presents a suitable solution: Instead of recomputing the convex hull (a global operation if implemented naively), new imaginary cells are inserted between the new vertex and the convex hull in order to preserve the original convex hull. (The effect can also be found at the bottom of Figure 4.9a.)

This is an efficient, local operation. However, it will also insert some new edges; therefore, a simplification process might run into an endless loop by collapsing edges that are instantly reconstructed by the insertion of imaginary cells. In order to avoid this problem, edge collapses should be avoided if the following three conditions are met: All edge neighbors of the collapsing edge are imaginary, one vertex is part of the boundary of the mesh, and all vertex neighbors of this vertex are imaginary. (For an example see the edge between the new imaginary cells in the right-hand side of Figure 4.10.)

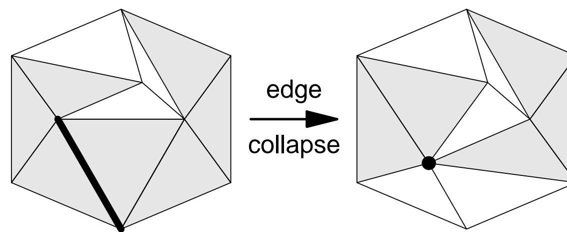


Figure 4.10 An edge collapse that could generate a non-convexity. Two new imaginary cells are inserted in order to preserve the original convex hull.

Incomplete Topology Preservation

Edge collapses in convexified meshes are considerably more powerful than edge collapses in the original meshes. For example, disconnected meshes can be joined in order to be simplified, tunnels in the boundary of a mesh can be closed, bridges between meshes can be broken, etc. In Figure 4.9a, a new connection between originally disconnected parts of the mesh may be found in the top-right corner and a disconnection of cells at the bottom.

However, not all of these features are always welcome; instead, it is more appropriate to have full control over the modifications of the topology of the mesh's boundary. Here, two very simple tests for edge collapses are presented that are sufficient to avoid topological changes of the mesh. The tests guarantee

the preservation of a certain *edge connectivity* but do not avoid all possible topological changes. Both tests involve only vertex neighbors of the collapsing edge. Before presenting these topological tests, some basic terms have to be defined:

Def.: The *type* of a cell is either imaginary or non-imaginary.

Def.: A cell T_1 is *connected* to a cell T_2 of the same type if the cells share a vertex (direct connection), or if T_1 is connected to a third cell T_3 of the same type that is connected to T_2 (indirect connection).

Def.: Two cells are *disconnected* if they are not connected.

Figure 4.11 shows an example of an edge collapse that establishes a new connection between two non-imaginary cells. In general, the collapse of an edge e between two vertices v_1 and v_2 *can* connect two previously disconnected cells if all of the following three conditions are met: All of the edge neighbors of e are of the same type t; at least one of the vertex neighbors of v_1 is not of type t; and at least one of the vertex neighbors of v_2 is not of type t. This is a necessary condition; therefore, it is sufficient to avoid edge collapses that fulfill it in order to avoid new connections between cells. The test is the same for triangular and tetrahedral meshes.

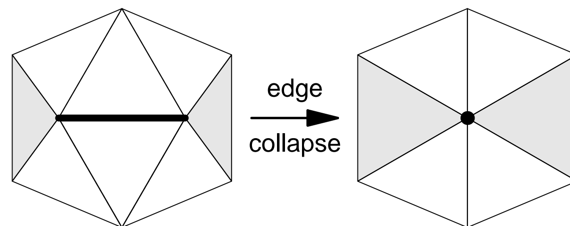


Figure 4.11 An edge collapse that connects two non-imaginary cells (*gray*).

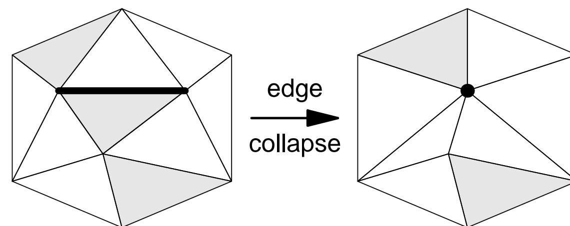


Figure 4.12 An edge collapse that disconnects two non-imaginary cells (*gray*).

Edge collapses are also able to disconnect cells; an example is depicted in Figure 4.12. In order to formulate a test for disconnecting edge collapses, one more definition is required:

Def.: An edge neighbor T of type t of an edge e is *isolated* if none of the faces of T that do not share e are shared by a face neighbor of T of type t.

This definition is needed for the statement that the collapse of an edge e *can* disconnect two previously connected cells if at least one of the edge neighbors of e is isolated. This is again a necessary condition, which works for triangular and tetrahedral meshes. (Note that the faces of a triangular cell are its edges.)

These two topological tests allow us to avoid new connections and/or disconnections simply by avoiding edge collapses that fulfill the conditions stated above. An example in three dimensions is presented in Figure 4.9b: Topological tests in a simplification of the convexified mesh of Figure 4.2 guarantee the preservation of edge connectivity; therefore, the simplification process is halted earlier than without these tests. Further simplification steps are possible with more general edge collapses.

Complete Topology Preservation

It is possible to extend the discussed concept of edge connectivity to faces of cells by defining a corresponding *face connectivity*:

Def.: A cell T_1 is *face connected* to a cell T_2 of the same type if the cells share a face (direct face connection), or if T_1 is face connected to a third cell T_3 of the same type that is face connected to T_2 (indirect face connection).

Def.: Two cells are *face disconnected* if they are not face connected.

Def.: An edge neighbor T of type t of an edge e is *actively isolated* if all of the faces of T that do not share e are shared by face neighbors of T that are not of type t.

The collapse of an edge e can face connect previously face disconnected cells if at least one of the edge neighbors of e is actively isolated.

It is necessary to test all edge neighbors of the collapsing edge for face disconnections. This is a sufficient test because face disconnections of cells that are not edge neighbors of the collapsing edge imply the existence of a face disconnection between edge neighbors of the collapsing edge.

This is again a simple, local test, which is easily implemented. However, even the combined tests for edge and face connectivity do not preserve the topology completely. Fortunately, a rigorous solution to this problem has been published by Dey et al. in [16]. In fact, the tests presented here turn out to be special cases of the *Link Conditions* for 3-complexes. (See Theorem C in [16].)

4.3 Cell Sorting for Non-Convex and Cyclic Meshes

Direct volume rendering based on cell projection as presented in Section 3.1 requires a visibility ordering of the cells of a mesh in order to composite the projections of cells correctly. A visibility ordering (or depth ordering) of a set of cells is an ordering of the cells such that a cell b precedes another cell a in the ordering if a obstructs b. (This defines a back-to-front ordering; reversing the ordering will result in a front-to-back ordering.) While the visibility ordering is trivial for certain meshes (e.g., rectilinear meshes), the ordering for unstructured or curvilinear meshes has to be computed explicitly. The computation of a visibility ordering is crucial for interactive volume visualization as it has to be performed for each viewpoint separately. Thus, several algorithms have been proposed that exploit particular properties of meshes (e.g., convexity or the Delaunay property) in order to improve the performance of the visibility sorting. An overview of some of these algorithms is given in Section 4.3.1.

Unfortunately, the price of exploiting certain properties of meshes is the restriction to the corresponding classes of meshes. The goal of the methods proposed in Sections 4.3.2 and 4.4, which were first published in [31], is to overcome one particular constraint, namely the acyclicity of tetrahedral meshes and meshes with convex polyhedral cells in general. Acyclic meshes are characterized by the absence of visibility cycles for all viewpoints. (Figures 4.17 and 4.21 show examples of visibility cycles formed by polygons and tetrahedra, respectively.)

While image-order algorithms (e.g., ray casting) are usually not affected by visibility cycles as they calculate the cells' visibility ordering for each viewing ray separately, most algorithms based on hardware-assisted cell projection cannot sort visibility cycles — simply because there is no visibility ordering of a visibility cycle. One possible approach to this problem is to split cells of a cyclic mesh appropriately in order to transform it into an acyclic mesh. However, the resulting additional cells will decelerate the visibility sorting. Moreover, numerical instabilities might still generate visibility cycles.

The solution proposed here is twofold: Firstly, the visibility sorting suggested by Williams in [80] and described in Section 4.3.1 is extended in order to compute and sort visibility cycles. This is achieved by computing visibility cycles with a standard graph algorithm and gathering all cells of each cycle in one "cell cluster" for each cycle. This allows us to find a visibility ordering of the combined set of single cells that are not part of a cycle and of cell clusters resulting from visibility cycles. This sorting algorithm is described in Section 4.3.2. While Sections 4.3.1 and 4.3.2 are restricted to convex meshes, extensions for non-convex meshes are discussed in Section 4.3.3.

Of course, sorting cyclic meshes is only one part of a solution, which is incomplete without a possibility to render cyclic cell clusters. Therefore, Section 4.4

presents an enhanced variant of an algorithm for resolving cyclic occlusions of polygons published by Snyder and Lengyel in [63], which allows us to render visibility cycles without splitting cells. However, the rendering time for each cluster is quadratic in the number of its cells.

4.3.1 Cell Sorting for Convex, Acyclic Meshes

Remarkably, the two most important approaches to the computation of visibility orderings of unstructured meshes are both based on results of Edelsbrunner [17] and published by Max et al. in [44]. The first method is to sort the cells of a tetrahedral mesh with respect to the tangential distance from the viewpoint to the circumscribing sphere of each cell. The resultant visibility orderings are exact for Delaunay meshes; however, they may be inexact if the Delaunay property does not apply. Note that the visibility ordering has to be inexact for a mesh with visibility cycles as there is no ordering of the cells of a visibility cycle. Nonetheless, variants of this ordering algorithm are often employed (see for example [12, 27, 83]), in particular because of the algorithm's high performance. Unfortunately, this algorithm appears to be inappropriate for the purpose of computing (and rendering) visibility cycles.

In [44], Max et al. also presented a second visibility sorting algorithm for convex, acyclic meshes, i.e., meshes with a convex boundary but without visibility cycles. This algorithm was published independently by Williams in [80], who named it MPVO (Meshed Polyhedra Visibility Ordering).

Williams defines three phases of the MPVO algorithm: Phase I is a preprocessing step, i.e., the computations are independent of the viewpoint. Thus, the results can be reused for different viewpoints. In particular, coefficients of the plane equations for all faces of all cells are calculated in this phase. Moreover, most of the data structures required in the following phases are constructed and initialized.

Phase II is a loop over all faces that are shared by two cells. For each face the behind relation "$<$" ($a < b$ if a is behind b, i.e., b obstructs a) between the two cells is calculated by evaluating the plane equation of the face for the given viewpoint. (This assumes a perspective projection; see [80] for the calculation of the behind relation for orthographic projections.) From these behind relations a directed graph is built, where each cell is represented by a node and each face shared by two cells is represented by an edge between two nodes. The directions of the edges of this graph correspond to the behind relation between the corresponding cells, i.e., the edge is directed from node a to b if $a < b$.

An example in two dimensions is sketched in Figure 4.13. The acyclic triangular mesh consists of eight labeled cells and is viewed from a point at the top of the figure. Figure 4.13a depicts the behind relation for each shared face

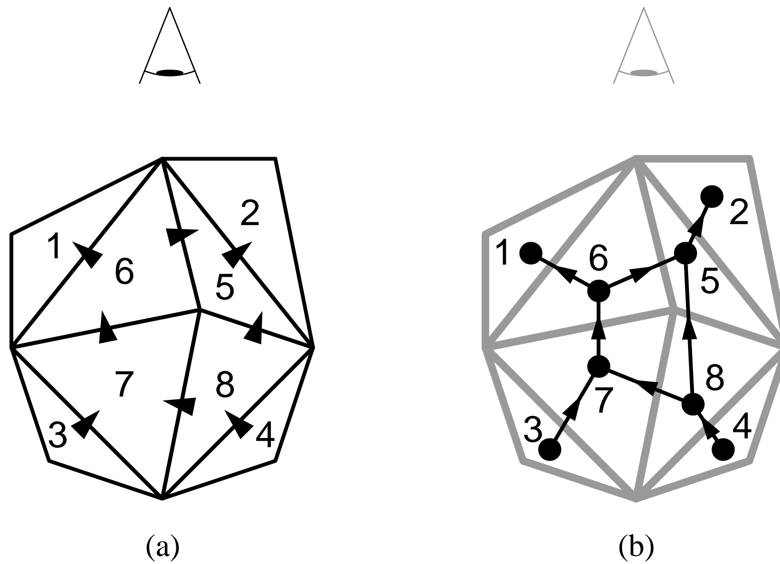


Figure 4.13 **(a)** Behind relations in an acyclic triangular mesh. **(b)** The directed acyclic graph corresponding to the behind relations in (a).

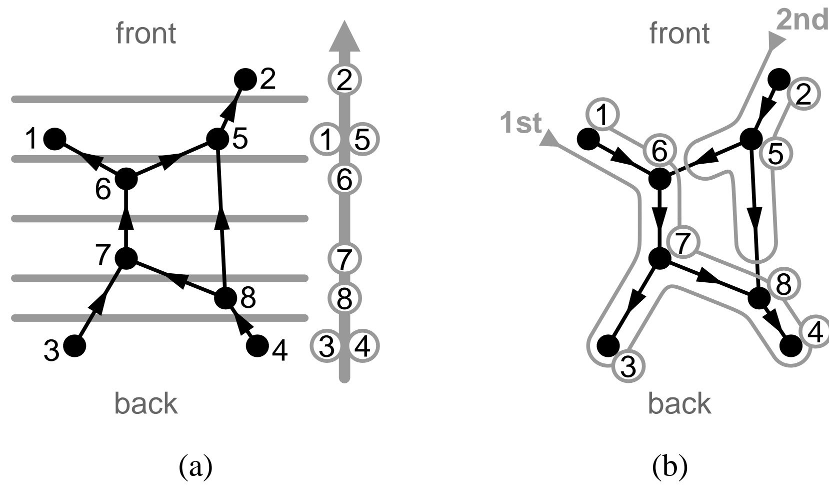
(i.e., shared edge of two triangular cells) while Figure 4.13b shows the resulting directed acyclic graph.

Phase III of the MPVO algorithm computes a visibility ordering of the mesh by topologically sorting the directed acyclic graph constructed in phase II. The topological sorting can be implemented by a breadth-first search (phase III-BFS) or a depth-first search (phase III-DFS). However, for the depth-first search the directions of all edges have to be reversed prior to the topological sorting.

The breadth-first search of the graph of Figure 4.13 is depicted in Figure 4.14a. It starts with the set of *source nodes*, i.e., nodes without incoming edges; in our example nodes 3 and 4. Note that the set of source nodes necessarily corresponds to a set of non-obstructing cells. In fact, the breadth-first search will result in a sequence of sets of non-obstructing cells. The corresponding sets of nodes are separated by gray horizontal lines in Figure 4.14a.

Figure 4.14b visualizes the depth-first search of the reversed directed acyclic graph of Figure 4.13. The search is in fact a sequence of recursive graph traversals starting at each of the source cells, which are nodes 1 and 2. (The set of source nodes of the graph with reversed edges in Figure 4.14b corresponds to the set of *sink nodes* of the graph in Figure 4.13, i.e., the nodes without outgoing edges.) The two traversals are marked by gray lines in Figure 4.14b.

An extension of the MPVO algorithm for non-convex meshes (called MPVO-NC) was also published by Williams in [80] and is described in Section 4.3.3. However, the computed visibility orderings are inexact for particular non-convex



(a) (b)

Figure 4.14 **(a)** Topological sorting of the graph of Figure 4.13b using a breadth-first search. **(b)** Same as (a) but using a depth-first search. Note that all directions have been reversed for the depth-first search.

meshes (see [80]). Several years later, Silva et al. published an extension of the MPVO algorithm for non-convex meshes called XMPVO in [62]. The performance of this algorithm was improved by the BSP-XMPVO algorithm proposed by Comba et al. in [13].

4.3.2 Cell Sorting for Convex, Cyclic Meshes

The MPVO algorithm and its descendants are limited to acyclic meshes as the topological sorting of the MPVO algorithm is not applicable to cyclic meshes. However, it is possible to compute visibility cycles by employing an algorithm for the computation of strongly connected components of a directed graph published by Tarjan in [66]. This approach is well-known for visibility cycles formed by polygons (see [20] or [63]).

Strictly speaking, there is no visibility ordering of a cyclic mesh. In particular, a phase III-DFS will detect a cycle and return an inexact visibility ordering. For example, Figure 4.15a depicts a cyclic mesh as the cells labeled 5, 6, and 7 form a visibility cycle, i.e., each of these cells (indirectly) obstructs each other. This cycle is more obvious in Figure 4.15b and corresponds to a non-atomic *strongly connected component* that includes the nodes labeled 5, 6, and 7. All other strongly connected components in this example consist of single nodes. (A strongly connected component of a directed graph is characterized by its property that all nodes of the component are mutually accessible.) Note that the algorithms discussed in this work cannot handle non-convex cells in general; however, the

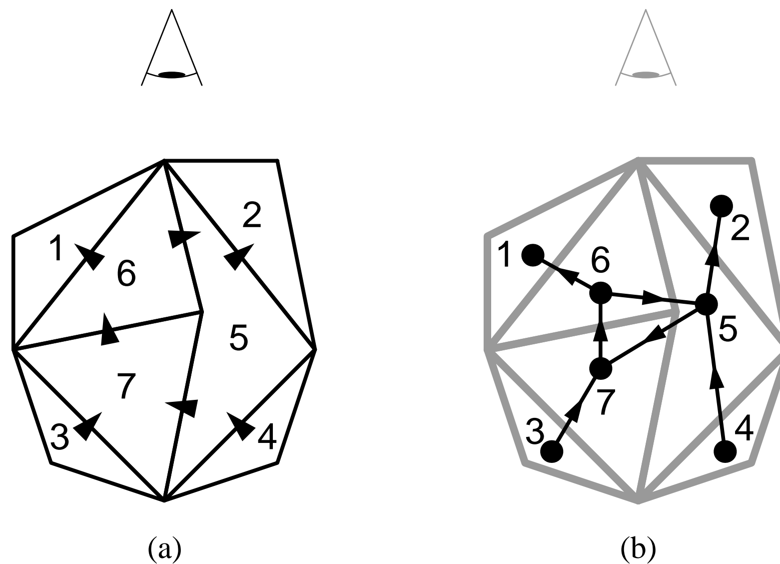


Figure 4.15 **(a)** Behind relations of a cyclic mesh. **(b)** The directed cyclic graph corresponding to the behind relations in (a).

non-convex cell 5 in Figure 4.15a allowed us to create a simple cyclic mesh in two dimensions. In fact, cyclic tetrahedral meshes are more easily constructed than cyclic triangular meshes; see Figure 4.21a.

A failed depth-first topological sort of the graph with reversed directions is visualized in Figure 4.16a: The cycle is detected when the algorithm descends from node 5 to node 6 (see the position marked with "!") as node 6 was visited earlier in the same descent.

In order to compute the strongly connected components of the visibility graph, a depth-first search algorithm published by Tarjan in [66] may be employed. Fortunately, this algorithm also computes a visibility ordering of the strongly connected components; thus, these components have to be rendered instead of single cells. If a component does not consist of a single cell, a visibility cycle has to be rendered as discussed in Section 4.4.

The basic idea of the algorithm by Tarjan is to keep track of the highest node (frontmost cell) that can be accessed from the current node. This node will usually be identical to the current node as outgoing edges are in general directed to the back of the mesh (compare Figures 4.14b and 4.16), i.e., by following any outgoing edge one is usually not able to reach a cell that is closer to the front than the current cell. The situation is, however, different for a cycle. For example, node 7 in Figure 4.16b has an outgoing edge to node 5, which has an edge to node 6, but cell 6 is in front of cell 7. This feature is exploited by Tarjan's algorithm to compute and topologically sort the strongly connected components of a directed

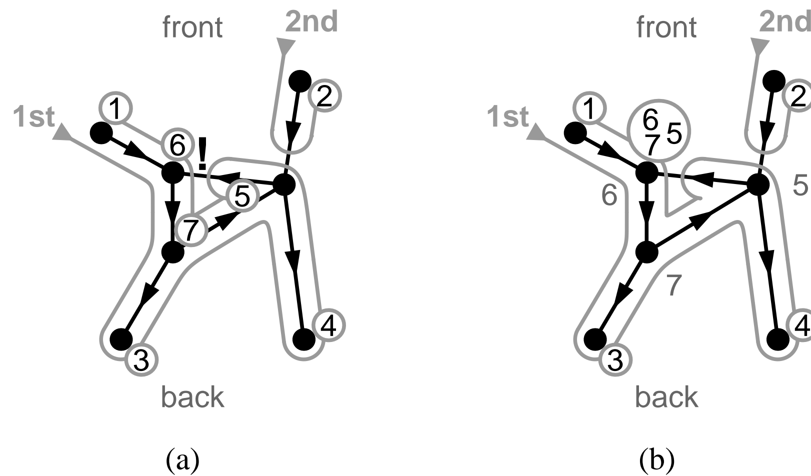


Figure 4.16 (a) Failed topological sort of the cyclic graph of Figure 4.15b (with reversed directions) using a depth-first search for acyclic graphs. **(b)** Topological sorting of the strongly connected components of the graph of Figure 4.15b (with reversed directions) using an extended depth-first search.

graph as illustrated in Figure 4.16b. The algorithm given in [66] is quite simple; however, the pseudo code presented here is based on a slightly improved version published in [58], which can be easily adapted to replace the phase III-DFS algorithm given in [80]. The resulting variant of the MPVO algorithm will be called MPVOC (C for cyclic).

MPVOC Phase III Algorithm

set $i := 0$;
empty stack S;
for each *cell* of the mesh
  set *cell*'s *num* := 0;
for each *cell* on the source cell list
  dfs(*cell*);

dfs(*cell*):
  push *cell* onto the stack S;
  set $i := i + 1$;
  set *front* := i;
  set *cell*'s *num* := *front*;

```
for each successor p of cell
   if p's num = 0
      set front := min(front, dfs(p))
   else
      set front := min(front, p's num)
if cell's num = front
   pop all cells from S until cell has been popped;
   set these cells' nums := 1+ number of the mesh's cells;
   output these cells (including cell);
return front.
```

When comparing this pseudo code with the pseudo code of the MPVO phase III-DFS algorithm in [80], note that the graph's edges have been reversed as depicted in Figure 4.16b. Consequently, the terms sink and source cells, and predecessor and successor switch their meanings. Also note that each cell of the mesh is required to store one additional integer variable (called *num*), which is not required by the original phase III-DFS algorithm. The time and space complexities of the MPVOC phase III algorithm are $O(n)$, where n is the number of cells of the mesh. These complexities are inherited from the algorithm by Tarjan (see [66]).

Strongly connected components are identified by the last if-statement in the procedure *dfs*. If one of these components consists of more than one cell, it corresponds to a visibility cycle and has to be rendered as discussed in Section 4.4. However, the MPVOC is advantageous even without the possibility to render cyclic obstructions: It turns out that the phase III-BFS algorithm is slower than the MPVOC phase III while it does not visit all cells of a mesh if a cycle exists (see [80]). Note that numerical errors can generate cycles even in acyclic meshes. The phase III-DFS algorithm is slightly faster but does not compute any information about the cycles. On the other hand, the MPVOC algorithm computes minimal sets of cyclic cells; thus, if only one cell in a group is visible (i.e., all other cells are either completely transparent, degenerate, or "imaginary"; see [80] and Section 4.1) then the MPVOC algorithm can still guarantee that the mesh is rendered exactly. This is also true if only two adjacent convex cells of a visibility cycle are visible.

4.3.3 Cell Sorting for Non-Convex, Cyclic Meshes

In [80], Williams proposed two extensions of the MPVO algorithm for non-convex meshes. Both may be integrated in the MPVOC algorithm presented in the previous section. The first adaptation of the MPVO algorithm for non-convex meshes (here called MPVONC) is a heuristic technique, the limitations of which are discussed in detail in [80]. The basic idea is to replace the source cell list (i.e., the

sink cell list for the non-reversed directions) by a new list that contains all cells with (at least) one boundary face that faces the viewpoint (i.e., is a front face of the cell). After sorting the list according to the distances of the centroids of the cells to the viewpoint, a depth-first search is performed for each cell in the list starting with the most distant one.

Obviously, this extension works without modifications for the MPVOC algorithm as the only differences of the MPVOC algorithm are encapsulated in the depth-first search, i.e, in the function *dfs* specified in Section 4.3.2. The resulting algorithm will be referred to as the MPVONCC algorithm.

Unfortunately, the MPVONCC (and the MPVONC) algorithm does not guarantee a correct visibility ordering for all non-convex meshes. Therefore, Williams also proposed an alternative extension of the MPVO algorithm, namely the convexification of non-convex meshes, which is described in detail in Section 4.1. For the purpose of cell sorting, the imaginary tetrahedra of a convexified mesh are treated in exactly the same way as all other cells, but they are simply ignored in the rendering step. In other words, convexified meshes may be sorted in the same way as any other convex mesh; therefore, the MPVOC algorithm is able to process any cyclic, non-convex mesh, provided the convexification of the mesh can be computed.

In order to compare the performance of the discussed sorting algorithms (MPVO and MPVOC for convexified meshes; MPVONC and MPVONCC for non-

Table 4.1 Sorting times for the convexified meshes with the MPVOC algorithm for convex, cyclic meshes. For comparison, the times of a slightly improved implementation and Williams' original implementation of the MPVO algorithm with a recursive depth-first search are included. The first number in parentheses is the time for phase II (computation of the behind relations), the second for phase III (topological sorting).

		sorting times (in seconds)		
convexified data set	number of tetrahedra	MPVOC	MPVO	MPVO (orig. impl.)
HeatSink	124,152	0.23 (0.12+0.11)	0.21 (0.13+0.08)	0.25 (0.17+0.08)
Bluntfin	190,852	0.34 (0.17+0.17)	0.31 (0.19+0.12)	0.34 (0.23+0.11)
Tapered Cylinder	630,726	1.20 (0.59+0.61)	1.07 (0.63+0.44)	1.26 (0.87+0.39)

Table 4.2 Sorting times for the (non-convex) meshes with the MPVONCC algorithm for non-convex, cyclic meshes. For comparison the times of a slightly improved implementation and Williams' original implementation of the MPVONC algorithm for non-convex, acyclic meshes are included. Times are specified as in Table 4.1.

		sorting times (in seconds)		
(non-convex) data set	number of tetrahedra	MPVONCC	MPVONC	MPVONC (orig. impl.)
HeatSink	121,668	0.27 (0.14+0.13)	0.25 (0.15+0.10)	0.25 (0.18+0.07)
Bluntfin	187,318	0.40 (0.20+0.20)	0.36 (0.22+0.14)	0.35 (0.24+0.11)
Tapered Cylinder	624,960	1.37 (0.66+0.71)	1.24 (0.72+0.52)	1.29 (0.88+0.41)

convex meshes), several tests were performed. Table 4.1 summarizes time measurements for the MPVOC algorithm compared to a slightly improved implementation (see [31]) and Williams' original implementation of the MPVO algorithm. As the MPVOC and MPVO algorithms are restricted to convex meshes, the convexified versions of the data sets presented in Section 4.1 have been used. The measurements were performed on an SGI Octane using one 250 MHz R10000 processor. Each number is the average of a set of sorting times for 100 different viewpoints. Note that no cycles were found in these data sets for any of the viewpoints and no rendering times are included.

Table 4.1 indicates that the time complexity of the MPVOC algorithm is indeed linear in the number of tetrahedra, and that the MPVOC algorithm is only about 12 % slower than a comparable implementation of the MPVO algorithm.

Table 4.2 includes the analogous measurements for the MPVONCC and MPVONC algorithms. As mentioned, this extension for non-convex meshes scarcely affects the new phase III of the MPVOC algorithm; therefore, the discussion of the differences between the MPVO and the MPVOC algorithm from above also applies to the comparison between the MPVONC and the MPVONCC algorithm. The times in Table 4.2 confirm this result.

Apart from the MPVOC and MPVONCC algorithms there are no visibility sorting algorithms for cyclic meshes. Ray casting algorithms, however, perform a visibility ordering for each pixel; therefore, they offer an alternative for rendering cyclic meshes. However, they are considerably slower for useful image sizes. For

example, Farias et al. report a total rendering time of 6 seconds for the Bluntfin data set for a 256×256 pixels image in [19]. Note that this time includes the computation and composition of color contributions of the cells.

The best exact visibility sorting algorithms for acyclic meshes (assuming that numerical errors do not generate cycles) are the MPVO algorithm for convex meshes and the BSP-XMPVO algorithm for non-convex meshes. The latter was published by Comba et al. in [13] and is considerably slower than the MPVO algorithm (about factor 3.5).

A visibility sorting algorithm for non-convex, acyclic meshes that is image-space correct was proposed by Cook et al. in [14]. In contrast to other variants of the MPVO algorithm, its complexity depends on the image size. For not too large images, the algorithm performs almost as well as the MPVONC algorithm.

Inexact visibility sorting algorithms for acyclic meshes are substantially faster than the MPVO algorithm according to [83]. However, these algorithms (see for example [12, 27, 83]) appear to be inappropriate for rendering visibility cycles in principle as no information about a possible cyclic structure is computed.

4.4 Cell Projection for Cyclic Meshes

This section presents a hardware-assisted algorithm for rendering possibly semi-transparent and/or intersecting polygons that may form visibility cycles (Section 4.4.1). The rendering of visibility cycles of convex polyhedral cells (as required by the MPVOC and MPVONCC algorithms presented in the previous section) will be discussed in Section 4.4.2.

The algorithm presented here was first published in [31]. It was derived from an algorithm by Snyder and Lengyel published in [63]. However, there are several crucial differences, which will be discussed below.

4.4.1 Rendering of Cyclic Occlusions of Polygons

Before specifying the algorithm, some formal definitions have to be introduced. Following the notation of [63], an image $I(P)$ of an orderless collection of n polygons $P := \{P_i\}$ with $i = 1, \ldots, n$ is represented as a two-dimensional array of 4-tuples with red, green, blue, and α components:

$$I(P) := [I_{\mathrm{r}}(P), I_{\mathrm{g}}(P), I_{\mathrm{b}}(P), I_{\alpha}(P)].$$

Note that the red, green, and blue components specify an associated color (also called pre-multiplied color), which are usually denoted $\tilde{C}$ in this work (see Section 2.4.3).

Two operations on images are required in the following, which are addition

$$A + B := [A_r + B_r, A_g + B_g, A_b + B_b, A_\alpha + B_\alpha]$$

and "out" (which has higher precedence than "+"):

$$A \,\text{out}\, B \;:=\; [A_r(1 - B_\alpha), A_g(1 - B_\alpha), A_b(1 - B_\alpha), A_\alpha(1 - B_\alpha)].$$

In plain words, $A \,\text{out}\, B$ is the image A attenuated by B. Based on these operations accumulator operators $\sum$ and OUT are defined, e.g.,

$$\sum_{\{A,B,C\}} = A + B + C \qquad \text{or} \qquad D \operatorname*{OUT}_{\{A,B,C\}} = ((D \,\text{out}\, A) \,\text{out}\, B) \,\text{out}\, C.$$

Note that the order of the arguments $\{A, B, C\}$ is not relevant. Additionally, the "over" operator is defined as

$$A \,\text{over}\, B := A + B \,\text{out}\, A,$$

which is the composition of a background image B with a foreground image A.

The proposed algorithm solves the following problem: Given a background image I^{bg} and a set of polygons $P := \{P_i\}$ with $i = 1, \ldots, n$, what is the final image $I(P) \,\text{over}\, I^{\text{bg}}$?

According to [63] $I(P)$ (for non-intersecting polygons) is given by

$$I(P) = \sum_{i=1}^{n} I(P_i) \operatorname*{OUT}_{\{I(P_j) | P_i < P_j\}},$$

where $P_i < P_j$ means that the polygon P_j occludes P_i; i.e., the image $I(P)$ is given by the sum of the images of all polygons P_i, each being attenuated by all polygons that occlude P_i. Since it is possible to replace all occlusion tests by z-tests, this equation is rewritten as

$$I(P) = \sum_{i=1}^{n} I(P_i) \operatorname*{OUT}_{j=1, j \neq i}^{n} I^*(P_j),$$

where $I^*(P_j)$ denotes the image of P_j rasterized with active z-test but disabled writing to the z-buffer. The z-buffer was set by rendering $I(P_i)$ with active writing into a previously cleared z-buffer.

The final image could be calculated by evaluating $I(P) \,\text{over}\, I^{\text{bg}}$. However, it is preferable to implement the equivalent expression $I^{\text{bg}} \,\text{out}\, I(P) + I(P)$ in order to avoid the need to buffer opacities. Inserting the result for $I(P)$ yields

$$I(P) = I^{\text{bg}} \operatorname*{OUT}_{i=1}^{n} I(P_i) + \sum_{i=1}^{n} I(P_i) \operatorname*{OUT}_{j=1, j \neq i}^{n} I^*(P_j), \tag{4.1}$$

i.e., the background image I^{bg} is attenuated by all polygons and the contribution of the attenuated images of these polygons are added.

It is straightforward to translate Equation (4.1) into an algorithm. The steps of this algorithm are:

1. Set the display buffer I^{d} to the background image: $I^{\text{d}} := I^{\text{bg}}$. (This is, of course, unnecessary if I^{bg} is already stored in I^{d}.)

2. For each polygon P_i with $i = 1, \ldots, n$ attenuate the display buffer:

$$I^{\text{d}} := I^{\text{d}} \operatorname{out} I(P_i).$$

3. For each polygon P_i with $i = 1, \ldots, n$ set the additional image buffer:

$$I^{\text{P}} := I(P_i) \operatorname*{OUT}_{j=1}^{n} I^{*}(P_j)$$

and add the resultant image to the display buffer: $I^{\text{d}} := I^{\text{d}} + I^{\text{P}}$.

Before discussing the main features of this algorithm, a simple step-by-step example is given.

Example: Ternary Cycle

The goal in this example is to render a visibility cycle formed by three polygons P_1, P_2, and P_3 in front of a white background I^{bg} as depicted in Figure 4.17b. The geometry of this scene is sketched in Figure 4.17a. Each polygon has a constant

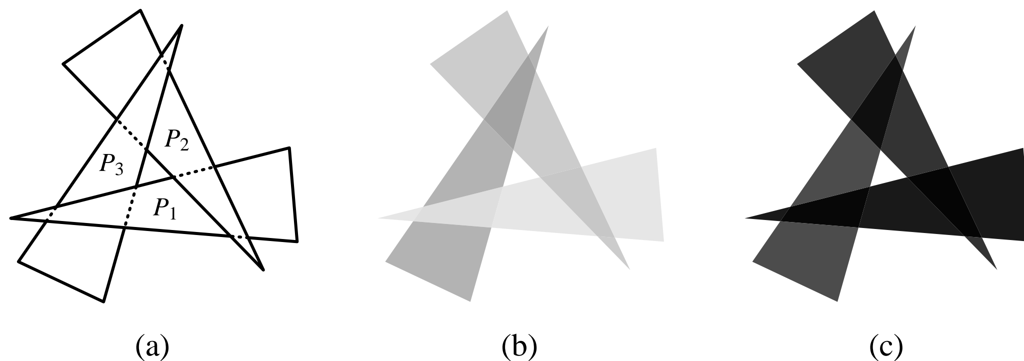
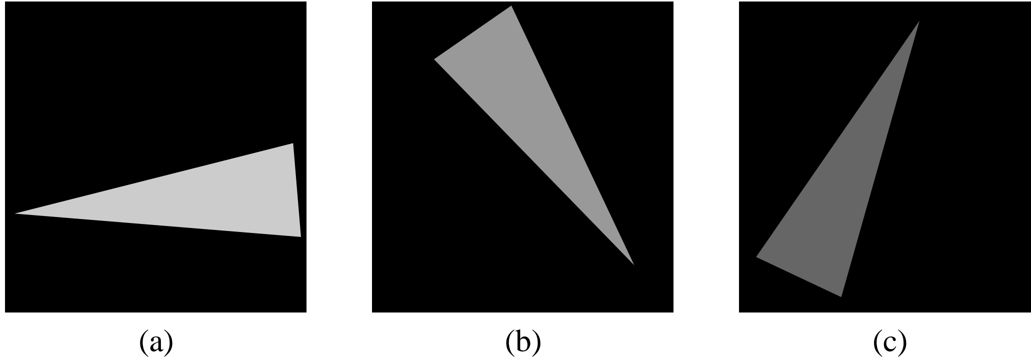
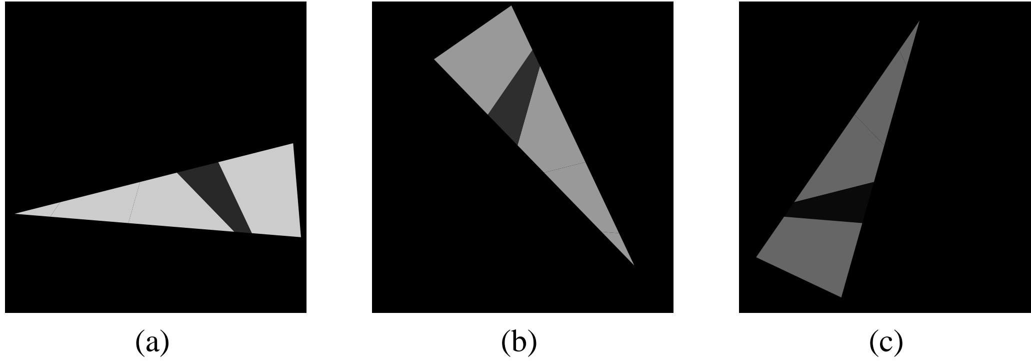


Figure 4.17 **(a)** Geometry of three polygons P_1, P_2, and P_3 that form a visibility cycle. **(b)** Colored rendering of the polygons P_1, P_2, and P_3 in front of a white background. **(c)** $I^{\text{bg}} \operatorname{out} I(P_1) \operatorname{out} I(P_2) \operatorname{out} I(P_3)$: the white background attenuated by the three polygons.



(a) (b) (c)

Figure 4.18 **(a)** $I(P_1)$: the image of polygon P_1; $I_\alpha(P_1)$ is 0.9 within the polygon and 0 otherwise. **(b)** $I(P_2)$; $I_\alpha(P_2)$ is 0.8 within the polygon and 0 otherwise. **(c)** $I(P_3)$; $I_\alpha(P_3)$ is 0.7 within the polygon and 0 otherwise.



(a) (b) (c)

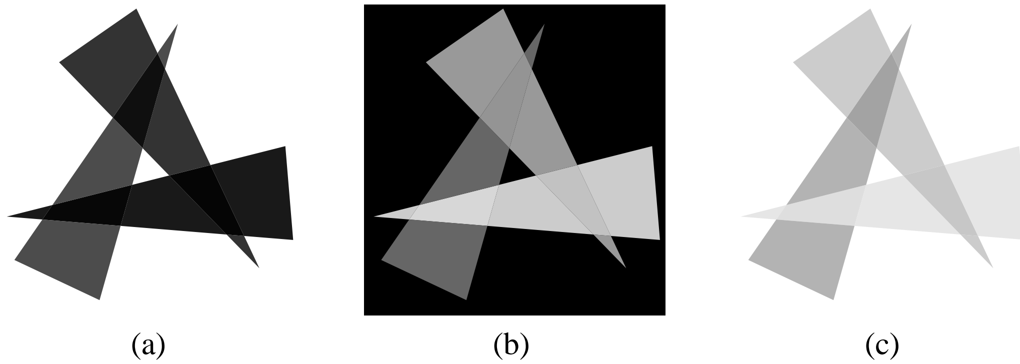
Figure 4.19 **(a)** $I(P_1)\,\mathrm{out}\,I(P_2)$: the image of polygon P_1 attenuated by the image of polygon P_2. **(b)** $I(P_2)\,\mathrm{out}\,I(P_3)$. **(c)** $I(P_3)\,\mathrm{out}\,I(P_1)$.

color (see Figure 4.18) and a constant α-value, which is 0.9, 0.8, and 0.7 for P_1, P_2, and P_3, respectively.

In terms of our formal specification of the algorithm, n is set to 3 and P to $\{P_1, P_2, P_3\}$. Moreover, I^{bg} is a completely white image. Therefore, the first step of the proposed algorithm is to copy this white background into the display buffer I^{d}.

The second step is to attenuate the display buffer by the images of the polygons. The result of the three (commutative) attenuation operations is depicted in Figure 4.17c.

The third step consists of a loop over the three polygons. Each polygon is attenuated by the occluding polygon, i.e., P_1 by P_2 (Figure 4.19a), P_2 by P_3 (Figure 4.19b), and P_3 by P_1 (Figure 4.19c). The sum of the three resulting images is shown in Figure 4.20b. Finally, this sum is added to the attenuated background (see Figure 4.20a). The result is depicted in Figure 4.20c. More precisely spoken, the sum corresponding to Figure 4.20b is evaluated and added to the attenuated



(a) (b) (c)

Figure 4.20 **(a)** I^{bg} out $I(P_1)$ out $I(P_2)$ out $I(P_3)$: the (white) background attenuated by the three polygons P_1, P_2, and P_3. **(b)** $I(P_1)$ out $I(P_2)$ + $I(P_2)$ out $I(P_3)$ + $I(P_3)$ out $I(P_1)$: the final contribution of the images of the polygons. **(c)** I^{bg} out $I(P_1)$ out $I(P_2)$ out $I(P_3)$ + $I(P_1)$ out $I(P_2)$ + $I(P_2)$ out $I(P_3)$ + $I(P_3)$ out $I(P_1)$: the final image.

background by sequentially adding the contribution of each polygon (Figure 4.19) to the display buffer, which initially holds the attenuated background image (Figure 4.20a).

Comparison with the Algorithm by Snyder and Lengyel

Compared to the original algorithm by Snyder and Lengyel published in [63], the proposed algorithm features several advantages:

- only one (instead of two) additional image buffer is required;
- no α-values (opacities) are buffered;
- no occlusion tests have to be performed in software;
- no occlusion graph is constructed or sorted;
- intersecting polygons do not need any particular treatment.

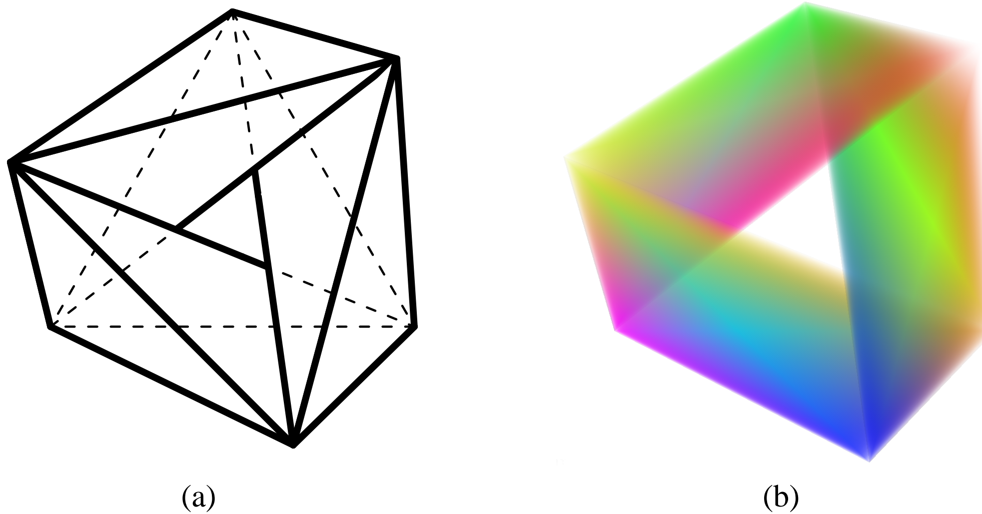
As all occlusion tests are performed by z-tests instead of complex calculations in software, a naive implementation of the algorithm will be rather small. In fact, a prototypical implementation of this algorithm is possible with about 50 lines of C code (excluding the code required to render a single polygon).

The disadvantages include the need for a z-buffer and additional image operations. As the algorithm is of time complexity $O(n^2)$, where n is the number of polygons, it is impractical for rendering complex scenes. However, it is useful for rendering visibility cycles formed by a few primitives.

4.4.2 Rendering of Cyclic, Convex Polyhedral Cells

Cyclic obstructions of convex polyhedral cells (instead of polygons) can be rendered by cell projection of the cells because cell projection reduces the problem of rendering a volumetric cell to the problem of rendering a polygon. However, the polygonal projection of each cell must be placed within the cell as the z-coordinates of the pixels of the polygons are used in z-tests. Note that cells with a volumetric intersection cannot be rendered this way. However, this kind of intersection is already excluded by the requirements of phase III of the MPVOC and MPVO algorithms (see Section 4.3.2).

The combination of a projected tetrahedra algorithm (see Section 3.1) with the presented algorithm for the rendering of visibility cycles of polygons allows us to render any visibility cycles of tetrahedra. For example, Figure 4.21a depicts three tetrahedra that form a visibility cycle. After defining colors and opacities at the vertices of the tetrahedra, the proposed method may be employed to render these tetrahedra in front of a white background as shown in Figure 4.21b. The geometry of this example is discussed in more detail in [80].



(a) (b)

Figure 4.21 **(a)** A visibility cycle formed by three tetrahedra. **(b)** Volume rendering of the tetrahedra depicted in (a). *(See also Figure C.6 on page 148 in the color plate section.)*

David's Mom: *I mean, it's not supposed to be like this.*
David: *It is not supposed to be anything.*

Dialog from the movie *Pleasantville*

Chapter 5

Non-Simplicial and Non-Adaptive Meshes

While the previous chapters have discussed several unpleasant geometric features of non-uniform meshes, there are also important advantages of certain non-uniform meshes. For example, meshes consisting of simplicial cells, i.e., tetrahedral meshes in three dimensions, offer the possibility of a continuous, piecewise linear interpolation as discussed in Section 2.7.3. In contrast, non-simplicial meshes lack this possibility; therefore, algorithms that rely on a linear interpolation are less suited for non-simplicial meshes. Two examples are the pre-integrated classification presented in Section 2.5 and many algorithms based on the topology of scalar fields. (The topology of a scalar function is introduced in Section 5.2.)

In this chapter, two such algorithms are modified in order to apply them to non-simplicial meshes, in particular uniform meshes. In both cases, approximations are employed in order to overcome the problems originating from the missing linear interpolation. More specifically, Section 5.1 presents pre-integrated volume rendering for uniform meshes, which has been published previously together with Klaus Engel and Thomas Ertl in [18], and in Section 5.2 an algorithm for "topology-guided" downsampling of uniform meshes is discussed, which has been published together with Thomas Ertl in [32]. Both algorithms are of particular interest for interactive direct volume visualization because of the hardware support for uniform meshes by modern graphics adapters.

Another important geometric feature of tetrahedral meshes — and unstructured meshes in general — is their adaptivity, i.e., the possibility of a locally adaptive resolution and an adaptive boundary, which uniform meshes lack. However, Section 5.3 presents a method to exploit the programmability of modern graphics hardware to emulate this feature with a two-level hierarchy of uniform meshes. This technique has been first published together with Thomas Ertl in [34].

5.1 Texture-Based Pre-Integrated Volume Rendering

This section presents a texture-based algorithm for hardware-accelerated volume visualization of uniform meshes that implements pre-integrated classification (see Section 2.5). The algorithm requires dependent texture lookups, i.e., the texture coordinates of one texture lookup are determined by a previous texture lookup (instead of a linear interpolation of texture coordinates). Texture-based volume rendering has been presented in Section 2.3: The basic idea is to render a stack of textured slices with either a stack of two-dimensional textures (see Figure 2.4a and [54]) or one three-dimensional texture (see Figure 2.4b and [7]).

The concepts of pre- and post-classification for direct volume rendering are discussed in detail in Section 2.4.2. For texture-based volume rendering, pre-classification is either implemented by applying transfer functions once for each texel and storing colors and opacities in the textures or with the help of paletted textures (see [28]), while post-classification is implemented by storing the original scalar data in textures and applying transfer functions during the rasterization of the slices after interpolating scalar values but before compositing the colors.

Unfortunately, pre-integrated volume rendering is not appropriate for uniform meshes since a linear interpolation between the data samples is assumed, while at least a trilinear interpolation within cells is necessary in order to achieve a continuous interpolation in uniform meshes. However, the approximation error of the discrete evaluation with pre-classification or post-classification is even worse since it is assumed that the data is almost constant between samples. On the other hand, the use of pre-integrated classification for texture-based volume rendering requires a sampling rate that is high enough to reconstruct a proper approximation of the original data by linear interpolation between samples.

With this assumption, pre-integrated classification may be implemented as follows: The textures contain the original scalar data of the volume analogously to the case of post-classification. For each rasterized pixel, pre-integrated classification requires the scalar data at the front and the back slice of each slab (see Figure 5.1a) between two adjacent slices (either object-aligned or view-aligned). Thus, the textures of the two slices have to be mapped onto one slice — either the front or the back slice; the latter case is illustrated in Figure 5.1b. This mapping of the two textures onto one slice requires multi-texturing and an appropriate calculation of texture coordinates. Thereby, the scalar values of both slices are fetched for every pixel of the slice corresponding to one slab. These two scalar values are necessary for a third texture lookup operation, which fetches pre-integrated colors and opacities from a two-dimensional texture. For view-aligned slices, the dependency on the length of the pre-integrated ray segment is often negligible since the

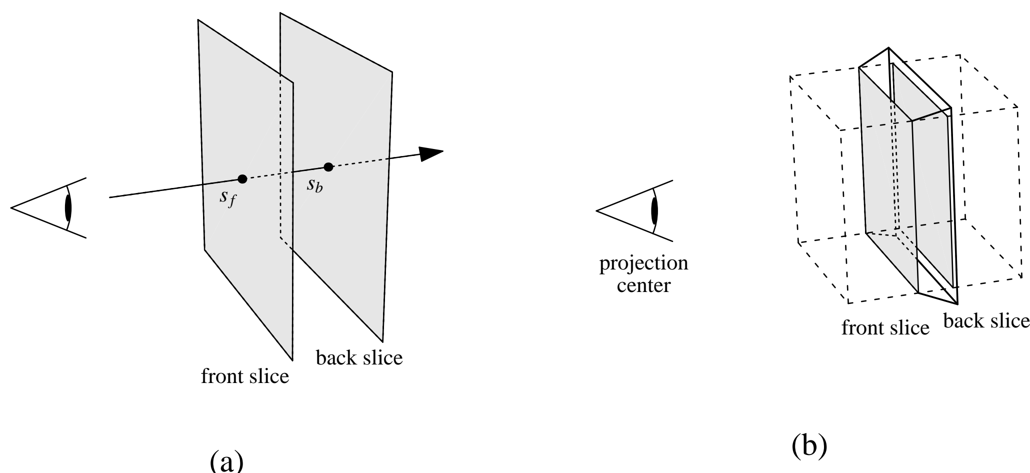


Figure 5.1 **(a)** A slab of the volume between two slices. The scalar value on the front (back) slice for a particular viewing ray is called s_f (s_b). **(b)** Projection of the front slice onto the back slice of a slab.

distances between slices are constant and, therefore, the lengths of the ray segments are approximately constant. As this texture lookup depends on previously fetched data, it is called a *dependent* texture lookup, which is supported by modern programmable graphics hardware, e.g., NVIDIA's GeForce3 or ATI's Radeon 8500.

In order to render shaded isosurfaces, the dependent texture has to contain color, transparency, and also an interpolation value if the isovalue is in between the front and back scalar value. The interpolation value is necessary in order to weight per-voxel gradient data fetched from an additional texture, since the interpolated (and normalized) gradient is required for lighting calculations. More details about the algorithm and a discussion of an implementation for NVIDIA's GeForce3 may be found in [18].

5.2 Topology-Guided Downsampling

As discussed in Section 5.2.1, it is an important feature of simplicial meshes with linear interpolation that all critical points are located at the positions of vertices of the mesh. Therefore, the application of topology-related concepts to non-simplicial — and in particular structured — meshes is rather uncommon, although these concepts have proven to be very useful for simplicial meshes. Thus, non-simplicial cells are also in this respect an unpleasant feature of meshes.

In order to show how to use topology-related concepts for structured meshes, this section presents a downsampling method for uniform volume meshes that

preserves much more of the topology of a scalar field than existing downsampling methods by preferably selecting scalar values of critical points. In particular, many critical points that are lost by traditional downsampling methods can be preserved. This section was first published together with Thomas Ertl in a similar form in [32].

Traditional downsampling methods include subsampling, i.e., successively deleting vertices, and the replacement of groups of vertices (for structured volume meshes usually $2 \times 2 \times 2$) by one vertex with the average data value as suggested for two-dimensional mip maps by Williams in [79] and for three-dimensional mip maps by Levoy and Whitaker in [40]. One generalization of this method is to filter a mesh before sampling it at a lower resolution; for a recent application see [26]. Many algorithms for volume visualization have been accelerated by employing downsampled meshes, e.g., ray casting [15, 40], splatting [38], and isosurface extraction [26, 60, 51, 75]. For all these techniques downsampling is an essential preprocessing step.

However, traditional downsampling methods ignore and, therefore, destroy the topology of the original scalar field. Unfortunately, there is no unique definition of the topology of a scalar field; related concepts are the *contour tree* [8], the *(hyper) Reeb graph* [21], and the *topology graph* [2]. All these concepts are, however, based on the *critical points* of a scalar field. Therefore, topology preservation of a scalar field is often defined as the preservation of all critical points; see for example [3, 23]. The theoretical framework for this definition is provided by Morse theory, see [48].

While the topology of a scalar field is not uniquely defined, the topology of surfaces — and isosurfaces in particular — is well defined. In fact, the topology of isosurfaces is strongly related to the critical points of the corresponding scalar field. Thus, the topology of isosurfaces extracted from downsampled meshes will usually deviate strongly from the topology of the original isosurfaces, i.e., the number of disconnected components, tunnels, and holes will strongly differ.

Unfortunately, the topology of an isosurface is in many cases its most important feature as it allows the user to navigate in a volume, to identify noise in a data set, or to estimate the quality and plausibility of extracted shapes or structures. Therefore, it is often useful to use topology-preserving simplification techniques in order to extract isosurfaces with the correct topology even from coarse meshes. Examples of such techniques have been published in [3, 23]. However, these approaches are limited to simplicial meshes and are, therefore, not very well suited for visualization algorithms for structured meshes.

Topology-guided downsampling, the method presented here, fills this gap by providing a simple algorithm for downsampling structured meshes without blindly destroying the topology of the scalar field. This is achieved by calculating critical points and determining the data values of the downsampled mesh from this clas-

sification. The method is named "topology-guided downsampling" as topology-preserving downsampling is impossible in general. However, even an approximate preservation of topology is highly desirable if isosurfaces are extracted from the downsampled volume mesh, e.g., for interactive previewing, because many topological features of the isosurfaces, e.g., the number of components, tunnels, and holes, are preserved.

After describing the algorithm in Section 5.2.1, some examples from medical and technical applications of volume visualization are presented in Section 5.2.2.

5.2.1 Algorithm

As topology-guided downsampling works as well in two dimensions as in three dimensions, the algorithm will be illustrated with the help of the two-dimensional scalar field $f(x,y)$ depicted in Figure 5.2a, which is defined by a bilinear interpolation between scalar values at the vertices of a two-dimensional uniform mesh. Isolines are extracted from this mesh by a decomposition into triangular cells and slicing the resultant height field with a horizontal plane as depicted in Figure 5.2b.

In order to (approximately) preserve the topology of this scalar field, its critical points have to be preserved. The first step is therefore to identify critical points in two- and three-dimensional structured meshes.

Critical Points in Two Dimensions

Critical points are local maxima, local minima, and saddle points. They indicate points where an isoline or isosurface changes its number of components or its genus. It is an important advantage of simplicial meshes, i.e., triangular meshes in two dimensions and tetrahedral meshes in three dimensions, that all critical points are located at vertex positions. Therefore, two-dimensional structured meshes are

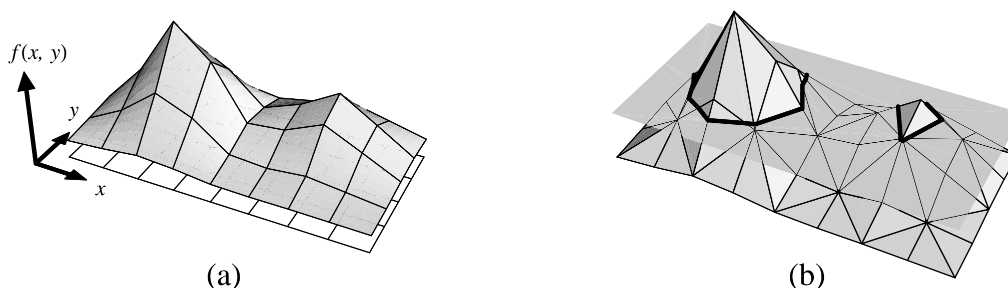


Figure 5.2 **(a)** A two-dimensional scalar (height) field. **(b)** Piecewise linear approximation to an isoline in the scalar field of (a).

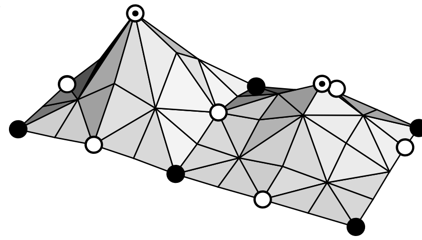


Figure 5.3 The critical points of the field of Figure 5.2a. Maxima are marked with dotted circles (⊙), minima with disks (•), and saddle points with empty circles (∘).

Figure 5.4 The surrounding polygon (*thick line*) of a vertex (◯) with **(a)** four neighbors and **(b)** eight neighbors.

usually decomposed into simplicial meshes as illustrated in Figures 5.2b and 5.3. It should be noted that this decomposition is only virtual, i.e., it is not stored in any data structures but performed on-the-fly whenever it is required.

In order to handle vertices at the boundary of the mesh, the missing neighbors are (virtually) generate by mirroring neighboring vertices across the boundary (see [23]). With this in mind, the decomposition into triangles employed in Figure 5.3 generates only two kinds of vertex neighborhoods: one with four neighbors depicted in Figure 5.4a and another with eight neighbors depicted in Figure 5.4b.

In analogy to [23], the corresponding *surrounding polygon of a vertex* is defined as the boundary of the adjacent triangles. The surrounding polygon defines an edge graph, which will be used in order to classify the surrounded vertex as a regular point, local maximum, local minimum, or saddle point.

This classification is achieved by marking each node of the edge graph, i.e., each vertex neighbor of the surrounding polygon of a vertex. A neighbor is marked with a 1 if its data value is greater than the value at the surrounded vertex, and a 0 otherwise. Then all edges between a 1 node and a 0 node are deleted and the number of remaining connected components of the edge graph is counted. The point is an extremum if this number is one. If it is two, then the point is regular; otherwise the point is a saddle point. The results of this classification for each

vertex of the mesh of Figure 5.2a is visualized in Figure 5.3. Note that this classification ignores any degeneracies. This is legitimate as we are only concerned with an approximate preservation of critical points.

Automatic Lookup Table Generation

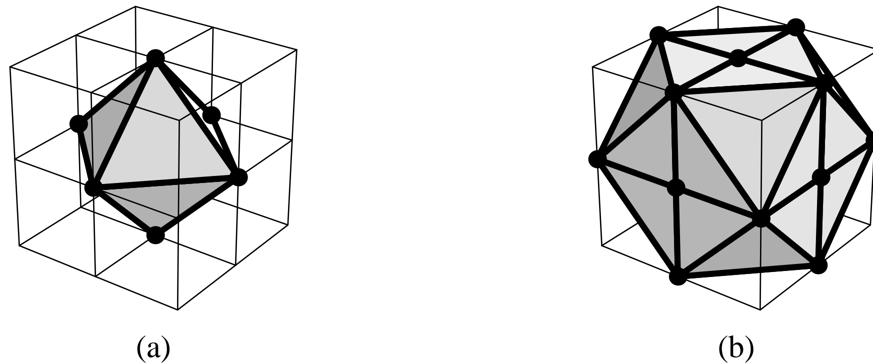
In order to speed up the classification of points discussed above, precalculated lookup tables may be employed. This is achieved by numbering all nodes of the edge graph. The number of each node corresponds to a bit position that holds the mark 0 or 1 of the corresponding node. The resultant bit pattern is interpreted as an integer index into a lookup table that specifies a classification code for each index. For example, the bit pattern for a local maximum will only consist of 0s, i.e., the index will be 0. Thus, the first entry in the lookup table will be the classification code for a local maximum. Similarly, the bit pattern for a local minimum will only consist of 1s, i.e., the last entry in the lookup table will be the classification code for a local minimum.

Note that local maxima and minima are easily identified; therefore, only one bit per entry in the lookup table is needed in order to distinguish between regular and saddle points; thus, the minimal table size for the surrounding polygon of Figure 5.4a is only $2^4/8 = 2$ bytes. The size of the corresponding lookup table for the surrounding polygon of Figure 5.4b is $2^8/8 = 32$ bytes.

Critical Points in Three Dimensions

The first problem of a generalization to three dimensions is to find a suitable tetrahedralization of a structured hexahedral mesh. In [8], Carr et al. discuss various decomposition schemes for three-dimensional structured meshes and choose a subdivision of each hexahedral cell into six square pyramids with their apices in the cell center although this requires that new data points are interpolated. In this work, new data points are avoided and therefore each hexahedron is subdivided into five tetrahedra. As mentioned in [8], this decomposition is not symmetrical as it generates two kinds of vertex neighborhoods. Also note that the decomposition is only virtual, i.e., it is performed on-the-fly.

In analogy to the two-dimensional case, the corresponding *surrounding polyhedron of a vertex* is defined by the boundary of the adjacent tetrahedra (see [23]). In the case of the decomposition of hexahedral cells into five tetrahedra, there are two different kinds of surrounding polyhedra: an octahedron and a triangulated cubeoctahedron; see Figures 5.5a and 5.5b. However, the approximative nature of the presented algorithm allows us to relax the need for a correct simplicial decomposition. Thus, the same surrounding polyhedron is used for all vertices. As



(a) (b)

Figure 5.5 Surrounding polyhedra of a vertex: **(a)** six neighbors define an octahedron. **(b)** 18 neighbors define a triangulated cubeoctahedron.

the triangulated cubeoctahedron generated better results in several experiments, it was employed for all examples in Section 5.2.2.

The classification of vertices as regular points, local maxima, local minima, and saddle points is performed in a similar way as in the two-dimensional case. In particular, nodes of the edge graphs defined by the surrounding polyhedra are marked in the same way: Nodes with a data value greater than the data value at the surrounded vertex with a 1, otherwise with a 0. All marks are collected in a bit pattern, which is used to index a precomputed lookup table containing the classification bit for the vertex as discussed above. The table size is $2^6/8 = 8$ bytes for the octahedron and $2^{18}/8 = 32768$ bytes for the triangulated cubeoctahedron.

Preservation of Critical Points

One of the results of Morse theory is that all critical points of a scalar field have to be preserved in order to preserve the topology of all its isosurfaces. However, it is not necessary to preserve the exact geometric position of the critical points. Nonetheless, the scalar values at all critical points have to be preserved exactly. Otherwise the topology of isosurfaces for isovalues in the interval between the old and the new scalar value at a critical point is changed. For example, if a local maximum is preserved but its scalar value v_{max} is decreased to $v'_{\text{max}} < v_{\text{max}}$, all isosurfaces for isovalues in the interval $[v'_{\text{max}}, v_{\text{max}}]$ will be modified topologically. This is what usually happens to local extrema with the traditional combination of filtering and downsampling.

An example is given in Figures 5.2, 5.6, and 5.7. The scalar field of Figure 5.2a is downsampled by averaging the scalar values (i.e., heights) over groups of four (or less) vertices as indicated in Figure 5.6 (for now the marks of the critical vertices should be ignored). Each group of vertices corresponds to one new

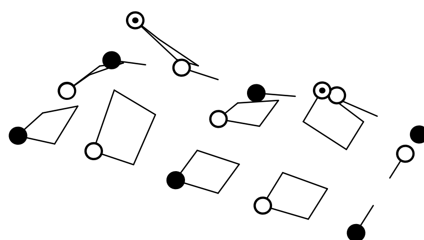


Figure 5.6 The partitioning of the mesh of Figure 5.2a employed for downsampling. Critical points are marked as in Figure 5.3.

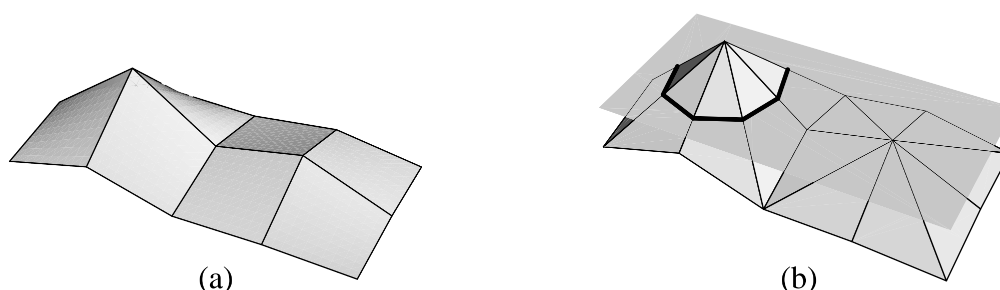


Figure 5.7 **(a)** The scalar field obtained by averaging downsampling of the mesh of Figure 5.2a. **(b)** Piecewise linear approximation to an isoline in the field of (a) for the same isovalue as in Figure 5.2b.

vertex of the downsampled mesh depicted in Figure 5.7a. Because of the averaging the height of both maxima is decreased in the new field. Therefore, the isolines for the same isovalue are topologically different for the original field and its downsampled version as illustrated by Figures 5.2b and 5.7b.

The goal of the presented method is to avoid these changes whenever possible; therefore, linear filtering has to be avoided. Thus, an appropriate downsampling principle is to select and thereby preserve the scalar values of critical points. Although this selection does not guarantee the preservation of critical points, the preservation of the selected scalar values is a necessary condition for the preservation of critical points.

The selection is illustrated in Figure 5.6, where all critical points are marked. In this example each group of vertices contains exactly one critical point. The scalar value of each critical point is then used for downsampling instead of the average height of the group of vertices. The resultant downsampled mesh is depicted in Figure 5.8a. Figure 5.8b demonstrates that the topology of the isoline of Figure 5.2b is preserved with this downsampling technique. The following section describes topology-guided downsampling for three-dimensional meshes in more detail.

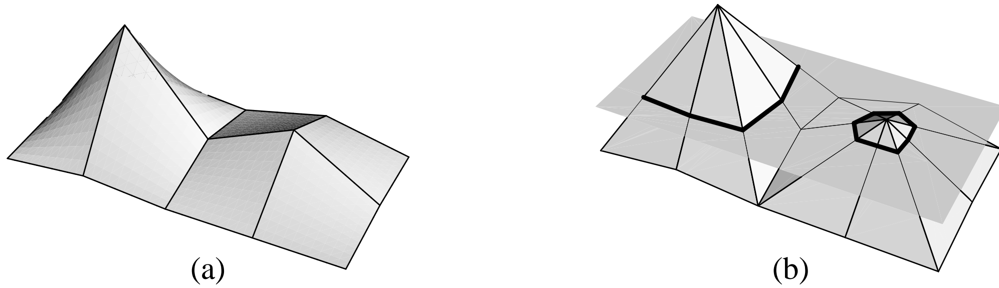


Figure 5.8 **(a)** Same as Figure 5.7a but for topology-guided downsampling. **(b)** Same as Figure 5.7b for the field in (a).

Steps of the Algorithm

Topology-guided downsampling reduces the number of vertices of a volumetric structured mesh with even dimensions by a factor of eight by replacing groups of $2 \times 2 \times 2 = 8$ vertices by one vertex. For each disjoint group of 8 vertices the following steps are performed in order to determine the scalar value of the new vertex. (If not given implicitly, the position of the new vertex is determined by the average position of the 8 vertices.)

1. For each vertex of the group, compute whether it is a regular point, a saddle point, or an extremum. Also, compute the average scalar value of these vertices.

2. If there is no critical point, the average scalar value is the result.

3. If there is only one critical point, its scalar value is the result.

4. If there are multiple saddle points but no extremum, the scalar value of the saddle point with the largest absolute distance to the average scalar value is the result.

5. If there are (multiple) saddle points but only one extremum, the scalar value of the extremum is the result.

6. Otherwise, the scalar value of the extremum with the largest absolute distance to the average scalar value is the result.

Steps 1 to 3 are motivated by the considerations described above. Steps 4 to 6 reflect an interest in the most “important” critical points, since many saddle points would not exist without a neighboring extremum and distant critical points are

likely to have more influence on the topology of isosurfaces than critical points close to the average scalar value.

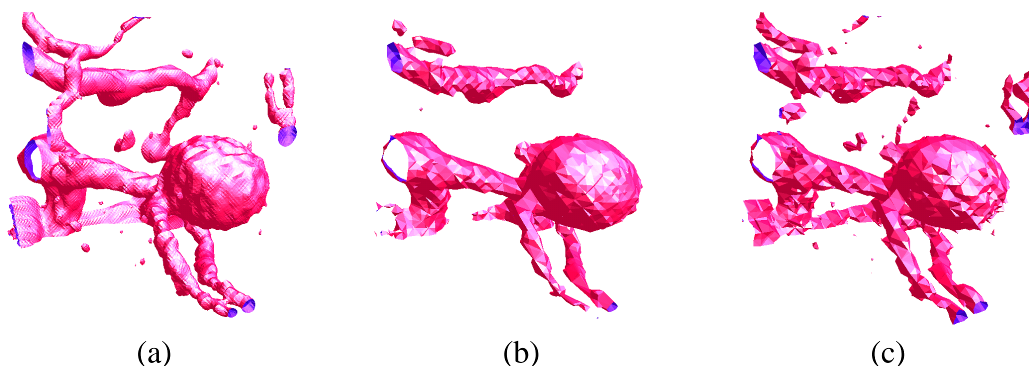
This downsampling procedure can be applied repeatedly — each time reducing the number of vertices by a factor of 8. However, in comparison to averaging downsampling methods, much more of the topological information is preserved by this algorithm as is demonstrated with the help of several examples in the next section.

5.2.2 Examples

Blood Vessels

The first example is a CTA (computer tomography angiography) volume data set showing blood vessels around an aneurysm. It is well suited to demonstrate topology-guided downsampling as it contains noise and structures of very different sizes. The resolution of this data set is $128 \times 128 \times 60$ voxels and 8 data bits per voxel. In order to visualize it, an isosurface for a fixed isovalue is extracted with a simple marching tetrahedra algorithm after decomposing the uniform mesh into tetrahedra as explained in the previous section. Figure 5.9a depicts the resultant isosurface of the original data set. All isosurfaces are rendered using flat shading with surface normals calculated directly from each triangle in order to emphasize the underlying mesh structure even for very fine meshs. Of course, pre-integrated volume rendering for uniform meshes (see Section 5.1) could also be used to render these images.

The downsampling results of the presented algorithm will be compared to a simple averaging downsampling that replaces eight vertices by one vertex with



(a) (b) (c)

Figure 5.9 **(a)** An isosurface extracted from a $128 \times 128 \times 60$ CTA volume data set. **(b)** Same isosurface extracted from a mesh downsampled to dimensions $32 \times 32 \times 15$ with averaging downsampling. **(c)** Same as (b) with topology-guided downsampling. *(See also Figure C.7 on page 148 in the color plate section.)*

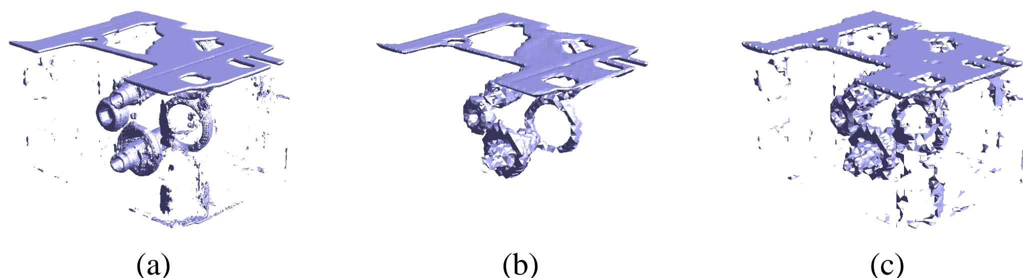
the average data value as employed in [79, 40, 15]. More general filtering and downsampling methods, e.g., [26], suffer essentially from the same problems for a comparable downsampling rate.

Figure 5.9b depicts the isosurface to the same isovalue as in Figure 5.9a but extracted from a downsampled volume of dimensions $32 \times 32 \times 15$ using traditional averaging downsampling. In contrast, Figure 5.9c depicts the result for the same settings but using topology-guided downsampling as described in the previous section with the cubeoctahedron being the surrounding polyhedron of all vertices. The compression rate is $(1/8)^2 \approx 1.6\%$ in both cases. Note that none of the two downsampling methods depends on a particular isovalue, i.e., the user may choose an isovalue after the downsampling, which is only a preprocessing step.

Obviously, the noise manifesting itself in small disconnected parts of the original isosurface in Figure 5.9a is partially preserved with topology-guided downsampling in Figure 5.9c but is almost completely lost with averaging downsampling in Figure 5.9b. More importantly, several crucial connections of blood vessels visible in Figure 5.9a become disconnected in Figures 5.9b and 5.9c. However, topology-guided downsampling preserves at least parts of the vessels while averaging downsampling results in larger gaps, or even the complete vanishing of parts of vessels, e.g., at the top of Figure 5.9b.

Engine Block

The second volume data set is a volumetric scan of an engine block. This example differs considerably from the first: Instead of a continuum of intensities, there are only three materials, i.e., air and two kinds of alloys. Due to noise, however, isosurfaces corresponding to isovalues between the two material values are likely to consist of many disconnected components as illustrated in Figure 5.10a.



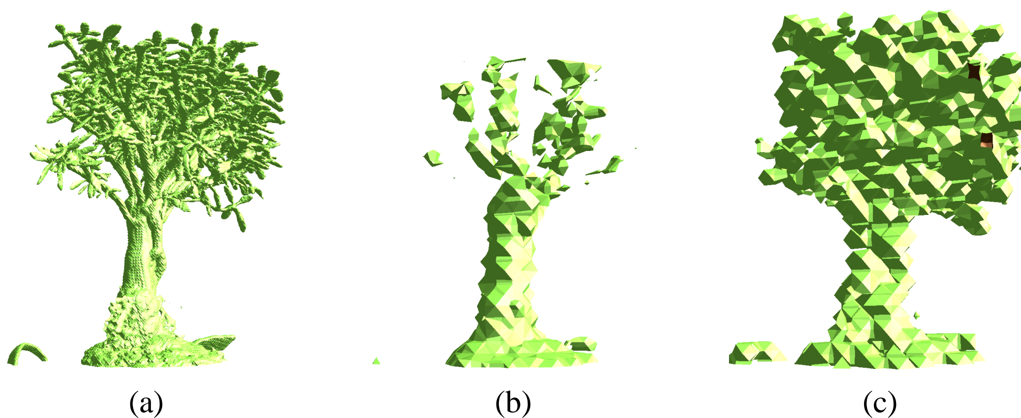
(a) (b) (c)

Figure 5.10 **(a)** Isosurface extracted from the original $256 \times 256 \times 110$ mesh. **(b)** Same isosurface extracted from a mesh of dimensions $64 \times 64 \times 28$ obtained by averaging downsampling. **(c)** Same as (b) but using topology-guided downsampling. *(See also Figure C.8 on page 149 in the color plate section.)*

This rather complicated topological structure cannot be preserved by averaging downsampling (see Figure 5.10b), while topology-guided downsampling preserves many more of the components of a typical isosurface (see Figure 5.10c). Of course, preservation of noise is not always desirable. However, for the purpose of previewing isosurfaces that might be distorted by noise the preservation of these topological features is very useful in order to recognize noise in isosurfaces even in a coarse preview.

Bonsai

The third and last example is a CT scan of a bonsai, which features a sharp but very complex border between air and the plant with many fine details. Figure 5.11a depicts the whole isosurface. The mesh's resolution of $256 \times 256 \times 128$ vertices is high enough to reconstruct single leaves.



(a) (b) (c)

Figure 5.11 **(a)** An isosurface extracted from a CT scan of a bonsai. **(b)** Same isosurface but extracted from a mesh of dimensions $32 \times 32 \times 16$ obtained with averaging downsampling. **(c)** Same as (b) with topology-guided downsampling. *(See also Figure C.9 on page 149 in the color plate section.)*

This way of representing a tree is related to shape modeling techniques based on voxelized scenes (see [26]). Again, we will show that topology-guided downsampling preserves more details of the shape for higher downsampling rates, which is crucial for this kind of applications.

Figures 5.11b and 5.11c show the same isosurface after three downsampling steps. While the shape is no longer recognizable after averaging downsampling in Figure 5.11b, topology-guided downsampling preserves a coarse representation of the original shape, as demonstrated by Figure 5.11c. (The isosurface in Figure 5.11c was clipped at the borders of the volume; this resulted in two dark holes.)

This example suggests that topology-guided downsampling is not only useful for scientific volume visualization but also for volume graphics, in particular if models have to be represented with different levels of detail.

5.3 Adaptive Volume Textures

One important advantage of simplicial meshes — and unstructured meshes in general — is their adaptive geometry, i.e., a locally adaptive resolution is possible and the mesh's boundary may be adapted to an arbitrary shape. On the other hand, structured meshes are less adaptive and, in particular, uniform meshes are almost perfectly non-adaptive. In order to compensate for this lack of adaptivity, uniform meshes usually have to have a higher resolution and a larger domain than unstructured meshes for the same problem. This, however, results in large mesh sizes, which are inappropriate for texture-based volume visualization algorithms because of the usually quite limited texture memory of graphics hardware. Thus, the main advantage of uniform meshes, i.e., the support by texturing hardware, is often severely diminished.

In order to overcome this dilemma and to offer at least a limited form of adaptivity in the two-fold sense of a locally adaptive resolution and an arbitrary boundary of the mesh, this section presents a two-level representation of mesh data that is appropriate for hardware-accelerated on-the-fly decoding with programmable texturing hardware. This kind of texture compression is particularly well suited for the texture-based volume rendering algorithms discussed in Sections 2.3 and 5.1. This section was first published together with Thomas Ertl in [34].

5.3.1 Adaptive Texture Mapping in Two Dimensions

Before discussing the case of adaptive volume textures in Section 5.3.2, the presented approach to adaptive texture mapping is discussed for two-dimensional textures in this section. Note that the example presented in this section is for the purpose of illustration only. In fact, the approach appears to be much more useful in three than in two dimensions; however, an explanation of the two-dimensional case is likely to be more comprehensible than an immediate discussion of the three-dimensional case.

This section begins with a specification of requirements for adaptive texture mapping, which are in fact restrictive enough to determine the basic data structures. These data structures lead to the decoding scheme, i.e., the algorithm for a texture lookup. Furthermore, a prototypical implementation of the decoder in current programmable graphics hardware, namely the ATI Radeon 8500, is presented and the generation of adaptive textures is discussed.

Requirements

As mentioned, the primary goal is an adaptive representation of texture data that features a locally varying resolution and an adaptive boundary of the texture's domain. Moreover, the decoder should be implemented within the rasterization pipeline of off-the-shelf programmable graphics hardware, such that the texture data is decoded "on the fly" for each texture lookup. This approach allows for a fast random access, which is usually required for texture decoding techniques (see [1, 4, 49]).

As the flexibility of current programmable graphics hardware, e.g., the ATI Radeon 8500 or NVIDIA's GeForce3, is rather limited, the data structures are limited to only one level of indirection, which can be implemented with dependent texture mapping. I.e., the data structures may include references but no nested references.

Furthermore, different interpolation schemes should be supported (see Section 2.7); in particular, nearest-neighbor interpolation and a continuous interpolation, i.e., linear, bilinear, or trilinear interpolation.

Representation of Adaptive Textures

In the context of meshing, there are basically two approaches to locally adaptive resolutions: hierarchical meshes and unstructured meshes. The latter are not suitable here because of the problem of cell location, which is part of any random access in an unstructured mesh. On the other hand, hierarchical meshes are also inappropriate as any hierarchy implies the need for nested references, which have to be avoided here. The solution proposed in the following is to employ a hierarchical representation that is restricted to just two levels. Thus, a restricted form of locally adaptive resolution can be retained and at the same time the nesting of references is avoided.

The first (upper) level of our representation is a coarse uniform mesh covering the domain of the texture. The data defined on this mesh will be called the *index data*. For each cell, these data consist of one reference to the texture data of the cell, which is called a *data block*, and a scaling factor specifying its resolution relatively to the maximum resolution; an example is given in Figure 5.12a. The second (lower) level contains the actual data blocks remapped to a uniform resolution such that all data blocks may be packed into one uniform mesh; therefore, these data will be called the *packed data*; see Figure 5.12b for an example. Although the cells of the coarse mesh are of uniform size, the packed data blocks are of different sizes depending on their resolution, i.e., data blocks of a high resolution will correspond to large blocks of the packed data because of the remapping to a uniform resolution.

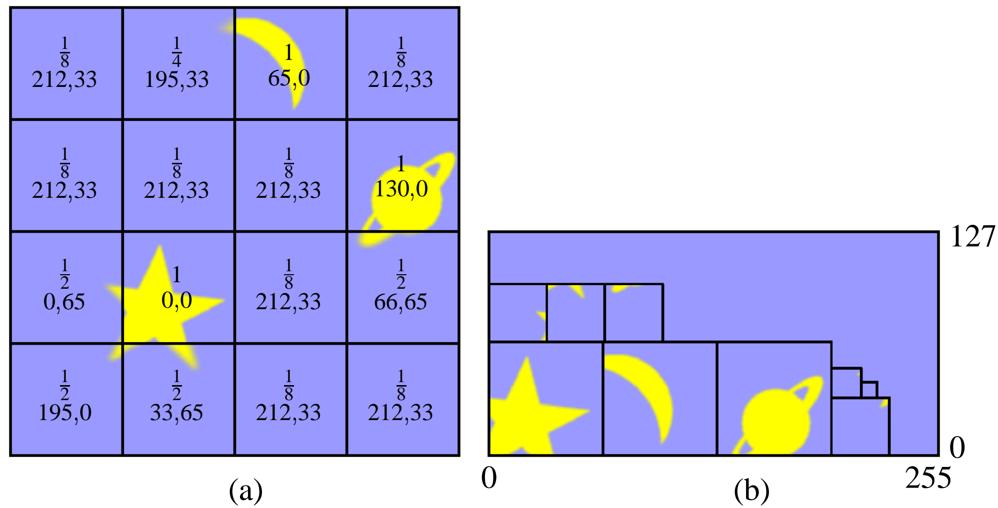


Figure 5.12 Representation of adaptive textures. **(a)** Index data: scale factors and coordinates of packed data blocks are stored for each cell of a 4×4 mesh representing the whole texture, which is included for the purpose of illustration only. (Actual coordinates are between 0 and 1.) **(b)** Packed data: the data blocks packed into a uniform mesh of 256×128 texels. The blocks' frames are for illustration purposes only. *(See also Figure C.10 on page 150 in the color plate section.)*

For a continuous interpolation of the texture data, we replicate the texels of the data blocks' boundaries and employ bilinear interpolation of the texels' data. This corresponds to the *space-filling block arrangement* suggested by Ning and Hesselink in [50] and is illustrated in Figure 5.13. Note that (in contrast to the OpenGL definition of textures) texel values are specified at vertices and the domain of valid texture coordinates is limited by the vertices' positions in order to allow for the bilinear interpolation. For a block of size $b \times b$ texels, the replication of boundary texels causes an increase in the amount of data by a factor of about $(b+1)^2/b^2$, e.g., an acceptable memory overhead of 13 % for image blocks of size 16×16 texels.

Neither the mentioned requirements nor the chosen data structures impose any restriction on multiple references to a single data block. In particular, it is useful to always include an *empty data block*, which is referenced (at least) by all cells of the coarse mesh outside the domain of the texture. All texels of the empty data block are set to an *empty texel value*, which depends on the particular application. For color images this value is usually the background's color or a completely transparent color. As the empty data block is perfectly homogeneous, it may be stored with the minimum resolution, i.e., the minimum block size.

Before presenting one particular way of computing the index data and packed data, the sampling of adaptive textures and its implementation is discussed in the

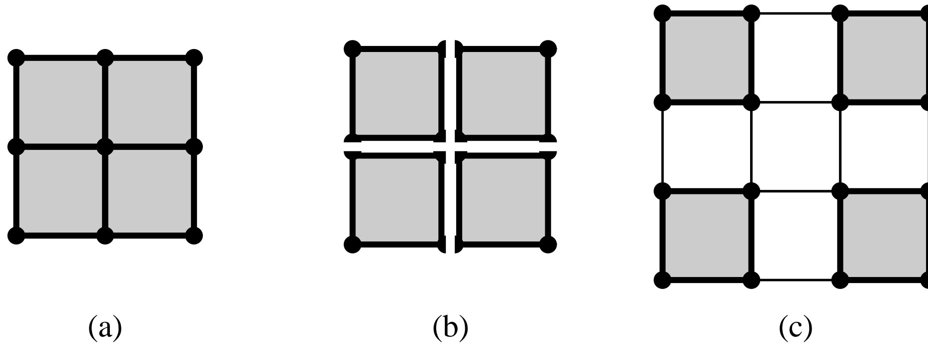


Figure 5.13 Decomposition of a 3×3 mesh into four blocks. Bilinear interpolation may be employed within the gray regions. **(a)** The original mesh. **(b)** Decomposition into four 2×2 blocks. Vertices at the blocks' boundaries are replicated. **(c)** Packing of the four blocks into one 4×4 mesh.

next two sections since the efficient sampling of adaptive textures is the most important requirement for the chosen representation.

Sampling of Adaptive Textures

A texture lookup in an adaptive texture at texture coordinates (s,t) (see also Figure 5.14) is performed in five steps:

1. determination of the cell of the index data that includes the point (s,t);
2. computation of the coordinates (s_o,t_o) corresponding to the origin of this cell;
3. lookup of the index data for this cell, i.e., of the associated scale factor m and the origin (s'_o,t'_o) of the data block in the packed data;
4. computation of the coordinates (s',t') in the packed data corresponding to (s,t) in the index data; and
5. lookup and interpolation of the actual texture data at (s',t') in the packed data.

The dimensions of the index data are denoted by n_s and n_t, i.e., there are $n_s \times n_t$ cells in the coarse mesh of the upper level of the adaptive texture's representation; n'_s and n'_t are the dimensions of the packed data, i.e., all data blocks are packed into a mesh of $n'_s \times n'_t$ texels. Additionally, the maximum resolution of a data block is defined by a maximum size of $b_s \times b_t$ texels. Note that replicated texels are included in b_s and b_t, e.g., the data blocks of Figure 5.13c are of size 2×2

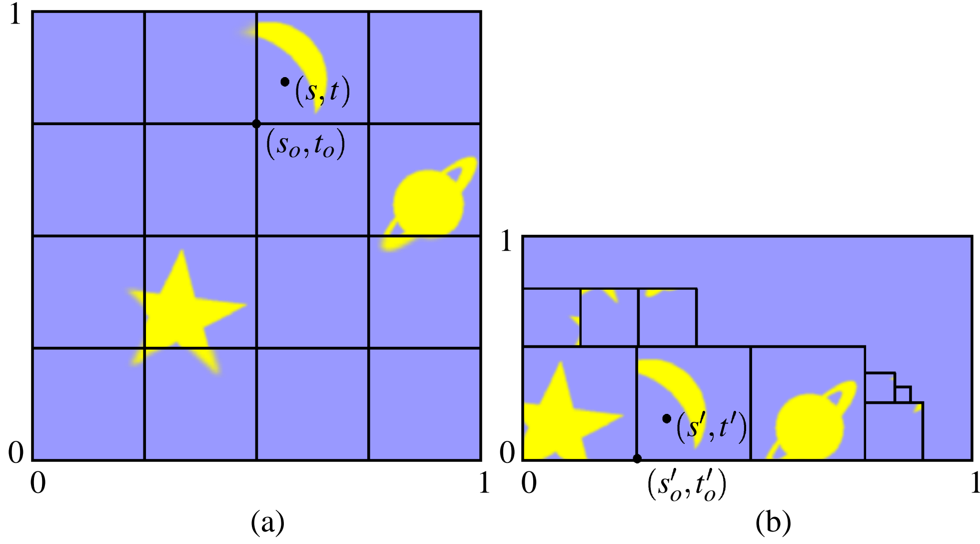


Figure 5.14 Texture lookup in an adaptive texture for texture coordinates s and t. **(a)** (s,t) specifies a cell of the index data, the origin of which is denoted by (s_o,t_o). **(b)** (s_o,t_o) corresponds to the origin (s'_o,t'_o) of a packed data block and (s,t) corresponds to a point (s',t') in that data block.

although the area for interpolation is only of the size of one texel, or in general of $(b_s-1)\times(b_t-1)$ texels. The scale factor m will be set to 1 for this maximum resolution. These definitions are particularly convenient if the adaptive texture is derived from a uniform texture image of size $(n_s(b_s-1))\times(n_t(b_t-1))$ because the maximum size of a data block is limited to $b_s\times b_t$ in this case.

With these definitions the origin (s_o,t_o) of the cell including the point (s,t) can be computed by

$$s_o = \frac{\lfloor s\, n_s \rfloor}{n_s} \quad \text{and} \quad t_o = \frac{\lfloor t\, n_t \rfloor}{n_t}, \tag{5.1}$$

where the floor function $\lfloor x \rfloor$ gives the largest integer less than or equal to x. The scale factor m and the origin (s'_o,t'_o) of the corresponding packed data block are given as functions of (s_o,t_o). Thus, the texture coordinates (s',t') in the packed data can be computed by

$$s' = s'_o + (s-s_o)m\frac{n_s(b_s-1)}{n'_s} \quad \text{and} \tag{5.2}$$

$$t' = t'_o + (t-t_o)m\frac{n_t(b_t-1)}{n'_t}, \tag{5.3}$$

i.e., the offset $(s,t)-(s_o,t_o)$ is scaled with m and two additional factors, and this scaled offset is added to the origin (s'_o,t'_o). (The reciprocal factors in Equation (2) and (3) in [34] are incorrect.)

These additional factors $(n_s(b_s - 1))/n'_s$ and $(n_t(b_t - 1))/n'_t$ stem from the scaling of all texture coordinates to the range between 0 and 1. In the example depicted in Figure 5.14, m is 1, n_s and n_t are 4, b_s and b_t are 65, n'_s is 256, and n'_t is 128. Thus, the factor $m\ (n_s(b_s - 1))/n'_s$ equals 1, but the factor $m\ (n_t(b_t - 1))/n'_t$ equals 2. Although the offsets $(s,t) - (s_o,t_o)$ and $(s',t') - (s'_o,t'_o)$ in Figure 5.14 appear to be equal on first sight, they are actually different as the coordinate system in Figure 5.14b is only half the height of that in Figure 5.14a. The factor $(n_t(b_t - 1))/n'_t = 2$ takes care of exactly this difference.

Implementation of Adaptive Texture Sampling

This section discusses an implementation of the texture lookup described in the previous section on ATI's Radeon 8500 using the "fragment shader" extension (see [25]). The two-dimensional texture lookup could also be implemented on NVIDIA's GeForce3 with the help of the "texture shader" extension (see [28]); however, this possibility is not discussed here since the required texture shader programming is less straightforward.

In the presented fragment shader implementation, the mesh dimensions n_s, n_t, n'_s, and n'_t are restricted to powers of two as these meshes are implemented with OpenGL textures. In particular, there is one texture of size $n_s \times n_t$ for the index data and one texture of size $n'_s \times n'_t$ for the packed data. While the latter employs bilinear interpolation and contains the actual image data, the texture for the index data uses nearest-neighbor interpolation and contains for each texel one pair of coordinates (s'_o,t'_o) specifying the origin of the corresponding data block in the texture for the packed data and one scale factor m, i.e., three components per texel, which can be stored in an RGB texture. Note that the limitation to 8 bits of precision for the specification of s'_o and t'_o limits n'_s and n'_t to a maximum value of 256. The scale factor m is restricted to values of the form 2^{-n} with $0 \leq n \leq 7$ as it is also specified by 8 bits. Therefore, the data blocks' dimensions are restricted to values of the form $2^n + 1$ with an integer n greater than or equal to 0, where the term $+1$ stems from the replication of texels at block boundaries.

A fragment shader program on the ATI Radeon 8500 consists of either one or two "passes" each consisting of up to six texture sampling and/or texture coordinate routing instructions followed by up to eight arithmetic instructions. Only the sampling instructions of the second pass may be dependent texture lookups, i.e., their texture coordinates may be the results of previous instructions. This makes a total of four blocks of instructions, which are illustrated in Figure 5.15. The particular instructions in the four blocks of the proposed fragment shader program are discussed in the next three paragraphs.

The first block of texture sampling instructions has to fetch the scale factor m and the coordinates s'_o and t'_o by one nearest-neighbor lookup in the $n_s \times n_t$

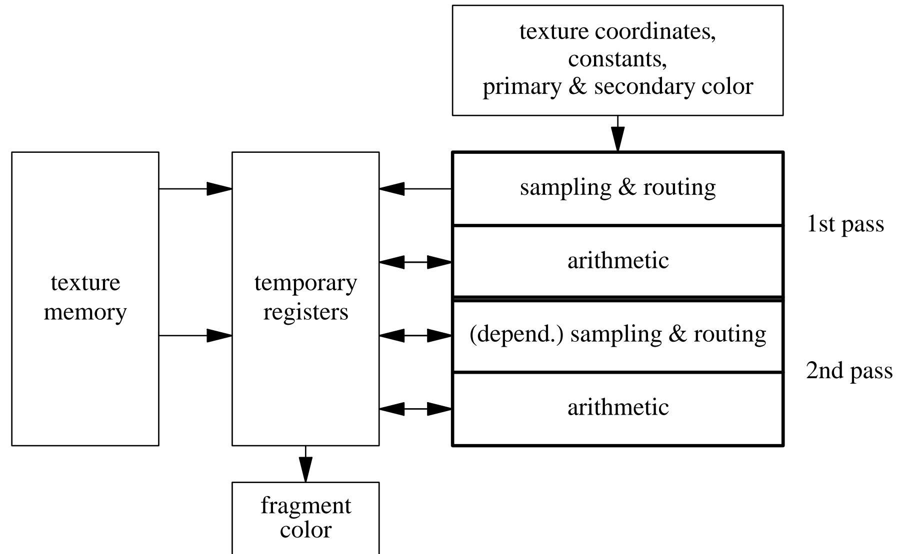


Figure 5.15 Scheme of the structure of a two-pass fragment shader program, its inputs, temporary registers, and the resulting fragment color.

index data mesh at texture coordinates s and t. Additionally, the coordinates s_o and t_o corresponding to the origin of the fetched texel (see Equation (5.1)) may be computed from s and t by a second nearest-neighbor texture lookup in another $n_s \times n_t$ texture containing coordinates $(i/n_s, j/n_t)$ in the (i, j)-th texel with $0 \leq i < n_s$ and $0 \leq j < n_t$. Alternatively, s_o and t_o may be computed separately by two texture lookups in two one-dimensional textures containing values i/n_s in the i-th texel with $0 \leq i < n_s$ and j/n_t in the j-th texel with $0 \leq j < n_t$, respectively.

The following block of arithmetic instructions computes the texture coordinates s' and t' for the lookup in the packed data as described above; see in particular Equations (5.2) and (5.3). Note that the terms $(n_s(b_s - 1))/n'_s$ and $(n_t(b_t - 1))/n'_t$ are constant for all texels; therefore, the computation reduces to one subtraction, two multiplications, and one addition for each coordinate, which can be performed by four fragment shader instructions since these vector instructions may affect up to four components (red, green, blue, and alpha). Moreover, one of the products and the final sum can be evaluated by a single “MAD” (multiply and add) instruction. However, an additional addition is necessary as the center of the (i, j)-th texel in the $n'_s \times n'_t$ texture is at the coordinates $((i + 1/2)/n'_s), (j + 1/2)/n'_t)$; thus, the constant vector $(1/(2n'_s), 1/(2n'_t))$ has to be added to (s', t'). Furthermore, additional operations are necessary in order to reduce artifacts stemming from the limited precision of the fragment shader’s arithmetic. In particular, decreasing m slightly (and at the same time increas-

ing $(1/(2n'_s), 1/(2n'_t)))$ helps to reduce artifacts significantly as this allows us to restrict the computed coordinates s' and t' to the correct packed data block. However, these modifications introduce new artifacts by causing additional discontinuities at block boundaries. A strong reduction of these artifacts is to be expected on future graphics hardware supporting more accurate arithmetic.

The second block of texture sampling instructions employs the coordinates s' and t' for a bilinearly interpolated texture lookup in the texture containing the packed data blocks. This completes the computations of the fragment shader program. Of course, additional texture lookups and arithmetic instructions are possible in the second pass, e.g., in order to blend the resulting color with the primary color and/or with colors resulting from further (standard) texture lookups.

Generation of Adaptive Textures

In order to complete the discussion of two-dimensional adaptive textures, one particular way of generating them from data defined on uniform meshes is presented next. If the data is not specified on a uniform mesh but on an unstructured mesh or in parametric form, it has to be resampled to a uniform mesh in order to apply the techniques presented in this section. Optimizations will not be discussed since the generation of an adaptive texture will usually be a pre-processing step.

In addition to the nomenclature introduced above, the dimensions of the original uniform mesh are denoted by N_s and N_t. If one of these dimensions is not a power of two, it has to be increased to the next greater power of two. In this case, the additional texels should be set to the empty texel value such that the additional empty regions are encoded efficiently.

The algorithm for generating an adaptive texture consists of the following steps, which are discussed in detail below (compare also with Figure 5.12):

1. Build a hierarchy of downsampled versions of the original mesh with the mesh of the i-th level being of size $2^{-i}N_s \times 2^{-i}N_t$ vertices.

2. Given the maximum data block size $b_s \times b_t$ introduced above, decompose the original mesh, i.e., the 0-th level of the hierarchy, into $n_s \times n_t$ cells of size $b_s \times b_t$ using the replication of boundary vertices explained above.

3. For each of the cells of step 2, test whether the data values of the cell are "sufficiently" close to the empty data value. In this case, mark the cell as empty; otherwise, determine an "appropriate" scale factor $m = 2^{-i}$ and copy a corresponding data block of size $(m(b_s - 1) + 1) \times (m(b_s - 1) + 1)$ from the data of the i-th level of the meshes' hierarchy.

4. Build a list of data blocks created in the previous step and append an empty data block, which is referenced by all marked cells.

5. Ensure consistent block boundaries by modifying the data blocks of neighboring cells such that data on shared boundaries is identical. This may be performed, for example, with the method proposed by Westermann et al. in [75]. However, the empty data block must not be modified.

6. Pack all data blocks into a mesh of size $n'_s \times n'_t$, which represents the packed data of the adaptive texture.

7. Based on the cells' references to data blocks established in steps 3 and 4, the scale factors of step 3, and the positions of the packed data blocks computed in step 6, assemble the cells' data in an $n_s \times n_t$ mesh, which represents the index data of the adaptive texture.

The downsampling of step 1 should include some filtering in order to minimize approximation errors, e.g., a simple averaging of neighboring vertices may be employed. The implementation may be simplified by choosing the vertex positions of a downsampled mesh from the vertex positions of the original mesh. In this case, it is advantageous to choose dimensions of the form $2^n + 1$ for N_s and N_t. Therefore, the size of the i-th mesh should be $(2^{-i}(N_s - 1) + 1) \times (2^{-i}(N_t - 1) + 1)$.

In order to choose appropriate dimensions b_s and b_t of the cells of step 2, several dependencies should be considered: The larger the cells are, the smaller is the amount of index data and the smaller is the memory overhead due to replicated boundary texels. On the other hand, the data representation becomes more adaptive with smaller cells, i.e., a larger region of the texture's domain may be covered by the memory-efficient empty data blocks and the resolution of individual cells may be adapted to local features more efficiently. For this reason, the cells depicted in the illustrative Figures 5.12a and 5.14a are far too large. The optimal choice depends not only on the size, shape, and dimensions of the texture's domain but also on the particular texture data. Therefore, the cells' dimensions should be optimized for each particular application or even for each texture.

In step 3, one possible criterion for data values "sufficiently" close to the empty data value is a user-specified limit of the L^∞-norm (maximum norm) of the difference between the interpolated data within the cell and the empty data value. The "appropriate" scale factor $m = 2^{-i}$ may be determined by calculating the L^∞-norm or the L^2-norm (depending on the application) of the difference between the interpolated data of the 0-th and the i-th level and choosing the maximum i that results in a norm still smaller than a user-defined limit. Of course, any other criterion could be employed, e.g., a position-dependent metric.

It should be noted that the proposed algorithm does not produce multiple references to data blocks apart from the references to the empty data block. If multiple references to other data blocks were allowed, it would be far more complicated

to guarantee identical data on shared cell boundaries, which are required for continuous interpolation. Moreover, this requirement is likely to restrict the use of multiple references to data blocks that feature only constant values; however, these blocks are already efficiently compressed because strongly downsampled versions of them are generated in step 3.

Step 6 requires an approximate solution to a variant of the well-known bin-packing problem. In this context, a simple recursive procedure is sufficient that fills a rectilinear empty region with blocks of only one size and calls itself recursively for the remaining empty region, which is decomposed into rectilinear parts. The block size is determined by searching for the largest unpacked block that still fits into the empty region.

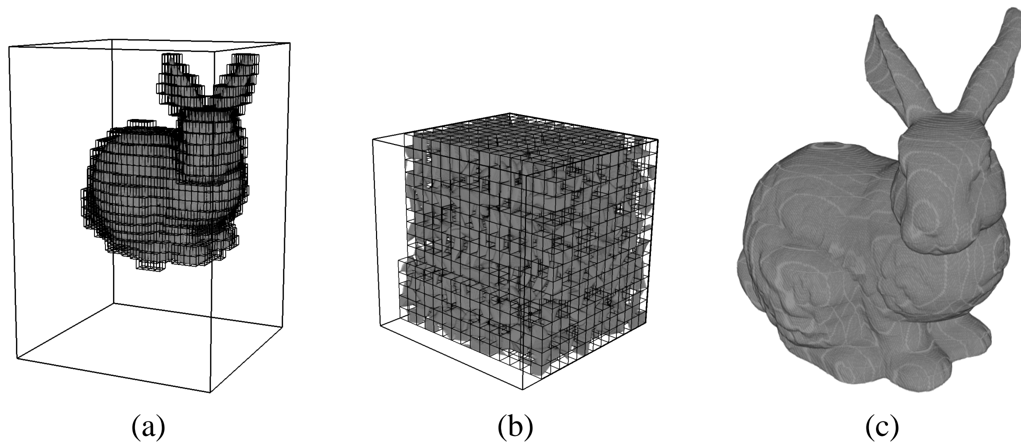
All steps of this algorithm may be generalized to three and more dimensions without complications. However, the implementation of the texture sampling to more dimensions is less obvious; therefore, it will be discussed in detail in the next section. Once more, it should be emphasized that this approach to adaptive texture mapping appears to be less useful for two-dimensional applications; in particular because there are more appropriate methods that are based on triangulations of texture images.

5.3.2 Volume Rendering with Adaptive Volume Textures

Volume data may be rendered with the help of three-dimensional texture mapping by blending a set of view-plane-aligned, textured slices into the frame buffer as discussed in Sections 2.3 and 5.1, and illustrated in Figure 2.4b. However, one of the main problems of this technique is the large amount of texture memory necessary for standard volume textures. Adaptive texture mapping, on the other hand, compresses the texture data and, therefore, allows us to employ three-dimensional texture mapping for considerably larger data sets.

The approach discussed in Section 5.3.1 is easily generalized to three dimensions, provided that the graphics hardware supports three-dimensional RGBA textures. Moreover, one more one-dimensional texture lookup is required in the first pass of the fragment shader program to determine the three texture coordinates of the origin of the current voxel in the index data since it is usually not practical to employ a (possibly large) three-dimensional texture for this purpose.

In Figures 5.16c, a volume rendering of the $512 \times 512 \times 360 \times 2$ bytes CT scan of the Stanford terra-cotta bunny is depicted. In order to generate the adaptive representation, the data was first converted to a $512^3 \times 1$ byte volume. Then, most of the noise was removed before computing the index and packed data in a variant of the algorithm described above for three-dimensional data. The non-empty cells of the resulting index data, which is stored in a small 32^3 RGBA texture, are depicted in Figure 5.16a. Figure 5.16b visualizes the packed data blocks, which fit



(a) (b) (c)

Figure 5.16 Volume rendering of a $512 \times 512 \times 360$ CT scan with adaptive texture mapping. **(a)** Non-empty cells of the 32^3 index data mesh. **(b)** Data blocks packed into a 256^3 texture. **(c)** Resulting volume rendering. *(See also Figure C.11 on page 150 in the color plate section.)*

into a 256^3 texture with luminance and alpha components requiring 32 MB of texture memory. A prototypical volume renderer based on the proposed algorithms achieves a performance of about 6 frames per second for a 512×512 image with about 500 slices on an ATI Radeon 8500 graphics board.

Higher frame rates are possible for smaller images, which are, however, not appropriate for a volume of virtually 512^3 voxels. As mentioned above, it appears to be impossible to remove all rendering artifacts at block boundaries on current hardware. In Figure 5.16c, these artifacts manifest themselves as rather large, lighter rings.

Margaret: *So what's it like?*
David: *What?*
Margaret: *Out there.*
David: *Well, it's ... it's louder, and ... scarier, I guess. And it's ... a lot more dangerous.*
Margaret: *Sounds fantastic.*

Dialog from the movie *Pleasantville*

Chapter 6

Geometrically Unpleasant Meshes in General

One result of this work is the identification of geometrically unpleasant features in basically any kind of mesh — at least with respect to volume visualization. This implies that there is probably no perfectly pleasant mesh for volume visualization. However, the existence of such a mesh is not very relevant since the decision for a particular mesh type is seldomly determined by the requirements of the visualization process in real-life applications. Thus, direct volume visualization will always have to deal with a variety of meshes, which implies the need for a variety of algorithms for different kinds of meshes with different, more or less unpleasant features.

Several of these volume visualization algorithms have been discussed in the previous chapters with a special emphasis on the restrictions on the geometry of the visualized meshes. For some of these algorithms it was demonstrated how to overcome particular restrictions (corresponding to particular unpleasant features of meshes) and, therefore, how to apply an algorithm to a larger class of meshes. In several cases, a generalization was not possible but new algorithms had to be developed in order to apply at least the underlying concepts of an algorithm to other classes of meshes. It is very likely that there are many more well-known algorithms for volume visualization that are overly restrictive with respect to the kinds of meshes they can process. Therefore, research on the generalization of volume visualization algorithms and concepts appears to be a particularly promising area for future research in volume visualization. In fact, it is one of the main goals of this work to suggest and motivate more research in this direction.

Of course, there are several reasons for the lack of more general algorithms and concepts: Research in volume visualization focuses (too?) often on specific applications and, therefore, on particular kinds of meshes without a strong motivation to generalize algorithmic solutions. Moreover, the generalization of algorithms and their prototypical implementations often requires considerably more effort and resources than specific solutions. The generalization of a underlying concept is often even more difficult to achieve. Furthermore, restrictions of volume visualization algorithms to certain kinds of meshes are (almost traditionally) played down in publications, thereby indirectly hindering the subsequent publication of more general solutions, which are often less efficient. On the other hand, this work led to several successful publications (in particular [18, 31, 32, 34, 35, 56, 69]) proving that research on this subject is definitely worthwhile.

Since the generalization of algorithms for volume visualization is a necessarily unlimited research topic, it was certainly not possible to cover it completely in this work. Thus, there are many known (and unknown!) further problems; a selection is given in Section 6.3. In order to help solving these problems, the following Section 6.1 summarizes the solutions proposed in the previous chapters, and Section 6.2 tries to generalize the ideas that led (or could have led) to these algorithms. More specifically spoken, several problem-solving strategies and their applications in this work are presented in order to demonstrate in which way these abstract strategies may be (and actually were) helpful for the development of algorithms for direct volume visualization.

6.1 Proposed Solutions

This section briefly summarizes the new algorithms presented in this work since these solutions (and their development) serve as examples for the problem-solving strategies discussed in Section 6.2.

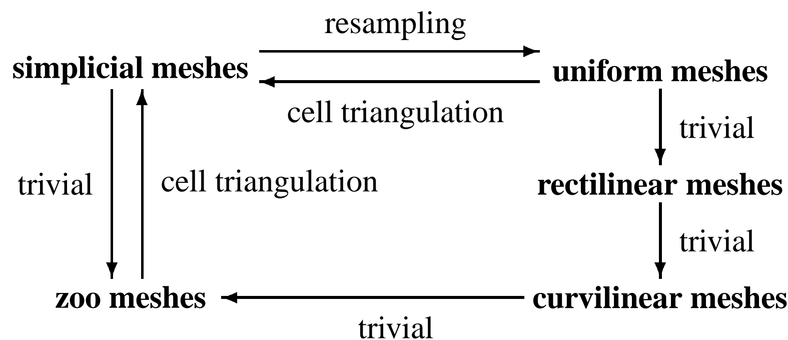


Figure 6.1 Trivial and common conversions between different kinds of meshes. (See Section 2.6 for a description of these meshes.)

Although the proposed solutions generalize previously published algorithms, they are usually restricted to either simplicial or uniform meshes. Uniform meshes are of particular interest because of their widespread use and the hardware-support by means of three-dimensional textures, while simplicial meshes offer the possibility of a continuous, piecewise linear interpolation. Moreover, many meshes are easily converted to simplicial meshes by triangulating (i.e., tetrahedralizing in three dimensions) all cells; see Figure 6.1, which summarizes trivial and common conversions between the kinds of meshes introduced in Section 2.6. Note that the direct conversions of rectilinear and curvilinear meshes to simplicial meshes by cell triangulation are implied by the trivial conversions to zoo meshes.

Figure 6.2 organizes the new algorithms presented in this work according to the kind of processed mesh (simplicial or uniform) and the kind of geometrically

unpleasant features of ...	... simplicial meshes	... uniform meshes
... individual cells	non-uniform cells ⇒ no native hardware-support ⇝ **pre-integrated cell projection (3.1)** ⇝ **hardware-based ray casting (3.3)**	non-simplicial cells ⇒ no linear interpolation within cells ⇒ pre-integration not applicable ⇝ **texture-based pre-integrated volume rendering (5.1)** ⇒ critical points not only at vertices ⇝ **topology-guided downsampling (5.2)**
... groups of cells	visibility cycles ⇝ **cell sorting for cyclic meshes (4.3)** ⇝ **cell projection for cyclic meshes (4.4)**	non-adaptive resolution ⇝ **adaptive volume textures (5.3)**
... the mesh's boundary	non-convex boundary ⇝ **edge collapses in non-convex meshes (4.2)**	non-adaptive boundary ⇝ **adaptive volume textures (5.3)**

Figure 6.2 New solutions proposed in this work (in **boldface**; references to the corresponding sections are given in parentheses).

unpleasant feature, which led to the development of each algorithm. The employed classification of geometrically unpleasant features distinguishes between features associated with the geometry of individual cells (here: a non-uniform or non-simplicial shape), the geometry of groups of cells (here: the existence of visibility cycles or of a non-adaptive resolution), and the geometry of the mesh's boundary (here: a non-convex or non-adaptive shape).

6.2 Problem-Solving Strategies

Before discussing particular problem-solving strategies and their applications to direct volume visualization some remarks are in order: The strategies mentioned in this section were chosen because they turned out to be useful in this work — or at least could have been useful. Thus, it is no coincidence that there is an exemplary application for each of them in this work. Most strategies were taken from [41], but no attempt was made to track references to the original publications. Usually, a single strategy is not sufficient to solve a problem, but a couple of strategies have to be combined to develop a solution. Also note that — regardless of the chosen strategy — it is always reasonable to seek help in the form of publications, experts, or available software implementations; publications from other subjects and/or disciplines are sometimes of particular interest. It should also be noted that strategies to find an optimal solution or to find a solution in optimal time were not included.

The strategies are grouped (compare [41]) into strategies helping to understand (Section 6.2.1), simplify (Section 6.2.2) and actually solve (Section 6.2.3) problems. Of course, the power of problem-solving strategies is limited. In fact, many algorithms (including several of the algorithms presented in this work) had to evolve slowly over time or were based on accidents, misunderstandings, or sudden inspirations. In other words, there is no guarantee that the following strategies lead to a good — or any — solution; however, they usually serve as a good starting point.

6.2.1 Understanding the Problem

The first step in solving a serious problem should be to gain a thorough understanding of it. This may include

- isolating,
- clarifying,
- formulating,

- defining, and
- visualizing

the problem. Often it is helpful to

- explain the problem to someone else,
- think of specific examples and extreme cases,
- imagine oneself to be part of the problem or its solution, or to
- search for key elements.

In order to understand the nature of a geometrically unpleasant feature of a mesh, it might be helpful to answer questions like:

- Why, exactly, is a particular mesh unpleasant?
- In what kind of meshes does this unpleasant feature occur? (If there are meshes in which it does not occur: Why? Are there any analogous features in these meshes?)
- Is it a problem of a specific algorithm or its implementation? (Is it a problem of the algorithm's assumptions? Its time or memory efficiency? The available hardware-support?)
- What are the worst-case and the expected consequences? (How relevant is the problem?)

For example, since the publication of [56] it was well understood how to apply pre-integrated classification (see Section 2.5.1) to tetrahedral meshes (see Section 3.1), but it was not obvious how to use pre-integrated classification for uniform meshes and in particular texture-based volume rendering. The problem is the assumption of a piece-wise linear interpolation in tetrahedral meshes (see Section 2.7.3), which is not valid for uniform meshes as their cells are not simplicial.

Based on this insight, the obvious solution would be to convert any uniform mesh into a tetrahedral mesh by decomposing each cubic cell into five tetrahedra (for an example of this approach see Section 5.2) and employ pre-integrated cell projection as discussed in Section 3.1 for the volume rendering. However, this approach leads to serious performance problems as a rather small data set of 256^3 voxels would already correspond to 83,886,080 tetrahedra — far too many for a real-time interactive volume visualization based on cell projection.

A deeper understanding of the problem led to the conclusion that the missing piece-wise linear interpolation only impedes an exact ray integration. However,

this possibility is usually already excluded by the discretized evaluation of the ray integral employed in texture-based volume rendering. Because of the associated discretization error, it is often legitimate to assume an approximately linear interpolation between samples. This realization led to texture-based pre-integrated volume rendering as presented in Section 5.1 and published in [18].

6.2.2 Simplifying the Problem

Many problems are much easier to solve if they are simplified first. This section discusses three popular strategies for this task and several applications in this work.

Redefining the Problem

Reformulating a problem or taking a different look at a problem may already help to solve it (or at least to gain a better understanding of it). However, redefining a problem usually implies an even deeper change of perspective, which often has better chances of helping to solve a problem.

The mentioned example of pre-integrated classification for uniform meshes (see Section 5.1) also serves as an example of a redefined problem: Instead of projecting cubic cells (as suggested by pre-integrated cell projection; see Section 3.1), ray segments between samples are pre-integrated in order to exploit the advantages of texture-based volume rendering.

Another example is topology-guided downsampling discussed in Section 5.2: As there is no topology-preserving downsampling of uniform meshes, the problem was redefined — in this case relaxed — to an approximative preservation of critical points.

Generalizing the Problem

Generalizing a problem usually means to make it worse. However, whenever there is a solution available for the more general problem, generalizing a problem might directly lead to a solution.

For example, one very popular approach to direct volume visualization of curvilinear meshes is to forget about the structure of a curvilinear mesh and convert it to a tetrahedral mesh by decomposing the cells into tetrahedra. Thus, the specific connectivity of a curvilinear mesh cannot be exploited in the volume visualization algorithm and, therefore, one might expect that the task of volume visualization becomes more difficult. However, cell projection algorithms for tetrahedral meshes (see Section 3.1) work very well for the converted meshes, while it

appears to be difficult to exploit the particular structure of curvilinear meshes for the purpose of volume visualization.

Dividing the Problem

One of the classical approaches in problem solving is to split a difficult problem into several less difficult problems. (A special case of this approach is the concept of "divide and conquer" algorithms.)

Several examples for this strategy can be found in this work, e.g., the problem of projecting cyclic meshes is split into the problem of sorting cyclic meshes (Section 4.3) and actual projecting visibility cycles (Section 4.4). Topology-guided downsampling (Section 5.2) is another example: The problem of preserving as much of the topology of a scalar field as possible is divided into the computation of critical points and an algorithm for the selection of downsampled values that tends to preserve critical points.

A special case of this strategy is to divide a problem into the task of finding any solution and the optimization of this solution. For example, texture-based pre-integrated volume rendering (Section 5.1) improves image quality considerably but the costly pre-integration of transfer functions was a serious drawback until the acceleration of the pre-integration discussed in Section 2.5.2 was discovered.

Another popular variant of dividing algorithmic problems is to find an algorithm for special cases and then generalizing the solution. As mentioned before, many of the algorithms discussed in this work are examples for this strategy as they generalize existing algorithms to larger classes of meshes.

6.2.3 Solving the Problem

In practice, it is often difficult to distinguish between understanding, simplifying, and actually solving a problem. In fact, the classification does not only depend on the problem itself but also on the point of view taken by the problem solver. Thus, there is definitely some overlap between the following four strategies for actually solving a problem and the simplification discussed in the previous section.

Directly Attacking the Problem

Directly attacking a problem is (and probably should be) the most common and usually the first approach in problem solving. As the resulting algorithmic solutions usually lack a certain elegance, they are sometimes qualified as "naive", "brute-force", or "standard" solutions. This, however, should not hinder anyone from employing them if they are appropriate.

The most important supplementary strategy for this method is to seek help, in particular from experts and published literature: If there is a direct solution to a particular (not too obscure) problem, then this solution has probably been published before and — equally important — it may probably be found by searching for the problem.

For example, the problems of sorting and projecting visibility cycles (Sections 4.3 and 4.4) were solved long before this work for visibility cycles formed by polygons instead of tetrahedra. Thus, the problem of sorting and projecting cyclic tetrahedral meshes was reduced to an adaptation of these published solutions.

Another example is the implementation of ray casting algorithms using programmable graphics hardware (Section 3.3). Ray casting is a well established technique for direct volume visualization without hardware-acceleration; see [22]. Thus, an implementation of these algorithms using graphics hardware is anything but far-fetched. The difficulty is just that graphics hardware has not been flexible enough for this kind of algorithms before the advent of programmable graphics hardware. In fact, it is likely that the enormous progress of graphics hardware development creates exactly this kind of situation for a lot more algorithms in computer graphics.

Avoiding the Problem

Avoiding problems is in many cases much more effective than dealing with them. In the context of developing algorithms for direct volume visualization of geometrically unpleasant meshes, avoiding such meshes means to convert them to more pleasant meshes *before* visualizing them. Although the conversion itself might be an extremely difficult problem, the fundamental advantage of this strategy is that the volume visualization algorithm does not need any modification at all.

Examples include resampling of unstructured meshes (Section 3.2), convexification of non-convex meshes (Section 4.1), or the conversion of cyclic into acyclic meshes by splitting particular cells, which was not discussed in this work.

Masking the Problem

Masking or hiding a problem is similar to by-passing it or finding a work-around. Usually, this strategy implies a very specific treatment of a problematic case such that it may be processed without difficulties. The difference with avoiding the problematic case is that the problem still exists and requires particular treatment. However, this treatment is usually very localized and on a rather low level, such that the (larger) rest of the solution does not need to take care of it.

For example, the virtual tetrahedralization utilized for topology-guided down-sampling (Section 5.2) hides the cubic shape of the cells, such that the rest of the algorithm works on a uniform mesh in the same way it would work on a tetrahedral mesh.

A less obvious example is texture-based volume rendering with object-aligned slices (Section 2.3): By choosing particular slice directions, the problem of three-dimensional texture lookups is masked and appears to the graphics hardware as a problem of two-dimensional texture lookups.

Another example of on-the-fly masking is provided by adaptive volume textures (Section 5.3): While they are sampled very similarly to usual volume textures, i.e., by the specification of three texture coordinates, their internal representation overcomes some important disadvantages of traditional volume textures.

Ignoring the Problem

Ignoring a problem usually means to accept its unpleasant consequences. Although this approach does not appear to be very attractive, it can be very effective if the consequences are unimportant.

For example, the approximation error caused by assuming a linear interpolation in uniform meshes for texture-based pre-integrated volume rendering (Section 5.1) is often acceptable because of the discretization error of texture-based volume rendering. Another example would be to accept the incorrect rendering of visibility cycles of cyclic meshes, as these cycles occur rarely in real-life applications and the resulting artifacts are often hardly visible.

6.3 Further Problems

In order to conclude this chapter, some of the problems of direct volume visualization of geometrically unpleasant meshes that are not covered by this work should be mentioned.

First of all, a general solution to *all* problems stemming from any of the mentioned unpleasant features is usually not available. For example, the discussion of non-simplicial meshes in Chapter 5 is limited to uniform meshes, while the discussion of non-uniform meshes in Chapter 3 covers only meshes that can be converted to simplicial meshes. Moreover, there is hardly any reason to assume that the solutions proposed in this work are in any sense optimal.

Following the classification employed in Figure 6.2, several unpleasant geometric features of meshes that were not discussed in this work are easily identified. For example, further unpleasant features that are associated with the geometry of individual cells include: badly shaped cells (e.g., so-called “slivers”, see for exam-

ple [10]), polyhedral cells (see for example [5]), and cells with non-planar faces (see for example [45]).

Another unpleasant feature associated with the geometry of groups of cells are “unfair” intersections of cells, i.e., intersections of two cells that are neither a shared face, nor a shared edge, nor a shared point, e.g., so-called “T-vertices”.

Also, particular problems of hierarchical and hybrid meshes are not covered in this work. Moreover, the complete topic of mesh generation, e.g., from meshless point data, is out of this work’s scope.

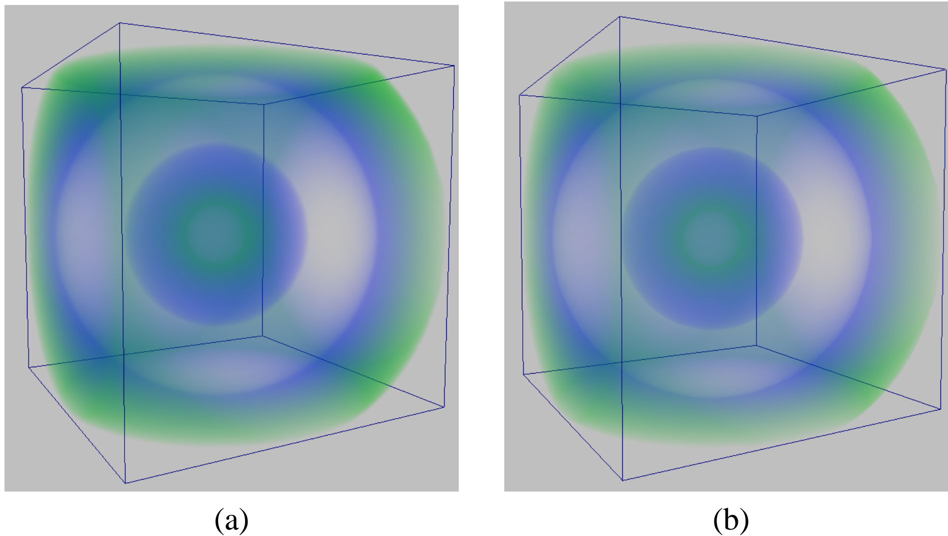
George: *So what’s gonna happen now?*
Betty: *I don’t know.*
Do you know what’s going to happen now?
George: *No. I don’t.*
Mr. Johnson: *I guess I don’t either.*

Dialog from the movie *Pleasantville*

Jennifer: *We're supposed to be in color!*

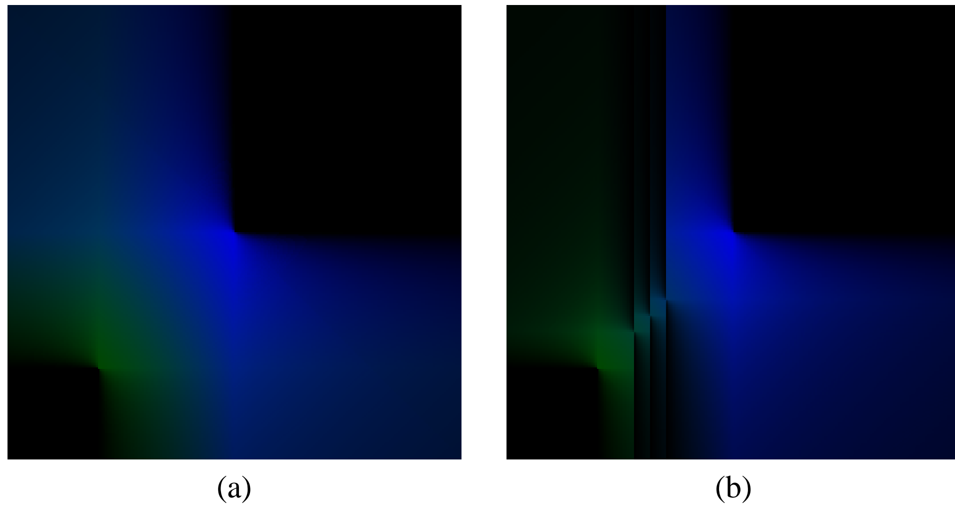
Quote from the movie *Pleasantville*

Color Plates



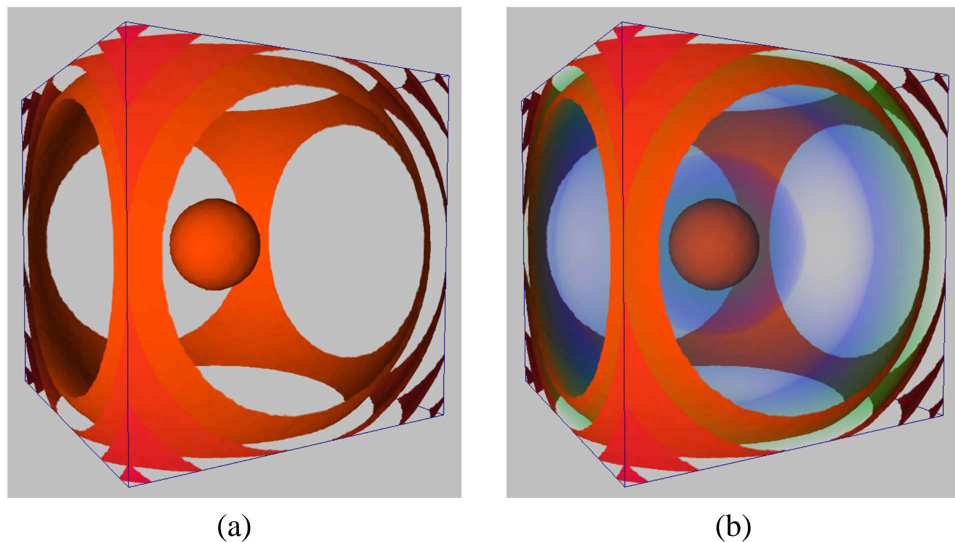
(a) (b)

Figure C.1 Visualization of a synthetic data set with non-linear transfer functions. **(a)** Pre-integrated classification with a three-dimensional texture of dimensions $64 \times 64 \times 64$ (1 MB). **(b)** Approximative pre-integrated classification with a two-dimensional texture of dimensions 256×256 (256 KB). *(See also Figure 3.3 on page 66.)*



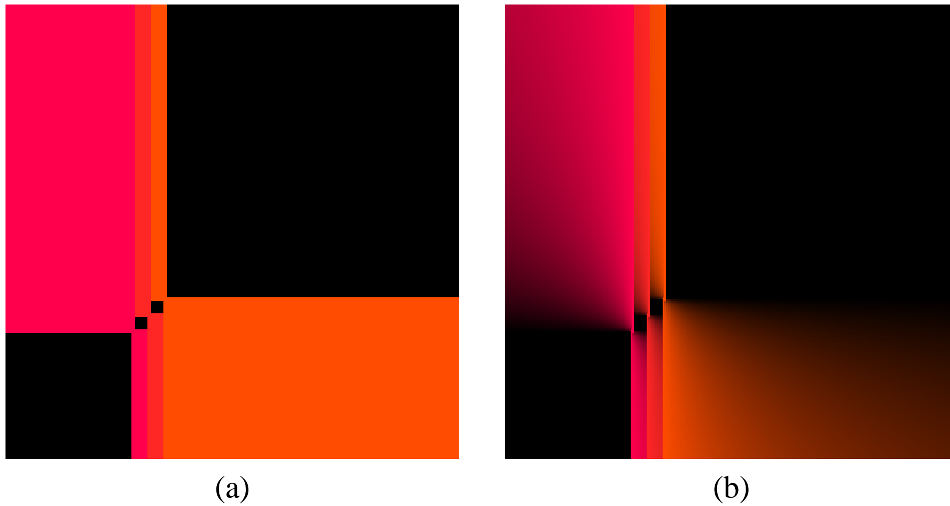
(a) (b)

Figure C.2 Textures for projected tetrahedra with pre-integrated classification. The horizontal (texture) coordinate is s_f, the vertical coordinate is s_b. Black pixels in these images correspond to completely transparent texels. **(a)** The texture employed for the semi-transparent volume in Figure 3.11b. **(b)** In this variant the integration is stopped at the isovalues, which correspond to opaque isosurfaces. *(See also Figure 3.5 on page 68.)*



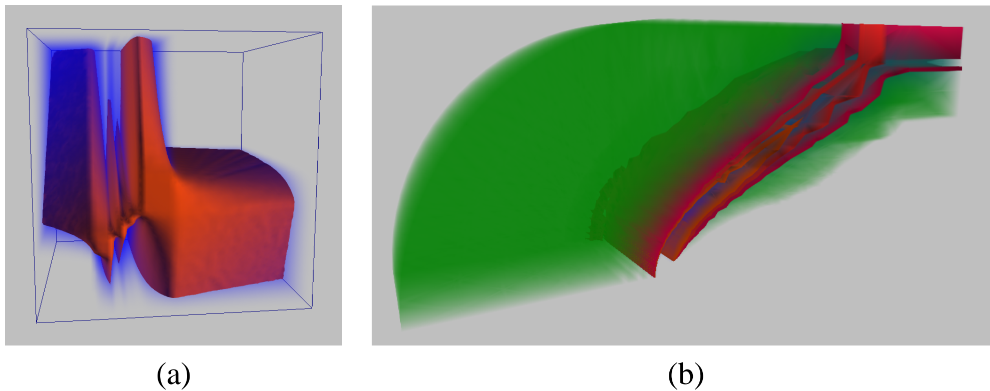
(a) (b)

Figure C.3 **(a)** Several isosurfaces extracted from the data set shown in Figures 3.3a and 3.3b. **(b)** Smooth combination of (a) with Figure 3.3b. *(See also Figure 3.9 on page 73.)*



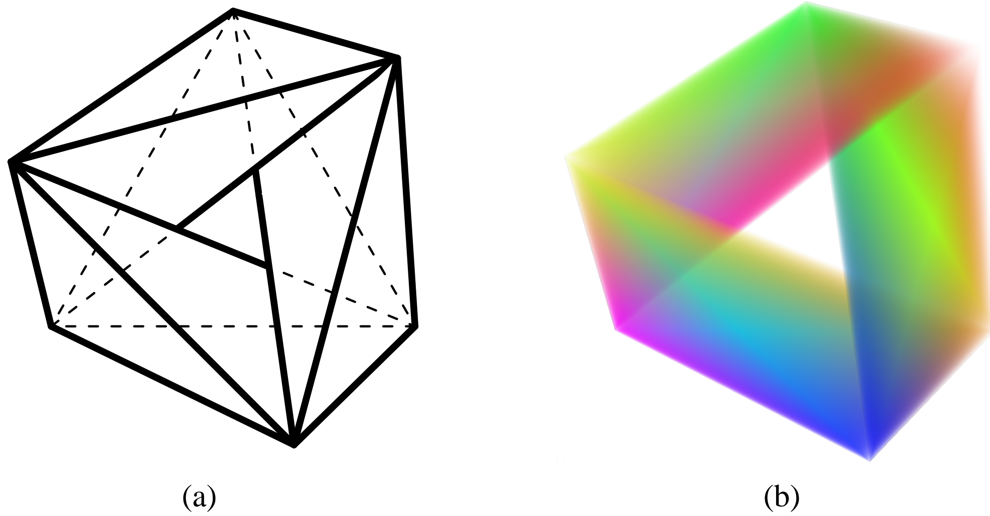
(a) (b)

Figure C.4 These two-dimensional textures of dimensions 256 × 256 were used to render the Bluntfin data set depicted in Figure 3.11b. The horizontal (texture) coordinate is s_f, the vertical coordinate is s_b. Back face triangles were textured with the image in **(a)**, while the texture in **(b)** was employed for the front face triangles. *(See also Figure 3.10 on page 74.)*



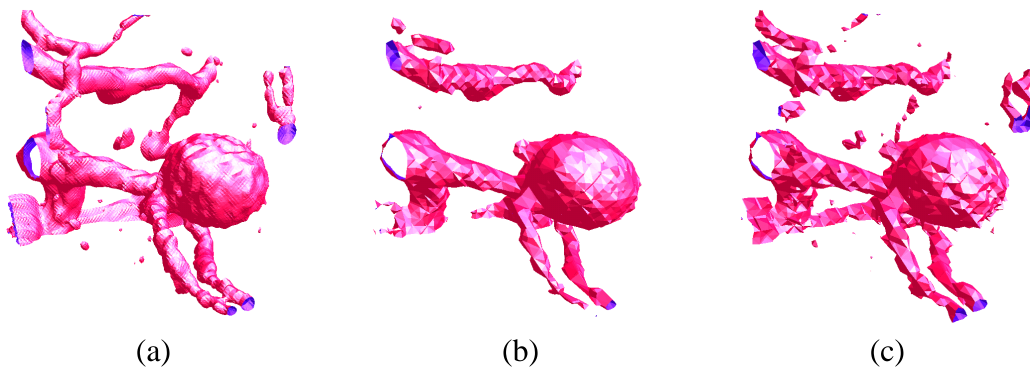
(a) (b)

Figure C.5 **(a)** Visualization of the opacity of the three-dimensional texture that corresponds to the two-dimensional texture in Figure 3.5b. The additional dimension parameterizes the length of the viewing ray within a tetrahedral cell. The isosurface represents opacity values of 0.25. **(b)** Visualization of the Bluntfin data set with three isosurfaces mixed with projected tetrahedra. *(See also Figure 3.11 on page 75.)*



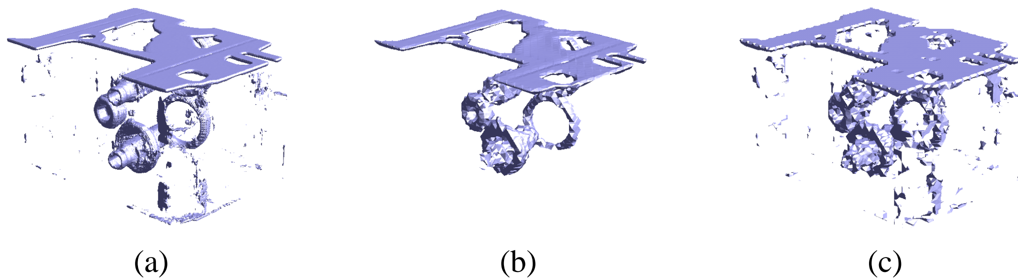
(a) (b)

Figure C.6 **(a)** A visibility cycle formed by three tetrahedra. **(b)** Volume rendering of the tetrahedra depicted in (a). *(See also Figure 4.21 on page 109.)*



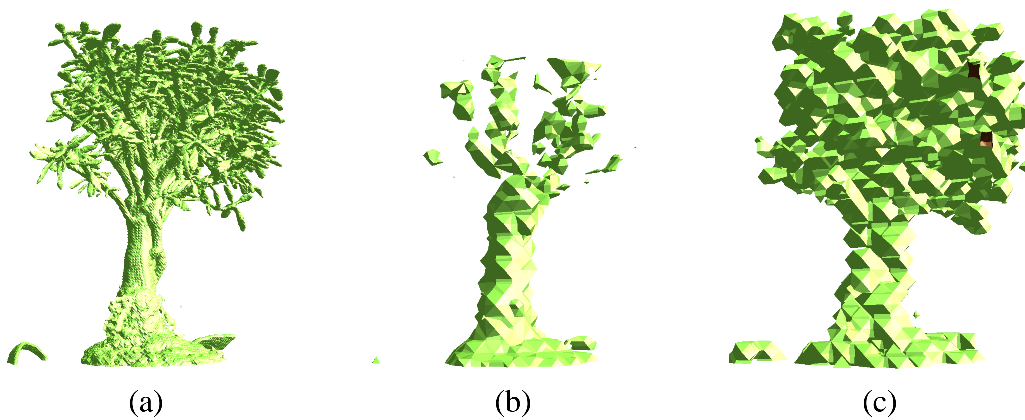
(a) (b) (c)

Figure C.7 **(a)** An isosurface extracted from a $128 \times 128 \times 60$ CTA volume data set. **(b)** Same isosurface extracted from a mesh downsampled to dimensions $32 \times 32 \times 15$ with averaging downsampling. **(c)** Same as (b) with topology-guided downsampling. *(See also Figure 5.9 on page 121.)*



(a) (b) (c)

Figure C.8 **(a)** Isosurface extracted from the original $256 \times 256 \times 110$ mesh. **(b)** Same isosurface extracted from a mesh of dimensions $64 \times 64 \times 28$ obtained by averaging downsampling. **(c)** Same as (b) but using topology-guided downsampling. *(See also Figure 5.10 on page 122.)*



(a) (b) (c)

Figure C.9 **(a)** An isosurface extracted from a CT scan of a bonsai. **(b)** Same isosurface but extracted from a mesh of dimensions $32 \times 32 \times 16$ obtained with averaging downsampling. **(c)** Same as (b) with topology-guided downsampling. *(See also Figure 5.11 on page 123.)*

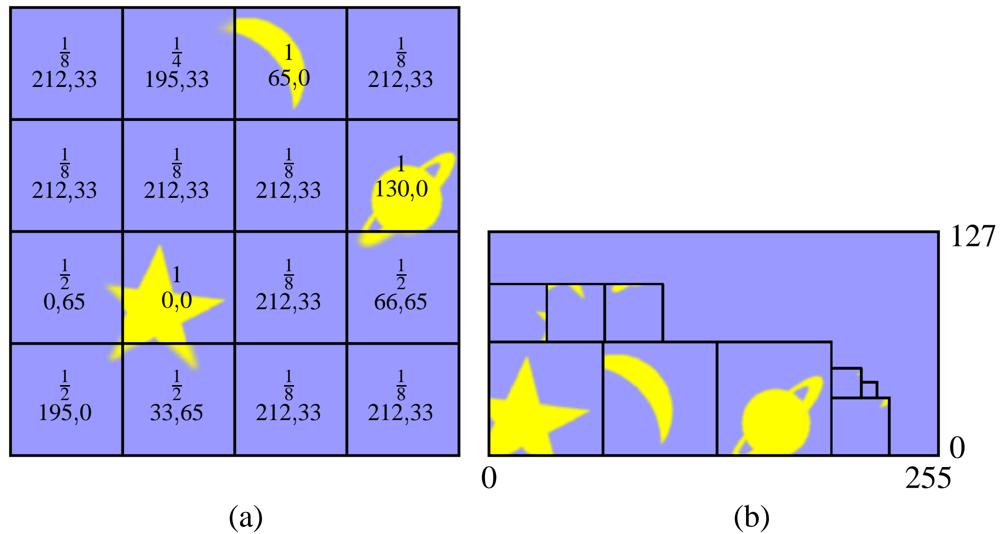


Figure C.10 Representation of adaptive textures. **(a)** Index data: scale factors and coordinates of packed data blocks are stored for each cell of a 4×4 mesh representing the whole texture, which is included for the purpose of illustration only. (Actual coordinates are between 0 and 1.) **(b)** Packed data: the data blocks packed into a uniform mesh of 256×128 texels. The blocks' frames are for illustration purposes only. *(See also Figure 5.12 on page 126.)*

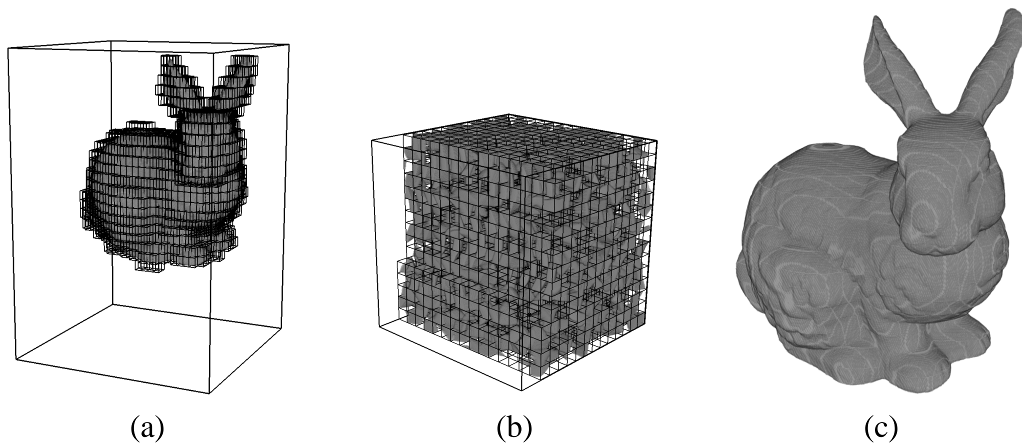


Figure C.11 Volume rendering of a $512 \times 512 \times 360$ CT scan with adaptive texture mapping. **(a)** Non-empty cells of the 32^3 index data mesh. **(b)** Data blocks packed into a 256^3 texture. **(c)** Resulting volume rendering. *(See also Figure 5.16 on page 134.)*

Bibliography

[1] C. L. Bajaj, I. Ihm, and S. Park. Compression-Based 3D Texture Mapping for Real-Time Rendering. In *Graphical Models*, 62(6):391-410, 2000.

[2] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Visualization of Scalar Topology for Structural Enhancement. In *Proceedings Visualization '98*, pages 51-58, 1998.

[3] C. L. Bajaj and D. R. Schikore. Topology Preserving Data Simplification with Error Bounds. *Computers & Graphics*, 22(1):3-12, 1998.

[4] A. C. Beers, M. Agrawala, and N. Chaddha. Rendering from Compressed Textures. In *Proceedings SIGGRAPH 96*, pages 373-378, 1996.

[5] J. Bennet, R. Cook, N. Max, D. May, and P. Williams. Parallelizing a High Accuracy Hardware-Assisted Volume Renderer for Meshes with Arbitrary Polyhedra. In *Proceedings 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pages 101-106, 2001.

[6] J. F. Blinn. Jim Blinn's Corner — Compositing, Part I: Theory. *IEEE Computer Graphics and Applications*, 14(5):83-87, 1994.

[7] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings 1994 Symposium on Volume Visualization*, pages 91-98, 1994.

[8] H. Carr, J. Snoeyink, and U. Axen. Computing Contour Trees in All Dimensions. In *Proceedings ACM-SIAM Symposium on Discrete Algorithms 2000*, pages 918-926, 2000.

[9] B. Chazelle, L. Palios. Triangulating a Nonconvex Polytope. *Discrete & Computational Geometry*, 5:505-526, 1990.

[10] S.-W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng. Sliver Exudation. *Journal of the ACM*, 47(5):883-904, 2000.

[11] P. Cignoni, D. Costanza, C. Montani, C. Rocchini, R. Scopigno. Simplification of Tetrahedral Meshes with Accurate Error Evaluation. In *Proceedings Visualization 2000*, pages 85-92, 2000.

[12] P. Cignoni, C. Montani, D. Sarti, and R. Scopigno. On the Optimization of Projective Volume Rendering. In R. Scateni, J. van Wijk, and P. Zanarini (editors), *Visualization in Scientific Computing '95*, pages 58-71. Springer-Verlag, 1995.

[13] J. Comba, J. T. Klosowski, N. Max, J. S. B. Mitchell, C. T. Silva, and P. L. Williams. Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids. In *Computer Graphics Forum (Proceedings EUROGRAPHICS '99)*, 18(3):369-376, 1999.

[14] R. Cook, N. Max, C. Silva, and P. Williams. Efficient, Exact Visibility Ordering of Unstructured Meshes. Lawrence Livermore National Laboratory technical report UCRL-JC-146582, 2002.

[15] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *Proceedings 1992 Workshop on Volume Visualization*, pages 91-98, 1992.

[16] T. K. Dey, H. Edelsbrunner, S. Guha, and D. V. Nekhayev. Topology Preserving Edge Contraction. *Publ. Inst. Math. (Beograd) (N.S.)*, 66:23-45, 1999.

[17] H. Edelsbrunner. An Acyclicity Theorem for Cell Complexes in d Dimensions. *Combinatorica*, 10(3):251-260, 1990.

[18] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Proceedings Graphics Hardware 2001*, pages 9-16, 2001.

[19] R. Farias, J. S. B. Mitchell, and C. T. Silva. ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering. In *Proceedings Volume Visualization and Graphics Symposium 2000*, pages 91-99, 2000.

[20] H. Fuchs, Z. M. Kedem, and B. Naylor. Predetermining Visibility Priority in 3-D Scenes. In *ACM Computer Graphics (Proceedings SIGGRAPH '79)*, 13(2):175-181, 1979.

[21] I. Fujishiro, T. Azuma, and Y. Takeshima. Automating Transfer Function Design for Comprehensible Volume Rendering Based on 3D Field Topology Analysis. In *Proceedings Visualization '99*, pages 467-470, 1999.

[22] M. P. Garrity. Raytracing Irregular Volume Data. *ACM Computer Graphics (Proceedings San Diego Workshop on Volume Visualization)*, 24(5):35-40, 1990.

[23] T. Gerstner and R. Pajarola. Topology Preserving and Controlled Topology Simplifying Multiresolution Isosurface Extraction. In *Proceedings Visualization 2000*, pages 259-266, 2000.

[24] S. Guthe, S. Roettger, A. Schieber, W. Strasser, and T. Ertl. High-Quality Unstructured Volume Rendering on the PC Platform. In *Proceedings Graphics Hardware 2002*, pages 119-125, 2002.

[25] E. Hart and J. L. Mitchell. *Hardware Shading with EXT_vertex_shader and ATI_fragment_shader*. ATI Technologies, 2001.

[26] T. He, L. Hong, A. Varshney, and S. W. Wang. Controlled Topology Simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171-184, 1996.

[27] M. Karasick, D. Lieber, L. Nackman, and V. Rajan. Visualization of Three-Dimensional Delaunay Meshes. *Algorithmica*, 19(1-2):114-128, 1997.

[28] M. J. Kilgard (editor), *NVIDIA OpenGL Extension Specifications*, NVIDIA Corporation, 2001.

[29] M. J. Kilgard (editor), *NVIDIA OpenGL Extension Specifications for the CineFX Architecture (NV30)*, NVIDIA Corporation, 2002.

[30] G. Knittel. Using Pre-Integrated Transfer Functions in an Interactive Software System for Volume Rendering. In *Proceedings Short Presentations EUROGRAPHICS 2002*, pages 119-123, 2002.

[31] M. Kraus and T. Ertl. Cell-Projection of Cyclic Meshes. In *Proceedings Visualization 2001*, pages 215-222, 2001.

[32] M. Kraus and T. Ertl. Topology-Guided Downsampling. In K. Mueller and A. Kaufmann (editors), *Volume Graphics 2001*, pages 223-234. Springer-Verlag, 2001.

[33] M. Kraus and T. Ertl. Implementing Ray Casting in Tetrahedral Meshes with Programmable Graphics Hardware. Technical Report 1/2002, Visualization and Interactive Systems Group at the University of Stuttgart, 2002.

[34] M. Kraus and T. Ertl. Adaptive Texture Maps. In *Proceedings Graphics Hardware 2002*, pages 7-15, 2002.

[35] M. Kraus and T. Ertl. Simplification of Nonconvex Tetrahedral Meshes. In G. Farin, B. Hamann, and H. Hagen (editors), *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 185-196. Springer-Verlag, 2003.

[36] K. Kreeger and A. Kaufman. Mixing Translucent Polygons with Volumes. In *Proceedings Visualization '99*, pages 191-198, 1999.

[37] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Proceedings SIGGRAPH 94*, pages 451-458, 1994.

[38] D. Laur and P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *ACM Computer Graphics (Proceedings SIGGRAPH '91)*, 25(4):285-288, 1991.

[39] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29-37, 1988.

[40] M. Levoy and R. Whitaker. Gaze-Directed Volume Rendering. *ACM Computer Graphics (Proceedings 1990 Symposium on Interactive 3D Graphics)*, 24(2):217-223, 1990.

[41] J. Malouff, *Fifty Problem Solving Strategies Explained*, `http://www.une.edu.au/psychology/staff/malouff/problem.htm`, 2002.

[42] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99-108, 1995.

[43] N. Max, B. Becker, and R. Crawfis. Flow Volumes for Interactive Vector Field Visualization. In *Proceedings Visualization '93*, pages 19-24, 1993.

[44] N. Max, P. Hanrahan, and R. Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *ACM Computer Graphics (Proceedings San Diego Workshop on Volume Visualization)*, 24(5):27-33, 1990.

[45] N. Max, P. Williams, C. Silva. Cell Projection of Meshes with Non-Planar Faces. In F. H. Post, G. M. Nielson, and G.-P. Bonneau (editors), *Data Visualization — The State of the Art (Proceedings Dagstuhl 2000, Seminar on Scientific Visualization)*. Kluwer Academic Publishers, 2002.

[46] M. Meißner, S. Guthe, and W. Straßer. Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. In *Proceedings Graphics Interface 2002*, pages 209-218, 2002.

[47] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. VIZARD II: A Reconfigurable Interactive Volume Rendering System. In *Proceedings Graphics Hardware 2002*, pages 137-146, 2002.

[48] J. Milnor. *Morse Theory.* Princeton University Press, 1963.

[49] P. Ning and L. Hesselink. Vector Quantization for Volume Rendering. In *Proceedings 1992 Workshop on Volume Visualization*, pages 69-74, 1992.

[50] P. Ning and L. Hesselink. Fast Volume Rendering of Compressed Data. In *Proceedings Visualization '93*, pages 11-18, 1993.

[51] M. Ohlberger and M. Rumpf. Hierarchical and Adaptive Visualization on Nested Grids. *Computing*, 59(4):365-385, 1997.

[52] J. Popović and H. Hoppe. Progressive Simplicial Complexes. In *Proceedings SIGGRAPH 97*, pages 217-224, 1997.

[53] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics (Proceedings SIGGRAPH 2002)*, 21(3):703-712, 2002.

[54] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization. In *Proceedings Graphics Hardware 2000*, pages 109-118, 2000.

[55] S. Roettger and T. Ertl. A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids. In *Proceedings Volume Visualization and Graphics Symposium 2002*, pages 23-28, 2002.

[56] S. Röttger, M. Kraus, and T. Ertl. Hardware-Accelerated Volume and Isosurface Rendering based on Cell-Projection. In *Proceedings Visualization 2000*, pages 109-116, 2000.

[57] J. Schulze. Personal communication, 2002.

[58] R. Sedgewick. *Algorithms*, 2nd edition. Addison-Wesley, 1988.

[59] K. S. Shanmugam. *Digital and Analog Communication Systems.* John Wiley & Sons, 1979.

[60] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill. Octree-Based Decimation of Marching Cubes Surfaces. In *Proceedings Visualization '96*, pages 335-342, 1996.

[61] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *ACM Computer Graphics (Proceedings San Diego Workshop on Volume Visualization)*, 24(5):63-70, 1990.

[62] C. T. Silva, J. S. B. Mitchell, and P. L. Williams. An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes. In *Proceedings 1998 Symposium on Volume Visualization*, pages 87-94, 1998.

[63] J. Snyder and J. Lengyel. Visibility Sorting and Compositing without Splitting for Image Layer Decomposition. In *Proceedings SIGGRAPH 98*, pages 219-230, 1998.

[64] O. G. Staadt and M. H. Gross. Progressive Tetrahedralizations. In *Proceedings Visualization '98*, pages 397-402, 1998.

[65] C. M. Stein, B. G. Becker, and N. L. Max. Sorting and Hardware Assisted Rendering for Volume Visualization. In *Proceedings 1994 Symposium on Volume Visualization*, pages 83-89, 1994.

[66] R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146-160, 1972.

[67] I. J. Trotts, B. Hamann, K. I. Joy, and D. F. Wiley. Simplification of Tetrahedral Meshes. In *Proceedings Visualization '98*, pages 287-295, 1998.

[68] M. Weiler and T. Ertl. Hardware-Software-Balanced Resampling for the Interactive Visualization of Unstructured Grids. In *Proceedings Visualization 2001*, pages 199-206, 2001.

[69] M. Weiler, M. Kraus, and T. Ertl. Hardware-Based View-Independent Cell Projection. In *Proceedings Volume Visualization and Graphics Symposium 2002*, pages 13-22, 2002.

[70] E. W. Weisstein. *Eric Weisstein's World of Mathematics.* `http://mathworld.wolfram.com/`, 2002.

[71] R. Westermann. The Rendering of Unstructured Grids Revisited. In D. Ebert, J. M. Favre, and R. Peikert (editors), *Data Visualization 2001 (Proceedings VisSym '01)*, pages 65-74. Springer-Verlag, 2001.

[72] R. Westermann and T. Ertl. The VSBUFFER: Visibility Ordering of Unstructured Volume Primitives by Polygon Drawing. In *Proceedings Visualization '97*, pages 35-42, 1997.

[73] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proceedings SIGGRAPH 98*, pages 169-177, 1998.

[74] R. Westermann, C. Johnson, and T. Ertl. A Level-Set Method for Flow Visualization. In *Proceedings Visualization 2000*, pages 147-154, 2000.

[75] R. Westermann, L. Kobbelt, and T. Ertl. Real-Time Exploration of Regular Volume Data by Adaptive Reconstruction of Isosurfaces. *The Visual Computer*, 15(2):100-111, 1999.

[76] L. Westover. Footprint Evaluation for Volume Rendering. *ACM Computer Graphics (Proceedings SIGGRAPH '90)*, 24(4):367-376, 1990.

[77] R. Whitaker, D. Breen, K. Museth, and N. Soni. Segmentation of Biological Volume Datasets Using a Level-Set Framework. In K. Mueller and A. Kaufmann (editors), *Volume Graphics 2001*, pages 249-263. Springer-Verlag, 2001.

[78] J. Wilhelms and A. van Gelder. A Coherent Projection Approach For Direct Volume Rendering. *ACM Computer Graphics (Proceedings SIGGRAPH '91)*, 25(4):275-284, 1991.

[79] L. Williams. Pyramidal Parametrics. *ACM Computer Graphics (Proceedings SIGGRAPH '83)*, 17(3):1-11, 1983.

[80] P. L. Williams. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, 11(2):103-126, 1992.

[81] P. L. Williams and N. Max. A Volume Density Optical Model. In *Proceedings 1992 Workshop on Volume Visualization*, pages 61-68, 1992.

[82] 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.

[83] C. M. Wittenbrink. CellFast: Interactive Unstructured Volume Rendering. *Proceedings Visualization 1999 Late Breaking Hot Topics*, pages 21-24, 1999.

[84] C. M. Wittenbrink, T. Malzbender, and M. E. Goss. *Opacity-Weighted Color Interpolation for Volume Visualization*. In *Proceedings 1998 Symposium on Volume Visualization*, pages 135-142, 1998.

[85] M. Woo, J. Neider, T. Davis, and D. Shreiner. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2, 3rd edition. Addison-Wesley, 1999.

I need to tell you now
As we leave
That it's much worse
Than you would believe

And no matter how far
You think you've been
The beginning
Well, that's where your are

So I'm using my last match
To put a fire up on every hill
And burn down Pleasantville

Verse from the song *Here in Pleasantville* by Jakob Dylan