Hardware-beschleunigte Volumenvisualisierung auf adaptiven Datenstrukturen

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

Manfred Weiler

aus Mellrichstadt

Hauptberichter:	Prof. Dr. T. Ertl
Mitberichter:	Prof. Dr. R. Westermann
	Prof. Dr. E. Gröller
Tag der mündlichen Prüfung:	27. September 2005

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2006

Danksagung

Wissenschaft findet nicht im luftleeren Raum statt und so haben eine Reihe von Personen einen nicht unerheblichen Beitrag an der Entstehung dieser Arbeit. Der erste herzliche Dank gilt meinem Doktorvater und wissenschaftlichen Betreuer Prof. Dr. Thomas Ertl, als kontinuierlich antreibende Kraft hinter dieser Dissertation. Er gab mir nicht nur Lohn und Brot sondern darüber hinaus die nötige Freiheit, die es mir ermöglichte, viele eigene Ideen zu verwirklichen.

Unter den zahlreichen Mitautoren der im Rahmen dieser Dissertation entstandenen Publikationen ist an erster Stelle Prof. Dr. Rüdiger Westermann zu nennen, dem ich neben Prof. Eduard Gröller auch für sein Engagement als Mitgutachter danke. Aus unserer direkten Zusammenarbeit entstand meine erste internationale Veröffentlichung, die den Grundstein für meine weitere wissenschaftliche Arbeit gelegt hat. Der Einführung in die Visualisierungs-Community verdanke ich außerdem die Bekanntschaft mit vielen netten und interessanten Leuten, unter anderem mit den weiteren Co-Autoren Chuck Hansen und Kurt Zimmerman. Rüdigers Arbeiten waren darüber hinaus direkte Inspiration für das hier eingeflossene Hardware-basierte Resampling unstrukturierter Gitter.

Zu den wichtigsten Mitautoren zählt definitiv auch Martin Kraus, dessen Zusammenarbeit dabei geholfen hat, viele latent vorhandene Ideen zutage zu fördern. Paula Mallón, unser spanischer Boomerang, hat mit Ihrer Erfahrung im Bereich der Komprimierung von Dreiecksnetzen nicht nur neue Aspekte zu dieser Dissertation beigetragen. Sie war es auch, die mich in die Spezialitäten spanischer Süßigkeitenkunst einführte und die mich dazu brachte, meine temporäre Wahlheimat Stuttgart und seine Umgebung besser kennenzulernen.

Für die nicht immer leichte Zusammenarbeit am RBF-Projekt, teilweise über Kontinente hinweg, möchte ich mich bei David Ebert, Kelly Gaither, Yun Jang, Jingshu Huang, Ralf Borchert und Simon Stegmaier bedanken. Vor allem schätze ich die Unterstützung von Matthias Hopf in zahlreichen Nachtsitzungen.

Allen Kollegen am Institut für Visualisierung und interaktive Systeme der Universität Stuttgart danke ich für den kameratschaftlichen Umgang, viele fruchtbare Diskussionen und unterhaltsame Stunden auch bei gemeinsamen Aktivitäten außerhalb der Arbeit. Besonders erwähnen möchte ich Marcelo Magallón, Martin Schulz, Ove Sommer, unsere Institutsperle Ulrike Ritzmann und meine Bürogenossen Klaus Engel und Guido Reina.

Meinen Leidensgenossen am OpenSG-Projekt, Thomas Klein und Michael Braitmaier, gilt mein herzlicher Dank. Die Arbeit war sicher nicht immer einfach.

Den Studien- und Diplomarbeitern Andreas Boronas und Markus Merz danke ich für ihre fruchtbare und erfolgreiche Zusammenarbeit.

Für ihre starke Unterstützung bei der Korrektur dieser Arbeit möchte ich mich bei Katrin Bidmon, Magnus Strengert und Daniel Weiskopf bedanken. Einige in letzter Minute in diese Arbeit eingeflossene Ideen resultierten aus Daniels Anmerkungen. Katrin war eine wertvolle Hilfe bei der Abwicklung des Promotionsverfahrens von Berlin aus.

Schließlich gilt großer Dank meinen Eltern, die mich immer nach Kräften unterstützt haben. Die Backkünste meiner Mutter Agnes Weiler werden in der Abteilung unvergessen bleiben. Leider konnte mein Vater Otmar Weiler den Abschluss dieser Arbeit nicht mehr erleben. Ihm möchte ich diese Arbeit widmen.

Danke an alle die ich vergessen habe – es ist nicht persönlich.

Manfred Weiler.

Inhaltsverzeichnis

Kurzfassung **11**

Abstract and Chapter Summaries **13**

1 Einleitung **17**
1.1 Adaptive Datenstrukturen 18
1.2 Gliederung der Arbeit 19

2 Grundlagen **21**
2.1 Visualisierung 21
2.2 Daten und Gittertypen 22
2.2.1 Strukturierte Gitter 23
2.2.2 Unstrukturierte Gitter 24
2.2.3 Multiblock-Gitter 24
2.2.4 Punktbasierte Daten und Wavelets 26
2.3 Graphik-Hardware 27
2.3.1 Rendering-Pipeline 27
2.3.2 Programmierbare Graphik-Hardware 32

3 Volumenvisualisierung **37**
3.1 Mathematische Grundlagen 38
3.1.1 Strahlintegration 38
3.1.2 Optische Modelle 39
3.1.3 Prä- und Post-Klassifikation 41
3.1.4 Numerische Integration 41
3.1.5 Prä-Integrierte Klassifikation 43

3.2 Beschleunigung der Prä-Integration 45
3.2.1 Inkrementelle Prä-Integration 46
3.2.2 Subrange-Integration 47
3.3 Algorithmen 49
3.3.1 Ray Casting 51
3.3.2 Texturbasierte Volumenvisualisierung 51
3.3.3 Zellprojektion 53

4 Hardware-basierte Zellprojektion 57
4.1 Scan-Konvertierung von Tetraedern 58
4.1.1 Blickpunktunabhängige Zellprojektion 58
4.1.2 Berechnung des Austrittspunkts 59
4.2 Vertexorientierter Ansatz 61
4.2.1 Berechnungen pro Fragment 61
4.2.2 Berechnungen pro Vertex 62
4.2.3 Konsistente Interpolation 65
4.2.4 Kantenartefakte 66
4.3 Fragmentorientierter Ansatz 68
4.3.1 Vertex-Shader-Programm 68
4.3.2 Fragment-Shader-Programm 69
4.4 Optische Modelle und Strahlintegration 70
4.4.1 Direktes Volume Rendering 71
4.4.2 Isoflächen 72
4.5 Zellprojektion von Tetraedergittern 73
4.6 Ergebnisse 75

5 Hardware-basiertes Ray Casting 77
5.1 Paralleles Ray Casting 78
5.1.1 Überblick 78
5.1.2 Nomenklatur 79
5.1.3 Strahlintegration 80
5.1.4 Zelltraversierung 82
5.2 Tetraederstreifen 83

5.3 Streifenerzeugung 85
5.4 Gitterrepräsentation 88
5.4.1 Tetraederlisten 88
5.4.2 Tetraederstreifen 89
5.5 Implementierung 92
5.5.1 Datenstrukturen 92
5.5.2 Strahlterminierung 94
5.5.3 Nicht-konvexe Tetraedergitter 94
5.6 Ergebnisse 96
5.6.1 DirectX-Implementierung 97
5.6.2 OpenGL-Implementierung 99
5.6.3 Diskussion 101

6 Volumenvisualisierung mit radialen Basisfunktionen 103
6.1 Radiale Basisfunktionen 104
6.2 Kodierung von Volumendaten 105
6.3 RBF-basierte Visualisierung 107
6.3.1 Hierarchische Strukturierung 108
6.3.2 Texturkodierung 108
6.3.3 Schnittebenen und Volumenvisualisierung 110
6.3.4 Splatting 112
6.4 Ergebnisse und Diskussion 113

7 Hardware-Software-balanciertes Resampling 119
7.1 Vorarbeiten 120
7.2 Hardware-unterstütztes Slicing 121
7.2.1 Ersatzgeometrie 122
7.2.2 Identifizierung des Schnittbereichs 123
7.2.3 Rasterisierung des Schnittpolygons 125
7.2.4 Hardware-unterstützte Dateninterpolation 126
7.2.5 Implementierungsdetails 127
7.3 Software-Resampling 129
7.4 Hybrides Resampling 131
7.5 Ergebnisse und Analyse 132

8 Level-of-Detail-Volumenvisualisierung mittels 3D-Texturen 137
8.1 Level-of-Detail-Texturrepräsentation . . . 138
8.1.1 Texturunterteilung . . . 139
8.1.2 Kontinuierliche Level-Übergänge . . . 140
8.1.3 Verschmelzen der Texturen . . . 141
8.1.4 Adaptive Darstellung . . . 143
8.2 Adaptives Slicing . . . 144
8.2.1 Anpassung der Opazität . . . 144
8.2.2 Opazitätskorrektur durch Polygon-Clipping . . . 146
8.3 Ergebnisse und Analyse . . . 147

9 Adaptive texturbasierte Volumenvisualisierung 151
9.1 OpenSG . . . 152
9.2 Hardware-Abstraktion . . . 153
9.3 Shader-Schnittstellen . . . 155
9.4 Volumenmodellierung in OpenSG . . . 156
9.5 Volumen-Shader . . . 158
9.5.1 Color-Table-Shader . . . 159
9.5.2 Isoflächen-Shader . . . 160
9.5.3 Prä-Integrations-Shader . . . 162
9.6 Volumen-Clipping . . . 164
9.7 Virtuelle Realität und Volumenvisualisierung . . . 166
9.7.1 Transparente Szenengraph-Replikation . . . 167
9.7.2 Client-Server-Anwendung . . . 168
9.7.3 Bildkompression auf Graphik-Hardware . . . 170

10 Zusammenfassung und Ausblick 175
10.1 Hardware-Software-Synthese . . . 176
10.1.1 Algorithmische Transformation . . . 176
10.1.2 Beseitigung von Kommunikationsflaschenhälsen . . . 177
10.1.3 Parallelisierung . . . 178
10.1.4 Hardware-Selektion . . . 179
10.1.5 Hardware-Software-Co-Design . . . 179

10.2 Partitionierungsstrategien für Tetraedergitter 180
10.2.1 CPU-zentrierte Partitionierung 180
10.2.2 Vertexzentrierte Partitionierung 180
10.2.3 Fragmentzentrierte Partitionierung 181
10.3 Ausblick 182

Anhang **185**
Abbildungsverzeichnis 185
Tabellenverzeichnis 188
Literaturverzeichnis 189

Kurzfassung

In vielen Disziplinen und Anwendungsgebieten spielen Volumendaten heutzutage eine immer wichtigere Rolle, was zu einer starken Nachfrage nach interaktiven Visualisierungsalgorithmen zur effizienten Analyse dieser komplexen Datenform geführt hat. Der Einsatz von Graphik-Hardware stellt dabei eine gute Möglichkeit dar, den mit der Volumenvisualisierung verbundenen Berechnungsaufwand zu meistern. Dem entgegen steht die bisher auf Graphikkarten übliche relativ starre Verarbeitungspipeline, die in ihrer Konzeption für die realistische Darstellung von 3D-Szenen entworfen wurde. Sie beschränkt den Einsatz auf wenige Typen von Volumendaten, insbesondere uniforme 3D Gitter, deren Struktur sich effizient auf die Mechanismen auf der Graphikkarte abbilden lässt.

Mit der kürzlich eingeführten freien Programmierbarkeit von Graphikkarten ist es zunehmend möglich, die massive Rechenleistung der Graphikchips, welche die Fähigkeiten aktueller Universalprozessoren bei weitem übersteigt, für die Visualisierung von Volumendaten auf adaptiven Datenstrukturen einzusetzen. Letztere gewinnen immer mehr an Bedeutung, weil sie mit den in der Simulation verwendeten Berechnungsmodellen konform sind, Vorteile hinsichtlich der effizienten Speicherung sehr großer Datensätze bieten und durch eine adaptive Traversierung eine interaktivere Darstellung erlauben.

Am Beispiel dreier wichtiger Vertreter dieser Gattung von adaptiven Datenstrukturen, Finite-Elemente-Netze in ihrer Ausprägung als Tetraedergitter, lokale Gitterverfeinerungen sowie gitterlose Punktrepräsentationen, untersucht diese Arbeit daher, in wieweit sich die dafür erforderlichen komplexen Visualisierungsalgorithmen durch den Einsatz moderner Graphik-Hardware beschleunigen lassen. Ziel es dabei, möglichst viele Teile der Berechnung in den Verantwortungsbereich der Graphikkarte zu bringen, um auch in Zukunft von dem raschen Fortschritt bei der Entwicklung von Graphik-Hardware zu profitieren, deren Leistungsfähigkeit sich schneller als die der Universalprozessoren, etwa alle 6 Monate, verdoppelt.

Ein wichtiges dabei zu untersuchendes Problemfeld stellt das Zusammenspiel der unterschiedlichen involvierten Berechnungseinheiten (Hauptprozessor, Vertexprozessor und Fragmentprozessor auf der Graphikkarte) dar, die für optimale Geschwindigkeit möglichst gleichmäßig ausgelastet werden müssen. Da jede Einheit für sich betrachtet einen Flaschenhals für den Gesamtalgorithmus darstellen kann, ist eine ausgewogene Balance bei der Verteilung der Berechnungsschritte auf die einzelnen Prozessoren erforderlich. Dies wird in der vorliegenden Arbeit demonstriert, indem unterschiedliche Algorithmen zur Visualisierung von Tetraedergittern entworfen werden, die sukzessive identifizierte Flaschenhälse bei der Berechnung und der Kommunikation

mit nachgelagerten Einheiten beziehungsweise dem assoziierten Speicher beseitigen. Am Ende steht ein Hardware-basierter Ray Casting-Algorithmus für Tetraedergitter, der vollständig auf der Graphik-Hardware abläuft und daher nur durch die Rechenleistung der Graphikprozessoren beschränkt ist.

Da Algorithmen, die auf der Graphikkarte ablaufen, ihre Daten aus dem begrenzten lokalen Speicher der Graphikkarte beziehen, behandelt diese Arbeit darüberhinaus unterschiedliche Möglichkeiten zur Kompression, um die maximale Größe der verarbeitbaren Datensätze zu erhöhen. Es wird eine neu entworfene kompakte Datenstruktur für Tetraedergitter vorgestellt, welche sowohl die Attribute der einzelnen Tetraeder als auch die Konnektivität zwischen den Tetraedern speichert und als Basis zur Visualisierung mittels Ray Casting dienen kann. Für punktbasierte Daten wird außerdem eine neue kompakte prozedurale Kodierung auf der Grundlage von radialen Basisfunktionen entwickelt, welche durch die Möglichkeit der Dekodierung mittels der programmierbaren Fragmenteinheit mit einer Vielzahl von Visualisierungslagorithmen kombinierbar ist und durch die vorgelagerte Kodierung anderer Datenstrukturen wie uniformer Gitter, Tetraedergitter oder gitterloser Daten eine universelle Visualisierungslösung darstellt.

Eine weitere Option zur interaktiven Visualisierung von Tetraedergittern bietet die Abtastung auf adaptiv verfeinerten uniformen Volumengittern, die besser für die Verarbeitung auf der Graphikkarte geeignet sind. Hier ermöglicht die Beschleunigung der Abtastung mittels Graphik-Hardware die dynamische Berechnung, jeweils entsprechend den Parametern des aktuell dargestellten Bildes. Die sonst beim Resampling auftretenden charakteristischen Probleme wie hoher Speicherbedarf, geringe Interaktionsraten oder niedrige Darstellungsqualität werden dadurch vermieden beziehungsweise minimiert.

Für typische Probleme, die bei der Visualisierung auf der Basis adaptiv verfeinerter uniformer Gitter auftreten, wie inkonsistente Interpolation zwischen unterschiedlichen Gitterbereichen oder Artefakte, die durch eine adaptive Abtastung auftreten, werden Lösungsansätze präsentiert.

Schließlich beschäftigt sich diese Arbeit mit Implementierungsaspekten einer adaptiven Volumendarstellung auf uniformen Gittern. Eine neuartige Software-Architektur mit flexiblen und erweiterbaren Shadern erlaubt es dabei, die Visualisierungsalgorithmen einfach auf die spezielle Graphik-Hardware der Zielplattform anzupassen, was in Anbetracht der heterogenen Graphikkarten-Landschaft einen wichtigen Baustein für die Entwicklung effizienter Visualisierungslösungen darstellt. Darüberhinaus wird gezeigt, wie sich programmierbare Graphik-Hardwere im Umfeld von virtuellen Umgebungen durch Hardware-beschleunigte Bildkompression und -dekompression geschwindigkeitssteigernd einsetzen lässt.

Die Arbeit bringt alle vorgestellten Algorithmen in einen Gesamtzusammenhang, der es erlaubt gemeinsame Gestaltungsrichtlinien für die erfolgreiche Umsetzung von (Visualisierungs-)Algorithmen auf Graphik-Hardware zu extrahieren. Diese können anderen Forschern als Muster und zur Anregung dienen und so die Entwicklung weiterer Hardware-beschleunigter Algorithmen inspirieren.

Abstract and Chapter Summaries

Volumetric data has become common in many scientific disciplines and application areas and there is a growing demand for direct volume rendering of such data at interactive framerates, since this has proven as an essential tool for the effective analysis of volumetric data. However, in particular for large data sets, direct volume rendering is computational expensive, which has led several researchers to the idea of exploiting the massive computational power of graphics hardware nowadays available in almost every standard PC.

Until a few years ago, the development of graphics hardware-accelerated visualization algorithms was limited by a relatively strict processing pipeline, which was originally designed for the rendering of realistic 3D scenes and not for the demands of visualization algorithms. The application of graphics hardware for visualization purposes was therefore restricted either to the rendering of visual representations computed by the main processor, e.g., extracted polygonal isosurfaces, or to data structures which map directly to mechanisms provided by the graphics hardware, e.g., three-dimensional scalar fields on a regular mesh, which can be represented by 3D textures. Volumetric data given on adaptive data structures like unstructured meshes, adaptive refined structured meshes, or point-based data representations usually had to be converted before the visualization, which either led to poor performance due to the memory overhead or to poor image quality.

Modern graphics hardware provides programmable vertex and fragment processing units instead of a fixed function pipeline. On the one hand, the flexibility offered by these units allows for much more complex visualization algorithms to be implemented directly on the graphics card and thus to benefit from its computational power. On the other hand, the implementation of optimized visualization algorithms now has to consider three different processing units, the main processor, the vertex processor, and the fragment processor, including the communication between these units and the access to the local memory of each unit. Since one overloaded processor or communication path limits the overall system performance, an optimal algorithm has to find a good workload distribution.

This thesis addresses the balancing of these units in the context of hardware-accelerated visualization algorithms for adaptive data structures. New visualization algorithms are designed based on the identified bottlenecks in common visualization algorithms. Furthermore, design principles that have proven to be successful in the context of hardware-software synthesis of embedded systems are adapted. Their application to typical visualization problems leads to the successful development of several new hardware-accelerated visualization algorithms for tetrahedral mes-

hes, point-based data representations, and adaptive structured meshes.

Beside from these actual applicable algorithms, this thesis also identifies common strategies and principles which lead to successful algorithmic designs. The overview of this strategies will help other researchers to develop hardware-accelerated visualization and rendering algorithms for areas not covered by this thesis.

Chapter 1 The thesis is structured as follows. After a brief introduction, Chapter 1 provides a motivation for this work and familiarizes the reader with adaptive data structures, which become more and more important since non-adaptive data structures often fail for very large data sets acquired by state-of-the-art measurement or simulation.

Chapter 2 An overview of the terms and methodologies of scientific visualization used in this thesis is given in Chapter 2. This not only includes a definition of scientific visualization and a description of the visualization pipeline, but also a survey of the relevant data types and mesh types, which are presented in the context of commonly applied classification schemes. As a fundament for the description of the following hardware-based visualization algorithms, the second part of Chapter 2 deals with the architecture of graphics hardware and its historical development from a fixed function rendering pipeline consisting of hardly parameterizable geometry processing, rasterization, and fragment processing to programmable vertex and fragment units. The description of these units includes the programming model, overall features, and special properties, which stem from the fact that vertex and fragment units are implemented as stream processors. Low-level as well as high-level programming interfaces are covered.

Chapter 3 The goal of Chapter 3 is the introduction of mechanisms and algorithms used in this thesis. Thus, first the basics of direct volume rendering are presented, including its physical foundation in the theory of radiative transport and the volume rendering integral, which effectively is a simplification neglecting scattering effects. “Optical model” is another important term which describes the physical mechanisms affecting light that passes through a volume. Since this is closely related to the actual form of the volume rendering integral used in visualization algorithms, a survey of important physically based and non-physically based models is given.

Pre-integrated classification as one of the most important recent improvements for direct volume rendering is discussed in detail since this is one of the most fundamental principles exploited in this thesis. It is based on a table of pre-integrated ray segments which is expensive to compute and limits the interactivity of volume rendering applications during transfer function modification. Several acceleration techniques are therefore presented including CPU methods based on incremental computation and the acceleration with graphics hardware.

In preparation for the graphics hardware-based implementations presented in Chapters 4 to 8, the most common algorithms for direct volume rendering are introduced, in particular ray casting, cell projection, and texture-based volume rendering.

The main part of this thesis consists of Chapters 4 to 8 and discusses several strategies for the acceleration of visualization algorithms for adaptive data structures. Tetrahedral meshes (Chapters 4 and 5), radial basis functions as point-based representation (Chapter 6), and adaptively refined uniform meshes (Chapters 7 and 8) are selected as examples. Each presented solution

removes bottlenecks of previous implementations. Additionally the presented strategies also document the different steps in the development of programmable graphics hardware.

Chapter 4 Chapter 4 demonstrates how programmable vertex processing can be exploited to remove significant bottlenecks associated with the main processor and the data transfer between the main processor and the graphics adapter. The basic idea is the introduction of tetrahedra as additional graphics primitives – similar to triangles, lines, and points – that can be directly scan converted by the graphics hardware and therefore allow for exploiting the very same optimization techniques used for polygonal graphics, e.g., OpenGL display lists and vertex arrays. An acceleration is achieved since a tetrahedral mesh stored in static vertex arrays or display lists may reside in the local memory of the graphics card which removes the need for data transfer. Alternatively the data is stored in the main memory in a form that is optimized for the data transfer. Two methods for the scan conversion of tetrahedra are discussed. The first exploits programmable vertex processing only and therefore requires only limited graphics card capabilities. However, correct results are restricted to orthographic view projection. The second variant removes this limitation by combining vertex and fragment processing.

Chapter 5 Algorithms that run solely on the graphics card – more precisely spoken in the fragment unit of the graphics card – and thus are only limited by the computational power of the graphics hardware are discussed in Chapter 5. Their implementation required the development of programmable fragment units. The presented hardware-based ray casting extends the idea of Chapter 4 which essentially consists of a ray casting algorithm applied to each tetrahedron individually. The approach includes acceleration strategies such as early-ray-termination and works for cyclic and/or non-convex meshes. The later are handled either natively by a technique similar to depth-peeling or by convexifying the mesh in a preprocessing step.

Another important issue is the compact representation of the tetrahedral mesh, since the ray casting approach requires that the data to be visualized resides in the limited texture memory of the graphics adapter. The possibility for mesh compression exploiting a data representation based on tetrahedral strips is discussed and a new compact data structure is proposed, that can be employed for a variety of visualization and filtering algorithms requiring access to the mesh connectivity.

A final comparison with cell projection algorithms discusses the advantages and drawbacks of the new hardware-based ray casting approach and gives some advice for which applications one or the other algorithm might be more suitable.

Chapter 6 An alternative approach which exploits a compact universal data representation scheme instead of a tetrahedral mesh for storing volumetric data in the local memory of the graphics card is presented in Chapter 6. It is based on a procedural encoding of the volumetric data with radial basis functions and thus allows for the visualization of volumetric data of different type and structure, e.g., uniform meshes, tetrahedral meshes, and meshless representations, as they can be converted to a procedural representation within acceptable error tolerances. Since the programmable fragment units of modern graphics adapters can evaluate the radial basis functions for each fragment independently of the rendered geometry, this procedural encoding is compatible with a variety of visualization techniques. This is demonstrated for scalar and vector data.

Chapter 7 Chapters 7 and 8 deal with adaptively refined uniform meshes, which are better suited for algorithms exploiting graphics hardware due to their locally regular structure. Subgrids can easily be mapped to 3D textures and texture-based volume rendering algorithms specifically perform at least one order of magnitude faster than visualization algorithm for tetrahedral meshes. In order to also benefit from these high framerates for the visualization of tetrahedral meshes, three resampling algorithms are presented in Chapter 7. They allow for the fast conversion of tetrahedral meshes into a hierarchy of adaptively refined uniform meshes. In this context an optimal balance for the workload distribution between the main processor and the graphics processor is investigated since the three algorithm exploit the graphics hardware to an increasing degree.

Chapter 8 The error free and adaptive rendering of locally refined uniform meshes is the subject of Chapter 8. The rendering is based on a hierarchy of texture maps, which is adjusted to guarantee consistent data interpolation even over borders between areas represented at different levels of detail. At the same time the hierarchy is constructed in a way that minimizes the amount of texture memory. High rendering performance is achieved by adapting the sampling frequency with respect to the selected level of detail, which usually leads to artifacts caused by non-corrected opacities at level-transitions. These artifacts are fixed by a special rendering scheme that clips slice polygons in erroneously rendered regions and renders them with adapted opacity.

Chapter 9 Chapter 9 covers implementation aspects for the adaptive rendering of volumetric data on uniform meshes. Due to the rapid development cycles of graphics hardware with one new graphics chip generation every year, application programmers are often faced with a highly heterogeneous list of graphics platforms, which beside from the adaptation of an application to the characteristics of the volume data set also enforces an adaptation to the capabilities of the specific graphics card an application runs on. The framework presented in Chapter 9 supports this adaptation by a flexible and extensible shader concept. Details of the integration of this framework into the open source scenegraph API OpenSG are provided. Additionally, aspects for volume rendering in virtual environments are discussed. This includes a distributed client server architecture which exploits several client nodes in order to relief the rasterization bottleneck imposed by volume rendering applications. The subimages computed by each client are sent to a display server using an image compression/decompression algorithm computed by the graphics hardware in order to avoid latencies and reduce bandwidth requirements.

Chapter 10 In Chapter 10 the concepts and algorithms presented in Chapters 4 to 8 are put into a unified context which has been adapted from the field of hardware-software synthesis for embedded systems. This allows us to identify common principles that lead to a successful implementation of the presented (visualization) algorithms on the graphics hardware. The compact presentation of these principles will also help other researchers to apply them to visualization algorithms for additional adaptive data structures and for areas not covered here. The thesis concludes with an outlook on unsolved problems and directions for future research.

Kapitel 1

Einleitung

Für viele Disziplinen und Anwendungsgebiete gehört der Umgang mit Volumendaten mittlerweile zur alltäglichen Problemstellung und die direkte Volumenvisualisierung hat sich in diesem Zusammenhang als wichtiges Werkzeug für die effiziente Analyse dieser Volumendaten etabliert. Aufgrund der oftmals komplexen dreidimensionalen Struktur sowie der Verdeckungsproblematik ist dabei die Möglichkeit der Interaktion von essentieller Bedeutung für das rasche und umfassende Verständnis der dargestellen Daten. Als allgemein anerkannte Minimalanforderung gelten dabei Bildwiederholraten von 5-10 Bildern pro Sekunde.

Um diese zu erreichen wird seit einigen Jahren erfolgreich Graphik-Hardware eingesetzt, da diese aufgrund ihrer massiven Rechenleistung geradezu prädestiniert ist, um die hohe Berechnungskomplexität von Algorithmen zur direkten Volumenvisualisierung zu meistern. Die Einsetzbarkeit von Graphik-Hardware für Visualisierungszwecke war dabei allerdings bisher stark eingeschränkt, da Graphik-Hardware vor allem für die realistische Darstellung von 3D-Szenen und nicht gemäß den Anforderungen der wissenschaftlichen Visualisierung entworfen wird. Effiziente Hardware-Umsetzungen entstanden dort, wo sich die Struktur der Daten besonders für die Mechanismen, die bereits auf der Graphik-Hardware vorhanden sind, eignen. Dies trifft z.B. im Falle der Volumenvisualisierung auf uniformen 3D-Gittern zu, die mit Hilfe von 3D-Texturen implementiert werden kann.

Einfache Datenstrukturen stoßen an ihre Grenze, wo immer schnellere und größere Supercomputer Datensätze von mehreren Gigabyte bis Terabyte produzieren. Probleme, die diese großen Datensätze aufwerfen, sind Speicherbedarf und Verarbeitungsgeschwindigkeit, die sich beide durch adaptive beziehungsweise hierarchische Datenstrukturen und adaptive Algorithmen angehen lassen. Adaptive Datenstrukturen entstehen aber nicht nur durch die Aufbereitung der Daten zum Zwecke der Visualisierung. Vielmehr macht die Möglichkeit der lokalen Optimierung und der Anpassung an das Berechnungsgebiet sie darüber hinaus zu einer attraktiven Repräsentation für viele physikalische und mathematische Berechnungen. Einige prominente Beispiele sind Finite-Elemente-Netze, lokale Gitterverfeinerung oder gitterlose Repräsentationen.

Der Einsatz von Graphik-Hardware für die beschleunigte Visualisierung von Volumendaten auf solchen adaptiven Datenstrukturen war bisher nur sehr eingeschränkt möglich. So ist die Sor-

tierung von Tetraedern beziehungsweise Finiten-Elementen, die Voraussetzung für eine korrekte Volumendarstellung ist, nur sehr schwer auf Graphik-Hardware abzubilden. Ähnliches gilt für die in adaptiven beziehungsweise hierarchischen Gittern erforderliche Traversierung oder die in Punktmengen sehr aufwändige Dateninterpolation, und dies sind nur einige Beispiele.

Viele Autoren konvertieren die adaptiven Datenstrukturen daher in reguläre Gitter, was zu Fehlern führt, welche die Qualität der Visualisierung beeinträchtigen und damit das Vertrauen der Berechnungsingenieure in die dargestellten Ergebnisse unterminieren. Darüber hinaus entstehen wiederum riesige Datenmengen, wenn die Auflösung des Zielgitters so gewählt wird, dass die Details der adaptiven Datenstruktur noch repräsentiert werden können.

Aufgrund dieser Nachteile besteht ein großes Interesse daran, Visualisierungsalgorithmen zu entwerfen, die direkt auf den adaptive Datenstrukturen arbeiten aber dennoch von der Rechenleistung der Graphik-Hardware profitieren. Die Voraussetzung dafür wurde mit einem Paradigmenwechsel in der Architektur der Graphik-Hardware geschaffen, die weg von einer festen Verarbeitungspipeline hin zu programmierbaren Einheiten führte, die frei definierbare Assembler-artige Programme sowohl auf Vertex- als auch auf Fragmentebene ausführen können. Die damit erreichte Flexibilität ermöglicht es heute, komplexe Visualisierungsalgorithmen in den Verantwortungsbereich der Graphikkarte zu bringen und dabei auch komplexere Datenstrukturen zu verwenden. Dies soll mit dieser Arbeit demonstriert werden.

Die gesteigerte Flexibilität erschwert auf der anderen Seite den Entwurf geeigneter Visualisierungsalgorithmen, da nunmehr drei verschiedene Berechnungseinheiten (CPU, Vertex- und Fragmentprozessor) involviert sind, von denen jede eine begrenzte Rechenkapazität besitzt. Darüber hinaus spielt die Kommunikation zwischen diesen Einheiten und mit dem lokal angebundenen Speicher eine entscheidende Rolle. Da eine einzelne überlastete Einheit oder ein überlasteter Kommunikationspfad die Leistung des Gesamtsystems beschränkt, ist eine ausgewogene Balancierung erforderlich. Die Identifikation von Flaschenhälsen und der Entwurf von Strategien zu deren Beseitigung ist daher ein weiteres Ziel dieser Arbeit.

1.1 Adaptive Datenstrukturen

Nicht nur in der Medizin sind heutzutage Datensätze im Gigabyte-Bereich an der Tagesordnung, wie das Visible Human Projekt der National Library of Medicine [NLM 1987] demonstriert. Um den Speicherbedarf und die Berechnungskomplexität bei der Verarbeitung sehr großer Datensätze zu reduzieren, werden in der Visualisierung adaptive Datenstrukturen eingesetzt. Die Reduzierung der Zellenanzahl oder der Datenpunkte in homogenen, leeren oder uninteressanten Bereichen verringert die Datenmenge in einer adaptive Repräsentation. Werden darüber hinaus unterschiedliche Auflösungsstufen des Datensatzes erzeugt, kann die verwendete Aufösung an die verfügbare Rechenleistung angepasst werden.

In der Technik werden adaptive Datenstrukturen vorwiegend zum Zwecke der numerischen Simulation komplexer physikalischer Eigenschaften eingesetzt. Aufgrund der Numerik der verwendeten Lösungsverfahren sowie geometrischer Randbedingungen sind dabei unterschiedliche

Datenstrukturen im Einsatz. Die Finite-Elemente-Analyse, eine Standardmethode zur numerischen Lösung partieller Differentialgleichungssysteme, unterteilt beispielsweise das Simulationsgebiet in eine Menge diskreter Elemente (Hexaeder, Pyramiden oder Tetraeder), was eine einfache Anpassung auch an komplexe Simulationsgebiete, wie den Innenraum einer Fahrgastzelle, erlaubt. Da die Berechnungskomplexität mit der Anzahl der Elemente im Gitter steigt, werden kleine Zellen nur dort verwendet, wo der Diskretisierungsfehler eine Verfeinerung erforderlich macht.

Eine weitere verbreitete Datenstruktur sind adaptiv verfeinerte Gitter, die beispielsweise in Form der adaptiven Gitterverfeinerung (Adaptive Mesh Refinement) Mitte der 80er Jahre vor allem in der Strömungsmechanik populär gemacht wurden. AMR-Gitter kombinieren die Einfachheit der regulären Gitter mit der Möglichkeit, die Genauigkeit der Berechnung an lokale Effekte anpassen zu können. So erzeugen selbst große Simulationsgebiete, die aufgrund von Randbedingungen selbst für sehr eng begrenzte Phänomene erforderlich sein können, moderate Berechnungsgitter, was sich vorteilhaft auf Speicherbedarf und Berechnungskomplexität auswirkt.

Methoden, die ohne Berechnungsgitter auskommen (Meshless Methods), werden zunehmend in der Strukturmechanik eingesetzt. Da sie auf einer Menge von unabhängigen Punkten basieren, die nicht über ein Gitter verbunden sind, entfällt bei ihnen die aufwändige Vernetzung. Außerdem sind sie unempfindlich gegenüber typischen Problemen, wie Risspropagierung, Fragmentierung oder starken Deformationen, die bei Finite-Elemente-Methoden eine kontinuierliche und oftmals globale Neuvernetzung erforderlich machen können.

1.2 Gliederung der Arbeit

Die vorliegende Arbeit gliedert sich wie folgt. Nach dieser Einleitung gibt Kapitel 2 zunächst eine kurze Einführung in grundlegende Begriffe und Methodologien der wissenschaftlichen Visualisierung. Als Basis für die darauffolgenden Hardware-basierten Visualisierungsalgorithmen wird außerdem die Architektur der Graphik-Hardware mit ihrer Entwicklung zu frei programmierbaren Einheiten präsentiert.

Kapitel 3 dient der Vorstellung der in dieser Arbeit verwendeten Mechanismen und Algorithmen. Es behandelt daher insbesondere die Grundlagen der direkten Volumenvisualisierung, wobei sowohl die physikalischen Gesetzmäßigkeiten als auch ihre Anwendung und die am meisten verbreiteten Algorithmen kurz vorgestellt werden. Prä-Integrierte Klassifikation als eine der wichtigsten mathematischen Verbesserungen der letzten Jahre wird diskutiert, da sie eines der fundamentalen Prinzipien darstellt, die in dieser Arbeit ausgenutzt wurden.

Der Hauptteil der vorliegenden Arbeit umfasst Kapitel 4 bis 8 und diskutiert am Beispiel von Tetraedergittern (Kapitel 4 und 5) sowie adaptiv verfeinerten uniformen Gittern (Kapitel 7 und 8) unterschiedliche Strategien, um die Visualisierung auf adaptiven Datenstrukturen mit Hilfe der Graphik-Hardware zu beschleunigen. Jede von ihnen beseitigt spezifische Flaschenhälse bisheriger Lösungen. Die gefundenen Strategien dokumentieren gleichzeitig die Entwicklung der programmierbaren Graphik-Hardware, die in mehreren Schritten verlief.

Kapitel 4 zeigt, wie programmierbare Vertexverarbeitung ausgenutzt werden kann, um den Hauptprozessor des Rechners sowie die Datenübertragung zur Graphik-Hardware als limitierenden Faktor zu beseitigen. Dazu werden Tetraeder als neue graphische Primitive, ähnlich den Dreiecken für Oberflächengraphik, vorgeschlagen und demonstriert, wie eine direkte Rasterisierung von Tetraedern durch die Graphik-Hardware realisiert werden kann.

Algorithmen, die vollständig auf der Graphikkarte – genauer gesagt der Fragmenteinheit der Graphikkarte – ablaufen und damit nur noch durch die Rechenleistung der Graphik-Hardware beschränkt sind, werden in Kapitel 5 behandelt. Letztere wurden erst durch das Aufkommen programmierbarer Fragmenteinheiten möglich. Da hierbei auch die zu visualisierenden Daten direkt im lokalen Speicher der Graphikkarte gehalten werden müssen, werden in diesem Umfeld weiterhin Methoden zur Datenkompression diskutiert, als Möglichkeit dem begrenzten Texturspeicher zu begegnen.

Ein Ansatz, der mit einer universellen Datenrepäsentation anstelle einer auf Tetraedergitter angepassten Datenstruktur arbeitet, um die Volumendaten im Speicher der Graphikkarte zu repräsentieren, wird in Kapitel 6 vorgestellt. Er basiert auf einer prozeduralen Kodierung der Volumendaten mittels radialer Basisfunktionen und erlaubt die Visualisierung unterschiedlicher Datenstrukturen wie uniformer Gitter, Tetraedergitter und gitterloser Daten. Da moderne Graphikkarten die transparente Auswertung der radialen Basisfunktionen mit Hilfe programmierbarer Fragmenteinheiten unterstützen, kann diese prozedurale Kodierung mit einer Vielzahl von Visualisierungstechniken kombiniert werden. Dies wird in Kapitel 6 am Beispiel von skalaren und vektorwertigen Volumendaten demonstriert.

Kapitel 7 und 8 befassen sich mit adaptiv verfeinerten uniformen Gittern, die aufgrund ihrer lokal regelmäßigen Anordnung der Datenwerte besser für die Verarbeitung auf Graphik-Hardware geeignet sind und deswegen im Allgemeinen mit höheren Bildwiederholraten als Tetraedergitter dargestellt werden können. In Kapitel 7 werden drei Algorithmen vorgestellt, mit denen sich Tetraedergitter in adaptiv verfeinerte uniforme Gitter überführen lassen. Im Umfeld dieser drei Algorithmen, die in unterschiedlichem Maße von der Graphik-Hardware Gebrauch machen, wird eine optimale Verteilung zwischen Hauptprozessor und Graphik-Hardware entworfen. Mit der fehlerfreien und adaptiven Darstellung verfeinerter uniformer Gitter beschäftigt sich schließlich Kapitel 8. Typische Probleme, die dabei auftreten, wie die inkonsistente Interpolation zwischen unterschiedlich aufgelösten Gitterbereichen oder die durch adaptive Abtastung entstehenden Artefakte werden vorgestellt und entsprechende Lösungen präsentiert.

Kapitel 9 beschäftigt sich mit den Implementierungsaspekten einer adaptiven Volumendarstellung auf uniformen Gittern. Die in der Praxis sehr heterogene Graphikkarten-Landschaft erzwingt neben der Adaption an den Datensatz und die Darstellungsparameter auch die Anpassung an die verwendete Graphik-Hardware. Die in Kapitel 9 vorgestellte Architektur unterstützt dies durch ein flexibles und erweiterbares Shader-Konzept. Daneben werden Aspekte für die Volumenvisualisierung in virtuellen Umgebungen diskutiert. In Kapitel 10 werden die in Kapitel 4 bis 8 vorgestellten Algorithmen in einen Gesamtzusammenhang gebracht, der es erlaubt, gemeinsame Gestaltungsprinzipien für die erfolgreiche Umsetzung von (Visualisierungs-) Algorithmen auf die Graphik-Hardware zu identifizieren. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick auf offene Probleme und zukünftige Forschungsrichtungen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, die für das Verständnis der weiteren Arbeit erforderlich sind. Dazu wird zunächst der Begriff der Visualisierung eingeführt und seine typische Interpretation als rückgekoppelte Pipeline vorgestellt. Es schließt sich eine Übersicht über charakteristische Datentypen und Datenstrukturen an, die in der wissenschaftlichen Visualisierung verwendet werden. Die vorgestellte Klassifikation der Datenstrukturen erlaubt eine Einordnung der für diese Arbeit relevanten adaptiven Datenstrukturen.

Mit den typischen Verarbeitungsprinzipien innerhalb der Graphik-Hardware beschäftigt sich ein weiterer Abschnitt, der darüber hinaus die Entwicklung der Graphik-Hardware von einer relativ starren Verarbeitungspipeline zu frei programmierbaren Einheiten skizziert.

2.1 Visualisierung

Die wissenschaftliche Visualisierung zielt darauf, durch Sichtbarmachen der in abstrakten Daten verborgenen Informationen Erkenntnisse zu gewinnen und beim Betrachter Verständnis über die Natur der Daten zu erzeugen. In der Literatur wird dieser Prozess häufig als rückgekoppelte lineare Pipeline beschrieben, die den Datenfluss von einer Quelle, z.B. Messung oder Simulation, bis hin zum fertigen Bild beschreibt. Daneben stellt sie eines der wichtigsten Konzepte für die Diskussion von Visualisierungsalgorithmen dar. Sie besteht, wie in Abbildung 2.1 dargestellt, aus den Stufen Filterung, Mapping und Rendering.

Die Filterung selektiert eine Teilmenge der Daten, die analysiert werden soll, die sogenannten Visualisierungsdaten, und extrahiert sie aus einer potentiell großen Menge von Rohdaten. Irrelevante Datenpunkte oder Parameter werden aussortiert, was zu einer Ausdünnung führt. Auch Dateninterpolation oder Resampling werden häufig zu den Filteroperationen gezählt. Die gefilterten Daten werden im Mapping-Schritt in eine darstellbare Repräsentation überführt. Diese besteht aus einer Menge von geometrischen Primitiven, denen zusätzliche Attribute wie Position, Farbe, Textur oder Transparenz zugeordnet werden können. Die anschließende Darstellung der

geometrischen Repräsentation wird als Rendering bezeichnet und wird in modernen Visualisierungssystemen in der Regel durch die Graphik-Hardware ausgeführt.

Ein wesentlicher Aspekt innerhalb dieses ganzen Prozesses ist die Interaktion des Benutzers, da vor allem bei unbekannten Daten ein exploratives Vorgehen nötig sein kann, das eine interaktive Manipulation der Visualisierungsparameter auf allen Stufen der Pipeline erfordert. Für traditionelle (nicht-stereoskopische) Displays verbessert eine interaktive Darstellung darüber hinaus den Tiefeneindruck und in virtuellen Umgebungen zerstört eine nicht-interaktive Darstellung (weniger als 10 Hz) den Eindruck der Immersion. Um diese hohen Interaktionsraten zu erreichen, soll in dieser Arbeit Graphik-Hardware nicht nur für das Rendering eingesetzt werden. Stattdessen wird auf Rohdaten oder den bereits gefilterten Daten aufgesetzt.

2.2 Daten und Gittertypen

Je nach der Art der Messung oder Simulation können Daten in der wissenschaftlichen Visualisierung in unterschiedlicher Form und Struktur vorliegen. Typische Attribute, die in gebräuchlichen Klassifikationschemata unterschieden werden, sind die Dimensionalität des Datengebietes (0D, 1D, 2D, 3D ...), der Datentyp und die Datenstruktur.

Bezüglich der Datentypen wird zwischen Skalar-, Vektor- und Tensorfeldern unterschieden, je nachdem ob 1, n oder $n \times m$ Attribute kodiert sind. Besitzen die Datenattribute noch höhere Dimensionalität spricht man von Tensoren höherer Stufe. Sogenannte multivariate Daten unterscheiden sich von den erstgenannten dadurch, dass die unterschiedlichen Attribute unabhängige Größen beschreiben, die nicht in Korrelation zueinander stehen müssen. Ein Beispiel sind Druck und Temperatur. Diese Arbeit beschränkt sich auf die Betrachtung von Skalardaten, die im dreidimensionalen Raum definiert sind.

Eine Vielzahl von Datenstrukturen wurde vorgeschlagen, um die unterschiedlichen Anforderungen spezifischer Anwendungen zu erfüllen. Im Falle von Volumendaten bestehen sie in der

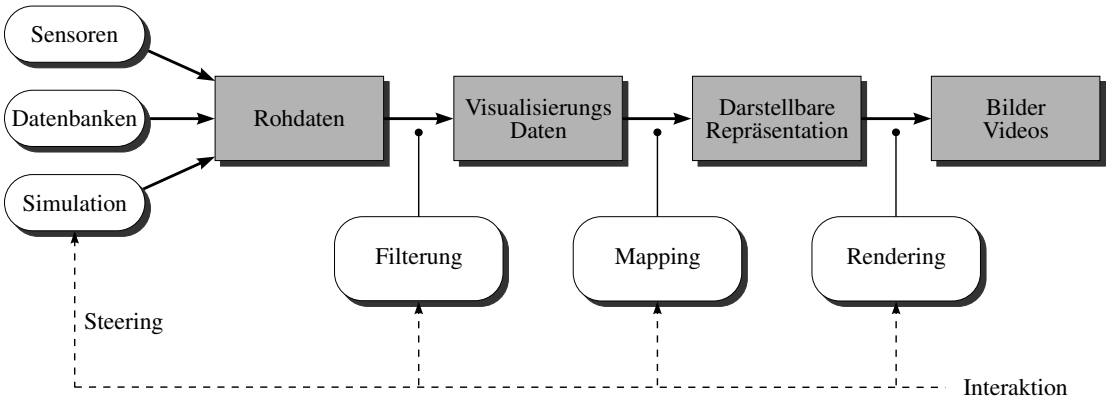


Abbildung 2.1: Die Visualisierungspipeline.

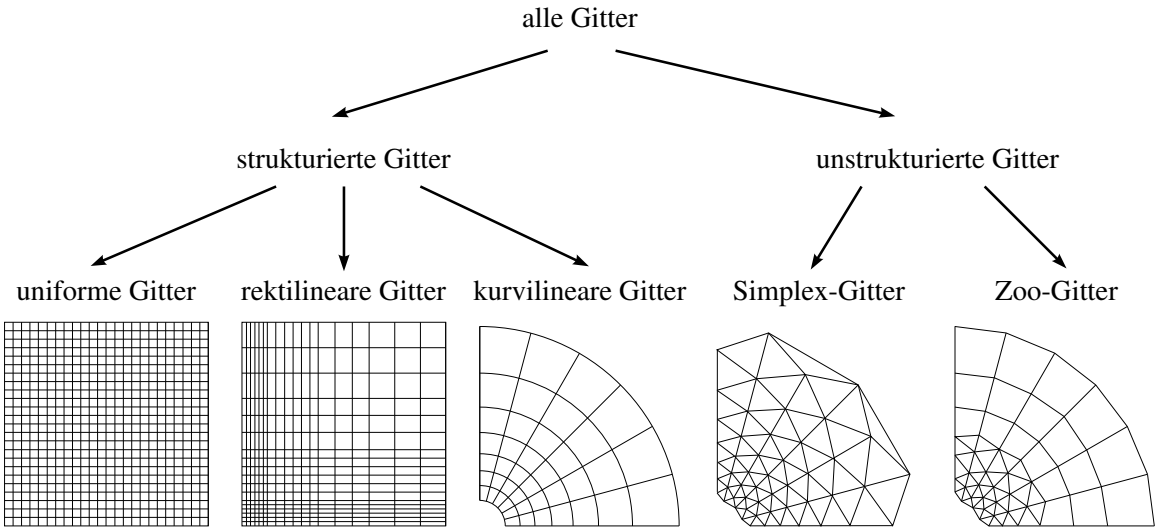


Abbildung 2.2: Klassifikation unterschiedlicher Gittertypen am zweidimensionalen Beispiel.

Regel aus einer Menge von Datenpunkten, die im dreidimensionalen Raum verteilt sind. Von Gittern spricht man, wenn es eine Verbindung (Konnektivität) zwischen den Datenpunkten gibt. Für gitterlose Daten ist auch der Begriff *Scattered Data* gebräuchlich.

2.2.1 Strukturierte Gitter

Abbildung 2.2 zeigt anhand von zweidimensionalen Beispielen eine grobe Klassifikation unterschiedlicher Gittertypen. Zunächst sind *strukturierte* und *unstrukturierte* Gitter zu unterscheiden. In strukturierten Gittern ist die Konnektivität implizit gegeben, in unstrukturierten Gittern ist sie frei definierbar, was auf der anderen Seite bedeutet, dass sie gesondert gespeichert werden muss und zusätzlichen Speicherplatz belegt.

Eine weitere Unterteilung der strukturierten Gitter kann in *uniforme*, *rektilineare* und *kurvilineare* Gitter erfolgen. In uniformen Gittern besitzen alle Zellen dieselbe Form und Größe. Eine Ausrichtung des Gitters an orthogonalen Achsen ist zwar nicht zwingend erforderlich wird aber häufig, insbesondere im Rahmen dieser Arbeit vorausgesetzt. Ausgehend von kubischen Zellen der Größe $\Delta x \times \Delta y \times \Delta z$ ergibt sich die Position der Vertizes $\mathbf{v}^{\text{uni}}_{i,j,k}$ eines uniformen Gitters der Dimension $n_x \times n_y \times n_z$ als

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

wobei $\mathbf{t}$ einen Translationsvektor und O eine Rotationsmatrix beschreibt.

Eine leichte Verallgemeinerung stellen die rektilinearen Gitter dar, bei welchen die Gitterabstände Δx, Δy und Δz nicht mehr konstant sind, sondern vom Index der Gitterposition abhängen

können. Entsprechend ergeben sich die Vertexpositionen zu

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

mit Koordinatenfunktionen $x(i), y(j)$ und $z(k)$.

Kurvilineare Gitter besitzen dieselbe Konnektivität wie uniforme oder rektilineare Gitter. Allerdings existiert hier keine implizite Definition für die Position eines Vertex $\mathbf{v}_{i,j,k}^{\text{kurv}}$. Stattdessen wird in einem kurvilinearen Gitter jeder Vertex als eigenständiger Vektor gespeichert.

2.2.2 Unstrukturierte Gitter

Unstrukturierte Gitter besitzen keine vorgegebene Konnektivität. Stattdessen wird neben den Vertexpositionen auch die Konnektivität explizit gespeichert. Der zusätzliche Speicherbedarf wird jedoch durch ein erhöhtes Maß an Flexibilität ausgeglichen. Unstrukturierte Gitter sind adaptiv, d.h. sie erlauben eine Anpassung an Berechnungsgebiete beliebiger Form und Topologie. Auch die Aufösung kann durch Variation der Zellengröße lokal adaptiert werden. Unstrukturierte Gitter spielen vor allem in der Finite-Elemente-Analyse eine wichtige Rolle, wo sie als Basis für die numerische Simulation dienen. Sie können anhand der Basisprimitive, aus denen sie konstruiert sind, weiter unterschieden werden.

Eine der wichtigsten Gruppen der unstrukturierten Gitter sind die *Simplex-Gitter*, also Gitter die lediglich aus den einfachsten n-dimensionalen Primitiven aufgebaut sind. Im Zweidimensionalen sind dies Dreiecke und im Dreidimensionalen Tetraeder. Die Möglichkeit der linearen Interpolation der Datenwerte macht sie zu einer attraktiven Plattform für eine ganze Reihe von Visualisierungsverfahren.

Daneben sind auch Gitter gebräuchlich, die aus unterschiedlichen Basisprimitiven, wie Prismen, Pyramiden oder Oktaedern aufgebaut sind. Solche Gitter werden auch als *Zoo-Gitter* bezeichnet. Zum Zwecke der Visualisierung werden sie häufig in ihre korrespondierenden Simplizes zerlegt. Die dadurch entstehende höhere Anzahl der Zellen wird in der Visualisierung oftmals durch die angenehmen mathematischen Eigenschaften der Simplizes aufgewogen. Außerdem vereinfacht die Beschränkung auf einen Zellentyp die Struktur der Visualisierungsalgorithmen deutlich. Die Zerlegung wird auch als *Tesselierung* bezeichnet.

2.2.3 Multiblock-Gitter

Neben den bisher behandelten einfachen Gittertypen werden vor allem im Bereich der Strömungssimulation sehr häufig sogenannte *Multiblock-Gitter* eingesetzt. Sie entstehen durch die Kombination mehrerer einfacher Gittertypen mit beliebiger Orientierung und beliebigem Datenlayout. Eine mögliche Klassifikation ergibt sich durch die Art und Weise wie die unterschiedlichen Gitterblöcke miteinander verbunden sind (Abbildung 2.3). *Konforme*-Gitterblöcke

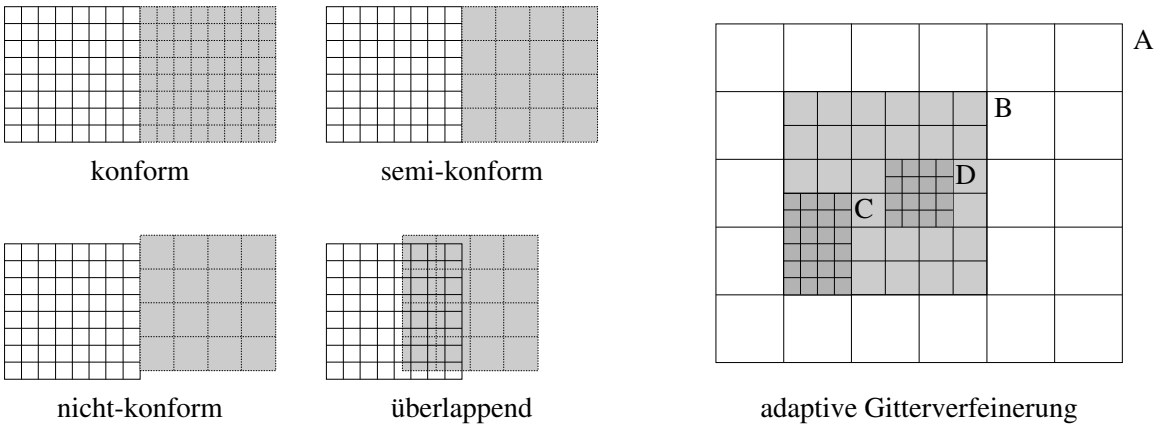


Abbildung 2.3: Klassifikation unterschiedlicher Multiblock-Gitter.

teilen sich alle Datenpunkte an der Verbindungsfäche, was in der Regel mit einer konsistenten Dateninterpolation über Gittergrenzen hinweg einher geht. Auch für *semi-konforme* Gitter, bei der Datenpunkte nur teilweise übereinstimmen, lässt sich ein konsistenter Anschluss in der Regel leicht konstruieren. Im Gegensatz dazu existiert für *nicht-konforme* oder *überlappende* Multiblock-Gitter bisher kein wohldefiniertes Interpolationsschema und es ist im Allgemeinen nicht klar, wie hier konsistente Übergänge zu schaffen sind.

Orthogonal dazu treten Multiblock-Gitter in hierarchischer und nicht-hierarchischer Form auf. Wichtige Vertreter für hierarchische Gitter sind der Oktalbaum und adaptive Gitterverfeinerungen. Ein Oktalbaum kodiert das Datengebiet durch rekursive Unterteilung ausgehend von einer einzigen Zelle, wobei in jeder Stufe durch gleichmäßige Teilung senkrecht zur x-, y- und z-Achse acht Nachfolger entstehen. Bei der Konstruktion von Oktalbäumen werden zwei Strategien unterschieden. Bottom-Up werden den inneren Knoten ausgehend von der feinsten Datenrepräsentation jeweils gefilterte Werte der acht Kindknoten zugewiesen. Bei Top-Down-Strategien entstehen die Werte an den Kindknoten ausgehend von den Eltern durch Verfeinerung der Berechnung.

Eine wichtige Spezialform der hierarchischen Multiblock-Gitter sind Adaptive Mesh Refinement (AMR) Gitter [Berger und Oliger 1984, Berger und Colella 1989]. Sie bestehen aus hierarchisch ineinander eingebetteten uniformen Gittern, wobei die Subgitter eine höhere Auflösung besitzen, was eine lokale Verfeinerung bewirkt. Für gewöhnlich erfüllen sie folgende Einschränkungen:

- Der Verfeinerungsfaktor ist ein ganzzahliger Wert, der sich jedoch für unterschiedliche Raumrichtungen unterscheiden kann.
- Subgitter sind komplett in ihren Eltern enthalten.
- Subgitter beginnen an den Zellgrenzen des Gitters, in das sie eingebettet sind, was bedeutet, dass eine Zelle des Eltern-Gitters entweder komplett verfeinert oder komplett unverfeinert vorliegt.

Der Vorteil der hierarchischen Gitter liegt darin, dass die Daten in unterschiedlichen Auflösungen vorliegen, was sich nicht nur zum Zwecke der Visualisierung ausnutzen lässt. Da für viele Algorithmen die Komplexität mit der Datenmenge steigt, erlaubt die adaptive Traversierung dieser Hierarchie eine Anpassung an die verfügbare Rechenkapazität. Aliasing-Artefakte lassen sich durch eine Anpassung an die Bildschirmauflösung reduzieren. Die Effektivität und Qualität einer Datenfilterung oder Segmentierung lässt sich durch Zugriff auf die in unterschiedlichen Auflösungsstufen kodierten unterschiedlichen Frequenzen in den Daten verbessern.

2.2.4 Punktbasierte Daten und Wavelets

Eine Wavelet-Repräsentation speichert ein gegebenes Datensignal als Hierarchie von hochpass- beziehungsweise tiefpassgefilterten Versionen des Signals mit jeweils reduzierter Auflösung. In jeder Stufe wird das Signal dabei durch Faltung mit einer sogenannten *Skalierungsfunktion* Φ^j in einen niederfrequenten Anteil sowie Detailinformation zerlegt, welche durch Filterung mit verschiedenen *Waveletfunktionen* Ψ_k^j entsteht. Der niederfrequente Anteil wird jeweils weiter unterteilt.

Die Rekonstruktion des Originalsignals entsteht umgekehrt durch Verdoppelung der Auflösung der Rekonstruktion auf der nächst gröberen Stufe, Filterung des Signals mit der inversen Skalierungsfunktion Φ'^j sowie den inversen Waveletfunktionen $\Psi_k'^j$ und Addition der Signalanteile. Ausgehend von der gröbsten Repräsentation erlaubt dies die iterative Rekonstruktion und Analyse des Signals auf verschiedenen Auflösungsstufen. Die Datenmenge verändert sich dabei nicht, da es sich bei der Wavelet-Transformation lediglich um eine Basistransformation handelt. Darüber hinaus ist die Zerlegung verlustfrei.

Die Besonderheit der Wavelet-Transformation liegt in der speziellen Wahl der Filterfunktionen, welche aus skalierten und translierten Versionen eines sogenannten *Mutterwavelets* bestehen und eine Orthonormalbasis eines hierarchischen Funktionenraums $V_0 \subseteq V_1 \subseteq \ldots$ des $L^2(\mathbb{R})$ aufspannen. Die fundierte mathematische Theorie hinter den Wavelets erlaubt akkurate Fehlerabschätzungen und macht sie damit zu einem interessanten Werkzeug nicht nur für Bild- und Signalverarbeitung sondern auch für die numerische Simulation.

Für die auf Wavelets basierende Visualisierung bieten sich zwei prinzipielle Vorgehensweisen an. Für die direkte Bilderzeugung aus der Wavelet-Repräsentation wurden eine Reihe von Techniken basierend auf Ray Casting vorgeschlagen [Westermann 1994, Ihm und Park 1998, Kim und Shin 1999, Rodler 1999]. Effizientere Methoden verwenden eine vorgelagerte (blockweise) Dekomposition der Wavelet-Repräsentation in (adaptive) uniforme Gitter, welche mittels Hardware-Beschleunigung mit hohen Interaktionsraten darstellbar sind [Guthe et al. 2002, Guthe und Straßer 2004, Strengert et al. 2004]. Die erfolgreiche Ausnutzung von Graphik-Hardware zur Wavelet-Dekomposition wurde in [Hopf und Ertl 2000] und kürzlich auf moderner programmierbarer Graphik-Hardware in [Hopf 2004] demonstriert.

Die Suche nach einem geeigneten Interpolationsschema stellt eine der größten Herausforderungen bei der Volumenvisualisierung punktbasierter Daten dar. Globale Interpolationsschemata sind dabei aufwändig, da sie den Einfluss aller Datenpunkte berücksichtigen müssen. Auf der

anderen Seite sind global glatte Approximationen mit lokalen Formulierungen nur schwer zu erreichen. Darüber hinaus macht die effiziente Suche nach lokal benachbarten Datenpunkten für eine gegebene Position im Raum eine hierarchische Strukturierung der Punktmenge erforderlich. In jüngster Zeit haben Methoden basierend auf radialen Basisfunktionen immer mehr Aufmerksamkeit erfahren, da dieses Interpolationsschema für seine exponentielle Konvergenz bekannt ist. Radiale Basisfunktionen wurden bereits erfolgreich in der wissenschaftliche Visualisierung eingesetzt [Co et al. 2003] und ermöglichen mittlerweile die Volumenvisualisierung punktbasierter Repräsentationen in Echtzeit [Jang et al. 2004].

2.3 Graphik-Hardware

Ziel dieser Arbeit ist es, Graphik-Hardware für die Volumenvisualisierung auf adaptiven Datenstrukturen einzusetzen, wobei der Fokus hier auf Standard-Graphik-Hardware liegt, die heutzutage relativ günstig ist und in nahezu jedem Schreibtisch-PC vorhanden ist. Die Entwicklung dieser Hardware über die letzten Jahre ist gekennzeichnet durch einen Paradigmenwechsel weg von einer festen Verarbeitungspipeline hin zu frei programmierbaren Einheiten, die es erst ermöglichten, komplexere Datenstrukturen zu verwenden oder komplexe Visualisierungsalgorithmen in den Verantwortungsbereich der Graphikkarte zu bringen. Ein wichtiger Schritt dahin war auch die Einführung der Datenverarbeitung mit Gleitkommagenauigkeit in allen Stufen der Graphikpipeline. Auch das Programmiermodell dieser neuen Funktionalität hat sich von einem Low-Level-Modell, das im Wesentlichen auf dem Setzen von Statusvariablen und Registerwerten der Graphikkarte basiert, zu einem High-Level-Modell mit kompilierten Shader-Hochsprachen wie z.B. Cg [nVidia 2002] oder OpenGL Shading Language [Kessenich et al. 2004] weiterentwickelt.

2.3.1 Rendering-Pipeline

Ausgangspunkt für die Hardware-beschleunigte Darstellung mit Hilfe von Graphikkarten ist eine dreidimensionale Szenenbeschreibung aus einfachen graphischen Primitiven, wie Punkten, Linien oder Dreiecken. Diese sind entweder explizit gegeben oder entstehen durch eine Unterteilung (Tesselierung) komplexerer Objekte durch die CPU. Die Überführung dieser dreidimensionalen Szenenbeschreibung in ein zweidimensionales Rasterbild ist bei nahezu allen Graphikkarten als feste Abfolge von Verarbeitungsschritten ausgeführt [Foley et al. 1990], die in Form einer Pipeline, wie sie in Abbildung 2.4 dargestellt wird, organisiert ist. Eingabe dieser Pipeline ist ein Strom von Vertizes, die durch ihre Position und weitere Attribute wie Farbe, Opazität oder Normale beschrieben werden. Als Ausgabe entsteht ein Pixelbild, das auf dem Monitor angezeigt wird. Die folgende Beschreibung der Pipeline basiert auf dem OpenGL-API [Shreiner et al. 2004], ist aber auf andere Programmierschnittstellen, insbesondere DirectX [Microsoft 2004], übertragbar. Als Untereinheiten der Rendering-Pipeline können die Geometrieverarbeitung, die Rasterisierung und die Fragmentoperationen identifiziert werden.

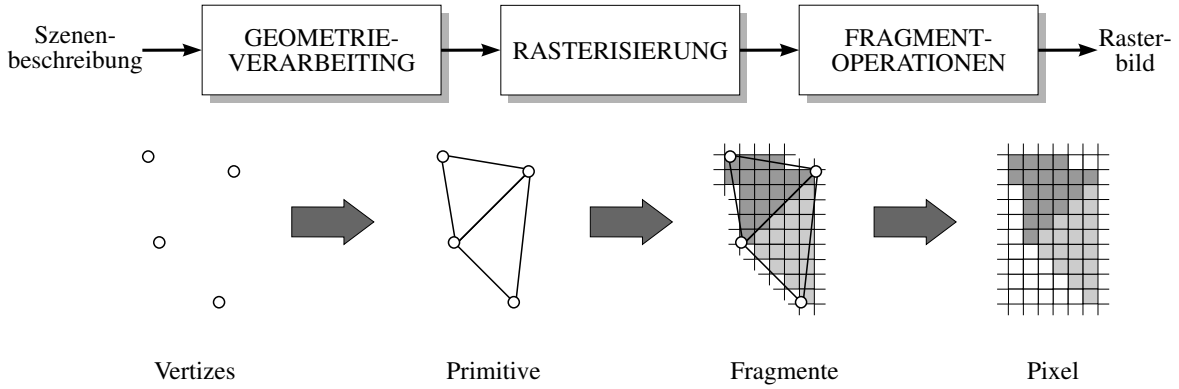


Abbildung 2.4: Standard-Rendering-Pipeline.

Geometrieverarbeitung

In der Geometrieeinheit der Graphikkarte werden Operationen auf den Vertizes ausgeführt. Dazu zählen affine Transformationen wie Translation, Rotation, Skalierung und Projektion der Vertizes. Daneben wertet die Geometrieeinheit lokale Beleuchtungsmodelle aus, weshalb diese Stufe auch oft als Transform & Lighting Einheit (T&L) bezeichnet wird. Die einzelnen Aufgaben der Geometrieeinheit können, wie in Abbildung 2.5 dargestellt, in folgende Blöcke unterteilt werden:

Modelview Transformation: Graphische Primitive werden in der Regel in einem lokalen Koordinatensystem definiert. Ihre Position und Orientierung in der Szene wird mit Hilfe der sogenannten Modelling Transformation spezifiziert, welche die Primitive in Weltkoordinaten überführt. Diese Transformation wird oftmals mit der Viewing Transformation kombiniert, welche die Position und Orientierung der virtuellen Kamera beschreibt. Diese kombinierte Modelview Transformation kann als 4×4 Matrix beschrieben werden, sofern die Berechnung mittels homogener Koordinaten erfolgt.

Beleuchtungsberechnung: In der klassischen Rendering-Pipeline erfolgt die Beleuchtungsberechnung je Vertex mit Hilfe des Phong Modells [Phong 1975]. Da dieses Informationen über Normalen und die Blickrichtung benötigt, kann die Beleuchtungsberechnung erst nach der Modelview Transformation erfolgen. Moderne programmierbare Fragmenteinheiten erlauben die Auswertung lokaler Beleuchtungsmodelle auch pro Fragment (Phong Shading) und beseitigen dadurch typische Probleme, wie sie beispielsweise bei einer vertexbasierten Berechnung spekularer Schlaglichter entstehen. Auch in diesem Fall bietet sich aus Gründen der Performanz an, räumlich linear variierende Terme der Beleuchtungsgleichung je Vertex zu berechnen.

Primitive Assembly: Die bisher beschriebenen Schritte der Geometriepipeline arbeiten auf einem Strom unkorrelierter Vertizes. Diese werden beim Primitive Assembly zu Linien und Polygonen verbunden. Auch die Tesselierung von Polygonen in einzelne Dreiecke findet

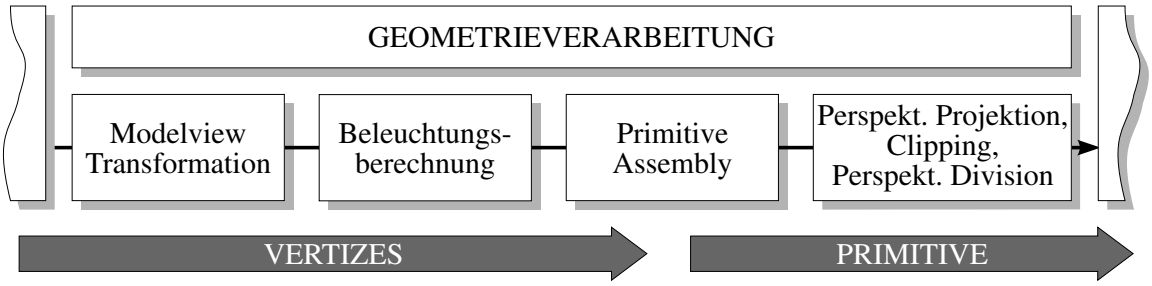


Abbildung 2.5: Geometrieverarbeitung als Teil der Standard-Rendering-Pipeline.

hier statt. Die Auswahl der Vertizes, die zu einem Primitiv beitragen, ist abhängig von dessen Typ, so wie er im Rahmen des Graphik-APIs spezifiziert wurde. So bilden in einer Liste von Dreiecken beispielsweise Blöcke von drei aufeinanderfolgenden Vertizes jeweils ein Dreieck, d.h. jeweils drei Vertizes aus dem Vertexstrom resultieren in einem Primitiv. Effizientere Typen erlauben die Generierung eines Primitivs mit jedem aus dem Strom gelesenen Vertex. Bei Dreiecksfächern (*Triangle Fans*) beispielsweise wird ein bestimmtes Dreieck durch den aktuellen Vertex, den vorangegangenen Vertex und den ersten Vertex im Fan bestimmt. In Dreiecksstreifen (*Triangle Strips*) ist ein einzelnes Dreieck durch den aktuellen und die beiden vorangegangenen Vertizes im Strom definiert. Für die effiziente Verarbeitung dieser komplexeren Primitivtypen stehen auf der Graphik-Hardware in der Regel effiziente Caching-Mechanismen für bereits transformierte Vertizes zur Verfügung.

Perspektivische Projektion: In dieser Stufe der Geometrieverarbeitung erfolgt die perspektivische Verzerrung der geometrischen Primitive. Dies geschieht wiederum durch Multiplikation der Weltkoordinaten mit einer 4×4 Matrix und resultiert in den sogenannten *Clip-Koordinaten*. Die Perspektivische Projektion definiert ebenfalls das sogenannte *View-Frustrum*, das Volumen, welches bestimmt, welche Objekte oder Objektteile auf dem Bildschirm sichtbar sind.

Clipping: Bei den nach der Projektion in Clip-Koordinaten lebenden Primitiven werden die Teile entfernt, die nicht auf dem Bildschirm zu sehen sind. Dies erfolgt durch Line Clipping bzw. Polygon Clipping am View-Frustrum.

Perspektivische Division: Am Ende der Geometriepipeline steht die Division der Koordinaten durch die jeweilige w-Komponente und durch Vernachlässigung der z-Komponente entstehen Primitive, die in der zweidimensionalen Bildebene leben.

Rasterisierung

Die Rasterisierung konvertiert die geometrischen Primitive in sogenannte Fragmente. Jedes Fragment korrespondiert zu einem Pixel im fertigen Bild, besitzt neben der Farbe jedoch noch zusätzliche Informationen, wie Tiefenwert oder Opazität, die beim Schreiben in den Framebuffer eine

Abbildung 2.6: Rasterisierung als Teil der Standard-Rendering-Pipeline.

Rolle spielen. Die Erzeugung dieser Fragmente kann entsprechend Abbildung 2.6 in drei Unterschritte zerlegt werden.

Polygonrasterisierung: Die Rasterisierung erzeugt die Menge an Fragmenten, die innerhalb des betrachteten Dreiecks liegt. Sie umfasst auch die Interpolation der visuellen Attribute wie Farbe, Normale, Texturkoordinaten oder Tiefenwert aus den Attributen der Vertizes.

Texturgenerierung: Texturen dienen dazu, Attribute zu definieren, die sich je Fragment ändern. Sie sind ein wichtiger Mechanismus zur Modellierung realistisch wirkender Oberflächen. In der Stufe der Texturgenerierung werden die Texturen, in der Regel zweidimensionale Rasterbilder, entsprechend den an den Vertizes definierten Texturkoordinaten auf die Dreiecke abgebildet. Für jedes Fragment werden die Texturkoordinaten perspektivisch korrekt linear interpoliert und mit der resultierenden Koordinate ein Nachschlag in der Textur durchgeführt. Bei diesem Prozess kann eine Interpolation der Texturwerte erforderlich sein, da in den wenigsten Fällen eine Texel-Position exakt getroffen wird. Das Ergebnis des Nachschlags ist ein Farbwert, oder ein anderes Fragmentattribut, welches für die Bestimmung der Fragmentfarbe herangezogen wird.

Texturanwendung: Die aus den Texturen nachgeschlagenen Werte können auf unterschiedliche benutzerdefinierbare Weise mit den übrigen Attributen des Fragments, insbesondere der primären Fragmentfarbe, kombiniert werden.

Fragmentoperationen

Die aus der Rasterisierung stammenden Fragmente werden letztlich in den Framebuffer geschrieben, um sie am Bildschirm darzustellen. Dabei werden sie mit den Werten, die bereits im Framebuffer gespeichert sind, kombiniert. Vor dem Schreiben durchlaufen die Fragmente eine Reihe von Tests, die abhängig von bestimmten Parametern und Bedingungen entscheiden, ob das Fragment geschrieben wird oder nicht. Die einzelnen Stufen der Fragmentverarbeitung sind in Abbildung 2.7 dargestellt:

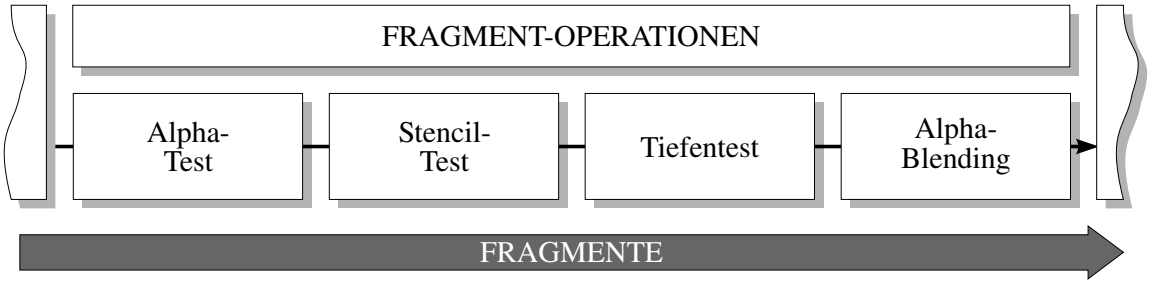


Abbildung 2.7: Fragmentoperationen als Teil der Standard-Rendering-Pipeline.

Alpha-Test: Ein Fragment kann abhängig vom Vergleich der Opazität des Fragments mit einem benutzerdefinierten Referenzwert verworfen werden.

Stencil-Test: Über den Stencil-Test können bestimmte geometrische Bereiche im Framebuffer für das Schreiben gesperrt werden. Die Schreibmaske wird dabei durch den Stencilbuffer definiert, der für jedes Pixel im Framebuffer einen entsprechenden Stencil-Wert speichert. Abhängig vom Vergleich dieses Wertes mit einen Referenzwert können Fragmente verworfen werden.

Tiefentest: Der Tiefentest ist der Standardmechanismus zur korrekten Ermittlung der Verdeckung im Bildraum. Dazu wird ein Tiefenpuffer, auch Z-Buffer genannt, verwendet, der den Tiefenwert jedes Pixels enthält. Ein neues Fragment wird mit dem Tiefenwert des korrespondierenden Pixels verglichen und bei einem größeren Abstand zum Betrachter verworfen.

Die Farbberechnung für Fragmente, die durch den Tiefentest verworfen werden, trägt nicht zum endgültigen Bild bei und stellt damit einen vermeidbaren Mehraufwand dar. Da dieser Mehraufwand insbesondere im Zusammenhang mit moderner programmierbarer Graphik-Hardware und immer komplexer werdenden Shader-Programmen signifikant sein kann, enthält moderne Graphik-Hardware oftmals einen zusätzlichen Tiefentest unmittelbar nach der Rasterisierung, der auch als vorgelagerter Tiefentest (*Early Depth Test*) bezeichnet wird. Sein effektiver Einsatz ist allerdings an gewisse Randbedingungen der gewählten Fragmentverarbeitung geknüpft.

Alpha-Blending: Die letzte Fragmentoperation kombiniert die Farbe des neu zu zeichnenden Fragments mit der Farbe des korrespondierenden Pixels oder überschreibt den alten Farbwert komplett. Aus der Menge der möglichen Blending-Operatoren ist dabei vor allem der *Over-Operator* [Porter und Duff 1984] relevant, da er für die Darstellung semitransparenter Objekte verwendet wird. Im Kontext von texturbasierter Volumenvisualisierung (Abschnitt 3.3.2) wird er für die Akkumulation des Volumenintegrals verwendet.

2.3.2 Programmierbare Graphik-Hardware

Die im vorangegangenen Abschnitt vorgestellte grundlegende Pipeline erfuhr in den letzten Jahren einen deutlichen Wandel. Die festverdrahtete Funktionalität, die nur wenig Einflussmöglichkeiten bot, wurde sukzessive durch flexiblere Einheiten ergänzt, was letztlich zu frei programmierbaren Vertex- und Fragmenteinheiten führte. Diese Entwicklung wurde insbesondere durch die Computerspieleindustrie vorangetrieben. Sie verlangt nach neuer Funktionalität für immer realistischere Darstellungen, die sich mit der fest verdrahteten Funktionalität nicht oder nur durch aufwändige Algorithmen mit mehreren Zeichendurchgängen erreichen lässt. Dieser Abschnitt beschreibt die wichtigsten Meilensteine auf dem Weg zu flexibel programmierbarer Graphik-Hardware.

Um die neue Funktionalität ansprechen zu können, definiert OpenGL einen Erweiterungsmechanismus, mit dem Hardware-Hersteller das OpenGL-API um neue Funktionsaufrufe ergänzen können. Für Programme, welche die neuen Möglichkeiten nicht nutzen, sind die Erweiterungen transparent, d.h. sie laufen ohne Modifikation auf Systemen mit oder ohne Unterstützung für eine bestimmte Erweiterung. Nur Programme, die eine Erweiterung verwenden sollen, müssen angepasst werden. Manche dieser Erweiterungen wurden im Laufe der Zeit in die Kernfunktionalität von OpenGL aufgenommen. Als zentrale Dokumentationsstelle für OpenGL-Erweiterungen dient die OpenGL Extension Registry [OpenGL ARB 2004].

Multi-Texturen

Eine erste wichtige Erweiterung der beschriebenen Pipeline wurde mit OpenGL 1.2 eingeführt. Es handelt sich um die Möglichkeit, innerhalb der Rasterisierungsstufe mehrere Texturwerte für ein Fragment nachzuschlagen und diese zu einem einzigen Farb- und Opazitätswert zu kombinieren. Eine typische Anwendung dafür findet sich in der Spieleprogrammierung, wo eine Textur, welche die Farbwerte eines Objekts enthält, mit der vorberechneten Beleuchtung aus sogenannten Light Maps oder Spiegelungen der Umgebung aus einer Environment Map kombiniert werden.

In OpenGL 1.2 ist Multi-Texturierung als strenge Sequenz von Texturierungsstufen spezifiziert, wobei in jeder Stufe ein eingehender Farbwert mit dem in der Textur nachgeschlagenen Farbwert kombiniert wird. Das Ergebnis dient dann als Eingabe für die nächste Stufe. Die Art der Kombination wird über die Texturumgebung der jeweiligen Texturierungsstufe bestimmt, wofür verschiedene fest vorgegebene Operatoren existieren. Einige wenige dieser Operatoren werden direkt von OpenGL definiert, daneben existieren eine Reihe von Erweiterungen, wie z.B. `ARB_texture_env_combine` oder `NV_texture_env_combine4` [OpenGL ARB 2004], welche mächtigere Operationen zur Verfügung stellen.

Konfigurierbare Rasterisierung

Die nächsten Stufe der Flexibilisierung wird repräsentiert durch die nVidia-Erweiterung `NV_register_combiners` [OpenGL ARB 2004], die mit der GeForce256 Graphikkarte eingeführt wurde. Sie ersetzt die Texturanwendung (siehe Abschnitt 2.3.1), also die Kombination von Fragmentfarbe und nachgeschlagenen Texturwerten, durch flexiblere Einheiten, sogenannte Combiner. Ein Combiner arbeitet auf einer Menge von Eingaberegistern, die mit Attributen des Fragments wie Farbe und Texturwerte aber auch benutzerdefinierbaren Konstanten gefüllt werden können. Diese Eingabewerte können mit einem festen Satz von arithmetischen Operationen wie Addition, Multiplikation oder Skalarprodukt verarbeitet und in Ausgaberegister geschrieben werden. Komplexere Operationen lassen sich durch eine Hintereinanderschaltung mehrerer Combiner erreichen, wobei bei der GeForce256 zwei, auf späteren Graphikkarten acht Combiner zur Verfügung stehen.

Parallel dazu wurde auch die Texturgenerierung flexibilisiert. Mit Hilfe der Erweiterung `NV_texture_shader` [OpenGL ARB 2004] tritt an die Stelle der fixen Abbildung von Texturkoordinaten auf gefilterte Texturwerte eine vierstufige programmierbare Einheit, in der jede Stufe eines von 21 möglichen Texture-Shader-Programmen ausführen kann. Das Ergebnis des Texturnachschlags in jeder Stufe hängt vom gewählten Programm und dem Ergebnis vorangegangener Stufen ab. Dadurch wurden erstmals abhängige Texturnachschläge (Dependent Texture Lookup), also Texturnachschläge, die als Eingabe das Ergebnis eines vorangegangenen Nachschlags verwenden, möglich. Da die Ergebnisse aller vier Texturstufen den Register Combiners als Eingabewerte zur Verfügung stehen, ist damit bereits eine sehr flexible Berechnung der endgültigen Fragmentfarbe möglich.

Auf Graphikkarten der Firma ATI steht ähnliche Funktionalität seit der Radeon 8500 zur Verfügung. Die OpenGL-Erweiterung `ATI_fragment_shader` kombiniert dabei die flexible Texturgenerierung und Texturanwendung in einer Einheit. Die Berechnung der endgültigen Fragmentfarbe erfolgt hier in zwei Phasen, die jeweils aus sechs Texturzugriffen und danach acht arithmetischen Operationen bestehen. Diese feste Reihenfolge macht, anders als bei den nVidia-Erweiterungen, mehrfach indirekte Texturzugriffe unmöglich, erlaubt aber durch eine größere Auswahl von Operationen ein breiteres Anwendungsspektrum.

Vertexprogramme

Der nächste logische Schritt lag in der Flexibilisierung der Geometriepipeline (siehe Abschnitt 2.3.1), die mit der Erweiterung `NV_vertex_program` beziehungsweise `ARB_vertex_program` erstmal bei dem nVidia GeForce3 Chip auftauchte. An die Stelle der fixen Modelview Transformation, Beleuchtungsberechnung und perspektivischen Projektion treten dabei kleine Assembler-ähnliche Programme, denen ein Register-Programmiermodell zugrunde liegt, so wie es in Abbildung 2.8(a) dargestellt ist. Eingabe dieser Programme sind neben den Vertexattributen die von außen setzbaren konstanten Programmparameter. Letztere lassen sich direkt oder indirekt über ein Adressregister ansprechen. Die Ausgabe erfolgt über verschiedene Ergebnisregister. Der Befehlssatz ist für vier-komponentige Vektoren optimiert und arbeitet

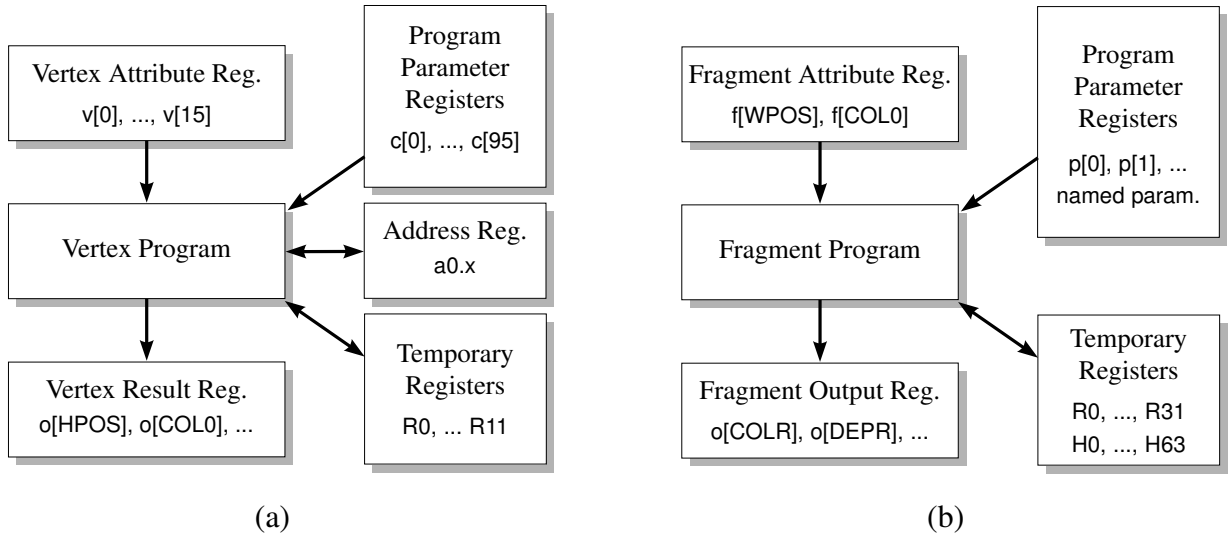


Abbildung 2.8: Vertex- und Fragmentprogramme als Ersatz für die statische T&L Berechnung beziehungsweise die Berechnung der Fragmentfarbe. Hier am Beispiel der Erweiterungen `NV_vertex_program` beziehungsweise `NV_fragment_program`.

ausschließlich auf Gleitkommawerten. Er stellt neben arithmetischen Instruktionen auch eine Reihe von Sonderfunktionen zur Verfügung, die komplexe Funktionen für die Beleuchtungsberechnung ausführen.

Erste Versionen der Vertexprogramme mussten noch ohne bedingte Anweisungen, Sprünge oder Schleifen auskommen. Diese wurden mit der OpenGL-Erweiterung `NV_vertex_program2` eingeführt. Jüngste Erweiterungen (`NV_vertex_program3`) ergänzen die Geometriepipeline sogar um die Möglichkeit von Texturnachschlägen auf der Vertexebene. Die wichtigste Einschränkung der Vertexprogramme ist, dass sie nach einen sogenanntes *Streaming*-Programmmodell arbeiten. Das bedeutet, dass Vertizes weder gelöscht noch erzeugt werden können und auch keine Information zwischen aufeinanderfolgenden Vertizes ausgetauscht werden kann.

Fragmentprogramme

Das erfolgreiche Modell der Vertexprogramme wurde bald auch auf die Fragmentebene übertragen. Seit der nVidia GeForceFX Graphikkarte ergänzen voll programmierbare Einheiten (siehe Abbildung 2.8(b)) die konfigurierbare Rasterisierung. Sie werden in einer Assembler-ähnlichen Sprache programmiert, wobei die Programme je nach verwendetem API Fragmentprogramme, Fragment-Shader oder Pixel-Shader genannt werden. Der Instruktionssatz, der in der Erweiterung `NV_fragment_program` beziehungsweise `ARB_fragment_program` definiert wird, ist dem der Vertexprogramme sehr ähnlich und enthält neben Vektoroperationen und skalaren Operationen Spezialbefehle für den Texturnachschlag und für das Verwerfen eines Fragments (Kill). Fragmentprogramme können neben Farbwerten auch Tiefenwerte ausgeben.

Eine wesentliche Errungenschaft im Zusammenhang mit Fragmentprogrammen war die Einführung von Gleitkommagenauigkeit in allen Stufen der Rasterisierungspipeline, einschließlich neuer Texturformate mit Gleitkommagenauigkeit und der Ausgabe in spezielle Gleitkomma-Framebuffer. Primär erlaubt dies höhere Präzision bei der Berechnung von Beleuchtungseffekten. Tatsächlich wurde dadurch aber der Weg für universelle Berechnungen auf der Graphikkarte geebnet, was mittlerweile unter dem Schlagwort „General Purpose Graphics Processing Unit (GPGPU)" ein aktives Forschungsgebiet ist. Ein guter Überblick dazu findet sich auf der Website von Mark Harris[1].

High-Level-Programmierschnittstellen

Viele der im letzten Abschnitt vorgestellten Ansätze zur Einführung von Programmierbarkeit in die Graphikpipeline sind als herstellerspezifische OpenGL-Erweiterungen ausgeführt, was die Ausnutzung dieser neuen Funktionalität schwierig machte und die Portabilität von Anwendungen zwischen unterschiedlichen Graphikkarten einschränkte. Verstärkt wurde dieser Effekt durch eine sehr inhomogene Schnittstellen-Landschaft mit unterschiedlicher Syntax und Semantik zur Realisierung derselben Funktionalität.

Mit der Standardisierung der programmierbaren Vertex- und Fragmenteinheiten durch das OpenGL Architectural Review Board trat zwar eine Verbesserung dieser Situation ein, dennoch bleibt die Programmierung der Einheiten in Assembler eine zeitraubende und fehlerträchtige Aufgabe. Die Entwicklung geht daher zunehmend zu abstrakten Shading Sprachen, wie sie im Bereich der nicht-interaktiven Graphik seit Jahren erfolgreich verwendet werden (siehe z.B. Renderman [Hanrahan und Lawson 1990]).

Shader, die in einer Hochsprache geschrieben sind, werden dabei von einem Compiler, der im Graphiktreiber oder als externes Programm realisiert ist, in Befehle übersetzt, die von der Graphikkarte interpretiert werden können. Neben der einfacheren Programmierung bietet dieses Konzept die Möglichkeit zur Virtualisierung der Graphik-Hardware, d.h. der Compiler kümmert sich um die Eigenheiten der spezifischen Hardware-Plattform und kann Beschränkungen hinsichtlich der Länge der Shader-Programme oder der Anzahl verfügbarere Ressourcen transparent für den Programmierer durch mehrere Rendering-Durchläufe auflösen.

Die erste dieser High-Level Shading Languages (HLSL) im Kontext von OpenGL war das von nVidia entwickelte „C for Graphics" (Cg) [Mark et al. 2003], dessen C-ähnliche Programme mit Hilfe eines externen Compilers in Fragment- und Vertexprogramme übersetzt werden. Prinzipiell erlauben dabei spezielle Compiler Backends die Anpassung an unterschiedliche Hardware-Architekturen und ermöglichen herstellerunabhängige Implementierungen. Diese Idee wird allerdings durch die Tatsache unterminiert, dass Backends die Sprachdefinition erweitern dürfen, wovon nVidia im Hinblick auf Anpassungen an die eigenen Graphikkarten ausgiebig Gebrauch gemacht hat. Bisher hat kein anderer Hersteller Backends für Cg veröffentlicht.

In der Zukunft dürfte Cg daher schnell an Bedeutung verlieren, insbesondere, da seit September 2004 die offizielle Spezifikation von OpenGL 2.0 [Segal und Akeley 2004] verfügbar ist,

[1] http://www.gpgpu.org/

die eine eigene Shading-Sprache [Kessenich et al. 2004] enthält. Dagegen spricht möglicherweise die Tatsache, dass Cg auch Backends für DirectX mitbringt, was es ermöglicht, denselben Shader für OpenGL und DirectX zu übersetzen. Allerdings besitzt DirectX auch eine eigene Shader-Hochsprache namens HLSL. Ein weiteres Problem von Cg ist, dass es als Aufsatz oberhalb des Graphikkarten-Treibers ausgelegt ist. Es dauert daher länger, bis neue Graphikkarten-Funktionalität in Cg zur Verfügung steht.

Kapitel 3

Volumenvisualisierung

Der Begriff Volumenvisualisierung [Drebin et al. 1988] beschreibt eine Menge von Techniken zur Darstellung dreidimensionaler Skalardaten, die im mathematischen Sinne als Funktion $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ beschrieben werden können. In der wissenschaftlichen Visualisierung spielen sie eine immer größere Rolle. Volumendaten entstehen aus Messungen, z.B. mittels Verfahren der medizinischen Bildgebung wie Computertomographie (CT) oder Magnetresonanztomographie (MR). Eine weitere Quelle sind Simulationen, wie sie beispielsweise im Rahmen der Strömungssimulation (Computational Fluid Dynamics – CFD) durchgeführt werden. Oft werden Volumendaten auch direkt berechnet oder synthetisch generiert, um Flüssigkeiten oder gasförmige Objekte wie Wolken, Nebel und Feuer darzustellen.

Bei den Verfahren zur Volumenvisualisierung lassen sich grob indirekte und direkte Methoden unterscheiden. Indirekte Methoden extrahieren zunächst eine Zwischenrepräsentation aus dem Volumen, welche dann oftmals mit geringerem Aufwand dargestellt werden kann. Einer der bekanntesten Vertreter dieser Klasse von Algorithmen ist die Darstellung von Isoflächen, d.h. der Menge aller Punkte, deren Skalarwert einem gewissen Schwellwert (Isowert) entspricht. Prinzipiell kann dies dadurch geschehen, dass entweder zunächst ein Dreiecksgitter der Isofläche erzeugt wird [Lorensen und Cline 1987], welches im Anschluss als reguläres Graphikprimitiv dargestellt wird, oder durch direkte Extraktion beim Rendering [Levoy 1988, Westermann und Ertl 1998]. Ein Nachteil dieser Methoden ist, dass durch die Beschränkung der Darstellung auf eine Teilmenge des Volumens ein Informationsverlust auftritt und dem Betrachter damit oftmals zu wenig Kontextinformation zur Verfügung steht, um die Volumendaten richtig zu interpretieren.

Die Methoden zur direkten Volumenvisualisierung versuchen dieses Manko zu beseitigen, indem sie eine „ganzheitliche" Darstellung der Volumendaten generieren. Dazu wird das Volumen als Wolke aus selbstleuchtendem Gas betrachtet, dem in jedem Punkt ein Farb- und Opazitätswert zugewiesen wird. Diese Zuweisung erfolgt abhängig von einer Dichteverteilung, welche durch die in den Volumendaten kodierte physikalische Eigenschaft definiert ist. Die Bilderzeugung erfolgt dabei direkt aus dem Volumen, indem für jedes Pixel das sogenannte *Volume Rendering Integral* durch Verfolgung der korrespondierenden Sehstrahlen durch das Volumen bestimmt wird. Diese Arbeit beschäftigt sich vor allem mit Methoden zur direkten Volumenvisualisierung, wes-

halb hier zunächst ein Einblick in die physikalischen Grundlagen gegeben wird, bevor einige grundlegende Konzepte und Algorithmen vorgestellt werden, die zum weiteren Verständnis notwendig sind. Im Folgenden meint der Begriff Volumenvisualisierung immer direkte Volumenvisualisierung.

3.1 Mathematische Grundlagen

Die direkte Bilderzeugung aus den Volumendaten basiert auf physikalischen Gesetzmäßigkeiten für die Interaktion von Licht mit Flächen und Volumen. Diese werden für die Näherung geometrischer Optik, das heißt unter Vernachlässigung von Beugung und Interferenz, durch die Transporttheorie beschrieben, deren Grundlage der Energieerhaltungssatz ist. Unter Vernachlässigung von Streuungseffekten führt dies zum sogenannten *Volume Rendering Integral* [Krüger 1990, Hege et al. 1993], dessen (approximative) Auswertung für alle Bildpunkte des Zielbildes der Kern aller Volumenvisualisierungsalgorithmen ist.

3.1.1 Strahlintegration

Mathematisch erfolgt die Auswertung des Volumenintegrals durch Integration der abgeschwächten Farbwerte und Absorptionskoeffizienten entlang des Sehstrahls durch jeden Bildpunkt. Dabei wird eine Parametrisierung des Sehstrahls $\mathbf{x}(\lambda)$ nach dem Abstand λ zur Betrachterposition vorausgesetzt. An jedem Punkt im Raum ist eine Farbdichte $\mathrm{color}(\mathbf{x})$ und eine Absorptionsdichte $\mathrm{extinction}(\mathbf{x})$ definiert. Ihre Einheiten sind Farbintensität bzw. Absorptionsstärke je Längeneinheit. Im Weiteren wird aber nur von Farb- bzw. Absorptionskoeffizient die Rede sein, wenn die genaue Bedeutung aus dem Kontext klar wird. Somit ergibt sich unter der Voraussetzung, dass keine Farbdichte für λ größer als ein maximaler Abstand D zum Betrachter und kleiner als 0 existiert, die Intensität eines Bildpunkts als

$$I = \int_0^D \mathrm{color}\big(\mathbf{x}(\lambda)\big) \exp\left(- \int_0^\lambda \mathrm{extinction}\big(\mathbf{x}(\lambda')\big) \mathrm{d}\lambda' \right) \mathrm{d}\lambda . \tag{3.1}$$

Für die Anwendung dieser Gleichung auf ein kontinuierliches Skalarfeld $s(\mathbf{x})$ muss jedoch noch die Berechnung der Farb- und Absorptionskoeffizienten spezifiziert werden. Dabei werden zwei Schritte unterschieden: die *Klassifikation* weist einem Skalarwert eine *Primärfarbe* und einen Absorptionskoeffizienten zu. Der Begriff Primärfarbe ist dabei OpenGL entlehnt, um anzudeuten, dass es sich um die Farbe vor dem *Shading* handelt. Für die Klassifikation werden Transferfunktionen für die Farbdichte $\tilde{c}(s)$ und die Absorptionsdichte $\tau(s)$ verwendet, welche die Skalarwerte $s(\mathbf{x})$ auf Farb- und Absorptionskoeffizienten abbilden.

Der zweite Schritt besteht aus dem eigentlichen *Shading* und berechnet die Farbbeiträge eines betrachteten Raumpunktes, d.h. die Funktion $\mathrm{color}(\mathbf{x})$. In das Shading fließt die Primärfarbe ein, aber ggf. auch andere Parameter wie der Gradient $\nabla s(\mathbf{x})$, ambiente und diffuse Farbe usw. Für

die folgenden Überlegungen spielt das Shading zunächst keine Rolle, weshalb hier von einem trivialen Shading ausgegangen wird, welches das Ergebnis der Klassifikation $\tilde{c}\big(s(\mathbf{x})\big)$ direkt als Farbwert verwendet. In analoger Weise wird der Absorptionskoeffizient mit $\tau\big(s(\mathbf{x})\big)$ identifiziert. Somit ergibt sich das Volume Rendering Integral als:

$$I = \int_0^D \tilde{c}\Big(s\big(\mathbf{x}(\lambda)\big)\Big) \exp\left(-\int_0^\lambda \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big) \mathrm{d}\lambda'\right) \mathrm{d}\lambda\,. \tag{3.2}$$

3.1.2 Optische Modelle

Im Zusammenhang mit dem Shading von Volumendaten tritt häufig der Begriff des optischen Modells in Erscheinung, welches die physikalischen Prinzipien hinter der Beleuchtungsberechnung zusammenfasst. Einfache Modelle, wie sie vor allem für Echtzeit-Algorithmen zur direkten Volumenvisualisierung verwendet werden, berücksichtigen dabei lediglich Emission und Absorption, während anspruchsvollere Modelle auch Streuungseffekte, Schatten und den Einfluss externer Lichtquellen berücksichtigen [Max 1995]. Obwohl die meisten optischen Modelle auf physikalischen Prinzipien basieren, haben sich in der Praxis auch alternative Mechanismen bewährt. An dieser Stelle soll lediglich ein kurzer Überblick über die gebräuchlichsten Modelle gegeben werden.

Emission und Absorption

Dieses am weitesten verbreitete Modell wurde bereits in Abschnitt 3.1.1 skizziert. Hinter ihm steht die Vorstellung eines selbstleuchtenden Gases, bei dem der Raum mit Partikeln erfüllt ist, die Licht emittieren und absorbieren. Die Dichte der Partikel entspricht dem Skalarwert des Volumens, d.h. die direkte Abbildung des Skalarwertes auf Emissions- beziehungsweise Absorptionskoeffizienten je Raumpunkt ist mit der Vorstellung konsistent, dass jedes Partikel ein gewisses Quantum an Licht emittiert bzw. absorbiert. Es ergibt sich die bereits vorgestellte Gleichung (3.2).

Reine Emission

Da bei diesem Modell keine Absorption auftritt, fällt der entsprechende Term im Volume Rendering Integral weg, die Vereinfachung ergibt:

$$I = \int_0^D \tilde{c}\Big(s\big(\mathbf{x}(\lambda)\big)\Big) \mathrm{d}\lambda\,. \tag{3.3}$$

Reine Absorption

Beim diesem Modell ist das Volumen erfüllt von kalten, perfekt schwarzen Partikeln, welche alles Licht, das auf sie eintrifft, verschlucken. Daher ist ein ambientes Umgebungslicht I_0 erforderlich,

damit überhaupt Licht auf der Bildebene ankommt. Die entsprechende Vereinfachung für das Volume Rendering Integral lautet

$$I = I_0 \exp\left(- \int_0^{\lambda} \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big) \mathrm{d}\lambda' \right). \tag{3.4}$$

Streuung, Beleuchtung und Schatten

Weitere Modelle berücksichtigen neben Emission und Absorption auch Streuung von Lichtquellen außerhalb des Volumens. Unter der Voraussetzung, dass das Licht ungehindert durch das Volumen wandern kann, kann die Berechnung der zusätzlichen Intensität dabei durch ein lokales Beleuchtungsmodell (z.B. Blinn-Phong [Phong 1975]) basierend auf den Gradienten des Volumens erfolgen. Wird das Licht beim Durchgang durch das Volumen abgeschwächt, ist für jeden Volumenpunkt zusätzlich die Integration der Absorptionsanteile entlang eines Strahls zur Lichtquelle erforderlich. D.h. die Berechnung von color$(\mathbf{x})$ beinhaltet selbst wieder eine Integration, was diese Modelle bisher für interaktive Volumendarstellung disqualifizierte. Durch die Verwendung flexibler Graphik-Hardware werden aber auch Methoden verfügbar, welche diese Effekte berücksichtigen [Kniss et al. 2002, Kniss et al. 2003].

Multiple Streuung betrachtet nicht nur Licht, das direkt von der Lichtquelle auf ein Partikel trifft. Zusätzlich wird eintreffendes Licht berücksichtigt, welches (mehrmals) an anderen Partikeln gestreut wurde. Interaktives Rendering mit physikalisch korrekter Mehrfachstreuung ist eines der ungelösten Probleme bei der Volumendarstellung.

Maximum Intensity Projection

Als Alternative zur Lösung für das Volume Rendering Integral wird vor allem in der Medizin häufig *Maximum Intensity Projection* (MIP) eingesetzt. In diesem Fall ist eine numerische Integration nicht erforderlich. Stattdessen ergibt sich die Intensität als Maximum der Farbwerte entlang des Sehstrahls:

$$I = \max_{0 \le \lambda \le D} \tilde{c}\Big(s\big(\mathbf{x}(\lambda)\big)\Big) \tag{3.5}$$

Der Vorteil von MIP liegt darin, dass besonders für Daten aus Computer-*Angiographien* unmittelbar aussagekräftige Darstellungen der Gefäßstruktur sichtbar werden, weil das dabei verwendete Konstrastmittel neben Knochenstrukturen das höchste Signal erzeugt. Eine besondere Wahl der Transferfunktion ist dabei nicht erforderlich. Es reicht die direkte Abbildung von Skalarwerten auf Farbkoeffizienten.

Mathematisch gesehen bieten sowohl das reine Emissions- und Absorptions-Modell als auch MIP den Vorteil, dass sie kommutativ sind. Dies erlaubt eine Betrachtung des Sehstrahls in Segmenten, die in beliebiger Reihenfolge verarbeitet werden können, und ermöglicht dadurch eine

effizientere algorithmische Umsetzung. Auf der anderen Seite geht dadurch die Tiefeninformation in den Ausgabebildern verloren, was die Gefahr der Missinterpretation der räumlichen Lage von Strukturen mit sich bringt.

3.1.3 Prä- und Post-Klassifikation

Techniken zur direkten Volumenvisualisierung unterscheiden sich teilweise deutlich darin, wie sie die Volume Rendering Gleichung (3.2) auswerten. Ein wichtiger und grundlegender Unterschied liegt in der Berechnung von $\tilde{c}(s(\mathbf{x}))$ und $\tau(s(\mathbf{x}))$. Oftmals wird das Skalarfeld $s(\mathbf{x})$ auf einem Gitter definiert, wobei der Skalarwert an einem beliebigen Punkt aus den Werten an den Gitterknoten $\mathbf{v}_i$ interpoliert wird.

Die Reihenfolge von Interpolation und Anwendung der Transferfunktion definiert den Unterschied zwischen *Prä-* und *Post-Klassifikation*. Bei Prä-Klassifikation findet die Anwendung der Transferfunktion vor der Interpolation statt, d.h. an den Gitterknoten $\mathbf{v}_i$ erfolgt die Berechnung der Farb- und Absorptionskoeffizienten, welche dann im Anschluss verwendet werden, um den Farbkoeffizienten $\tilde{c}(s(\mathbf{x}))$ und den Absorptionskoeffizienten $\tau(s(\mathbf{x}))$ zu interpolieren. Im Gegensatz dazu werden bei Post-Klassifikation die Farb- und Absorptionskoeffizienten aus dem interpolierten Skalarwert $s(\mathbf{x})$ bestimmt.

Offensichtlich liefern Prä- und Post-Klassifikation unterschiedliche Ergebnisse, sobald die Interpolation nicht mit der Anwendung der Transferfunktion vertauschbar ist. In uniformen Gittern ist dies typischerweise der Fall, da hier trilineare Interpolation üblich ist, die nur dann mit der Anwendung der Transferfunktion vertauschbar ist, wenn diese konstant ist oder der Identität entspricht. Ähnliches gilt für prozedural beschriebene Volumen, bei denen in der Regel Interpolationsfunktionen höherer Ordnung verwendet werden. Für Tetraedergitter ist zumindest eine lineare Transferfunktion erforderlich, damit sie mit der linearen Interpolation vertauschbar ist.

In allen anderen Fällen liefern Prä- und Post-Klassifikation unterschiedliche Ergebnisse, wobei von den meisten Autoren Post-Klassifikation als korrekt angesehen wird, da sie dem Verständnis eines kontinuierlichen (interpolierten) Skalarfeldes entspricht. Dennoch hat auch Prä-Klassifikation seine Berechtigung, insbesondere im Zusammenhang mit texturbasierter Volumenvisualisierung, da manche Hardware-Architekturen nur Prä-Klassifikation direkt unterstützen. Segmentierung im medizinischen Umfeld, also die Zuweisung bestimmter Volumenelemente zu Strukturen, wie zum Beispiel Organen, ist ebenfalls eine Prä-Klassifikationstechnik.

3.1.4 Numerische Integration

Die gebräuchlichste numerische Approximation für das Volume Rendering Integral aus Gleichung (3.2) ist die Berechnung einer Riemannsumme für n Strahlsegmente der jeweils gleichen Länge $d := D/n$. (Siehe auch Abbildung 3.1 und Abschnitt IV.A in [Max 1995]). Die folgenden Betrachtungen lassen sich jedoch leicht auf Strahlsegmente unterschiedlicher Länge verallgemeinern.

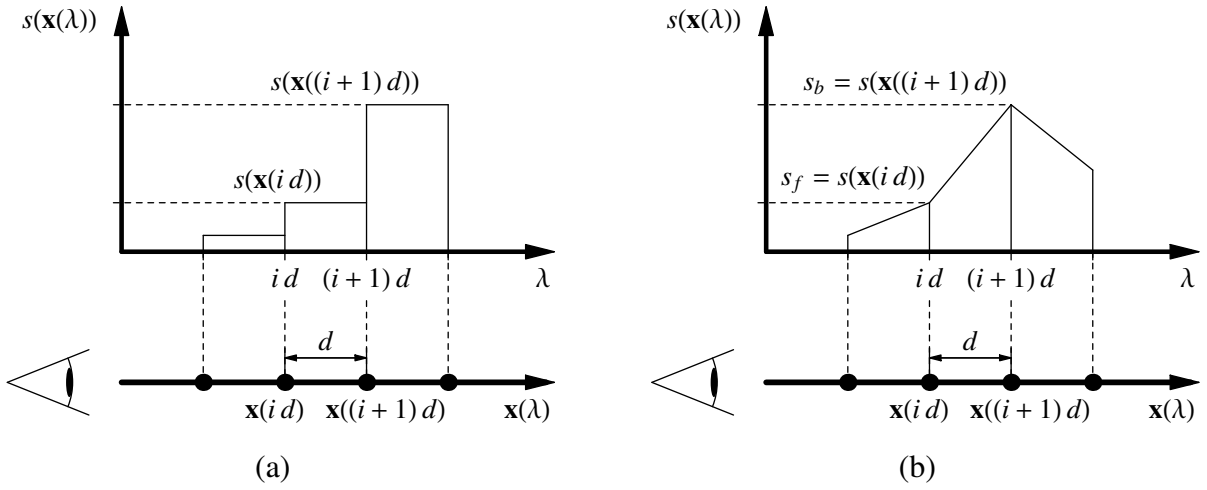


Abbildung 3.1: **(a)** Stückweise konstante Approximation der Funktion $s(\mathbf{x}(\lambda))$ entlang des Sehstrahls. **(b)** Stückweise lineare Approximation von $s(\mathbf{x}(\lambda))$.

Zunächst wird die *Opazität* α_i des i-ten Strahlsegments definiert und unter der Annahme eines stückweise konstanten Skalarwertes, wie in Abbildung 3.1(a) dargestellt, approximiert:

$$\alpha_i := 1 - \exp\left(-\int_{i\,d}^{(i+1)d} \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big)\mathrm{d}\lambda'\right) \tag{3.6}$$

$$\approx 1 - \exp\Big(-\tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)d\Big) \tag{3.7}$$

Das Ergebnis wird für kleine Werte von d häufig weiter durch $\alpha_i \approx \tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)d$ abgeschätzt. Der Term $1-\alpha_i$ wird auch als *Transparenz* des i-ten Strahlsegments bezeichnet.
In ähnlicher Weise lässt sich die emittierte Farbe $\tilde{C}_i$ des i-ten Strahlsegments, die als

$$\tilde{C}_i := \int_{i\,d}^{(i+1)d} \tilde{c}\Big(s\big(\mathbf{x}(\lambda)\big)\Big) \exp\left(-\int_{i\,d}^{\lambda} \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big)\mathrm{d}\lambda'\right)\mathrm{d}\lambda \tag{3.8}$$

definiert ist, approximieren. Die Selbstabschwächung innerhalb des Strahlsegments wird dabei in der Regel vernachlässigt, was durch $-\int_{i\,d}^{(i+1)d}\tau(s)\mathrm{d}\lambda' = 0$ ausgedrückt wird und in der Vereinfachung $\tilde{C}_i \approx \tilde{c}\big(s(\mathbf{x}(i\,d))\big)d$ resultiert, die letztlich auf die bekannte diskrete Form der Volume Rendering Gleichung führt:

$$I \approx \sum_{i=0}^{\lfloor D/d \rfloor} \tilde{C}_i \prod_{j=0}^{i-1}(1-\alpha_j). \tag{3.9}$$

Die Aufsummierung der Beiträge erfolgt in einem iterativen Prozess, wobei zwei Traversierungsstrategien unterschieden werden. Bei *Back-to-Front Compositing* werden die Sehstrahlen begin-

nend hinter dem Volumen auf den Betrachter zu abgearbeitet. Die entsprechende Compositing-Gleichung lautet:

$$\tilde{C}'_i = \tilde{C}_i + (1-\alpha_i)\tilde{C}'_{i+1} \qquad \text{bzw.} \qquad \tilde{C}'_i = \alpha_i C_i + (1-\alpha_i)\tilde{C}'_{i+1} \tag{3.10}$$

wobei die linke Version *vormultiplizierte* Farben verwendet, die in der Literatur alternativ auch mit den Begriffen *opazitätsgewichtete Farben* oder *assoziierte Farben* bezeichnet werden. Im Zusammenhang mit dem Mischen von Bildern ist die Compositing-Gleichung auch unter dem Begriff *Over-Operator* bekannt [Porter und Duff 1984].

Eine Alternative stellt das *Front-to-Back Compositing* dar, welches bei der klassischen Strahlverfolgung mittels Ray Casting Anwendung findet. Hierbei wird parallel zur Akkumulation der Farbe $\tilde{C}'_i$ zusätzlich in jedem Compositing-Schritt die akkumulierte Opazität α'_i benötigt.

$$\tilde{C}'_i = \tilde{C}'_{i-1} + (1-\alpha'_{i-1})\tilde{C}_i \qquad \text{und} \qquad \alpha'_i = \alpha'_{i-1} + (1-\alpha'_{i-1})\alpha_i. \tag{3.11}$$

Die Compositing-Gleichungen lassen sich unmittelbar aus dem Volume Rendering Integral ableiten und gelten unabhängig von der verwendeten Näherungsmethode.

3.1.5 Prä-Integrierte Klassifikation

Ein wesentliches Problem der im letzten Abschnitt vorgestellten Integrationsmethode für das Volume Rendering Integral ist die diskrete Abtastung, die nur dann zu einer hinreichenden Approximation führt, wenn die Abtastung mit einer Rate erfolgt, die höher als die Nyquist-Frequenz ist. Dabei reicht es nicht, die Grenzfrequenz des Skalarfeldes zu betrachten, weil die Integration über die aus der Transferfunktion nachgeschlagenen Farb- und Absorptionskoeffizienten erfolgt. Ausschlaggebend für die Abtastung ist vielmehr das Produkt der Nyquist-Frequenzen des Skalarfeldes $s(\mathbf{x})$ mit dem Maximum der Nyquist-Frequenzen der Transferfunktionen $\tau(s)$ und $\tilde{c}(s)$ – beziehungsweise $c(s)\tau(s)$ im Falle vormultiplizierter Farben (siehe auch [Schulze et al. 2003]).

Sprünge in der Transferfunktion, wie sie in Abbildung 3.2 beispielsweise zur Simulation einer Isoflächendarstellung verwendet wurden, führen zu hohen Frequenzen in der Transferfunktion und erfordern eine unter Umständen massive Überabtastung des Volumens, die auf Kosten der Performanz geht. Eine nicht an die Transferfunktion angepasste Abtastrate führt zu Sampling-Artefakten.

Um dieses Problem zu beheben schlagen Kraus, Röttger und Ertl eine Verbesserung der Approximation für das Volume Rendering Integral vor, welche den Einfluss der Transferfunktion aus der Strahlintegration beseitigt und damit eine Abtastung des Skalarfeldes unabhängig von den Eigenschaften der Transferfunktion ermöglicht. Das Verfahren ist heute unter dem Namen *Prä-Integrierte Klassifikation* bekannt und wurde erstmals im Zusammenhang mit der Visualisierung von Tetraedergittern [Röttger et al. 2000] veröffentlicht. Später erfolgte eine Adaption auf texturbasierte Volumenvisualisierung [Engel et al. 2001]. Für Ray Casting wurde es erstmals in [Knittel 2002] verwendet.

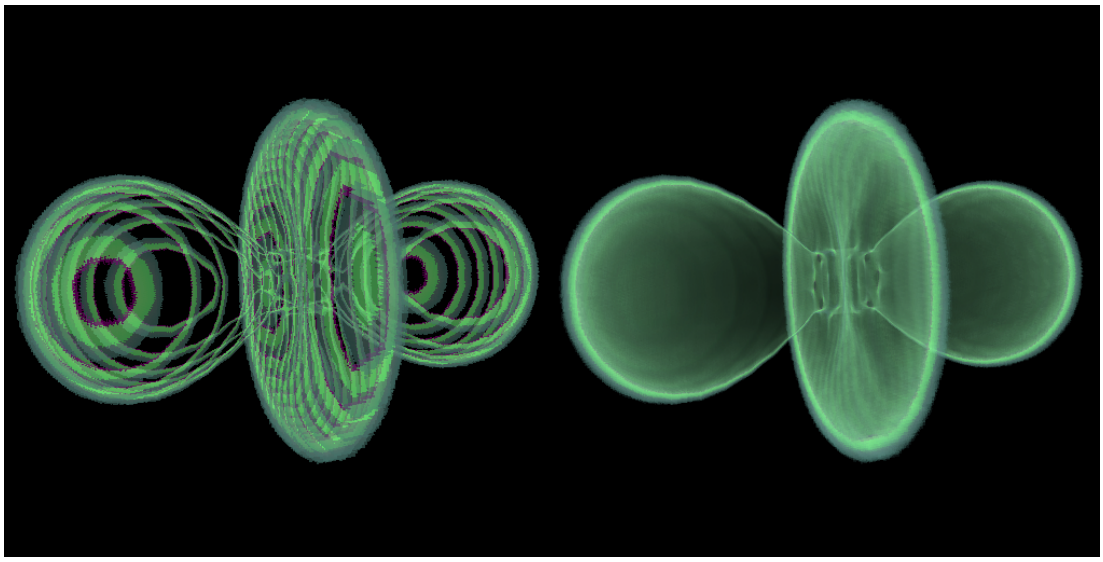


Abbildung 3.2: Vergleich von Volumendarstellungen mit Post-Klassifikation (links) und Prä-Integration (rechts). In beiden Fällen wird dieselbe Abtastrate verwendet.

Die Idee beruht auf der Konstruktion eines stückweise linearen Skalarfeldes aus den Abtastpunkten entlang des Sehstrahls (siehe Abbildung 3.1(b)). Sie erlaubt die Rekonstruktion des Skalarwertes an jedem Punkt des i-ten Strahlsegments durch lineare Interpolation aus dem Skalarwert am Start (Front) des Strahlsegments $s_f := s\big(\mathbf{x}(id)\big)$ und dem Skalarwert am Ende (Back) des Strahlsegments $s_b := s\big(\mathbf{x}((i+1)d)\big)$. Für die Approximation der Opazität α_i des i-ten Strahlsegments ergibt sich dann:

$$\alpha_i = 1 - \exp\Bigg(-\int_{i\,d}^{(i+1)d} \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big)\,\mathrm{d}\lambda'\Bigg) \tag{3.12}$$

$$\approx 1 - \exp\Bigg(-\int_0^1 \tau\big((1-\omega)s_f + \omega s_b\big)d\,\mathrm{d}\omega\Bigg) \tag{3.13}$$

In analoger Weise lassen sich die (assoziierten) Farben $\tilde{C}_i$ annähern:

$$\tilde{C}_i \approx \int_0^1 \tilde{c}\big((1-\omega)s_f + \omega s_b\big)\exp\Bigg(-\int_0^\omega \tau\big((1-\omega')s_f + \omega' s_b\big)d\,\mathrm{d}\omega'\Bigg)d\,\mathrm{d}\omega. \tag{3.14}$$

Der wesentliche Vorteil dieser Approximation ist, dass die Integrale über die Transferfunktionen in Tabellen vorberechnet werden können, wobei die Strahlparameter s_f, s_b und d als Indizes in die Tabelle dienen. Der Nachschlag beseitigt die Notwendigkeit für eine Überabtastung des Volumens. Wie in [Engel et al. 2001] gezeigt wurde, kann damit sowohl die Darstellungsqualität (weniger Unterabtastung) als auch die Geschwindigkeit (weniger Abtastpunkte) erhöht werden. Ein Vergleich der Volumendarstellung mit Post-Klassifikation und Prä-Integration unter gleichen Abtastbedingungen ist in Abbildung 3.2 zu sehen.

3.2 Beschleunigung der Prä-Integration

Wie in Abschnitt 3.1.5 vorgestellt, basiert die Prä-Integrierte Klassifikation auf Tabellen der für beliebige Strahlstücke vorberechneten Volumenintegrale. Die Tabellen enthalten die vorintegrierte Farbe $\tilde{C}(s_f, s_b, d)$ und die vorintegrierte Opazität $\alpha(s_f, s_b, d)$ für n diskrete Werte von s_f beziehungsweise s_b. Die Berechnung dieser Tabellen ist relativ aufwändig und muss für jede Modifikation der Transferfunktion wiederholt werden. Oftmals erfordert die Suche nach geeigneten Transferfunktionen den Eingriff des Benutzers, der dazu interaktives Feedback benötigt. Es wurden daher eine Reihe von Möglichkeiten vorgeschlagen, diese Berechnungen zu beschleunigen [Engel et al. 2001, Röttger und Ertl 2002, Lum et al. 2004].

Eine interessante Möglichkeit stellt Berechnung mittels Graphik-Hardware dar [Röttger und Ertl 2002], da die einfache Parallelisierbarkeit der numerischen Integration der Architektur der Graphik-Hardware entgegenkommt. Darüber hinaus benötigen Hardwareunterstützte Visualisierungsalgorithmen die Prä-Integrationstabellen in der Regel als Textur, was im Falle einer Berechnung durch die CPU zusätzlichem Mehraufwand durch den Transfer der Daten zur Graphikkarte und durch die Reinitialisierung der Textur erfordert. Durch eine Berechnung direkt auf der Graphikkarte fällt dieser Mehraufwand weg, da ein Rendering direkt in die Prä-Integrationstextur erfolgen kann. Letzteres erfordert jedoch spezielle OpenGL-Erweiterungen und ist damit auf bestimmte Graphik-Hardware beschränkt. Auch ist die Berechnung anfällig für Fehler durch die teilweise reduzierte interne Berechnungsgenauigkeit mancher Graphikkarten. Hier sollen daher vor allem algorithmische Optimierungen betrachtet werden, da diese im Rahmen der Numerik exakt sind und universell, insbesondere auch für die Berechnung auf der CPU, ausgenutzt werden können.

Eine erste naheliegende Optimierungsmöglichkeit ergibt sich aus der Tatsache, dass lokale Modifikationen eine inkrementelle Aktualisierung der Prä-Integrationstabelle erlauben. Für einen geänderten Skalarwert s müssen lediglich diejenigen Einträge von $\tilde{C}(s_f, s_b, d)$ und $\alpha(s_f, s_b, d)$ neu berechnet werden, für die s im Bereich $s_f \leq s \leq s_b$ beziehungsweise $s_b \leq s \leq s_f$ liegt. Schlimmstenfalls betrifft dies die Hälfte aller Einträge. Am Anfang der Suche nach einer geeigneten Transferfunktion stehen jedoch oftmals globale Änderungen, wo der Beschleunigungseffekt durch lokale Aktualisierung gering ist. Hier sind Methoden zur Reduzierung der Berechnungskomplexität erforderlich.

Für die komplette Berechnung der Tabelle mit n Diskretisierungsstufen für s_f beziehungsweise s_b liegt der Aufwand bei $O(n^4)$, weil die Tabelle drei Dimensionen besitzt und jeder Eintrag die Integration über die Länge des Strahlsegments erfordert. In [Engel et al. 2001] wurde gezeigt, dass unter einer Näherung, welche die Selbstabschwächung innerhalb eines Strahlsegments vernachlässigt, der Aufwand auf $O(n^3)$ reduziert werden kann. Dazu werden tabellierte Stammfunktionen für die Integrale $\int \tau(s)\mathrm{ds}$ und $\int \tilde{c}(s)\mathrm{ds}$ berechnet, aus denen sich die Einträge der Prä-Integrationstabelle jeweils mit konstantem Aufwand ableiten.

Für viele Anwendungen ist darüber hinaus eine Reduktion der Prä-Integrationstabelle, die eigentlich von den drei Parametern s_f, s_b und d abhängt, möglich. Hier kann die Dimensionalität der Tabelle durch Beschränkung auf einen einzigen Wert d reduziert werden. Dies gilt zum Beispiel

für Ray Casting mit äquidistantem Abstand der Abtastpunkte oder texturbasierte Volumenvisualisierung mit Schnittebenen, die parallel zur Bildebene ausgerichtet sind. Der Berechnungsaufwand sinkt entsprechend von $O(n^4)$ auf $O(n^3)$. Ohne die Beschränkung auf nur eine Strahlsegmentlänge oder die Verwendung von Approximationen kommen Methoden zur inkrementellen Berechnung aus.

3.2.1 Inkrementelle Prä-Integration

Inkrementelle Prä-Integration [Weiler et al. 2003a] nutzt die Tatsache, dass die Tabellen mit der Farbe $\tilde{C}(s_f, s_b, d)$ und der Opazität $\alpha(s_f, s_b, d)$, die entsprechend den Gleichungen (3.13) und (3.14) berechnet werden, eine begrenzte Menge an Einträgen besitzen. Somit liegen s_f, s_b und d, welches der Länge des Strahlsegments entspricht, quantisiert vor.

Vorausgesetzt dass die Prä-Integrationstabellen bereits für alle Einträge mit $d < d'$ berechnet wurde, können alle Einträge für weitere Strahllängen $d = d' + \Delta d$ dadurch berechnet werden, dass das Strahlintegral entsprechend Abbildung 3.3 in einen Teil der Länge d' und einen Teil der Länge Δd aufgespalten wird. Der Skalarwert s_p am Schnittpunkt wird linear zwischen s_f und s_b interpoliert, d.h. $s_p = (d' s_f + \Delta d\, s_b)/(\Delta d + d')$.

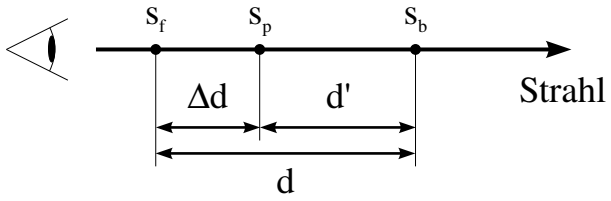


Abbildung 3.3: Unterteilung von Strahlsegmenten der Länge d in zwei Teile der Länge Δd beziehungsweise d'.

Da diese Teilintegrale bereits von der Tabelle abgedeckt werden, reduziert sich die Auswertung auf die Interpolation der nachgeschlagenen Strahlintegrale und eine Blending-Operation. Die Interpolation ist unabhängig von der Quantisierung der Skalarwerte erforderlich, weil s_p beliebige Zwischenwerte annehmen kann. Genauer ausgedrückt ergeben sich $\tilde{C}(s_f, s_b, d)$ und $\alpha(s_f, s_b, d)$ durch folgende Gleichungen:

$$\tilde{C}(s_f, s_b, d' + \Delta d) = \tilde{C}(s_f, s_p, \Delta d) + (1 - \alpha(s_f, s_p, \Delta d))\tilde{C}(s_p, s_b, d') \tag{3.15}$$

$$\alpha(s_f, s_b, d' + \Delta d) = \alpha(s_f, s_p, \Delta d) + (1 - \alpha(s_f, s_p, \Delta d))\alpha(s_p, s_b, d') \tag{3.16}$$

Als weitere Randbedingungen gelten, dass $\tilde{C}(s_f, s_b, 0) = 0$ und $\alpha(s_f, s_b, 0) = 0$ für alle Werte von s_f und s_b. Außerdem müssen $\tilde{C}(s_f, s_b, \Delta d)$ und $\alpha(s_f, s_b, \Delta d)$ ein einziges Mal durch numerische Auswertung der Integrale in Gleichung (3.13) und (3.14) berechnet werden.

Die Interpolation der tabellierten Integrale erzeugt einen Fehler, dessen Größe von der Auflösung der verwendeten Tabelle abhängt. Dieser Fehler entsteht durch die Nachschläge für $\tilde{C}(s_f, s_p, \Delta d)$

und $\alpha(s_f, s_p, \Delta d)$. Da sich diese Fehler nicht akkumulieren können, sind sie unkritisch. Potentiell akkumulieren können sich jedoch die Fehler, die aus dem Nachschlag von $\tilde{C}(s_p, s_b, d')$ und $\alpha(s_p, s_b, d')$ entstehen. In der Praxis wird diese Fehlerakkumulation durch die Reihenfolge, wie die Anteile für Δd und d' in den Blending-Gleichungen (3.15) und (3.16) auftauchen, unterdrückt.

Im Bezug auf den Berechnungsaufwand für inkrementelle Prä-Integration gilt, dass die erste Ebene der Integrationstabellen in $O(n^3)$ zu berechnen ist, da für jeden der $n \times n$ Einträge der ersten Ebene eine numerische Integration durchgeführt werden muss. Alle weiteren $n \times n \times (n-1)$ Einträge erfordern lediglich konstante Rechenzeit, weshalb der Gesamtaufwand ebenfalls $O(n^3)$ ist. Varianten dieser Technik wurden unabhängig von mehreren Forschern entwickelt [Guthe 2002], soweit bekannt aber bisher nicht veröffentlicht.

3.2.2 Subrange-Integration

Das Prinzip der inkrementellen Berechnung für Farbe und Opazität der Strahlstücke bei Prä-Integrierter Klassifikation lässt sich nicht nur für die Berechnung unterschiedlicher Strahlsegmentlängen d ausnutzen. Für konstante Strahllängen wurde in [Lum et al. 2004] ein Verfahren vorgeschlagen, welches eine zweidimensionale Prä-Integrationstabelle, deren Berechnungskomplexität bei naivem Vorgehen $O(n^3)$ ist, in $O(n^2)$ bestimmt. Die Idee beruht darauf, die Zwischenergebnisse beim Compositing der Farbe $\tilde{C}(s_f, s_b, d)$ und der Opazität $\alpha(s_f, s_b, d)$ eines Strahlsegments durch inkrementelle Berechnung möglichst effektiv wiederzuverwenden.

Im Folgenden wird von dem typischen Fall ausgegangen, dass die Transferfunktion als diskrete Tabelle von Farb- und Opazitätswerten beschrieben wird. Ein Tabelleneintrag entspricht dabei dem Farb- und Opazitätskoeffizient für einen diskreten Skalarwert s_i mit $0 \leq i \leq n$. Der Unterschied zwischen den Skalarwerten benachbarter Einträge in der Tabelle beträgt Δs und entspricht einer Länge Δd entlang des Sehstrahls. Dem vorderen Skalarwert s_f entspricht der Index f dem hinteren Skalarwert s_b der Index b. Schrittweises Compositing des Wertes $\tilde{C}(s_f, s_b, d)$ liefert folglich die Zwischenergebnisse $\tilde{C}(s_f, s_{f+1}, \Delta d)$, $\tilde{C}(s_f, s_{f+2}, 2\Delta d)$ … $\tilde{C}(s_f, s_{b-1}, d - \Delta d)$, $\tilde{C}(s_f, s_b, d)$.

Leider existiert keine einfache Möglichkeit $\tilde{C}(s_i, s_{j+1}, d)$ aus $\tilde{C}(s_i, s_j, d)$ herzuleiten, was daher rührt, dass im zweiten Fall Δs einer anderen Strahllänge entspricht und daher eine Korrektur der Opazität erforderlich wäre. Diese kann nicht im Nachhinein erfolgen, sondern muss für jeden Schritt des Compositing durchgeführt werden. Tatsächlich ist das Ergebnis, wenn zu $\tilde{C}(s_i, s_j, d)$ ein weiterer Wert s_{j+1} hinzugefügt wird, $\tilde{C}(s_i, s_{j+1}, d + \Delta d)$, wobei Δd der Länge von Δs entlang des Sehstrahls entspricht.

Wie in Abbildung 3.4 dargestellt, besteht die Idee nun darin, durch sukzessives Compositing zunächst alle Integrale aufzubauen, bei denen Δs einer festen Strahllänge $\Delta d = d/N$ entspricht, da hierbei keine Probleme mit unterschiedlichen Opazitäten auftreten. In einem zweistufigen Prozess werden dazu zunächst alle Teilintegrale der Form $\tilde{C}(s_i, s_k, |k-i|\Delta d)$ beziehungsweise $\tilde{C}(s_k, s_j, |j-k|\Delta d)$ für $|j-i| = N$ berechnet und daraus durch nur jeweils einen weiteren

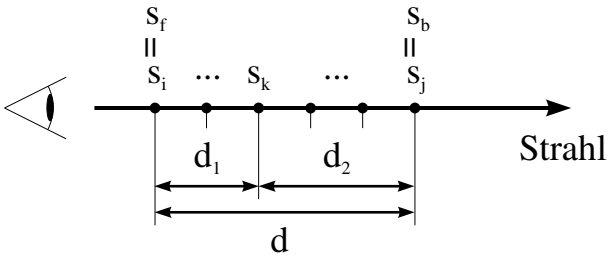


Abbildung 3.4: Unterteilung eines Strahlsegments, welches den Skalarwertbereich $s_f \rightarrow s_b$ abdeckt, in zwei Teilbereiche $s_f \rightarrow s_k$ beziehungsweise $s_k \rightarrow s_b$.

Compositing-Schritt alle Integrale $\tilde{C}(s_f, s_b, d)$ zusammengesetzt. Danach wird mit $N+1$ fortgefahren, wodurch der Aufbau der Prä-Integrationstabelle entlang von diagonalen Linien erfolgt.

Das Compositing zur Gesamtfarbe und -opazität des Strahlsegments aus den Teilintegralen $\tilde{C}(s_f, s_k, d_1)$ und $\tilde{C}(s_k, s_b, d_2)$ für die Strahlfarbe beziehungsweise $\alpha(s_f, s_k, d_1)$ und $\alpha(s_k, s_b, d_2)$ für die Opazität erfolgt entsprechend der Gleichungen (3.17) und (3.18). Dabei ergeben sich die Längen der Teilsegmente durch Teilung der Gesamtstrahllänge d im Verhältnis der Skalarwerte, beziehungsweise der Indizes der Skalarwerte in der Transferfunktion. Genauer gesagt ist $d_1 = |k-f|/|b-f|d$, wenn f der Index von s_f in der Transferfunktion ist und b der Index von s_b. Entsprechend berechnet sich $d_2 = |b-k|/|b-f|d$.

$$\tilde{C}(s_f, s_b, d_1 + d_2) = \tilde{C}(s_f, s_k, d_1) + (1 - \alpha(s_f, s_k, d_1))\tilde{C}(s_k, s_b, d_2) \tag{3.17}$$

$$\alpha(s_f, s_b, d_1 + d_2) = \alpha(s_f, s_k, d_1) + (1 - \alpha(s_f, s_k, d_1))\alpha(s_k, s_b, d_2) \tag{3.18}$$

Für eine effiziente Organisation der Berechnung der Teilintegrale wird der Wertebereich der Transferfunktion in Blöcke der Größe N unterteilt (siehe Abbildung 3.5), so dass in jedem Block die inkrementelle Berechnung der zugehörigen Integrale entweder durch Front-to-Back oder durch Back-to-Front Compositing erfolgen kann. Die Struktur ist so organisiert, dass zwei benachbarte Teilintegrale jeweils zu einem Strahlintegral $\tilde{C}(s_f, s_b, d)$ verbunden werden können.

Mit Hilfe dieser Struktur gelingt es, die Prä-Integrationstabelle der Größe $n \times n$ in der Zeit $O(n^2)$ zu berechnen. Da sie insgesamt $2n$ Diagonalen besitzt, muss die Teilbereichs-Integration für $2n$

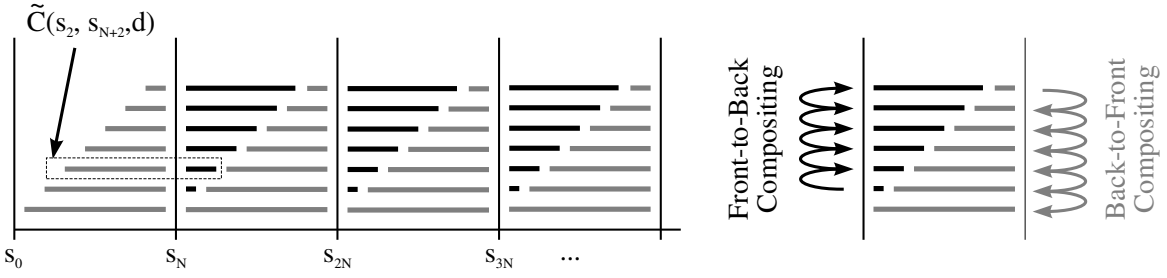


Abbildung 3.5: Organisation der Strahlteilintegrale der Länge N.

unterschiedliche Strahllängen durchgeführt werden. Jede dieser Teilschritte füllt eine Datenstruktur entsprechend Abbildung 3.5, die für eine Transferfunktion aus n Einträgen n/N Blöcke besitzt, von denen jeder $2N$ Compositing-Operationen benötigt. In der Gesamtbetrachtung benötigt jeder einzelne Eintrag also konstante Rechenzeit. Die Subrange-Integration stellt damit eine hinsichtlich der Berechnungskomplexität optimale Lösung dar, die sich auch auf dreidimensionale Prä-Integrationstabellen erweitern lässt. Dazu ist lediglich eine Schleife über die unterschiedlichen Strahllängen entlang der dritten Tabellendimension erforderlich.

3.3 Algorithmen

Seit der Einführung der direkten Volumenvisualisierung vor etwa 20 Jahren wurden eine Vielzahl unterschiedlicher Algorithmen und Techniken entwickelt, die sich in drei Hauptgruppen einteilen lassen: Bildraumverfahren, Objektraumverfahren und Frequenzraumverfahren.

Die wesentliche Idee der *Bildraumverfahren* besteht darin, ausgehend vom Zielbild den Beitrag des gesamten Volumens für die Farbe jedes einzelnen Bildpunktes zu bestimmen. Dies erfolgt, wie in Abbildung 3.6 links dargestellt, durch die Betrachtung von Strahlen durch jeden Bildpunkt. Entlang des Strahls wird die Farbe und Opazität an bestimmten Stützpunkten ermittelt und zur endgültigen Farbe zusammengefasst. *Ray Casting* ist ein typischer Vertreter dieser Klasse von Algorithmen und wird in Abschnitt 3.3.1 näher behandelt.

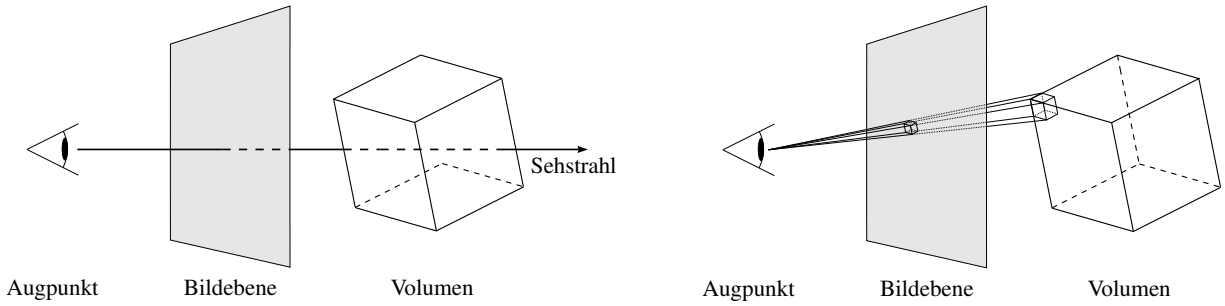


Abbildung 3.6: Bei der links dargestellten direkten Volumenvisualisierung mittels Ray Casting wird je Pixel ein Strahl durch das Volumen verfolgt. Zellprojektion (rechts) projiziert die Elemente des Volumens auf die Bildebene.

Objektraumverfahren bearbeiten umgekehrt das Volumen Element für Element und berechnen den Beitrag jedes Volumenelements zum Gesamtbild (vgl. Abbildung 3.6 rechts). Da die Projektion jedes Elements (z.B. Punkt oder Gitterzelle) hierbei im Prinzip mehrere Pixel des Zielbildes überdecken kann, wurden im Laufe der Zeit effiziente Methoden entwickelt, um den Beitrag einer Zelle zum Gesamtbild zu bestimmen. Eine der wichtigsten ist die *Zellprojektion*, die im Abschnitt 3.3.3 näher betrachtet werden soll. Beim *Splatting* [Westover 1990] wird beispielsweise lediglich das Zentrum der Volumenelemente in den Bildraum projiziert und in der Umgebung

ein sogenannter *Footprint* in das Ergebnisbild eingefügt, welcher durch Faltung des Volumenelements mit einem radialsymmetrischen Filterkernel im Rahmen eines Vorberechnungsschrittes entsteht.

Über die Zugehörigkeit der erstmals in [Cabral et al. 1994] vorgeschlagenen texturbasierten Volumenvisualisierung zu den Bildraumverfahren beziehungsweise Objektraumverfahren herrscht in der Literatur eine gewisse Uneinigkeit. Wie in Abbildung 3.7 dargestellt, werden dabei Schnittebenen entweder parallel zu den Hauptachsen des Volumens oder parallel zur Bildebene gezeichnet, dabei von der Graphik-Hardware mit den Volumenwerten texturiert und mittels Blending in den Framebuffer geschrieben. Offensichtlich erfolgt das Zeichnen der Polygone in den Koordinaten des Volumens, also im Objektraum. Auf der anderen Seite erfolgt die Texturierung in der Graphik-Hardware pixelorientiert, im Bildraum. Texturbasierte Volumenvisualisierung wird daher auch als hybrider Algorithmus betrachtet.

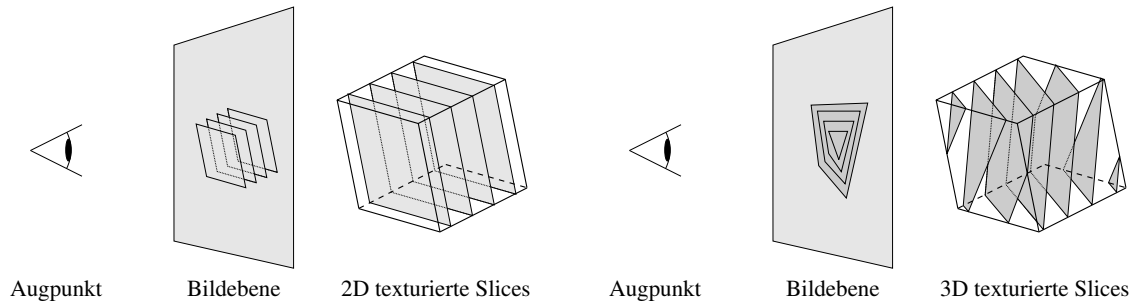


Abbildung 3.7: Bei der links dargestellten direkten Volumenvisualisierung mittels zweidimensionaler Texturen werden Schnittpolygone ausgerichtet an einer Achse des Volumens gezeichnet. Bei dreidimensionalen Texturen werden Schnittpolygone parallel zur Bildebene gezeichnet.

Ähnliches gilt für den Shear-Warp-Algorithmus [Lacroute und Levoy 1994], der in gewisser Weise mit der texturbasierten Volumenvisualisierung auf zweidimensionalen Texturen verwandt ist. Statt die Graphik-Hardware einzusetzen, um die Schnittebenen zu texturieren und auf die Bildebene zu projizieren, verwendet er eine Rasterisierung durch die CPU, die aufgrund einer geschickten Zerlegung der Kameratransformation sehr effizient implementiert werden kann.

Die Grundidee bei *Frequenzraumverfahren* besteht darin, das Volumen in einem Vorverarbeitungsschritt mittels 3D-Fourier-Transformation in den Frequenzraum zu überführen [Malzbender 1993, Totsuka und Levoy 1993]. Eine Projektion des Volumens im Ortsraum kann nun sehr einfach durch Extraktion einer Schnittebene im Frequenzraum bestimmt werden und das Ergebnis mittels inverser 2D-Fourier-Transformation zurück in den Ortsraum übertragen werden. Der Vorteil dieses Vorgehens ist, dass aufgrund der Vernachlässigung des Vorverarbeitungsschrittes die Berechnung des Volumenbildes in $O(n^2)$ erfolgen kann. Jedoch ist abgesehen von Veränderungen der Kameraparameter keine interaktive Modifikation der Bildparameter möglich. Da Volume Rendering im Fourierraum darüber hinaus auf rein emissive optische Modelle beschränkt ist, wird es in dieser Arbeit nicht weiter berücksichtigt.

3.3.1 Ray Casting

Ray Casting gehört zu den frühesten Volume Rendering Techniken (vergleiche [Levoy 1988] und [Garrity 1990]). Da die Strahlverfolgung beim Ray Casting direkt den physikalischen Lichttransport durch das Volumen nachvollzieht, werden mittels Ray Casting berechnete Bilder darüber hinaus oftmals als Referenz für den Vergleich mit anderen Algorithmen herangezogen. Zusätzlich eignet sich Ray Casting ausgezeichnet zur Parallelisierung, weil die Bearbeitung der Sehstrahlen unabhängig voneinander erfolgt.

Die grundlegende Idee bei Ray Casting besteht darin, Strahlen ausgehend vom Auge durch das Skalarfeld des Volumens zu verfolgen. Entlang jedes Strahles wird dabei die Farbe und Opazität an diskreten Abtastpunkten bestimmt und mittels Front-to-Back Compositing zur finalen Farbe jedes Pixels kombiniert. Gewöhnlich wird dabei mindestens ein Strahl je Pixel betrachtet. Da bei diesem Vorgehen offensichtlich der meiste Berechnungsaufwand in der großen Anzahl von Interpolationsoperationen steckt, die beim Sampling der Sehstrahlen entstehen, zielen Optimierungstechniken auf die Reduktion dieser Sampling-Operationen.

So kann beispielsweise das Compositing der Farbwerte, wie in Gleichung (3.11) beschrieben, abgebrochen werden, sobald α'_i einen gewissen Schwellwert nahe bei 1 erreicht hat, da in diesem Fall der Farbbeitrag jedes weiteren Abtastpunktes mit $(1-\alpha'_i) \approx 0$ multipliziert wird und daher vernachlässigt werden kann. Diese Beschleunigungstechnik wird gewöhnlich als *Early Ray Termination* oder *Adaptive Ray Termination* bezeichnet [Levoy 1990]. Auch der Abstand der Abtastpunkte entlang eines Sehstrahls muss nicht notwendigerweise konstant sein. Oftmals lässt sich der Abstand adaptiv erhöhen, um leere Bereiche des Volumens oder solche, die durch die Transferfunktion ausgeblendet werden, schnell zu überspringen. Diese Technik ist als *Empty Space Skipping* oder auch *Space Leaping* bekannt [Levoy 1990].

Da sich diese Techniken, die für eine effiziente Implementierung von Ray Casting unbedingt erforderlich sind, nur sehr schwer auf Graphik-Hardware abbilden lassen, war der Einsatz von Ray Casting beim Volume Rendering bis vor Kurzem auf nicht-interaktive CPU-Implementierungen sowie spezielle Hardware-Designs beschränkt [Meißner et al. 2002]. Dies hat sich jedoch dramatisch geändert. So wurde in [Krüger und Westermann 2003, Röttger et al. 2003] gezeigt, dass die Implementierung eines vollen Ray Casting Algorithmus für uniforme Volumengitter inklusive Sampling-Optimierung auf dem Graphikprozessor möglich ist. Dass Ray Casting auf programmierbarer Graphik-Hardware jedoch nicht nur auf uniforme Gitter beschränkt ist, sondern auch auf adaptive Datenstrukturen wie Tetraedergitter übertragbar ist, wird in Kapitel 5 demonstriert.

3.3.2 Texturbasierte Volumenvisualisierung

Seit ihrer Einführung in [Cabral et al. 1994] hat sich für die Hardware-beschleunigte Volumendarstellung auf uniformen Gittern die texturbasierte Volumenvisualisierung zur erfolgreichen Standardtechnik entwickelt, da sie unter Ausnutzung der Textur-Hardware moderner Graphikkarten vergleichsweise hohe Bildwiederholraten erreicht. Die Idee besteht darin, äquidistante Schnittebenen, sogenannte *Slices*, innerhalb der Bounding Box des Volumens zu betrachten, sie

mit den Volumenwerten zu texturieren und beginnend mit der vom Betrachter am weitesten entfernten Ebene mittels Alpha-Blending in den Framebuffer zu zeichnen. Da hierbei das Alpha-Blending mittels *Over Operator* dem *Back-to-Front Compositing* entspricht und die Texturierung mit dem Sampling des Volumens identifiziert werden kann, ergibt sich durch dieses Vorgehen in jedem Pixel die Berechnung einer Riemannsumme für das Volume Rendering Integral.

Für die praktische Implementierung dieser Technik bieten sich zwei Möglichkeiten an. Das Volumen kann entweder als dreidimensionale Textur definiert werden (vgl. Abbildung 3.7 rechts), wobei in diesem Fall Schnittebenen parallel zur Bildebene als sogenannte *Proxy Geometry* verwendet werden. Alternativ dazu kommt ein Stapel von zweidimensionalen Texturen zusammen mit Schnittebenen, die an einer der Hauptachsen des Volumens ausgerichtet sind, zum Einsatz (vgl. Abbildung 3.7 links). Dabei wird zur Vermeidung von visuellen Artefakten immer die Hauptachse gewählt, die den kleinsten Winkel zur Blickrichtung bildet. Entsprechend müssen in diesem Fall drei Stapel von zweidimensionalen Texturen verwendet werden, ein Stapel je möglicher Orientierung. Dies führt zu einer Verdreifachung des Speicherbedarfs.

Wesentliche Vorteile der 3D-Methode liegen darin, dass nur eine einzige dreidimensionale Textur erforderlich ist. Das Sampling des Volumens erfolgt mittels trilinearer Interpolation, was im Allgemeinen in einer besseren Bildqualität resultiert. Außerdem lässt sich durch zusätzliche Schichtpolygone, welche die Sampling-Rate durch Verringerung des Schichtabstandes erhöhen, die Bildqualität weiter steigern. Letzteres ist bei der 2D-Methode nicht ohne weiteres möglich, weil hierbei die Anzahl der Schnittpolygone auf die Anzahl der zweidimensionalen Texturen beschränkt ist. Außerdem treten typischerweise Artefakte durch die Umschaltung zwischen den Texturstapeln auf und der variierende Winkel zwischen der Blickrichtung und der Orientierung der Slices führt zu einer ständigen Variation in der Abtastrate, die durch die Anpassung der Opazität ausgeglichen werden muss.

Der große Vorteil der 2D-Methode liegt darin, dass sie mit 2D-Texturen auskommt, die von nahezu allen Graphikarchitekturen unterstützt werden. Außerdem ist die Geschwindigkeit moderner Graphik-Hardware mit zweidimensionalen Texturen im Allgemeinen deutlich schneller, weil die bilineare Interpolation nur vier benachbarte Texel benötigt und außerdem die geringere Größe der zweidimensionalen Texturen ein effizientes Caching in der Graphik-Hardware erlaubt. Weitere Details und eine Diskussion über die Vorteile und Nachteile der verschiedenen Techniken findet sich in [Rezk-Salama 2002].

Von der Entwicklung moderner Graphik-Hardware konnte die texturbasierte Volumenvisualisierung in besonderem Maße profitieren. So wurde beispielsweise in [Rezk-Salama et al. 2000] gezeigt, dass mit Hilfe von Multi-Texturen auch bei zweidimensionalen Texturen beliebige Schichtabstände realisierbar sind und auch die Speichervervielfachung durch die drei Texturstapel vermieden werden kann. Mit derselben Slicing-Technik konnten basierend auf einer Technik von Westermann [Westermann und Ertl 1998] interaktive Isoflächen mit diffuser und spekularer Reflexion realisiert werden. Mit der Adaptierung der Prä-Integrierten Klassifikation in [Engel et al. 2001] erreicht texturbasierte Volumenvisualisierung mittlerweile Darstellungen, deren Qualität mit der von analytisch integrierten Ray Casting Bildern konkurrieren kann. In dieser Arbeit wird texturbasierte Volumenvisualisierung zur Darstellung adaptiver Hierarchien von uniformen Volumengittern verwendet.

3.3.3 Zellprojektion

Ausgehend von den ersten Projektionsalgorithmen, die noch ohne Unterstützung durch Graphik-Hardware arbeiteten [Max et al. 1990], hat sich Zellprojektion zur vermutlich am weitesten verbreiteten Methode zum Hardware-beschleunigten Rendering von unstrukturierten Gittern im Allgemeinen und Tetraedergittern im Besonderen entwickelt. In der Vergangenheit basierten nahezu alle Zellprojektionsalgorithmen auf dem *Projected Tetrahedra* (PT) Algorithmus, der erstmals in [Shirley und Tuchman 1990] veröffentlicht wurde.

Dieser Algorithmus nutzt die beschleunigte Scan-Konvertierung der Graphik-Hardware, indem er die Projektion eines Tetraeders in eine Reihe von Dreiecke zerlegt, denen an den Vertizes die korrekte Farbe und Opazität entsprechend dem Emissions-Absorptions-Modell oder einem anderen optischen Modell zugewiesen wird. Die Dreiecke können somit sehr effizient von der Standard-Graphik-Hardware gezeichnet werden. Die erforderlichen Schritte des Algorithmus können dabei wie folgt zusammengefasst werden:

1. Zerlege das Volumen in Tetraeder. Hexaederzellen können dabei in fünf oder sechs Tetraeder zerlegt werden. Für Prismen und Pyramidenzellen ergeben sich entsprechend drei beziehungsweise zwei Tetraeder. Skalarwerte sind an den Vertizes der Tetraeder definiert. Innerhalb des Tetraeders wird ein linearer Skalarwerteverlauf angenommen.

2. Sortiere die Zellen entsprechend ihrer Sichtbarkeit von hinten nach vorne.

3. Klassifiziere jeden Tetraeder entsprechend seines Projektionsprofils und zerlege die Projektion entsprechend Abbildung 3.8 in Dreiecke.

4. Ermittle durch Strahlintegration die korrekte Farbe und Opazität für die Vertizes der Dreiecke.

5. Zeichne die Dreiecke mit Hilfe der Graphik-Hardware.

Viele Forschergruppen haben sich seitdem mit der Weiterentwicklung und Verbesserung dieser Methode beschäftigt, wobei zwei Problemkreise zu identifizieren sind. Effiziente Sortierverfahren für unstrukturierte Gitter [Max et al. 1990, Williams 1992, Cignoni et al. 1995, Wittenbrink 1999], im Besonderen für nicht-konvexe [Stein et al. 1994, Comba et al. 1999] und zyklische Gitter [Kraus und Ertl 2001] wurden untersucht, da die Sortierung aufgrund der Schere zwischen der Leistungsfähigkeit des Hauptprozessors und der Graphik-Hardware zunehmend den Flaschenhals der Methode darstellt. Dies gilt insbesondere, nachdem die in der Regel sukzessiv entstehenden Ergebnisse der Sortierung nicht mit der vollen Bandbreite des Übertragungskanals zur Graphikkarte transportiert werden können.

In jüngster Zeit wird sogar Graphik-Hardware zur Unterstützung der Sortierung eingesetzt. Bei [Callahan et al. 2004] führt die CPU lediglich eine grobe Sortierung der Tetraeder dergestalt durch, dass die Feinsortierung mit Hilfe von Fragmentoperationen auf der Graphik-Hardware erledigt werden kann. Die hohe Geschwindigkeit dieser hybriden Technik rührt daher, dass die

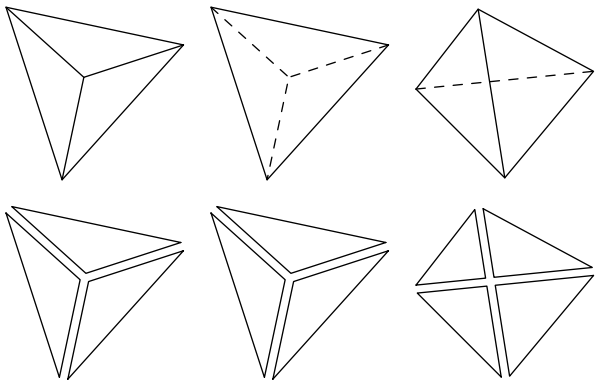


Abbildung 3.8: Klassifikation der nicht-degenerierten Fälle bei Shirly-Tuchman-Zellprojektion (obere Reihe), sowie deren Zerlegung in Dreiecke.

Vorsortierung nicht exakt sein muss und somit sehr schnelle Näherungsverfahren verwendet werden können.

Ein weiteres Feld von Arbeiten beschäftigt sich darüber hinaus mit der korrekten Strahlintegration. Der ursprüngliche PT-Algorithmus interpoliert Farbe und Opazität linear zwischen den Vertizes der Dreiecke. Dies stellt lediglich eine Approximation dar und resultiert in Rendering-Artefakten. In [Stein et al. 1994] wird daher vorgeschlagen, eine zweidimensionale Textur zu verwenden, die mit der Dicke l des projizierten Tetraeders sowie dem zwischen Eintritts- und Austrittsfläche gemittelten Absorptionskoeffizienten τ parametrisiert wird. Diese wird zum Nachschlag der exakten Opazität $\alpha = 1 - \exp(-\tau l)$ an jedem Pixel verwendet. Die Farbe wird immer noch zwischen den Vertizes linear interpoliert. Weil darüber hinaus Texturkoordinaten linear interpoliert werden, ist dieser Ansatz auf lineare Transferfunktionen $\tau = \tau(s(\mathbf{x}))$ beschränkt.

Eine Verallgemeinerung dieser Idee stellt die in Abschnitt 3.1.5 vorgestellte Prä-Integration dar, die erstmals in [Röttger et al. 2000] für die Zellprojektion verwendet wurde. Sie verwendet dreidimensionale Texturen, um sowohl die korrekte Farbe als auch Opazität je Fragment zu bestimmen. Dazu wird an jedem Vertex der gezeichneten Dreiecke der Skalarwert an der Eintrittsfläche s_f (Front Face) sowie der Skalarwert an der Austrittsfläche s_b (Back Face) und die Dicke l des Tetraeders als Vertexparameter definiert. Die von der Graphik-Hardware linear über das gezeichnete Dreieck interpolierten Parameter dienen als Koordinaten in die als Textur gespeicherte Prä-Integrationstabelle. Wie in [Röttger et al. 2000] gezeigt wurde, können mit ein und derselben Zellprojektionstechnik somit einerseits semitransparente Volumendarstellungen mit beliebiger Transferfunktion, Isoflächendarstellungen mit mehreren Isoflächen wie auch gemischte Darstellungen von Volumen und Isoflächen erreicht werden.

Probleme entstehen bei Zellprojektion mit Prä-Integrierter Klassifikation jedoch unter perspektivischer Projektion, da die lineare Interpolation der Texturkoordinaten im Bildraum in diesem Fall nicht konsistent mit der erforderlichen linearen Interpolation der Skalar- und Dickenwerte

im Objektraum ist. Die Lösung dazu liegt in der Verwendung homogener Koordinaten, welche, wie in [Kraus et al. 2004] vorgeschlagen, eine perspektivisch korrekte Interpolation für den PT-Algorithmus erlauben. Zusammen mit weiteren Verbesserungen, wie der Erhöhung der Genauigkeit in der Repräsentation der Prä-Integrationstabelle oder der Verwendung eines Framebuffers im Gleitkommaformat, wird damit erstmals ein interaktives und artefaktfreies Rendering von Tetraedergittern mittels moderner programmierbarer Graphik-Hardware möglich.

In dieser Arbeit wird korrekte Strahlintegration durch pixelexaktes Ray Casting innerhalb eines Tetraeders beziehungsweise innerhalb des gesamten Tetraedergitters erreicht (vgl. Kapitel 4 und 5). Um außerdem von der vollen Leistungsfähigkeit der schnellen Graphik-Hardware profitieren zu können, wird versucht, die Sortierung in der CPU komplett zu vermeiden, was einerseits durch die Ausnutzung geeigneter optischer Modelle aber auch durch die Verwendung von Techniken aus dem Ray Casting gelingt.

Kapitel 4

Hardware-basierte Zellprojektion

Zellprojektion ist eine Standardmethode für die Volumenvisualisierung von unstrukturierten Gittern im Allgemeinen und Tetraedergittern im Besonderen. Die ersten Ansätze, z.B. [Max et al. 1990], waren als reine CPU-Implementierungen ausgelegt und noch heute ist dieser Ansatz aufgrund seiner Flexibilität und der einfachen Möglichkeit zur Parallelisierung attraktiv. In den letzten Jahren hat sich jedoch der Projected Tetrahedra (PT) Algorithmus, der erstmals von Shirley und Tuchman [Shirley und Tuchman 1990] veröffentlicht wurde, als Quasi-Standard etabliert. Er nutzt die in der Graphik-Hardware implementierte Scan-Konvertierung von Dreiecken und erreicht damit unter den meisten Umständen signifikant höhere Bildwiederholraten.

Von der in den letzten Jahren zunehmenden Transformations- und Rasterisierungsleistung moderner Graphikkarten konnten PT-basierte Systeme zur Volumenvisualisierung jedoch nicht in gleichem Maße profitieren wie Anwendungen für polygonale Graphik. Die Gründe dafür sind einerseits durch die PC-Systemarchitektur begründet und andererseits algorithmischer Natur. Alle Stufen der Graphik-Hardware sind für die Verarbeitung von Dreiecken als Basisprimitive ausgelegt und Anwendungen für polygonale Graphik verwenden oftmals Hardware-Optimierungen wie OpenGL Display Lists, Vertex Arrays oder DirectX Vertex Buffers, um eine stete Auslastung der parallelen Pipelines auf der Graphikkarte und damit optimale Performanz zu garantieren. Die Menge der Daten, die über den AGP-Bus transferiert werden muss, wird dabei reduziert, bzw. die Transferrate durch geeignete Vorverarbeitung und die Übertragung im Burst-Modus erhöht. Dies stellt eine wesentliche Verbesserung dar, weil sich die Datenübertragung von der CPU zur Graphikkarte in den letzten Jahren immer mehr zum Flaschenhals entwickelt hat.

Im Kontext von Zellprojektion kann in der Regel kein ausreichender Datenstrom für die Graphikkarte geliefert werden. Die Gründe liegen in der erforderlichen Sortierung bzw. der Tatsache, dass die dargestellten Polygone abhängig von der Betrachterposition dynamisch neu berechnet werden müssen. Aufgrund dieser blickpunktabhängigen Berechnung verbieten sich auch Optimierungen basierend auf statischen Display Lists etc. Erforderlich wäre dazu eine Graphik-Hardware, die Tetraeder als Graphikprimitive direkt unterstützt. Verschiedene Ansätze dazu wurden bereits vorgeschlagen [Wittenbrink 2001, King et al. 2001] und eine dramatische Beschleunigung der Volumendarstellung von unstrukturierten Gittern in Aussicht gestellt. Nachdem keine dieser Architekturen bisher verwirklicht wurde, soll in diesem Abschnitt ge-

zeigt werden, dass sich moderne Standard-Graphik-Hardware einsetzen lässt, um eine direkte Scan-Konvertierung von Tetraedern zu implementieren. Dieser Teil der Arbeit wurde auch unter dem Titel *„Hardware-Based View-Independent Cell Projection"* in [Weiler et al. 2002, Weiler et al. 2003b] veröffentlicht. Ein alternativer Ansatz zur Abbildung des PT-Algorithmus auf die Vertexeinheit moderner Graphikkarten wurde etwa zur selben Zeit entwickelt und in [Wylie et al. 2002] vorgestellt.

4.1 Scan-Konvertierung von Tetraedern

Wie bereits in der Einleitung erwähnt, soll erreicht werden, dass die Graphik-Hardware eine blickpunktunabhängige Scan-Konvertierung jedes einzelnen Tetraeders durchführen kann. Dadurch ergibt sich die Möglichkeit, eine statische Datenstruktur für das Tetraedergitter zu verwenden, was den Einsatz von Optimierungen wie OpenGL Display Lists, Vertex Arrays oder DirectX Vertex Buffers ermöglicht. Leider verbietet sich dazu ein Ansatz basierend auf dem ursprünglichen PT-Algorithmus, weil bei diesem bei der Projektion der Tetraeder unterschiedliche Fälle erzeugt werden, die blickpunktabhängig ausgewählt werden müssen. Der hier vorgestellte Ansatz verschiebt alle blickpunktabhängigen Berechnungen auf die Graphik-Hardware und verwendet dazu einen dem Ray Casting verwandten Ansatz.

4.1.1 Blickpunktunabhängige Zellprojektion

Zunächst soll ein einzelner Tetraeder betrachtet werden. Für jedes Pixel, das durch die Projektion des Tetraeders entsteht, wird ein Sehstrahl betrachtet und das korrespondierende Volumenintegral ermittelt. Letzteres erfolgt entsprechend der in Abschnitt 3.1.5 vorgestellten Prä-Integrierten Klassifikation durch den Nachschlag in einer Tabelle, die als Textur implementiert wird. Die Parameter für den Texturnachschlag sind dabei der Skalarwert s_f am Eintrittspunkt des Strahls in den Tetraeder, der Skalarwert s_b am Austrittspunkt und die Länge l des Strahlsegments im Inneren des Tetraeders (siehe Abbildung 4.1).

Um diese Parameter zu ermitteln, werden die dem Betrachter zugewandten Seiten (Front Faces) des Tetraeders gezeichnet. Die Skalarwerte an den Knoten werden dabei als Farbwerte definiert. Man beachte, dass allein durch die Auswahl der Front Faces kein blickpunktabhängiges Element in dem Algorithmus entsteht, weil die Front Faces identisch mit den sichtbaren Flächen des Tetraeders sind, die sich bei geeigneter Spezifikation der Flächennormalen durch OpenGL Backface Culling extrahieren lassen. Es kann demnach eine statische Liste mit allen Flächen aller Tetraeder verwendet werden.

Durch die Wahl der Farbwerte an den Knoten und die lineare Interpolation der Knotenparameter durch die Graphik-Hardware ist nun bereits der Skalarwert am Eintrittspunkt s_f bekannt. Somit muss eigentlich nur noch die Länge l des Strahlsegments bestimmt werden, weil sich daraus auch der Skalarwert am Austrittspunkt s_b unter Zuhilfenahme des Gradientenvektors $\mathbf{g}$ für den Tetraeder und des Vektors der normalisierten Blickrichtung $\mathbf{d}$ bestimmen lässt:

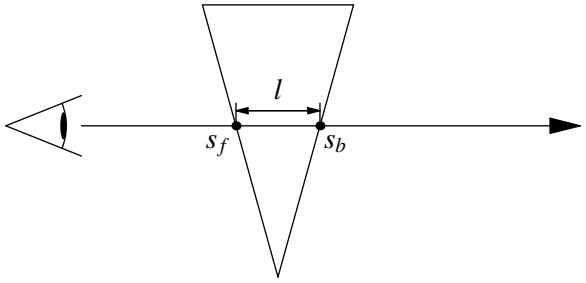


Abbildung 4.1: Schnitt eines Tetraeders mit einem Sehstrahl. s_f und s_b sind die Skalarwerte am Schnittpunkt mit der dem Betrachter zugewandten Fläche (Front Face) und der abgewandten Fläche (Back Face). l bezeichnet die Länge des Strahlsegments und entspricht der Dicke des Tetraeders.

$$s_b = s_f + (\mathbf{g} \cdot \mathbf{d})l \tag{4.1}$$

Somit verbleibt nur noch das Problem, die korrekte Länge des Strahlsegments pro Fragment zu ermitteln, was äquivalent zu dem Problem ist, bei gegebenem Eintrittspunkt des Strahls, den Abstand zum Austrittspunkt zu ermitteln. Dies soll zunächst ohne Betrachtung von möglichen Einschränkungen durch die Graphik-Hardware diskutiert werden.

4.1.2 Berechnung des Austrittspunkts

Es sei zunächst angenommen, dass jede beliebige Kenngröße des Tetraeders auf der Fragmentebene verfügbar ist. Im Speziellen sind das die Normalen aller Seiten sowie der Gradientenvektor für den Tetraeder. Auch die Koordinaten des Strahleintrittspunktes für das betrachtete Fragment sollen als bekannt vorausgesetzt werden. Letzteres lässt sich einfach erreichen, indem entweder die Koordinaten der Tetraederknoten explizit auch als Texturkoordinaten der korrespondierenden Vertizes definiert werden oder eine geeignete Parametrisierung der automatischen Texturkoordinatengenerierung von OpenGL erfolgt.

Algorithmisch ist die Suche nach der Länge des Strahlsegments im Inneren des Tetraeders ähnlich zu dem Problem des dreidimensionalen Line Clipping an einem konvexen Polyeder, der in diesem speziellen Fall ein Tetraeder ist. Allerdings ist hier nur der Austrittspunkt interessant. Abbildung 4.2(a) stellt die Situation im zweidimensionalen Fall dar. Für ein betrachtetes Fragment sei $\mathbf{p}$ der korrespondierende Eintrittspunkt. Der Tetraeder wird durch vier Flächen f_i begrenzt, die über ihre Normalenrichtungen $\mathbf{n}_i$ definiert sind.

Gemäß dem bekannten Algorithmus zum parametrischen Line Clipping an konvexen Polyedern, der unter anderem von Cyrus und Beck [Cyrus und Beck 1978] vorgeschlagen wurde, wird zunächst der Strahl mit jeder Seite des Tetraeders geschnitten und der Strahlparameter für den

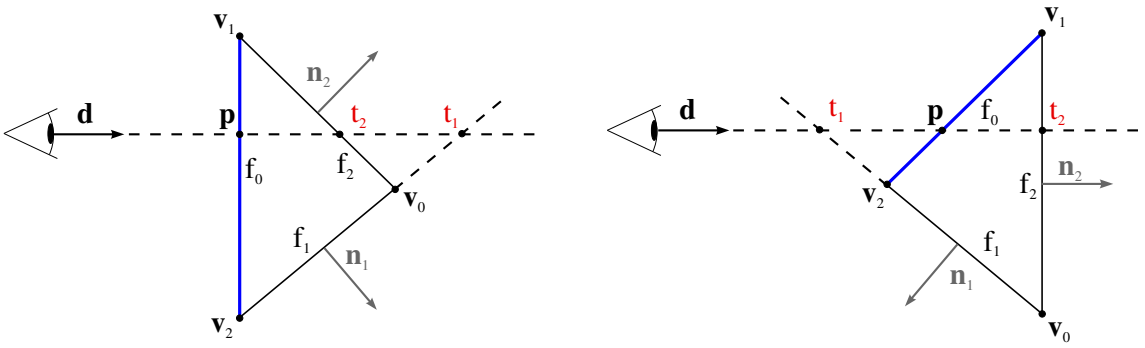


Abbildung 4.2: (a) Veranschaulichung des fragmentbasierten Ray Casting im analogen zweidimensionalen Fall. Der Sehstrahl startet am Eintrittspunkt **p**, welcher einem Fragment des gezeichneten Front Face entspricht. Von dort aus erfolgt ein Schnitt mit allen Seitenflächen des Tetraeders. (b) Potentielle Eintrittspunkte können durch Betrachtung des Vorzeichens des korrespondierenden Strahlparameters eliminiert werden. Alle möglichen Eintrittspunkte außer dem eigentlichen Eintrittspunkt gehören zu negativen Strahlparametern.

Schnittpunkt berechnet. Danach erfolgt eine Klassifikation in potentiell eintretend oder austretend anhand eines Vergleiches mit den nach außen zeigenden Flächennormalen. Der gesuchte Austrittspunkt ergibt sich schließlich als der potentiell austretende Schnittpunkt mit dem kleinsten Strahlparameter.

In diesem Fall startet der Sehstrahl jedoch nicht vom Augpunkt sondern auf einer Fläche des Tetraeders, die somit als Kandidat für den Austrittspunkt ausscheidet. Es müssen demnach nur noch die drei verbleibenden Flächen getestet werden. Für die Berechnung des Schnittpunktes mit den Flächen kommt eine parametrische Beschreibung des Sehstrahls ($\mathbf{r} = \mathbf{p} + t\mathbf{d}$) zum Einsatz, bei der $\mathbf{d}$ im Falle einer perspektivischen Projektion der normierte Vektor vom Augpunkt zum jeweiligen Eintrittspunkt ist. Für den Fall einer orthographischen Projektion ist $\mathbf{d}$ der normierte Vektor der Blickrichtung, der aus den Kameraparametern bestimmt werden kann. Wenn a_i den konstanten Term in der Ebenengleichung der Fläche f_i beschreibt, ergibt sich der Strahlparameter t_i für den Schnittpunkt gemäß

$$t_i = \frac{a_i - (\mathbf{p} \cdot \mathbf{n}_i)}{(\mathbf{d} \cdot \mathbf{n}_i)} \; . \tag{4.2}$$

Unter den hier getroffenen Annahmen ist die Klassifikation des Schnittpunktes einfacher als für das Line Clipping im Allgemeinen. Dies wird aus Abbildung 4.2(b) deutlich: Entsprechend der Normalenrichtung $\mathbf{n}_1$ müsste der Schnittpunkt mit Fläche f_1 als potentiell eintretend klassifiziert werden. Der Startpunkt des Strahls ist jedoch hier identisch mit dem korrekten Eintrittspunkt, dessen Strahlparameter das Maximum der Strahlparameter aller eintretenden Schnittpunkte ist. Nachdem dieser Parameter den Wert 0 hat, müssen notwendigerweise alle weiteren potentiell eintretenden Schnittpunkte negativen Strahlparametern entsprechen. Weiterhin wurden normierte Flächennormalen angenommen, wodurch das Minimum aller positiven Strahlparameter t_i gleichzeitig bereits die gesuchte Dicke l des Tetraeders entlang der Strahlrichtung darstellt.

Im nächsten Abschnitt werden zwei Umsetzungen dieser Idee auf programmierbare Graphik-Hardware vorgestellt. Beide erfordern ein Zusammenspiel von programmierbarer bzw. parametrisierbarer Vertex- und Fragmenteinheit. Im ersten Ansatz wird die meiste Rechenarbeit auf Vertexebene erledigt. Der zweite Ansatz verwendet hauptsächlich Fragmentoperationen.

4.2 Vertexorientierter Ansatz

Obwohl die pro Fragment erforderlichen Berechnungen, die in Abschnitt 4.1.2 beschrieben wurden, nicht sehr komplex sind, erfordern sie doch bereits den Einsatz einer frei programmierbaren Rasterisierungspipeline, wie sie erst mit dem ATI Radeon R300, dem nVidia GeForce-FX oder dem 3Dlabs P10 Chip vorgestellt wurden. Hauptgrund dafür ist die begrenzte Länge von Fragmentprogrammen aber auch semantische Beschränkungen. So besteht z.B. die Rasterisierungspipeline der GeForce3/4 Chip-Serie von nVidia aus den sogenannten Texture Shaders und Register Combiners [OpenGL ARB 2004]. Diese lassen arithmetische Operationen nur *nach* Texturoperationen zu. Der hier vorgestellte Ansatz erfordert jedoch zumindest einen abhängigen Texturnachschlag am Ende nach der arithmetischen Berechnung der Strahlschnittpunkte. Die OpenGL-Erweiterung `ATI_fragment_shader` erlaubt zwar das Mischen von arithmetischen Operationen mit Texturnachschlägen, allerdings werden vom ATI R200 Chip nur maximal 8 arithmetische Operationen vor dem letzten Texturnachschlag unterstützt, was hier unzureichend ist. Jedoch verfügt bereits der GeForce3 Graphik-Chip über eine frei programmierbare Vertexeinheit, die über die OpenGL-Erweiterung `NV_vertex_program` [OpenGL ARB 2004] ansprechbar ist und genügend Flexibilität bietet, um den hier vorgestellten Ansatz umzusetzen. Die meisten erforderlichen Operationen können damit auf Vertexebene umgesetzt werden. Allerdings erfordert dies eine Beschränkung auf orthographische Projektion.

Für orthographische Projektionen gilt, dass der Vektor der Blickrichtung für alle Fragmente konstant ist. Daraus resultiert ein lineares Verhalten der Strahlparameter t_i für alle Fragmente einer gezeichneten Tetraederfläche. Dies kann ausgenutzt werden, indem die Parameter nur noch an den Knoten der gezeichneten Flächen ausgerechnet werden. Die Graphik-Hardware übernimmt dann die korrekte Interpolation der Parameter für jedes Fragment. Im Folgenden wird eine Implementierung basierend auf dem GeForce3 Chip von nVidia beschrieben.

4.2.1 Berechnungen pro Fragment

Nachdem die Strahlparameter bereits auf Vertexebene berechnet werden und als linear interpolierte Vertexattribute zur Verfügung stehen, verbleibt als einzige pro Fragment durchzuführende Operation die Berechnung des positiven Minimum der drei Strahlparameter. Sie werden als Texturkoordinaten übergeben, um ein automatisches Abschneiden (Clamping) auf den Bereich $[0,1]$ zu vermeiden, wie es bei Farbattributen der Fall ist. Die Minimum-Berechnung kann auf unterschiedliche Weise erfolgen. Eine Möglichkeit ist die Verwendung von bedingten Zuweisungsoperationen (Conditional Set), wie sie bereits im Befehlssatz des ATI Radeon R200 Chips enthalten sind.

In der hier vorgestellten Implementierung für den nVidia GeForce3 Chip wird das Minimum der drei Strahlparameter durch den Nachschlag in einer dreidimensionalen Textur bestimmt. Diese Textur enthält als Wert für jeden Texel (r,s,t) das Minimum dieser drei Texturkoordinaten-Komponenten. Gültige Werte liefert dies jedoch nur, wenn die Texturkoordinaten auf das Intervall $[0,1]$ beschränkt sind, was für die nach Gleichung (4.2) berechneten Strahlparameter t_i in keinster Weise der Fall ist. Dieses Problem kann durch eine Division der Strahlparameter durch die maximale Kantenlänge gelöst werden.

Zusätzlich zu den Strahlparametern in den Komponenten r, s und t der Texturkoordinaten wird die maximale Kantenlänge des jeweiligen Tetraeders in der homogenen Texturkoordinate q gespeichert. Der Vorteil dieses Vorgehens gegenüber der Division jeder Komponente durch die maximale Kantenlänge ist eine höhere Genauigkeit, weil die homogene Division erst nach der Interpolation der Texturkoordinaten erfolgt. Quantisierungsartefakte durch die Interpolation werden dadurch vermindert.

Selbst nach dieser „Normalisierung" können immer noch Werte größer als 1 für t_i auftreten. Für das korrekt berechnete Minimum muss sich nun jedoch ein Wert ≤ 1 ergeben, nachdem der Strahl im Inneren eines Tetraeders maximal eine Strecke zurücklegen kann, die der maximalen Kantenlänge entspricht. Somit muss unter den berechneten Strahlparametern mindestens einer ≤ 1 existieren und größere Werte können gefahrlos durch die OpenGL Texturumgebung `CLAMP_TO_EDGE` abgeschnitten werden. Derselbe Mechanismus kann auch für negative Strahlparameter eingesetzt werden, die potentiell eintretenden Schnittpunkten entsprechen und somit verworfen werden müssen. Dies erfolgt durch Überschreiben der negativen Werte mit einem großen positiven Wert.

Ein wichtiger Punkt für die Genauigkeit der Berechnung und des endgültigen Farbwertes ist die Auflösung der Minimumtextur. Gerade bei langgestreckten Tetraedern wird entlang der kurzen Kanten nur ein kleiner Teil des möglichen Datenbereiches ausgenutzt, was auf der verwendeten Hardware zu Quantisierungsfehlern führen kann. Auf der anderen Seite stellen zu große Texturen einen zu vermeidenden Mehraufwand an Speicher dar. Nach den hier gemachten Erfahrungen stellt eine Auflösung von 128^3, was zu einem Speicherbedarf von 4 MB für eine Luminanz-Alpha-Textur[1] führt, einen guten Kompromiss dar.

4.2.2 Berechnungen pro Vertex

Ziel des hier vorgestellten Ansatzes ist eine blickpunktunabhängige direkte Scan-Konvertierung von Tetraedern durch die Graphikkarte. Daher erfolgt die Bestimmung der Vertexattribute, die für die korrekte Berechnung der Sehstrahlen pro Fragment erforderlich sind, hier nicht durch die CPU sondern mit Hilfe der programmierbaren Vertexeinheit der Graphikkarte, wie sie z.B. durch die OpenGL-Erweiterung `NV_vertex_program` [OpenGL ARB 2004] zur Verfügung gestellt wird. Vertexprogramme werden für jeden Vertex unabhängig aufgerufen und ersetzen die Standardtransformation und -beleuchtungsberechnung von OpenGL. Sie bestehen im Falle

[1] Die Notwendigkeit einer Luminanz-Alpha-Textur ergibt sich als Randbedingung einer Implementierung unter Ausnutzung der Texture Shader (siehe Abschnitt 4.4).

```
1   !!VP1.0
2   # Input
3   # -----
4   # V[OPOS].xyz = v   (vertex position)
5   # V[OPOS].w   = a   (facial plane offset)
6   # v[COL0].xyz = g   (gradient)
7   # v[COL0].w   = s_f (front scalar)
8   # v[NRML]           (adjacent face normal)
9   # v[TEX0].x   = n0  (normal component 0)
10  # v[TEX0].y   = n1  (normal component 1)
11  # V[TEX0].z   = ch  (output channel)
12  # v[TEX0].w   = w   (maximum edge length)
13
14
15  # Vertex transformation
16  # ---------------------
17  # c[11] = {-1, -1, 1, 0}
18  #
19  # R9   := transformed vertex
20  #
21  MOV R8.w, c[11].z; # overwrite homogen.
22                     # coordinate with '1'
23  MOV R8.xyz, v[OPOS];
24  DP4 R9.x, c[4], R8;
25  DP4 R9.y, c[5], R8;
26  DP4 R9.z, c[6], R8;
27  DP4 R9.w, c[7], R8;
28  MOV o[HPOS], R9;
29
30
31  # Reconstruct Normal
32  # ------------------
33  # c[11] = {-1, -1, 1, 0}
34  #
35  # R3   := (n0, n1, sqrt(1 - n0^2 - n1^2))
36  #
37  MOV R4.z, c[11];
38  MUL R4.xy, v[TEX0], v[TEX0];
39  MOV R3.xy, v[TEX0];
40  DP3 R3.z, c[11], R4;
41  RSQ R3.z, R3.z;
42  RCP R3.z, R3.z;
43
44
45  # Compute distance
46  # ----------------
47  # c[21] = d (localized view direction)
48  # R3    = n (face normal)
49  #
50  # R5.xyzw  := (<v;n> - a) / <d;n>
51  #
52  DP3 R6.x, v[OPOS], R3;
53  ADD R5, R6.x, -v[OPOS].w;
54  DP3 R6.y, c[21], R3;
55  RCP R6.y, R6.y;
56  MUL R5, R5, R6.y;
57
58
```

```
59  # Wrap negatve/zero values
60  #      to large positive value
61  # ----------------------------
62  # c[13] = {500, x, x, x}
63  #
64  SGE R10, -R5, c[11].w;
65  MAD R5, -R10, R5, R5;
66  MAD R5, R10, c[13].x, R5;
67
68
69  # Set texture coordinate shared with next
70  #     vertex to 'undef' if edge visible
71  # ---------------------------------------
72  # c[18] = {0,       0, 1000, 0}
73  # c[19] = {1000,    0,    0, 0}
74  # c[20] = {0,    1000,    0, 0}
75  # c[21] = d (view direction)
76  #
77  DP3 R8, c[21], v[NRML];
78  SLT R0, R8, c[18].w;
79  ARL A0.x, v[TEX0].z;
80  MUL R4, R0, c[A0.x + 18];
81
82
83  # Store scaled and biased distance in
84  #       channel given by V[TEX0].z
85  # -----------------------------------
86  # c[24] = {1, 0 ,0 ,0}
87  # c[25] = {0, 1 ,0 ,0}
88  # c[26] = {0, 0 ,1 ,0}
89  #
90  MAD o[TEX0], c[A0.x + 24], R5, R4;
91
92  # Set tetra scaling factor as
93  #     homogenious texture coordinate
94  # ----------------------------------
95  MOV o[TEX0].w, v[TEX0].w;
96
97
98  # Store scaled gradient and scalar
99  #       value in 2nd texture stage
100 # --------------------------------
101 # o[TEX1]  := (w*g_d, 0, 0, s_f)
102 # o[TEX2]  := (0, 0, 0, s_f)
103 #
104 DP3 R11, v[COL0], -c[21];
105 MOV o[TEX1].yw, c[11].w;
106 MUL o[TEX1].x, R11, v[TEX0].w;
107 MOV o[TEX1].z, v[COL0].w;
108 MOV o[TEX2].xyw, c[11].w;
109 MOV o[TEX2].z, v[COL0].w;
110
111 # ---------------------------------------
112 # -- SHADING: here emission with gamma --
113 # ---------------------------------------
114 MUL R11, v[TEX0].w, c[22].x;
115 MOV o[COL1].xyz, R11;
116 END
```

Abbildung 4.3: Vertexprogramm zur blickpunktunabhängigen Zellprojektion mit emissivem optischen Modell.

der GeForce3/4 aus bis zu 128 Befehlen, die auf einer verhältnismäßig großen Anzahl von vierkomponentigen Vektorregistern arbeiten und mit Gleitkommagenauigkeit rechnen. Dies ist für den Zweck der blickpunktunabhängigen Zellprojektion bereits vollkommen ausreichend. Im Folgenden sollen die einzelnen erforderlichen Schritte erläutert werden. Das komplette Programm ist in Abbildung 4.3 dargestellt.

Das hier verwendete Vertexprogramm erledigt einerseits die Transformation in die Bildebene, die sich entsprechend [OpenGL ARB 2004] über eine Sequenz aus vier Skalarprodukten berechnen lässt, andererseits die Berechnung der Strahlparameter entsprechend der Gleichung (4.2). Konzeptionell ist dabei, wie in Abschnitt 4.1.2 erklärt, die Berechnung der Schnittparameter für

drei Flächen des Tetraeders erforderlich. Dies würde voraussetzen, für jeden Knoten die Ebenengleichungen aller drei Flächen als Vertexattribute zu spezifizieren, da im Streaming-Modell der Vertexeinheit ein Datenaustausch zwischen verschiedenen Vertizes nicht möglich ist. Allerdings würde dies die Datenmenge pro Vertex deutlich vergrößern. Tatsächlich lässt sich die Menge der Vertexattribute durch nachfolgende Überlegung deutlich reduzieren. Abbildung 4.4 zeigt den Schnitt eines Strahls mit einem Tetraeder.

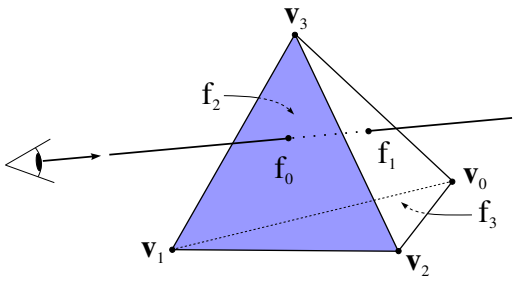


Abbildung 4.4: Nomenklatur für Tetraeder.

Angenommen die hervorgehobene Fläche des Tetraeders f_0, die dem Vertex $\mathbf{v}_0$ gegenüber liegt, wird als Front Face gezeichnet. In diesem Fall können mögliche Austrittspunkte nur auf den Flächen f_1, f_2 und f_3 liegen. Somit müssten die korrespondierenden Parameter t_1, t_2 und t_3 an den Vertizes $\mathbf{v}_1$, $\mathbf{v}_2$ und $\mathbf{v}_3$ berechnet und für jedes Fragment auf f_0 interpoliert werden. Für den Vertex $\mathbf{v}_1$ ist der korrekte Wert von t_2 jedoch bereits ohne Berechnung bekannt. Schließlich liegt $\mathbf{v}_1$ auf f_2 und t_2 muss folglich 0 sein. Analoges gilt für t_3, der ebenfalls 0 ist, nachdem $\mathbf{v}_1$ auch auf f_3 liegt. In der Summe muss Gleichung (4.2) also nur noch für t_1 ausgewertet werden und am Vertex $\mathbf{v}_1$ werden nur die Parameter der Ebenengleichung für f_1 benötigt. Allgemein gilt: am Vertex $\mathbf{v}_i$ werden nur die Parameter der Ebenengleichung von f_i benötigt und müssen als Vertexparameter spezifiziert werden.

In der Regel wird eine Ebenengleichung durch vier Skalare beschrieben, einen Normalenvektor und den Abstand der Ebene zum Koordinatenursprung, im Englischen auch als *Offset* bezeichnet. In dem hier beschriebenen Ansatz soll jedoch die Datenübertragung von der CPU zur Graphikkarte beschleunigt werden, was sich durch Datenvermeidung am einfachsten erreichen lässt. Daher werden die Normalen nicht als drei-komponentige Vektoren sondern nur mit zwei Komponenten gespeichert. Dies ist möglich, nachdem Normalen Einheitsvektoren sind. Somit lässt sich die dritte Komponente entsprechend folgender Gleichung rekonstruieren:

$$\mathbf{n} = (n_0, n_1, n_2) = \left(n_0, n_1, \sqrt{1 - n_0^2 - n_1^2} \right), \quad \text{für } n_2 > 0. \tag{4.3}$$

Die Umsetzung erfordert nur einige wenige Instruktionen im Vertexprogramm. Allerdings gilt die Gleichung nur für positive Werte von n_2. Für die Auswertung von Gleichung (4.2) spielt es aber keine Rolle, ob mit $\mathbf{n}$ oder $-\mathbf{n}$ gerechnet wird; d.h. bei jeder Flächennormale, deren dritte Komponente negativ ist, kann die Orientierung der Normale einfach umgedreht werden.

4.2.3 Konsistente Interpolation

Wie bereits dargestellt, erfolgt die Bestimmung des korrekten Strahlparameters je Fragment durch lineare Interpolation durch die Graphik-Hardware. Die Strahlparameter t_i werden dazu in den Komponenten *r*, *s* und *t* der Texturkoordinaten gespeichert. Nur einer dieser Parameter wird je Vertex tatsächlich berechnet, die beiden anderen sind 0. Um eine konsistente Interpolation zu erreichen, müssen jedoch alle drei Vertizes der gezeichneten Tetraederfläche darin übereinstimmen, welcher der Strahlparameter t_i in welcher Komponente gespeichert ist. Ohne die Möglichkeit zur Kommunikation zwischen den Vertizes, was im Streaming-Modell der Vertexeinheit nicht vorgesehen ist, kann dies nur von außen gesteuert erfolgen. Daher wird beim Zeichnen für jeden Vertex ein Index spezifiziert, der angibt, wo der berechnete Strahlparameter gespeichert werden soll. Ein Index von 0 steht dabei für die *r*-Komponente, während 1 und 2 entsprechend für *s* und *t* stehen.

Für das Umlenken des berechneten Parameters t_i auf die jeweilige Komponente der Texturkoordinate innerhalb des Vertexprogramms wurden verschiedene Möglichkeiten gefunden. Alle beruhen auf der Idee, einen Maskierungsvektor zu konstruieren, der genau in der Zielkomponente den Wert 1 besitzt. Dieser wird mit dem berechneten Strahlparameter skalar multipliziert. Der Maskierungsvektor lässt sich ausgehend vom Index der Zielkomponente auf unterschiedliche Art konstruieren: Eine Möglichkeit benutzt eine Kombination aus „Set on Less Than" (SLT) und „Set on Greater Equal" (SGE) Instruktionen. Sie ist in Abbildung 4.5 links dargestellt. Der Maskierungsvektor entsteht dabei durch Multiplikation der Ergebnisse einer SLT- und einer SGE-Operation mit geschickt gewählten Vergleichsvektoren.

```
# c[8]  = {1, 2, 3, 0}
# c[9]  = {0, 1, 2, 0}
# R1.x  = computed ray parameter

SLT R1, v[TEX0].z, c[8];
SGE R2, v[TEX0].z, c[9];
MUL R0, R1, R2;
MUL o[TEX0], R1.x, R0;
```

```
# c[18] = {1, 0, 0, 0}
# c[19] = {0, 1, 0, 0}
# c[20] = {0, 0, 1, 0}
# R1.x  = computed ray parameter

ARL A0.x, v[TEX0].z;
MUL R0, R1.x, c[A0.x + 18];
ADD o[TEX0], R0;
```

Abbildung 4.5: Umlenken des berechneten Strahlparameters auf eine Komponente der Texturkoordinaten. Links wird die Implementierung mittels SLT und SGE Instruktionen dargestellt, rechts eine Implementierung mittels ARL Instruktion. Der Index der Zielkomponente ist in `v[TEX0].z` gespeichert.

Eine weitere Möglichkeit ergibt sich unter Verwendung indirekter Adressierung mittels des „Address Register Load" (ARL) Befehls (siehe Abbildung 4.5 rechts). Die möglichen Maskierungsvektoren werden dabei als Konstanten definiert und es wird lediglich mittels des Index für die Zielkomponente die richtige Konstante ausgewählt. Diese Lösung wurde in der hier beschriebenen Implementierung verwendet (siehe auch Zeile 90 in Abbildung 4.3).

Mit der OpenGL-Erweiterung `NV_vertex_program2` ist eine weitere Möglichkeit unter Verwendung der „Condition Codes" dazugekommen (siehe Abbildung 4.6). Diese wird aber erst seit

dem nVidia GeForceFX Chip unterstützt. Bei nahezu jedem Befehl im Vertexprogramm lässt sich dabei durch Anhängen eines „C" ein spezielles internes Register auf das Ergebnis eines komponentenweisen Vergleichs des Ausgaberegisters mit Null setzen. Mögliche Ergebnisse sind dabei unter anderem „gleich" (EQ), „größer als" (GT) oder „kleiner gleich" (LE). Mit diesem Vergleichswert lassen sich die Komponenten des Ausgaberegisters eines nachfolgenden Befehls maskieren.

```
# c[30] = {0, 1, 2, 0}
# R1.x  = computed ray parameter

ADDC CC, -v[TEX0].z, c[30];
MOV o[TEX0], (EQ.xyz), R1.x;
```

Abbildung 4.6: Umlenken des berechneten Strahlparameters auf eine Komponente der Texturkoordinaten. Implementierung mittels Condition Code.

4.2.4 Kantenartefakte

Im vorigen Abschnitt wurde beschrieben, wie die Strahlparameter an den Vertizes berechnet, als Texturkoordinaten definiert und durch lineare Interpolation an jedem Fragment zur Verfügung gestellt werden. Leider führt der beschriebene naive Ansatz zu Problemen an den Kanten des Tetraeders. Der Effekt ist auf der linken Seite von Abbildung 4.7 dargestellt. Die Abbildung zeigt die Länge des Strahlsegments aufgetragen als Helligkeitswert.

Die Begründung für diese Artefakte liegt in der Verteilung der Texturkoordinaten, die auf der rechten Seite der Abbildung dargestellt ist. Bei der Rasterisierung der Fläche f_1 repräsentiert die Komponente r der Texturkoordinaten den Abstand von der Fläche f_0. Nachdem der Abstand entlang der Blickrichtung gemessen wird, ist dieser offensichtlich negativ. Mit anderen Worten handelt es sich beim Strahlschnittpunkt mit der Fläche f_0 um einen potentiell eintretenden Schnittpunkt, der nicht zur Bestimmung des tatsächlichen Austrittspunktes heranzuziehen ist. Für die meisten Fragmente, die aus der Rasterisierung der Fläche f_1 entstehen, ist dies der Fall, da die Behandlung von negativen Strahlparametern, die in Abschnitt 4.2.1 beschrieben wird, den Einfluss des negativen Parameters t_0 bei der Minimumsuche eliminiert.

Entlang der Kante zwischen f_0 und f_1 ist jedoch das Interpolationsgewicht für die Attribute von Vertex $\mathbf{v}_0$ Null. Daher geht die Klassifikation für t_0 verloren und der Strahlparameter wird als Parameter eines austretenden Schnittpunktes missinterpretiert. Die Minimumbestimmung liefert demnach entlang der Kante den falschen Wert 0. Dies könnte vermieden werden, wenn OpenGL die Spezifikation eines minimalen Interpolationsgewichtes erlauben würde. In diesem Fall könnte ein kleiner Offset für die Interpolation definiert und vor dem Texturnachschlag mittels Texur-Bias wieder korrigiert werden. Auch die Integration des Offsets in die Minimumtextur wäre möglich.

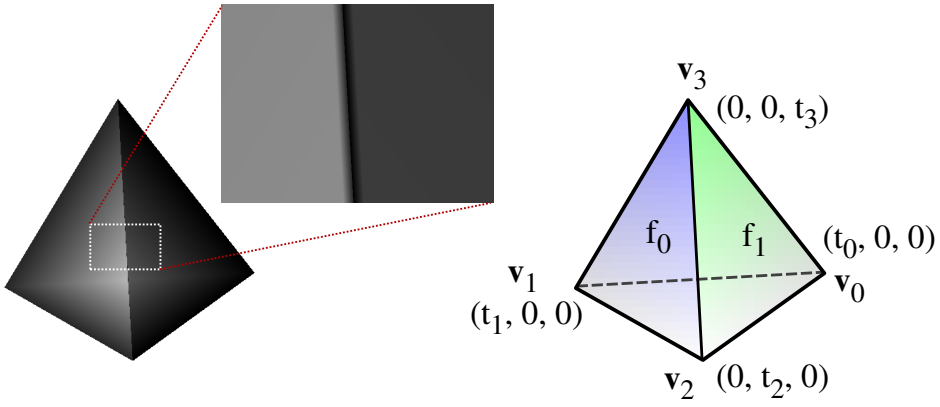


Abbildung 4.7: Wird das Minimum der interpolierten Strahlparameter mit der Dicke des Tetraeders identifiziert, so führt dies zu Artefakten an den Kanten zwischen zwei sichtbaren Tetraederflächen. Fälschlicherweise wird hier eine Dicke von 0 angenommen, was in der helligkeitskodierten Darstellung der Dicke zu deutlich sichtbaren schwarzen Linien führt.

Es sei darauf hingewiesen, dass diese fälschliche Klassifizierung nur darauf zurückzuführen ist, dass die Berechnung mit Hilfe von Vertexprogrammen statt pro Fragment erfolgt. Sie ist damit auch ein weiterer Grund für die Beschränkung des Ansatzes auf orthographische Projektion. Mit einer Berechnung der Klassifikation pro Fragment kann, wie im Abschnitt 4.3 gezeigt wird, die Dicke des Tetraeders immer korrekt ermittelt werden.

Aber auch durch Berechnung pro Vertex lässt sich das Problem beheben. Es muss lediglich sichergestellt sein, dass zusätzlich mindestens ein Vertex auf der Kante die korrekte Klassifikation von t_0 besitzt. Dadurch wird sie auch entlang der Kante richtig interpoliert. Um diesen Vertex zu bestimmen, werden alle Vertizes einer Tetraederfläche von Außen betrachtet im Gegenuhrzeigersinn sortiert und damit ein Umlaufsinn und eine Richtung für die Kanten festgelegt. Jeder Vertex der Fläche kann somit eindeutig als Startpunkt einer Kante betrachtet werden. Jetzt wird jeder Startpunkt einer Kante zusätzlich mit dem Normalenvektor der angrenzenden Fläche als Vertexattribut versorgt. Entsprechend der Behandlung von negativen Strahlparametern, wie sie in Abschnitt 4.2.1 beschrieben ist, kann im Vertexprogramm überprüft werden, ob der Schnittpunkt mit der angrenzenden Fläche einem potentiell eintretenden Schnittpunkt entspricht. Ist dies der Fall, kann die korrespondierende Texturkoordinate statt auf 0 auf einen großen positiven Wert gesetzt werden.

Dies soll nochmals an einem Beispiel verdeutlicht werden. Betrachten wir wiederum die Fläche f_1 im Beispiel von Abbildung 4.7. In diesem Fall ist $\mathbf{v}_3$ der Startpunkt der Kante von $\mathbf{v}_3$ nach $\mathbf{v}_2$ und das Vertexprogramm würde die Komponente r der Texturkoordinate auf einen großen positiven Wert setzen. Die r Komponente von $\mathbf{v}_2$ bleibt für das Zeichnen von f_1 unverändert. Für die Fläche f_0 gilt umgekehrt, dass $\mathbf{v}_2$ der Startpunkt der Kante von $\mathbf{v}_2$ nach $\mathbf{v}_3$ ist, und damit würde die Komponente r von $\mathbf{v}_2$ verändert, weil hier t_1 der zu f_1 korrespondierende Strahlparameter ist und für alle Fragmente von f_0 jeder Schnittpunkt mit f_1 ein potentiell eintretender Schnittpunkt ist.

Selbst mit diesem Schema bleibt die Situation einer fehlenden Klassifikation für einen Punkt der Kante erhalten. Um allerdings in jedem Punkt die korrekte Klassifikation zu errechnen, würde jeder Vertex die Normalenvektoren aller drei Tetraedeflächen benötigen, was auf zusätzliche Vertexattribute hinausläuft. In diesem Ansatz wird zugunsten der höheren Transferrate darauf verzichtet. Der resultierende Fehler betrifft im schlimmsten Fall ein Pixel je Tetraeder und tritt in der Praxis oftmals gar nicht in Erscheinung.

4.3 Fragmentorientierter Ansatz

Der in Abschnitt 4.2 vorgestellte vertexorientierte Ansatz war hauptsächlich durch die Beschränkungen der Fragmenteinheit motiviert. Programmierbare Vertexverarbeitung stand seit der GeForce3 Chip-Serie und damit deutlich früher als programmierbare Fragmentverarbeitung zur Verfügung. Beginnend mit der ATI Radeon 9700 Mitte 2002 haben jedoch mittlerweile alle wichtigen Graphikkarten-Hersteller Produkte mit mehr Flexibilität sowohl auf Vertex- als auch auf Fragmentebene vorgestellt. Neben der größeren Auswahl von Instruktionen ist auch die Länge der Programme gewachsen. So spezifiziert die OpenGL-Erweiterung `ARB_fragment_program` [OpenGL ARB 2004], die seit ATI Radeon 9700 und nVidia GeForceFX unterstützt wird, mindestens 48 arithmetisch-logische Operationen und 24 Texturoperationen. Mehrere Ansätze für Shader-Hochsprachen wie Cg von nVidia [nVidia 2002] oder die von 3Dlabs vorgeschlagene OpenGL 2.0 Shading Language [3Dlabs 2002], die seit Juni 2004 vom OpenGL Architectural Review Board als offizieller Standard einer OpenGL Shading Language akzeptiert wurde [Kessenich et al. 2004], erleichtern darüber hinaus die einfache und Hardware-unabhängige Programmierung der flexiblen Einheiten. Unter Verwendung dieser Shading Language soll im Folgenden eine Variante des in Abschnitt 4.2 vorgestellten Algorithmus vorgestellt werden, der einige der wesentlichen Beschränkungen beseitigt. Er besteht aus einem Vertex-Shader-Programm und einem Fragment-Shader-Programm und wurde beispielhaft mit Hilfe einer Wildcat VP870 Graphikkarte realisiert. Die Darstellung hier erfolgt lediglich konzeptionell, für Details der Implementierung wird auf [Weiler et al. 2003b] verwiesen.

4.3.1 Vertex-Shader-Programm

Während in der vertexorientierten Implementierung die Mehrzahl der Berechnungen von der Vertexeinheit durchgeführt wird, erfolgt nun der Hauptteil der Berechnung der Strahlschnitte und der Strahlintegration auf Fragmentebene. Als Aufgabe der Vertexeinheit verbleibt die Versorgung der Fragmenteinheit mit den erforderlichen Daten.

Dazu gehören für jede gezeichnete Tetraederfläche die Normalen der drei übrigen Tetraederflächen, der gegenüberliegende Vertex zur Bestimmung der begrenzenden Ebenen des Tetraeders, der für alle Flächen des Tetraeders konstante Gradient sowie der Skalarwert an jedem Vertex, der von der Graphik-Hardware linear interpoliert wird. Die Position wird neben der Verwendung zur normalen View-Transformation als weiterer Parameter definiert, um vom Fragmentprogramm

darauf zugreifen zu können. Um Fragmentoperationen zu sparen wird auch die Strahlrichtung im Vertexprogramm als Differenz aus der in lokale Koordinaten transformierten konstanten Betrachterposition und den jeweiligen Vertexkoordinaten bestimmt und muss so lediglich pro Fragment normalisiert werden. Dies ist möglich, nachdem Differenzbildung und lineare Interpolation kommutativ sind.

Alle genannten Parameter werden von der Anwendung als Attribute (`glVertexAttribute`) an den Vertizes definiert und sogenannten `varying` Variablen zugewiesen, welche die Interpolatoren der Graphikkarte ansprechen und damit die Schnittstelle zum Fragmentprogramm darstellen. Leider ist in der aktuellen Spezifikation der OpenGL Shading Language keine Möglichkeit für konstante Attribute pro Primitiv vorgesehen. Daher müssen auch Attribute, die in dem jeweils gezeichneten Dreieck konstant sind, wie Normalen und Gradient, je Vertex definiert werden. Dies erfordert zusätzlichen Speicherplatz und belegt unnötige Ressourcen der Graphik-Hardware.

4.3.2 Fragment-Shader-Programm

Ein Fragmentprogramm für die Darstellung von Isoflächen in projizierten Tetraedern ist in Abbildung 4.8 zu sehen. Es führt dieselben Operationen aus, die im Abschnitt 4.2 durch eine Kombination von Berechnungen auf der Vertexeinheit und festverdrahteter OpenGL-Funktionalität erreicht wurden. Es wertet eine leichte Variation von Gleichung (4.2) aus:

$$t_i = \frac{(\mathbf{v} - \mathbf{p}) \cdot \mathbf{n}_i}{(\mathbf{d} \cdot \mathbf{n}_i)} \,, \tag{4.4}$$

wobei $\mathbf{v}$ einen Vertex der i-ten Tetraederfläche bezeichnet.

Analog zur in Abschnitt 4.2 beschriebenen Lösung wird das Minimum aller potentiell austretenden Schnittpunkte bestimmt. Dies erfolgt auf der Basis der linear interpolierten Vertex-Shader-Variablen. Für jede der drei verbleibenden Tetraederflächen wird dabei der Strahlparameter bestimmt und mit dem bisher bekannten Minimum verglichen. Die if-Bedingung, die jeden Update der Strahlparameter-Variablen umschließt, stellt dabei sicher, dass nur dem Betrachter abgewandte Flächen berücksichtigt werden. Das Skalarprodukt aus Normale und Strahlrichtung kann dabei für die Berechnung des Strahlparameters wiederverwendet werden. Eine Normalisierung der Strahlrichtung ist nicht erforderlich, was die Effizienz erhöht. Um lokale Variablen und Instruktionen zu sparen, wurde intensiv von der Wiederverwendung von Variablen Gebrauch gemacht. Dies hat sich aufgrund von Limitierungen der Wildcat VP870 Graphikkarte, für welche die Implementierung erfolgte, als essentiell herausgestellt.

Am Ende des Programms steht die Berechnung der Fragmentfarbe, die in diesem Ansatz nach der Methode der Prä-Integration bestimmt wird. Für die Isoflächendarstellung ist dazu nur der Skalarwert am Austrittspunkt erforderlich, der in Zeile 23 bestimmt wird. Zusammen mit dem Skalarwert am Eintrittspunkt dient er als Koordinate für den Nachschlag in einer 2D-Textur. Für ein rein emissives optisches Modell ist zusätzlich noch die Multiplikation mit der Länge des Strahlsegments, die linear in die Emission eingeht, erforderlich.

```
1    uniform sampler2D preint;
2    uniform float floatmax;

4    varying vec3 normal0, normal1, normal2, point;
5    varying vec3 gradient, position, dir;
6    varying float scalar;
7
8    void main(void) {
9        float f_reg1;
10       float f_reg2 = floatmax;
11       vec3 v_reg1;
12
13       v_reg1 = point - position;
14       f_reg1 = dot(dir,normal0);
15       if (f_reg1 > 0.0) f_reg2 = dot(v_reg1,normal0) / f_reg1;
16
17       f_reg1 = dot(dir,normal1);
18       if (f_reg1 > 0.0) f_reg2 = min(dot(v_reg1,normal1) / f_reg1, f_reg2);
19
20       f_reg1 = dot(dir,normal2);
21       if (f_reg1 > 0.0) f_reg2 = min(dot(v_reg1,normal2) / f_reg1, f_reg2);
22
23       v_reg1.0 = dot(gradient,dir) * f_reg2 + scalar;
24       v_reg1.1 = scalar;
25
26       vec4 v_reg2 = texture2D(preint,vec2(v_reg1));
27
28       gl_FragColor = vec4(v_reg2);
29   }
```

Abbildung 4.8: Fragment-Shader für fragmentorientierte Zellprojektion mit Isoflächendarstellung.

Für ein reines Absorptionsmodell ist zusätzlich noch die Länge des Strahlsegments erforderlich, die als dritte Texturkoordinate für den Nachschlag in einer 3D-Textur verwendet wird. Sie ergibt sich direkt aus der Länge des Vektors vom Eintrittspunkt des Sehstrahls zum Austrittspunkt `f_reg2 * dir`. Als Texturkoordinate muss sie im Interval $[0,1]$ liegen, was eine Normalisierung erforderlich macht. Dies erfolgt durch Division der Länge durch die maximale Kantenlänge aller Tetraeders, die als weitere Shader-Konstante definiert wird.

4.4 Optische Modelle und Strahlintegration

Konzeptionell berechnet der vorgestellte Ansatz zur blickpunktunabhänigen Zellprojektion für jedes Pixel das Strahlsegment eines Sehstrahls durch den Tetraeder. Für die Bestimmung des Farbbeitrags und der Opazität wird auf die Methode der Prä-Integration zurückgegriffen [Röttger et al. 2000, Engel et al. 2001]. Wir verwenden den Skalarwert am Austrittspunkt s_b, den Skalarwert am Eintrittspunkt s_f und die Länge l des Strahlsegments, die aus der Scan-Konvertierung der Tetraeder, wie sie in den vorangegangenen Abschnitten beschrieben wurde, hervorgeht. Sie dienen als Texturkoordinaten für den Nachschlag in einer 2D- bzw. 3D-Textur.

Sowohl für den vertexorientierten Ansatz als auch für den fragmentorientierten Ansatz werden dieselben Texturen verwendet. Beim vertexorientierten Ansatz (Abschnitt 4.2) erfolgt

der Texturnachschlag basierend auf dem Ergebnis des Nachschlags in der Minimumtextur. Dieser abhängige Texturnachschlag wurde unter Verwendung der OpenGL-Erweiterung `NV_texture_shader` erreicht. Die entsprechenden Einstellungen sind in Abbildung 4.9 dargestellt (Details siehe [Weiler et al. 2002]). Im fragmentorientierten Ansatz erfolgt der Nachschlag einfach als Instruktion des Fragmentprogramms (siehe Zeile 26 in Abbildung 4.8).

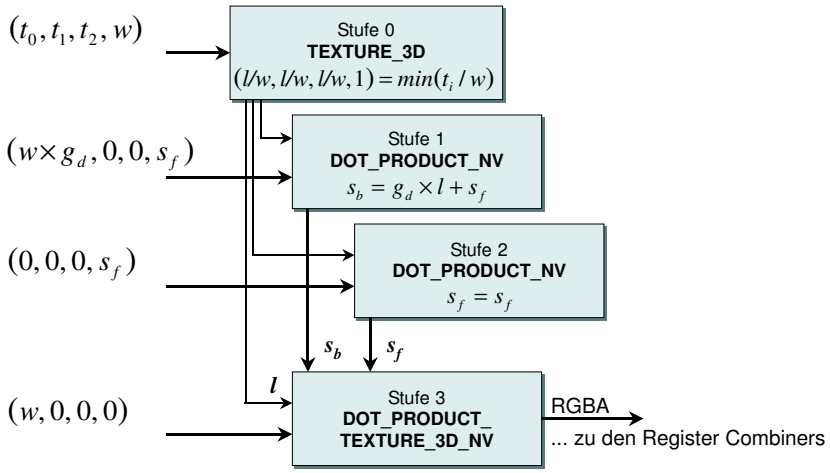


Abbildung 4.9: Einstellung für den Texture Shader bei einem abhängigen 3D-Texturnachschlag basierend auf dem minimalen Strahlparameter aus der Minimumtextur.

4.4.1 Direktes Volume Rendering

Durch die Speicherung von Farb- und Opazitätsbeitrag in einer Textur lassen sich unterschiedliche Effekte erzielen. Allerdings sind für unsortierte Zellen nur solche optischen Modelle mit einer kommutativen Blending-Funktion möglich, d.h. solche Modelle, die ein Compositing in beliebiger Reihenfolge erlauben. Unter ihnen sind Maximum Intensity Projection (MIP) und eine Beschränkung des kompletten Density-Emitter-Modells lediglich auf den Emissions- oder alternativ den Absorptionsterm. Alle diese Modelle lassen sich mit einer beliebigen Transferfunktion kombinieren, da diese nur für die Erzeugung der Textur eine Rolle spielt.

Für Maximum Intensity Projection wird beispielsweise eine 2D-Textur verwendet, die an jeder Position (s_f, s_b) das Maximum der Transferfunktion im Interval $[\min(s_f, s_b), \max(s_f, s_b)]$ enthält. Das Ergebnis wird mit einer Maximum-Blending-Funktion von OpenGL auf einen schwarzen Hintergrund gezeichnet. Eine rein emissive Darstellung addiert die Beiträge der einzelnen Tetraeder. In diesem Fall kommt eine 2D-Textur zum Einsatz, die an jeder Position (s_f, s_b) das Integral der Transferfunktion im Intervall $[\min(s_f, s_b), \max(s_f, s_b)]$ enthält. Ein reines Absorptionsmodell benötigt eine 3D-Textur mit dem Absorptionsterm des Volumenintegrals entlang der Strahlrichtung.

Um den Tiefeneindruck zu erhöhen, lassen sich alle diese optischen Modelle durch sogenanntes *Intensity Depth Cuing* [Foley et al. 1990] erweitern. Beispiele dafür finden sich in Abbildung 4.10.

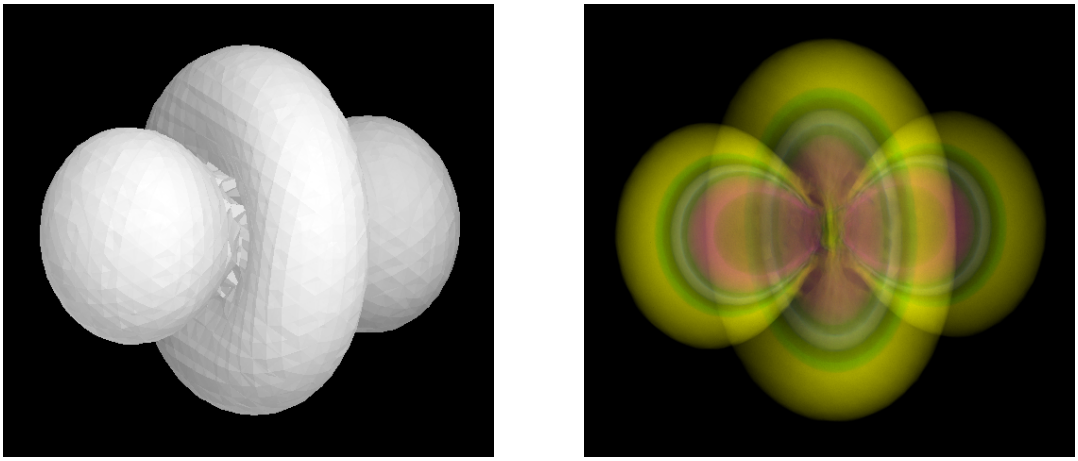


Abbildung 4.10: Beide Bilder zeigen denselben orbitalähnlichen künstlichen Datensatz aus 150.000 Tetraedern unter unterschiedlichen optischen Modellen. Links wird eine Isofläche dargestellt, wohingegen im rechten Bild ein rein emissives Modell verwendet wurde. Außerdem wurde im rechten Bild der Tiefeneindruck durch Intensity Depth Cuing erhöht.

4.4.2 Isoflächen

Der Ansatz erlaubt auch die Darstellung mehrerer beleuchteter Isoflächen ohne vorhergehende Sortierung der Zellen. Dazu wird eine Textur entsprechend Abbildung 4.11 (vergleiche auch [Röttger et al. 2000]) verwendet. Die korrekte Verdeckung der Isoflächenfragmente wird dabei selbst ohne Sortierung der Zellen des unstrukturierten Gitters durch eine Kombination von OpenGL-Alpha-Test und Tiefentest erreicht.

Die grundlegende Idee ist offensichtlich. Überall dort, wo ein Sehstrahl innerhalb des Tetraeders die Isofläche trifft, gilt entweder $s_f < s_{\text{iso}} < s_b$ oder $s_f > s_{\text{iso}} > s_b$. An diesen Stellen liefert die Textur die ambiente Materialfarbe der Isofläche. An allen anderen Stellen wird der Alpha-Kanal der Textur auf 0 gesetzt, wodurch Fragmente, die nicht zur Isofläche gehören, durch den Alpha-Test entfernt werden. Für mehrere Isoflächen ist eine „Sortierung" erforderlich. Für $s_f < s_b$ schaut man entlang des Gradienten des Skalarfeldes. Damit verdecken Isoflächen mit kleineren Isowerten solche mit größeren Isowerten. Analoges gilt im umgekehrten Fall. Die Darstellung mehrerer Isoflächen erlaubt auch die Zuweisung unterschiedlicher Farben je Isofläche, wie in Abbildung 4.11 zu sehen. Allerdings ist das Zeichnen auf Flat Shading beschränkt, wobei die Anzahl der unterstützten Lichtquellen im Wesentlichen nur durch die Anzahl der Vertex-Shader-Konstanten beschränkt ist.

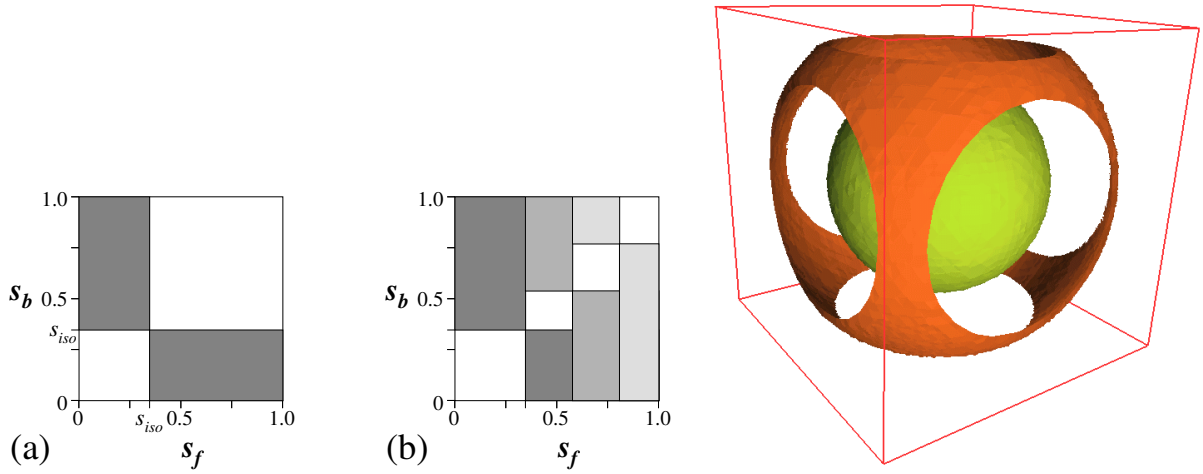


Abbildung 4.11: Blickpunktunabhängige Isoflächendarstellung. Links ist die 2D-Textur für eine (a) bzw. mehrere Isoflächen (b) dargestellt, in der Texel, die zu einer Isofläche korrespondieren, die Materialfarbe der Isofläche besitzen. Ein Alpha-Wert von 0 entspricht der Abwesenheit einer Isofläche. Das Beispiel rechts demonstriert die Möglichkeit unterschiedlicher Materialfarben.

4.5 Zellprojektion von Tetraedergittern

Die Darstellung von Tetraedergittern mittels Zellprojektion besteht nicht nur aus der Projektion eines einzelnen Tetraeders, sondern auch aus der Sortierung der Tetraeder und der Übertragung großer Mengen von Tetraedern zur Graphikkarte, was in diesem Abschnitt diskutiert wird.

Volumenvisualisierungsanwendungen basierend auf der Shirly-Tuchman-Zellprojektion sind nicht in der Lage, die Maximalleistung moderner Graphikkarten auszunutzen [King et al. 2001]. Dafür gibt es zwei Gründe: Erstens können die zerlegten Dreiecke aufgrund des Mehraufwands für die Sortierung nicht in ausreichender Geschwindigkeit generiert werden und zweitens ist die Datenübertragung zwischen der CPU und der Graphikkarte insbesondere auf PC-Architekturen beschränkt. Die maximale Datenrate wird dort nur im Burst-Modus erreicht, wenn die Übertragung blockweise erfolgt. Die Ausnutzung dieses Burst-Transfers bietet somit ein großes Potential zur Verbesserung der Performanz von Volumenvisualisierungsanwendungen für unstrukturierte Gitter. Theoretische Überlegungen [King et al. 2001] liefern Beschleunigungen bis zum Faktor 20. Die charakteristische Erzeugung der Dreiecke in PT-basierten Algorithmen liefert eher einen kontinuierlichen Strom von Dreiecken, der sich auf eine blockweise Übertragung nicht ohne Zwischenspeicherung und den damit verbundenen signifikanten Mehraufwand abbilden lässt.

Die Motivation für den hier vorgestellten Ansatz der blickpunktunabhängigen Zellprojektion ist die Überwindung dieses Flaschenhalses in der Datenübertragung von der CPU zur Graphikkarte. Die blickpunktunabhängige Beschreibung des Tetraedergitters erlaubt es, in einem Vorverarbeitungsschritt eine statische Datenstruktur zu definieren, die in jedem Frame wiederverwendet werden kann. Dies erlaubt zahlreiche Optimierungen, wie das Caching der Daten oder die

Vorverarbeitung zum Zweck der optimierten Übertragung. Die in OpenGL dafür vorgesehenen Mechanismen heißen Display Lists und Vertex Arrays [Shreiner et al. 2004].

Die grundlegende Idee von Display Lists besteht im Caching von OpenGL-Instruktionen, dergestalt, dass dieselbe Geometrie nur einmal definiert wird und dann durch einfaches Ausführen der Display Lists mehrfach gezeichnet werden kann. Die Beschleunigung wird dadurch erreicht, dass die Display Lists bereits bei der Definition in eine Form gebracht werden, die kompatibel mit der Graphik-Hardware oder den darüberliegenden Software-Schichten ist. Daneben kann ihre Speicherung in speziellen Speicherbereichen im Graphiktreiber oder in direkt von der Graphik-Hardware adressierbaren Speicherbereichen erfolgen.

Vertex Arrays dienen eigentlich der Reduktion der Anzahl von OpenGL-Funktionsaufrufen. Alle Vertexdaten werden dabei in einigen wenigen Arrays gespeichert, wodurch letztlich nur ein Funktionsaufruf für das Zeichnen sehr vieler graphischer Primitive erforderlich ist. In ähnlicher Weise wie Display Lists können Vertex Arrays dabei in einem Cache vorgehalten werden und durch Vorverarbeitung optimiert werden. Neuere OpenGL-Erweiterungen, wie z.B. `NV_vertex_array_range` [OpenGL ARB 2004] ermöglichen eventuell sogar ein Vorhalten der Daten im lokalen Speicher der Graphikkarte oder alternativ dazu in dedizierten Speicherbereichen im Adressraum der Anwendung mit direktem Zugriff durch die Graphikkarte mittels Direct Memory Access (DMA). Die einzige Information, die von der Anwendung geliefert werden muss, sind Indizes in die Vertex Arrays oder die Anzahl der zu zeichnenden Primitive.

Die hier vorgestellte Implementierung verwendet Vertex Arrays, nachdem sich in unseren Untersuchungen auf aktueller PC-Graphik-Hardware kein messbarer Geschwindigkeitsgewinn durch die Verwendung von Display Lists gezeigt hat. Dies kann jedoch abhängig von Graphikkarte und Treiberversion unterschiedlich sein, so dass im Zweifelsfall beide Varianten ausprobiert werden sollten. Tatsächlich hat die Verwendung von Vertex Arrays gegenüber der Verwendung von Display Lists einige Nachteile, insbesondere die fehlende Möglichkeit, Parameter je Graphikprimitiv zu definieren. In Vertex Arrays sind nur mit den Vertizes assoziierte Parameter möglich, was zu der Notwendigkeit der Datenreplikation führt. Auch benötigt in der vorgestellten Implementierung jede Fläche eines Tetraeders individuelle Vertexinstanzen, so dass sich die Verwendung von Dreiecksstreifen mit nur sechs Vertizes pro Tetraeder verbietet. Stattdessen werden für jeden Tetraeder vier Dreiecke mit insgesamt zwölf Vertizes benötigt.

Indem die Daten in Vertex Arrays gespeichert werden, lässt sich das gesamte Tetraedergitter mit lediglich einem einzigen Aufruf von `glDrawArrays` zeichnen. Mit der Annahme, dass die Vertexdaten auf der Graphikkarte liegen, reduziert der Ansatz zur blickpunktunabhängigen Zellprojektion die Datenmenge, die zur Graphikkarte übertragen werden muss, auf wenige Bytes je Frame, was dem Graphikprozessor erlaubt, mit voller Kapazität zu arbeiten, ohne auf Daten warten zu müssen.

Das skizzierte Szenario lässt sich aber nur erreichen, wenn keine Sortierung der Tetraeder erfolgt. Schließlich liegen die Elemente im Vertex Array in einer festen Reihenfolge vor. Dies beschränkt die Wahl des optischen Modells. Prinzipiell erlaubt der Ansatz jedoch auch die Darstellung mit korrekter Sichtbarkeitssortierung. Dazu wird dasselbe Array verwendet, nun aber wird anstelle eines einzigen Aufrufs von `glDrawArray` jeweils ein Aufruf pro Tetraeder abgesetzt, der den

jeweils richtigen Tetraeder entsprechend der Sortierreihenfolge anspricht. Ein Sortieralgorithmus muss dazu lediglich eine sortierte Liste der Tetraederindizes liefern, was den Ansatz kompatibel zu jedem publizierten Sortieralgorithmus für Tetraeder macht.

4.6 Ergebnisse

Der vertexorientierte Ansatz wurde auf zwei Systemen getestet: Einem Linux PC mit Athlon 1200 MHz Prozessor und 512 MB RAM mit einer 64 MB GeForce3 Graphikkarte und einem PC mit GeForce4 Ti mit 128 MB lokalem Speicher in einem Athlon XP 1800+ System mit 512 MB RAM. Es sei jedoch darauf hingewiesen, dass der Einfluss der CPU vernachlässigbar ist, da der Ansatz die meisten Berechnungen auf die Graphikkarte verlagert.

Tabelle 4.1 zeigt die Performanz der beiden Systeme für unterschiedliche Datensätze. Sie wird weder durch die Wahl des optischen Modells noch durch die Transferfunktion beeinflusst, da beide nur den Inhalt einer 2D-Textur definieren. Selbst beim aufwändigeren Nachschlag in einer 3D-Textur, der für das reine Absorptionsmodell erforderlich ist, bleibt die Performanz vergleichbar, was darauf schließen lässt, dass die Implementierung nicht durch die Rasterisierung limitiert ist.

Die Zeiten in Tabelle 4.1 reflektieren das Ergebnis intensiver Optimierung hinsichtlich der auf die Graphikkarte übertragenen Datenmenge. Letztlich konnte durch Zusammenfassen von Vertexattributen und der Umstellung des Datentyps von Float nach Short, wo immer dies ohne Effekt auf die Bildqualität möglich war, der Speicherverbrauch von 720 Byte/Tet auf 456 Byte/Tet reduziert werden, wobei die erzielte Bildwiederholrate proportional zur gesparten Bandbreite anstieg.

Dennoch bleibt die durchschnittliche Datenrate von etwa 215 MByte/sek signifikant unter dem theoretischen Maximalwert einer $4\times$AGP-Verbindung von 1000 MByte/sek zurück. Der Grund dafür liegt weder in der Rasterisierung noch der Verarbeitung in der Vertexeinheit, die zuerst verdächtigt wurde, weil sie aufgrund der relativ aufwändigen Vertexprogramme mit 30-50 Instruktionen einen signifikanten Mehraufwand im Vergleich zur festverdrahteten Transformation und Beleuchtungsberechnung darstellt. Im Speziellen gilt dies für den GeForce3 Chip, der nur eine Vertexeinheit besitzt.

Tatsächlich liegt der Flaschenhals in der Datenübertragung. Für Datensätze relevanter Größe konnte die OpenGL-Erweiterung `NV_vertex_array_range` nicht ausgenutzt werden, die in Messungen zwei- bis dreimal schneller als jeder andere Ansatz war. Der Performanz-Vorteil resultiert aus der Tatsache, dass die Vertex Arrays mit Hilfe dieser Erweiterung entweder lokal auf der Graphikkarte oder in Speicherbereichen liegen, die direkt von der Graphikkarte mittels DMA-Pull zugreifbar sind. Vertex Range Arrays sind jedoch selbst auf Graphikkarten mit 128 MB Speicher auf weniger als 32 MB beschränkt, was lediglich für Datensätze von maximal 90.000 Tetraedern ausreicht. Für normale Vertex Arrays scheint der Flaschenhals durch die Bandbreite zum Hauptspeicher definiert zu werden. Dies wird gestützt durch die Beobachtung, dass der Performanzunterschied zwischen den Testsystemen genau dem Bandbreitenunterschied in den jeweiligen Hauptspeicherbussen entspricht.

Tabelle 4.1: Vergleich der Performanz des vertexorientierten Ansatzes zur Zellprojektion zwischen dem GeForce3 und GeForce4 Chip von nVidia.

Datensatz	Num. Tetraeder	GeForce3			GeForce4		
		Fps	Tet/Sek	MB/Sek	Fps	Tet/Sek	MB/Sek
Bluntfin	224.874	1,67	375 K	167	2,13	479 K	213
Heatsink	121.668	3,02	367 K	163	3,81	464 K	207
Orbital	148.955	2,53	377 K	168	3,23	481 K	214
Sphere	148.955	2,53	377 K	168	3,23	481 K	214

Der fragmentorientierte Ansatz wurde auf einer 3Dlabs Wildcat VP870 Graphikkarte unter Verwendung von OpenGL 2.0 Betatreibern (Version 3.01-0115) getestet. Diese Version enthält zahlreiche Beschränkungen, welche die Implementierung der Fragmentprogramme deutlich behinderten und massive Optimierungen in Hinblick auf die Anzahl der Operationen und Variablen erforderlich machten, da weder Funktionsaufrufe noch die an sich vorgesehene automatische Verteilung auf mehrere Zeichendurchgänge unterstützt werden.

Aufgrund dieser Limitierungen konnte das vollständige Programm aus Abschnitt 4.3.2 nicht getestet werden, da es selbst nach der Optimierung noch mehr Instruktionen enthält, als die Test-Hardware unterstützt. Durch Auskommentierung der Zeilen 20 und 21 konnte jedoch der Strahlparameter zumindest für zwei Tetraederflächen richtig berechnet werden, was in einem korrekten Bild resultiert, solange die gezeichnete Fläche des Tetraeders nicht die einzige zum Betrachter gerichtete Fläche ist. Auf aktueller Graphik-Hardware stehen genügend Fragmentinstruktionen zur Verfügung und das demonstrierte Programm liefert korrekte Ergebnisse.

Kapitel 5

Hardware-basiertes Ray Casting

Die Ausnutzung der Möglichkeiten moderner Graphikkarten wird ein immer wichtigerer Aspekt für das Design von Graphikalgorithmen und im Speziellen Visualisierungsalgorithmen. Die hohe Rechenleistung der Graphik-Hardware, die durch die intrinsische Parallelität und die hohen internen Speicherbandbreiten erreicht wird, kann dabei jedoch nur zum Tragen kommen, wenn die Kommunikation mit der CPU als limitierender Faktor ausgeschaltet wird. Im Idealfall lässt sich dies erreichen, wenn die gesamte Berechnung auf der Graphikkarte erfolgen kann. Dies wird aufgrund der flexiblen Programmierbarkeit und der im Gleitkommaformat arbeitenden Arithmetik moderner Graphikkarten für eine zunehmende Anzahl von Algorithmen möglich.

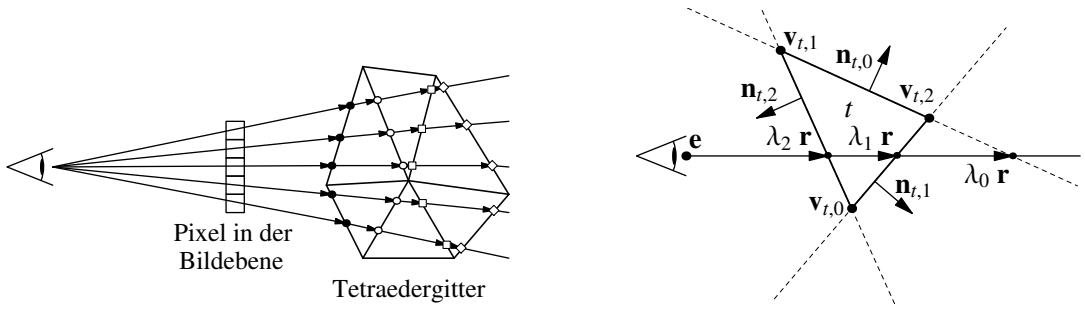
Der im vorangegangenen Kapitel vorgestellte Ansatz zur Hardware-basierten Zellprojektion scheidet als Kandidat für eine vollständige Implementierung auf der Graphikkarte aus, weil bisher kein Algorithmus für die komplette Sortierung der Zellen auf der Graphikkarte existiert. Lediglich die Möglichkeit einer Feinsortierung auf der Graphikkarte basierend auf einer mittels CPU berechneten groben Vorsortierung, wurde in [Callahan et al. 2004] demonstriert. Für die vollständige Zellsortierung auf der Graphikkarte wurden Lösungsansätze in Form der R-Buffer-Architektur [Wittenbrink 2001, King et al. 2001] vorgeschlagen, jedoch bisher nicht verwirklicht.

Weitergedacht führt der Ansatz der Hardware-basierten Zellprojektion, der im Wesentlichen auf der Anwendung von Ray Casting für jeden einzelnen Tetraeder beruht, zu der Idee, Ray Casting auf das gesamte Tetraedergitter anzuwenden und dies auf der Graphikkarte zu implementieren. Ray Casting bietet einige Vorzüge. So erfolgt beispielsweise die Sortierung intrinsisch und wird pro Pixel bestimmt, wodurch zyklische Tetraedergitter kein Problem darstellen. Durch Techniken zum frühen Strahlabbruch (Early Ray Termination) können darüber hinaus signifikante Performanzsteigerungen erzielt werden. Für die Umsetzung auf Graphik-Hardware bietet sich Ray Casting insbesondere aufgrund seiner einfachen Parallelisierbarkeit und der Tatsache, dass keine wahlfreien Speicherzugriffe erforderlich sind, an. Letzteres kommt der Streaming-Architektur der Graphik-Hardware entgegen. Ein Vorschlag zur Realisierung von Ray Casting auf Tetraedergittern mit Hilfe von programmierbarer Graphik-Hardware wird in diesem Abschnitt vorgestellt. Veröffentlichungen dazu finden sich in [Weiler et al. 2003a, Weiler et al. 2004b]. Für uniforme Gitter wurden parallel zu dem hier vorgestellten Verfahren zwei Ansätze entwickelt [Röttger et al. 2003, Krüger und Westermann 2003], die teilweise ähnliche Mechanis-

men verwenden. Ein ausführlicher Vergleich findet sich in [Weiler et al. 2006]. Ein alternatives Hardware-beschleunigtes Verfahren zum Ray Casting wurde in [Westermann und Ertl 1997] vorgestellt.

5.1 Paralleles Ray Casting

Als Grundlage für das hier vorgestellte Ray Casting auf programmierbarer Graphik-Hardware dient ein Ansatz zur Strahlpropagierung, der sich an einer in [Garrity 1990] vorgestellten Idee orientiert. Wie in Abbildung 5.1(a) zu sehen, wird dabei jeder Sehstrahl ausgehend vom Betrachter so lange von einer Zelle zur nächsten propagiert, bis das gesamte Tetraedergitter traversiert ist. Der Strahl startet dabei ausgehend vom ersten Schnittpunkt mit dem Gitter, der in einer Initialisierungsphase ermittelt wird. Die Traversierung erfolgt in mehreren Durchläufen. In jedem Durchlauf wird der Farb- und Opazitätsbeitrag entsprechend dem aktuellen Tetraeder berechnet und im Framebuffer akkumuliert.



(a) (b)

Abbildung 5.1: Prinzip des Hardware-basierten Ray Casting: (a) Für jedes Pixel wird ein Strahl traversiert, der jeweils an den Schnittpunkten mit den Flächen der Tetraeder anhält. Der erste Schnittpunkt ist mit Punkten (•) gekennzeichnet, weitere Schnittpunkte mit Kreisen (○), Quadraten (□) und Rauten (◇). (b) Schnittberechnung des Sehstrahls mit den Flächen des Tetraeders.

Eine CPU-Lösung bearbeitet dabei typischerweise die Pixel und damit die Sehstrahlen sequentiell. Ray Casting ist jedoch einfach parallelisierbar, da die Berechnung der Pixel unabhängig erfolgt. Dies ist der Schlüssel für eine Implementierung mit programmierbarer Graphik-Hardware, bei der jedes Fragment unabhängig bearbeitet wird.

5.1.1 Überblick

In dem hier vorgestellten Ansatz werden Fragmentprogramme verwendet, um alle für die Strahlpropagierung erforderlichen Berechnungen auf der Graphik-Hardware zu erledigen. Für jeden Schritt der Propagierung wird ein bildschirmfüllendes Rechteck gezeichnet, um gleichsam für alle Pixel und damit für alle Sehstrahlen parallel eine Zelle weiterzulaufen.

Dazu muss Traversierungsinformation über die Schnittpunkte der Strahlen mit den Tetraedern zwischen aufeinanderfolgenden Traversierungsschritten ausgetauscht werden. Dies erfolgt mit Hilfe von mehreren 2D-Texturen, die jeweils dieselbe Größe wie der Framebuffer besitzen und in jedem Schritt gelesen und aktualisiert werden. Die Texturen beinhalten den aktuellen Schnittpunkt des Strahls mit der Tetraederfläche sowie den Index des Nachbartetraeders, in den der Strahl läuft.

Um alle für einen Zeichendurchgang erforderlichen Berechnungen durchführen zu können, benötigt das Fragmentprogramm Zugriff auf die Daten des Tetraedergitters. Daher werden die Vertexdaten, die Normalen der Tetraederflächen und Nachbarschaftsinformation in mehreren Texturen gespeichert. Dies erlaubt nicht nur wahlfreien Zugriff durch das Fragmentprogramm, sondern auch, dass die Daten im lokalen Speicher der Graphikkarte lagern, wodurch der vorgestellte Algorithmus sehr niedrige Bandbreitenanforderungen besitzt.

Für jeden Sehstrahl sind die folgenden Schritte erforderlich:

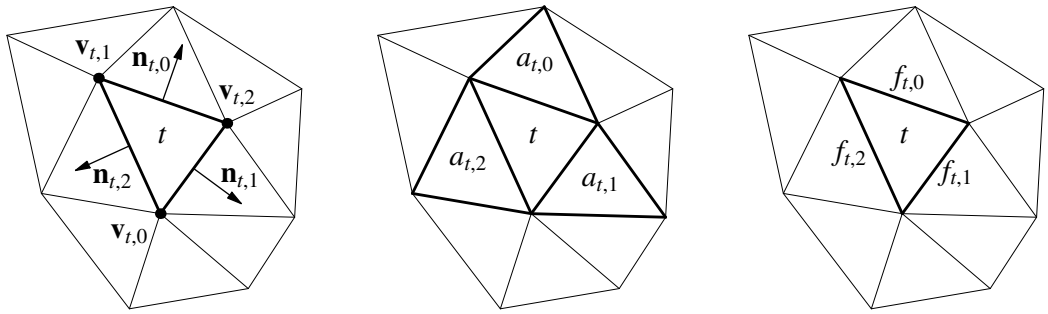
1. Initialisierung
2. Solange noch innerhalb des Tetraedergitters
   (a) Berechne Austrittspunkt aus dem aktuellen Tetraeder
   (b) Ermittle Skalarwert am Austrittspunkt
   (c) Berechne Strahlintegral im Inneren der aktuellen Zelle
   (d) Blende Farb- und Opazitätsbeitrag in den Framebuffer
   (e) Fahre mit der Nachbarzelle am Austrittspunkt fort

Zunächst wird der Framebuffer gelöscht und mit dem ersten Schnittpunkt jedes Sehstrahls, einem Schnittpunkt mit der Umrandung des Gitters, initialisiert. Dies kann erfolgen, indem die Randflächen des Tetraedergitters rasterisiert werden. Allerdings kann genauso gut eine Berechnung durch die CPU erfolgen, nachdem die Zahl der Randflächen typischerweise klein ist und dieser Schritt daher nicht zeitkritisch ist. Die übrigen Schritte lassen sich entsprechend ihrer Zugehörigkeit zu Strahlintegration oder Traversierung unterteilen. Auf beide Themenkreise wird nach einer kurzen Einführung der Nomenklatur eingegangen.

5.1.2 Nomenklatur

Zur Beschreibung der Berechnungen für einen Zeichendurchgang werden einige Notationen eingeführt. Vergleiche auch Abbildung 5.2. Ein Tetraeder aus einem Gitter von n Zellen wird über seinen Index 0 bis $n-1$ identifiziert. Ein beliebiger Index wird im Folgenden häufig mit t bezeichnet. Jeder Tetraeder t besitzt vier Flächen, deren Normalen mit $\mathbf{n}_{t,i}$ bezeichnet sind. Dabei beschreibt i den Index der Fläche und kann 0, 1, 2 oder 3 sein. Es wird angenommen, dass die Normalen von der Zelle nach außen zeigen. Jeder Tetraeder t definiert außerdem vier Vertizes

$\mathbf{v}_{t,i}$, wobei Vertex $\mathbf{v}_{t,i}$ gegenüber der i-ten Fläche liegt (vergleiche Abbildung 5.2(a)). Wie in Abbildung 5.2(b) zu sehen, wird der Nachbar, der mit dem Tetraeder t die i-te Fläche teilt, mit $a_{t,i}$ bezeichnet. Der Index der Fläche von $a_{t,i}$, die zur i-ten Fläche von t korrespondiert, wird durch $f_{t,i}$ beschrieben (vergleiche Abbildung 5.2(c)).



(a) (b) (c)

Abbildung 5.2: Nomenklatur für Tetraeder: (a) Der Vertex $\mathbf{v}_{t,i}$ liegt in Tetraeder t gegenüber der i-ten Fläche ($i \in \{0,1,2,3\}$), welche die Normale $\mathbf{n}_{t,i}$ besitzt. (b) Die Nachbarzelle $a_{t,i}$ hat mit Tetraeder t die i-te Fläche gemeinsam. (c) Für die Flächenindizes $f_{t,i}$ des Nachbarn gilt, dass die i-te Fläche von t zur $f_{t,i}$-ten Fläche des Nachbarn $a_{t,i}$ von t korrespondiert.

5.1.3 Strahlintegration

Die Strahlintegration stellt die zentrale Aufgabe innerhalb eines Zeichendurchgangs des vorgestellten Algorithmus zum Ray Casting von Tetraedergittern dar. Für jeden Sehstrahl muss der Farb- und Opazitätsbeitrag bezogen auf die aktuelle Zelle bestimmt werden. Dafür wird hier wiederum der Ansatz der Prä-Integration [Röttger et al. 2000] verwendet, der vorberechnete Farb- und Opazitätswerte in Texturen speichert, die mit dem Skalarwert am Eintrittspunkt, dem Skalarwert am Austrittspunkt und der Länge des Strahlsegments parametrisiert werden.

Der Eintrittspunkt sowie sein Skalarwert können einfach aus den zur Speicherung von Traversierungsinformation verwendeten Texturen gelesen werden, wie sie in Abschnitt 5.1.1 eingeführt wurden. Diese Texturen identifizieren auch die Zelle, die der Strahl an diesem Punkt betritt. Der korrespondierende Austrittspunkt wird durch die Berechnung der drei Schnittpunkte mit den übrigen Flächen des aktuellen Tetraeders berechnet. Von diesen wird der Schnittpunkt gewählt, der dem Betrachter am nächsten liegt, jedoch nicht auf vom Betrachter sichtbaren Flächen. Es wird darauf hingewiesen, dass der Schnittpunkt mit der Eintrittsfläche nicht berechnet werden muss.

Ausgehend von der Eintrittsfläche j, dem Augpunkt $\mathbf{e}$ und der normalisierten Richtung des Sehstrahls $\mathbf{r}$ (siehe Abbildung 5.1(b)), ergeben sich die drei Schnittpunkte mit den Flächen des Tetraeders t zu $\mathbf{e} + \lambda_i \mathbf{r}$ mit $0 \leq i < 4 \ \wedge \ i \neq j$ und

$$\lambda_i = \frac{(\mathbf{v}-\mathbf{e})\cdot\mathbf{n}_{t,i}}{\mathbf{r}\cdot\mathbf{n}_{t,i}}, \quad \text{wobei} \quad \mathbf{v} := \mathbf{v}_{t,3-i}. \tag{5.1}$$

Diese Gleichung lässt sich einfach mit Hilfe von Fragmentoperationen berechnen. Eine Fläche ist sichtbar, wenn der Nenner des Bruchs negativ ist. Die Sichtbarkeit ergibt sich also nahezu ohne weitere Kosten. Werden die λ_i für alle sichtbaren Flächen auf einen großen positiven Wert gesetzt, ergibt sich der gesuchte Austrittspunkt als $\min\{\lambda_i | 0 \leq i < 4 \ \wedge \ i \neq j\}$. Auch diese Funktion lässt sich einfach auf Fragmentoperationen abbilden. Gegebenenfalls kann eine im Befehlssatz der Graphik-Hardware fehlende Minimumfunktion durch eine Sequenz von bedingten Zuweisungsoperationen emuliert werden. Weitere Möglichkeiten werden in Abschnitt 4.2.1 diskutiert.

Sobald das Minimum aller λ_i und die zugehörige Fläche i bestimmt ist, kann der Schnittpunkt $\mathbf{x}$ entsprechend $\mathbf{x} = \mathbf{e} + \lambda_i\mathbf{r}$ berechnet werden. Der Skalarwert $s(\mathbf{x})$ am Punkt $\mathbf{x}$ kann gemäß

$$s(\mathbf{x}) = \mathbf{g}_t \cdot (\mathbf{x}-\mathbf{x}_0) + s(\mathbf{x}_0) = \mathbf{g}_t \cdot \mathbf{x} + (-\mathbf{g}_t \cdot \mathbf{x}_0 + s(\mathbf{x}_0)), \tag{5.2}$$

berechnet werden, wobei $\mathbf{g}_t$ der Gradient des Skalarfeldes ist, der innerhalb eines Tetraeders konstant ist. Der Punkt $\mathbf{x}_0$ ist ein beliebiger Punkt innerhalb des Tetraeders, z.B. einer der Vertizes. Man beachte, dass Gleichung (5.2) impliziert, dass nur ein Vektor $\mathbf{g}_t$ und ein Skalar $\hat{g}_t = -\mathbf{g}_t \cdot \mathbf{x}_0 + s(\mathbf{x}_0)$ für jeden Tetraeder gespeichert werden muss, um das Skalarfeld an jedem Punkt innerhalb des Tetraeders auswerten zu können. Darüber hinaus benötigt diese Rekonstruktion nur ein Skalarprodukt und eine skalare Addition, was sehr viel effizienter implementiert werden kann als die komplette lineare Interpolation aus den Skalarwerten an den Vertizes. Dies gilt insbesondere, weil die komplette Interpolation noch die Knotengewichte erfordert.

Mit den Skalarwerten am Eintrittspunkt und am Austrittspunkt und der Länge des Strahlsegments, die als Abstand zwischen den Koordinaten des Eintritts- und Austrittspunkts berechnet wird, kann nun das Strahlintegral durch einen Nachschlag in einer Prä-Integrationstextur bestimmt werden. Wie bereits in Abschnitt 4.4 gezeigt, sind dabei unterschiedliche optische Modelle, einschließlich einem reinen Emissions- oder Absorptionsmodell, Maximum Intensity Projection und der Darstellung von beleuchteten Isoflächen, möglich. Im Falle von Ray Casting werden, aufgrund der intrinsischen Sortierung der Strahlsegmente, darüber hinaus auch optische Modelle unterstützt, die nicht-kommutatives Blending voraussetzen. Im Besonderen trifft dies für das Density-Emitter-Modell [Williams und Max 1992] zu, welches für gewöhnlich im Volume Rendering Anwendung findet.

Im letzten Schritt wird der Farb- und Opazitätsbeitrag jedes Strahlsegments in den Framebuffer geschrieben. Da das Blending in der Reihenfolge von vorne nach hinten erfolgt, ergibt sich die akkumulierte assoziierte (d.h. vormultiplizierte) Farbe $\tilde{C}'_k$ und Opazität α'_k nach k Durchläufen gemäß

$$\tilde{C}'_k = \tilde{C}'_{k-1} + (1-\alpha'_{k-1})\tilde{C}_k \quad \text{und} \quad \alpha'_k = \alpha'_{k-1} + (1-\alpha'_{k-1})\alpha_k,$$

wobei $\tilde{C}_k$ und α_k die assoziierte Farbe und Opazität des Strahlsegments aus dem k-ten Zeichendurchgang bezeichnen.

5.1.4 Zelltraversierung

Wie bereits erwähnt, erfolgt die Traversierung der Sehstrahlen durch sukzessives Zeichnen bildschirmfüllender Rechtecke. Mit jedem Zeichendurchgang wird eine Schicht von Zellen des Tetraedergitters durchschritten. Die Traversierung wird kontrolliert vom Index der aktuellen Zelle, der je Sehstrahl in Zwischentexturen von derselben Größe wie der Framebuffer gespeichert wird. Das verwendete Fragmentprogramm liest den Index der aktuellen Zelle aus dieser Textur und überschreibt den Texel mit dem Index der Zelle, die an der Fläche angrenzt, durch den der Strahl die aktuelle Zelle verlässt. Dieser Index ergibt sich durch Nachschlag von $a_{t,i}$, wenn t die aktuelle Zelle und i der Index ist, der dem ermittelten minimalen Strahlparameter λ_i entspricht.

Für Zellen am Rand ist dieser Index -1, so dass einfach ermittelt werden kann, wenn der Sehstrahl das Gitter verlässt und die Bearbeitung des korrespondierenden Pixels beendet werden kann. In diesem Fall wird der neue Austrittspunkt auf den alten Eintrittspunkt gesetzt, was in einem Strahlsegment der Länge Null resultiert, welches keinen Einfluss auf den Framebuffer-Inhalt hat. Eine effizientere Lösung, die den vorgelagerten Tiefentest moderner Graphikkarten nutzt, wird in Abschnitt 5.5.2 präsentiert.

Dieses Vorgehen ist jedoch nur für konvexe Gitter sinnvoll, da bei nicht-konvexen Gittern der Sehstrahl wieder eintreten kann. Dies kann durch eine vorgelagerte Konvexifizierung des Gitters behoben werden. Entsprechend der in [Williams 1992] vorgeschlagenen Idee wird dabei der Bereich zwischen dem Rand des Tetraedergitters und seiner konvexen Hülle durch imaginäre Zellen aufgefüllt. Beim Zeichnen müssen diese Zellen gesondert behandelt werden, da sie nicht zur Farbe und Opazität des Sehstrahls beitragen dürfen. Der Austrittspunkt und die nachfolgende Zelle müssen jedoch wie bei gewöhnlichen Zellen bestimmt und in die Traversierungsdatenstruktur geschrieben werden. Eine alternative Methode für die Unterstützung nicht-konvexer Gitter wird in Abschnitt 5.5.3 beschrieben.

Die Traversierung der Sehstrahlen muss so lange fortgesetzt werden, bis alle Strahlen das Gitter verlassen haben. In [Kraus und Ertl 2002] wird vorgeschlagen, diesen Test auf der CPU zu erledigen, indem die entsprechende Komponente der Zwischentextur, welche den Index der aktuellen Zelle enthält, ausgewertet wird. Dies erfordert allerdings das aufwändige Lesen der Textur in den Hauptspeicher. Glücklicherweise können moderne Graphikkarten, z.B. in Form von Occlusion Queries, ein gewisses Maß an Informationen zur Anwendung zurückliefern. Dies lässt sich, wie in Abschnitt 5.5.2 gezeigt wird, ausnutzen, um die Stoppbedingung durch die Graphik-Hardware auszuwerten.

Weiterhin bietet die Steuerung der Traversierung über den Index der aktuellen Zelle einen Mechanismus, um auf einfache Weise Early Ray Termination zu realisieren, was aufgrund der reduzierten Rasterisierung einen großen Einfluss auf die Performanz hat. Sobald die akkumulierte Opazität einen gewissen Schwellwert überschreitet, wird der Index der aktuellen Zelle auf -1 gesetzt.

Bevor im Folgenden auf die Repäsentation der Daten des Tetraedergitters auf der Graphikkarte eingegangen wird, ist ein Blick auf kompakte Datenstrukturen angebracht, weil der Graphikkartenspeicher den limitierenden Faktor für die Größe der verarbeitbaren Datensätze darstellt.

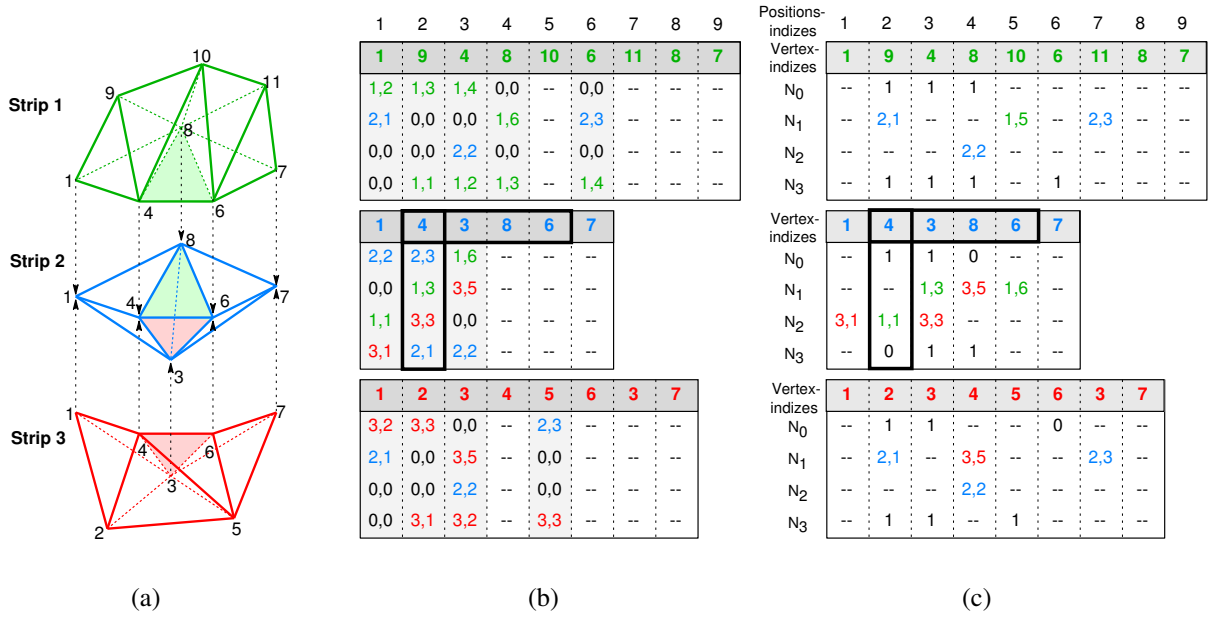
5.2 Tetraederstreifen

Tetraederstreifen (Strips) bieten als Verallgemeinerung von Dreiecksstreifen eine sehr kompakte Repäsentation der Daten eines Tetraedergitters. Die Tetraeder werden dabei als String von Indizes, die in eine Liste von Vertizes verweisen, repräsentiert. Jeweils vier aufeinanderfolgende Indizes eines Strings kodieren einen Tetraeder, so dass eine Liste von n Tetraedern mit lediglich $n + 3$ Indizes repräsentiert wird. Um ein Tetraedergitter so zu speichern, dass es zum Ray Casting verwendet werden kann, muss diese Repräsentation jedoch um Information über die Konnektivität der Tetraeder erweitert werden. Dabei sollte der Mehraufwand minimal sein, um den wesentlichen Vorzug der Tetraederstreifen zu behalten. Wichtig ist darüber hinaus, dass der wahlfreie Zugriff auf einzelne Tetraeder möglich sein muss und der Zugriff auf die vier Nachbartetraeder in konstanter Zeit erfolgen kann. Dies ist bei den klassischen Kompressionsalgorithmen für Dreiecksnetze [Touma und Gotsman 1998, Gumhold und Straßer 1998, Rossignac 1999], für die auch Erweiterungen für Tetraedergitter entwickelt wurden [Gumhold et al. 1999, Szymczak und Rossignac 1999], nicht der Fall. Immer muss vor dem Zeichnen das komplette Gitter entpackt werden. Dieser Abschnitt stellt ein neuartiges Schema für die kompakte Speicherung von Nachbarschaftsinformation für die Tetraeder eines Streifens vor. Es wird zwar im Zusammenhang mit der direkten Volumendarstellung von Tetraedergittern beschrieben, lässt sich aber auch für eine Vielzahl anderer Konnektivitätsinformation erfordernde Vor- und Nachverarbeitungsschritte auf Graphik-Hardware oder in der CPU einsetzen. Anders ausgedrückt ist das vorgestellte Schema universell und erlaubt die Speicherung sowohl von sequentiellen als auch von generalisierten Tetraederstreifen.

Die folgende Beschreibung basiert auf der verbreiteten Konvention, dass die i-te Fläche des Tetraeders dem Vertex v_i ($i \in \{0,1,2,3\}$) gegenüberliegt. Die Einführung erfolgt anhand des Beispiels in Abbildung 5.3(a), welches drei benachbarte Streifen aus insgesamt elf Vertizes zeigt. Die mittlere Tabelle in Abbildung 5.3(b) zeigt die Organisation der Streifen in Form mehrerer Listen, welche die Indizes der Vertizes sowie die jeweiligen Nachbarn enthalten. Insgesamt werden vier unterschiedliche Nachbarschaftslisten benötigt, eine pro Tetraederfläche. Sie sind mit N_0, N_1, N_2 und N_3 bezeichnet. Die Nachbarschaftsinformation befindet sich jeweils unter dem Index des ersten Vertex eines Tetraeders. Für den Tetraeder $\{4,3,8,6\}$, der mit einer schwarzen Umrandung markiert ist, steht sie unterhalb von Vertex 4.

Ein Eintrag in der Nachbarschaftsliste besteht aus zwei Indizes: dem Index des Streifens, in dem der Nachbar liegt, und der Position des ersten Vertex innerhalb des Tetraederstreifens. Aus Gründen der Übersicht sind die Indexpositionen oberhalb der Listen dargestellt. Um ein Beispiel zu geben: Der Nachbar des Tetraeders $\{4,3,8,6\}$ bezüglich der Fläche 0 ist $\{3,8,6,7\}$ und wird als $(2,3)$ kodiert, weil 3 der dritte Vertex in Streifen 2 ist. Der Tetraeder $\{4,3,8,6\}$ selbst hat den Index $(2,2)$. Wichtig ist, dass Streifen und Tetraeder beginnend mit 1 gezählt werden. Das vermeidet die spezielle Behandlung von nicht-existierenden Nachbarn. Sie können einfach mit $(0,0)$ repräsentiert werden.

Um zu beschreiben, wie dieselbe Information kompakter gespeichert werden kann, soll zunächst der Fall sequentieller Streifen betrachtet werden, d.h. solche ohne Vertexreplikationen. Ein Beispiel für einen sequentiellen Streifen ist Streifen 2 in Abbildung 5.3(b). Unterschiedliche Typen



(a) (b) (c)

Abbildung 5.3: Beispiel für generalisierte Tetraederstreifen. Das Tetraedergitter ist in (a) dargestellt, gefolgt vom Layout der Nachbarschaftsinformation (b) und dem verbesserten Layout in (c).

von Nachbarschaftsinformation können unterschieden werden:

Nachbarn im selben Streifen: Abgesehen vom ersten und letzten Tetraeder besitzt jeder Tetraeder wenigstens zwei Nachbarn im selben Tetraederstreifen. Der nachfolgende Tetraeder im Streifen ist der Nachbarn bezüglich der Fläche 0 (N_0) und der vorangegangene Tetraeder der Nachbar bezüglich der Fläche 3 (N_3). Für $\{4,3,8,6\}$ ist beispielsweise der N_0-Nachbar $(2,3)$ und der N_3-Nachbar $(2,1)$. Diese Nachbarn werden als „implizite Nachbarn" bezeichnet, da ihre Konnektivität durch die Reihenfolge innerhalb des Streifens gegeben ist.

Nachbarn, die in einen anderen Streifen verweisen: Der Nachbartetraeder bezüglich der Flächen 1 (N_1) und 2 (N_2), zum Beispiel $(1,3)$ und $(3,3)$, verweisen typischerweise auf einen Tetraeder eines anderen Streifens. Daher müssen sie weiterhin als explizite Indizes gespeichert werden und werden deshalb „explizite Nachbarn" genannt.

Enden des Streifens: Der erste Tetraeder in einem Streifen kann über den N_3 Nachbar mit einem anderen Streifen verbunden sein. In dem behandelten Beispiel trifft dies für $\{1,4,3,8\}$ in Streifen 2 zu, der den N_3 Nachbar $(3,1)$ besitzt. In ähnlicher Weise kann der letzte Tetraeder über N_0 mit einem anderen Streifen verbunden sein (siehe z.B. $(1,6)$ in Tetraeder $\{3,8,6,7\}$ von Streifen 2).

Unter Ausnutzung der impliziten Nachbarn kann die Datenstruktur, die in Abbildung 5.3(b) präsentiert wird, komprimiert werden. Dies ist in Abbildung 5.3(c) dargestellt. Man beachte,

dass hier die Nachbarschaftsinformation um eine Stelle nach rechts verschoben wurde, d.h. sie befindet sich nun unterhalb des zweiten Vertex eines jeden Tetraeders. Die Nachbarschaftsinformation von Tetraeder $\{4,3,8,6\}$ in Streifen 2 befindet sich nun beispielsweise unterhalb von Vertex 3.

Für die meisten Tetraeder müssen nunmehr nur noch die N_1- und N_2-Nachbarn explizit kodiert werden, wohingegen N_0 und N_3 durch zwei Flags ersetzt werden, die lediglich anzeigen, ob der Tetraeder implizite (Wert 1) oder explizite (Wert 0) N_0- und N_3-Nachbarn besitzt. Bei einem Wert von 0 ist es immer noch notwendig, die Nachbarn explizit zu speichern. Glücklicherweise kommt dies bei sequentiellen Streifen nur bei dem ersten und letzten Tetraeder vor. Um diese expliziten N_0- und N_3-Nachbarn ohne zusätzlichen Mehraufwand in der Datenstruktur zu speichern, werden die bisher nicht benutzten N_1- und N_2-Einträge des nächsten bzw. vorherigen Tetraeders verwendet. Nachdem die Nachbarschaftsinformation um eine Stelle nach rechts verschoben wurde, steht nun ein freier Platz am Streifenanfang und zwei freie Plätze am Streifenende zur Verfügung. Somit kann der explizite N_0-Nachbar als N_1-Eintrag in der nächsten Spalte und der N_3-Nachbar als N_2-Eintrag in der vorangegangenen Spalte gespeichert werden. Dies wird mit Hilfe der Pfeile in Abbildung 5.3(c) angedeutet.

Für generalisierte Tetraederstreifen sind Vertexreplikationen erlaubt, um die Fläche, über die der Streifen fortgesetzt wird, zu ändern. Dies ermöglicht die Konstruktion längerer Streifen. Beispielsweise wurde in Streifen 3 in Abbildung 5.3(a) der Vertex 3 repliziert, um $\{3,4,5,6\}$ mit $\{5,6,3,7\}$ fortsetzen zu können. Als Folge ist $\{3,4,5,6\}$ zweimal in demselben Streifen enthalten: als $\{3,4,5,6\}$ und als $\{4,5,6,3\}$. Allerdings wird die Nachbarschaftsinformation nur einmal betrachtet und zwar für den ersten der beiden Tetraeder unterhalb von Vertex 3 (siehe Abbildung 5.3(b)). Die zweite Instanz wird als „virtueller Tetraeder" bezeichnet.

Aufgrund dieser virtuellen Tetraeder sind N_0 und N_3 nunmehr explizit, selbst wenn sie auf Tetraeder im selben Streifen verweisen. Glücklicherweise sind die N_1- und N_2-Einträge des virtuellen Tetraeders unbenutzt. Nimmt man dasselbe Schema wie für die expliziten Nachbarn am Anfang und am Ende des Streifens an, dann werden diese Einträge automatisch für die nun expliziten Nachbarn verwendet. Als Beispiel sei auf den N_3-Nachbarn von Tetraeder $\{5,6,3,7\}$ verwiesen, der als N_2-Eintrag unterhalb von Vertex 5 kodiert wird.

5.3 Streifenerzeugung

Der Speicherbedarf eines Tetraedergitters, welches mit Tetraederstreifen repräsentiert wird, hängt sehr stark von der Qualität des Algorithmus zur Streifenerzeugung ab. Im Kontext von Dreiecken wurde die Streifenerzeugung (*Stripification*) bereits ausgiebig erforscht [Akeley et al. 1990, Evans et al. 1996, Xiang et al. 1999, Speckmann und Snoeyink 1997], und eine Reihe von frei verfügbaren Werkzeugen existiert [Akeley et al. 1990, Beeson und Demer 2002], die aus einem gegebenen Dreiecksnetz eine Repräsentation aus Dreiecksstreifen, gegebenenfalls unter Berücksichtigung der optimalen Ausnutzung der Graphik-Hardware, generieren. In der Regel versuchen die publizierten Ansätze

mittels Heuristiken eine suboptimale Lösung zu finden, da das Problem der Generierung von Dreiecksstreifen aus einer gegebenen Dreiecksliste NP-hart ist.

Trotz dieser umfassenden Literatur zur Streifenerzeugung von polygonalen Netzen scheint es kaum Arbeiten zur Erzeugung von Tetraederstreifen zu geben. Eine Ausnahme stellt [King et al. 2001] dar. Dort wird ein Greedy-Ansatz vorgeschlagen, der verschiedene Heuristiken verwendet, um zu entscheiden, welcher Tetraeder als nächstes in den Streifen eingefügt wird.

Der hier vorgeschlagene Ansatz basiert ebenfalls auf einem Greedy-Algorithmus, der speziell versucht, isolierte Tetraeder zu vermeiden, da sie einen Mehraufwand von drei Indizes erzeugen. Zur Erzeugung sowohl von sequentiellen als auch von generalisierten Tetraederstreifen wird jeweils dieselbe Datenstruktur verwendet. Jeder Tetraeder speichert alle seine Nachbarn zusammen mit einem Flag, das anzeigt, ob der Tetraeder bereits besucht wurde. Die Streifenerzeugung läuft ausgehend von einem Starttetraeder zunächst so lange in eine Richtung, bis kein nachfolgender Tetraeder mehr gefunden wird. Danach wird versucht, den Streifen in der entgegengesetzten Richtung vom Starttetraeder aus fortzusetzen.

Sowohl um den Starttetraeder für einen Streifen zu finden, als auch um den nächsten Tetraeder zu bestimmen, werden unterschiedliche Heuristiken verwendet. Für die Suche nach dem Starttetraeder werden vier Warteschlangen ($queue_i$, $i \in [1,4]$) verwaltet, in welche die Tetraeder entsprechend der Anzahl noch nicht besuchter Nachbarn i eingetragen werden. Startpunkte werden immer aus der nicht-leeren Warteschlange mit dem kleinsten Index entnommen. Diese Strategie hat zu der geringsten Anzahl von isolierten Tetraedern geführt. Weiterhin konnte die Korrelation zwischen den Streifen dadurch erhöht werden, dass nur Nachbarn von bereits besuchten Tetraedern in die Warteschlangen eingetragen wurden. Dies führt im Allgemeinen zu deutlich längeren Streifen, weil die Fragmentierung des Tetraedergitters während der Streifenerzeugung reduziert wird. Nur wenn keine unbesuchten Nachbarn mehr existieren, wird ein beliebiger Tetraeder als neuer Startpunkt ausgewählt.

Um den längsten möglichen Streifen ausgehend von einem definierten Starttetraeder zu finden, werden alle möglichen Vertexpermutationen betrachtet. Im Falle von vier unbesuchten Nachbarn sind dies bis zu 24. Da jedoch für einen Tetraeder mit lediglich einem unbesuchten Nachbarn die Reihenfolge der Vertizes im Streifen frei wählbar ist, können in diesem Fall stattdessen die Permutationen des Nachbarn untersucht werden. Zur weiteren Erhöhung der Streifenlänge kann anstelle von einem einzelnen Startpunkt eine Menge von Startpunkten betrachtet werden. Die Streifenerzeugung wird dann parallel für all diese Startpunkte ausgeführt und der längste Streifen in die Kodierung übernommen.

Um während der Streifenerzeugung zu entscheiden, welcher Tetraeder als nächster gewählt werden soll, bieten sich zwei Heuristiken an:

Sequentielle Streifen: Wähle den nächsten Tetraeder, der den sequentiellen Streifen fortsetzt.

Generalisierte Streifen: Wähle den Tetraeder mit den wenigsten unbesuchten Nachbarn und wähle im Falle von Zweideutigkeiten den Tetraeder, der die wenigsten Vertexreplikationen erzeugt.

Tabelle 5.1: Vergleich der Ergebnisse für die Erzeugung sequentieller (Seq) und generalisierter (Gen) Tetraederstreifen.

Datensatz	# Tetra	# Vert	# Streifen		∅ Länge		Max. Länge		% isoliert		% Kompression	
			Seq	Gen	Seq	Gen	Seq	Gen	Seq	Gen	Seq	Gen
Cube	1.715	512	436	205	3,93	8,36	34	143	6,06	3,32	66,10	59,87
Fighter	70.125	13.832	16.169	7.274	4,34	9,64	281	2.249	7,56	3,15	63,44	59,20
Spx	103.488	37.320	24.009	11.161	4,31	9,27	281	2.235	8,06	3,38	63,60	59,46
Heatsink	121.668	23.576	27.855	12.503	4,37	9,73	198	2.505	7,86	2,79	63,26	56,53
Sphere/Ell	148.955	32.768	34.319	15.201	4,34	9,80	154	1.037	7,95	2,84	63,42	57,58

Alternativ kann auch eine hybride Strategie verwendet werden, die primär den nächsten Tetraeder für eine sequentielle Fortsetzung des Streifens wählt und alternativ die Strategie für generalisierte Streifen verfolgt, falls dies nicht möglich ist. Dies führte in den durchgeführten Tests jedoch ausnahmslos zu schlechteren Ergebnissen als die reine Anwendung der Strategie für generalisierte Streifen. Tabelle 5.1 zeigt Ergebnisse, die bei der Erzeugung von sequentiellen und generalisierten Tetraederstreifen erzielt wurden.

Der vorgestellte Algorithmus erzeugt sequentielle Tetraederstreifen mit einer durchschnittlichen Länge von $4,3$ Tetraedern und weniger als 8% isolierte Tetraeder, woraus eine Kompressionsrate von 63% bis 66% resultiert. Die Kompression ergibt sich dabei aus dem Verhältnis des Speicherbedarfs im Vergleich mit einer Tetraederliste aus t Tetraedern:

$$\frac{3(t+3n)}{8t} = \frac{3}{8} + \frac{9}{8}\frac{n}{t} \tag{5.3}$$

In der Tetraederliste benötigt jeder Tetraeder 8 Indizes für die vier Vertizes und vier Nachbarn. Das in Abschnitt 5.2 beschriebene Schema erfordert demgegenüber für die Speicherung in Form von n sequentiellen Streifen $3(t+3n)$ Indizes; drei Indizes für den Vertex und die Nachbarn je Tetraeder und drei zusätzliche Indizes je Streifen. Theoretisch ergibt sich demnach entsprechend Gleichung (5.3) eine maximale Kompression von $3/8 \approx 37\%$.

Unter Verwendung von generalisierten Tetraederstreifen lässt sich die Anzahl der isolierten Tetraeder mehr als halbieren und die durchschnittliche Streifenlänge nahezu auf $9,4$ verdoppeln, wobei replizierte Vertizes nicht mitgezählt wurden. Werden sie berücksichtigt, ergibt sich lediglich eine moderate Verbesserung der Kompression auf 56% bis 60%.

Beim Vergleich mit den in [King et al. 2001] präsentierten Zahlen muss beachtet werden, dass King generalisierte Tetraederstreifen nicht mit Hilfe von Vertexreplikationen realisiert, sondern über zwei Bit Kontrollinformation, die für jeden neuen Vertex angeben, welcher Vertex des vorangegangenen Tetraeders ersetzt werden soll. Diese Zusatzinformation fehlt in den Statistiken völlig. Dennoch bleiben die hier präsentierten Ergebnisse etwas hinter den Zahlen von King et al. zurück, was weiteren Raum für Verbesserungen lässt.

5.4 Gitterrepräsentation

Der Schlüssel, um Hardware-basiertes Ray Casting direkt auf der Graphikkarte zu implementieren, liegt in der Möglichkeit, die kompletten Daten des Tetraedergitters in Texturen auf der Graphikkarte zu speichern und somit einen wahlfreien Zugriff auf die Gitterdaten zu erlauben. Seit Aufkommen von DirectX9-kompatibler Hardware, wie der ATI Radeon 9700 oder der nVidia GeForceFX, ist dies mit 32 Bit Gleitkommagenauigkeit ohne Präzisionsverlust möglich. Ziel muss es dabei sein, eine kompakte Form der Speicherung zu erreichen, da die limitierte Speichermenge auf der Graphikkarte die maximale Größe für verarbeitbare Datensätze bestimmt.

Je weniger Informationen die Datenstruktur jedoch enthält, umso höher wird der Aufwand, um die für die Strahlberechnung erforderlichen Parameter zu ermitteln. Im Extremfall kann dies sogar eine Implementierung auf bestimmter Hardware verhindern. Immer hat es jedoch Einfluss auf die Performanz, so dass gegebenenfalls ein guter Kompromiss zwischen Speicherbedarf und Geschwindigkeit gefunden werden muss. Im Folgenden werden zwei unterschiedliche Datenstrukturen vorgestellt, sowie ihre Einsetzbarkeit auf unterschiedlicher Graphik-Hardware diskutiert.

5.4.1 Tetraederlisten

Die erste vorgestellte Datenrepräsentation wurde für die ATI Radeon 9700 entworfen, welche die erste verfügbare DirectX9-Hardware war. Verglichen mit späteren Chips stellt die Anzahl der verfügbaren Texturoperationen (32) und arithmetisch-logischen Operationen (64) noch eine signifikante Einschränkung dar. Damit das Ray Casting dennoch implementiert werden konnte, musste ein relativ einfacher Zugriff auf die Daten des Tetraedergitters ermöglicht werden. Daher werden hier alle Größen, die in Abschnitt 5.1.2 vorgestellt wurden, explizit gespeichert. Die Repräsentation entspricht der einer Tetraederliste. Einen Überblick über die erforderlichen Texturen bietet Tabelle 5.2.

Die Zellindizes werden dabei in zwei Texturkoordinaten bzw. zwei Farbkomponenten zerlegt, da ihr Bereich leicht den Datenbereich für eine einzige Texturkoordinate sprengt. Der untere Index

Tabelle 5.2: Texturen zur Speicherung des Gitters in Form von Tetraederlisten.

Tetraederlisten	Texturkoordinaten			Texturdaten			
	u	v	w	r	g	b	α
Vertizes	t		i	$\mathbf{v}_{t,i}$			—
Flächennormalen	t		i	$\mathbf{n}_{t,i}$			$f_{t,i}$
Nachbarschaft	t		i	$a_{t,i}$		—	—
Skalardaten	t		—	$\mathbf{g}_t$			$\hat{g}_t$

t_v ist dabei der Rest der Division des Zellindex durch die Gröβe der verwendeten Textur in der v-Dimension. Der obere Index t_u ergibt sich als ganzzahliger Anteil der Division. Die Speicherung des Tetraedergitters erfolgt in jeweils einer dreidimensionalen Textur für die Vertizes, die Flächennormalen und die Nachbarschaftsinformation und einer zweidimensionalen Textur für die Rekonstruktion der Skalardaten (siehe Abschnitt 5.1.3). Über die Nachbarschaftsinformationen kann auch eine Kennzeichnung imaginärer Zellen erfolgen. Das Vorzeichen des α-Kanals aller Flächennormalen gibt an, ob eine Zelle imaginär ist oder nicht.

Alle Texturen liegen im Gleitkommaformat vor. Der Zugriff auf die Texturen erfolgt über den Zellindex, der in der ersten und zweiten Texturkoordinate spezifiziert wird. Die w-Komponente der dreidimensionalen Texturen bezeichnet den Index des Vertex, der Fläche oder des Nachbarn. Somit ergibt sich, dass die Größe der dritten Dimension aller Texturen immer vier ist, da jeder Tetraeder vier Vertizes, Normalen und Nachbarn besitzt. Die Größe in u- und v-Dimension kann im Rahmen der maximalen Texturgröße beliebig an die Größe des Datensatzes angepasst werden.

Zur Beschleunigung des Datennachschlags und damit des gesamten Algorithmus wird in [Bernardon et al. 2004] vorgeschlagen, die dreidimensionalen Texturen durch zweidimensionale Texturen zu ersetzen, um von der im Allgemeinen besseren Cache-Kohärenz für zweidimensionale Texturen zu profitieren. Dabei werden die vier einzelnen Schichten der dreidimensionalen Textur nicht übereinander sondern stattdessen nebeneinander entlang der v-Achse angeordnet. Die Vertizes $\mathbf{v}_{t,0}$, $\mathbf{v}_{t,1}$ und $\mathbf{v}_{t,2}$ ergeben sich somit durch Nachschlag an der Position (t_u, t_v), $(t_u, t_v + dv)$ und $(t_u, t_v + 2dv)$, wenn dv die ursprüngliche Gröβe der Textur in v-Dimension ist. Darüber hinaus lässt sich $\mathbf{v}_{t,3}$ im α-Kanal der übrigen drei Vertizes speichern, wodurch die Anzahl der Texturnachschläge reduziert wird. Analoges gilt für die Speicherung von Normalen und Nachbarschaftsdaten, wodurch es gelingt, lediglich drei zweidimensionale Texturen für die Speicherung aller Daten zu verwenden.

Besser ist unter dem Gesichtspunkt der Cache-Kohärenz jedoch, die Vertizes, Normalen und Nachbarschaftsdaten eines Tetraeders sequentiell in der Textur zu speichern, also beispielsweise die Vertizes $\mathbf{v}_{t,0}$, $\mathbf{v}_{t,1}$ und $\mathbf{v}_{t,2}$ an den Positionen $(t_u, 3t_v + 0)$, $(t_u, 3t_v + 1)$ und $(t_u, 3t_v + 2)$ abzulegen. Auch kann eine Verteilung der Schichten der dreidimensionalen Textur auf mehrere Textureinheiten sinnvoll sein, wenn diese über einen eigenen Textur-Cache verfügen oder einen echt parallelen Zugriff erlauben.

5.4.2 Tetraederstreifen

Die Implementierung von Hardware-basiertem Ray Casting mittels der in Abschnitt 5.2 vorgestellten Datenstruktur für Tetraederstreifen erzeugt höhere Anforderungen an die Graphik-Hardware. Im Besonderen sind zusätzliche Fragmentoperationen nötig, um die erforderlichen Gitterdaten nachzuschlagen und um Größen, die nicht direkt gespeichert sind, zu berechnen. Dies ist mit den 64 Instruktionen einer ATI Radeon 9700 nicht zu erreichen ohne den Algorithmus in mehrere Zeichendurchgänge je Traversierungsschritt zu unterteilen. Die im Folgenden vorgestellte Datenrepräsentation basierend auf Tetraederstreifen erfordert daher mindestens die Fähigkeiten einer nVidia GeForceFX, die mit ihren bis zu 1024 Instruktionen langen Fragment-

programmen die erste Graphik-Hardware ist, auf der eine Realisierung möglich war. Auf der aktuellen Graphikkartengeneration, vertreten durch die Nachfolgeserien nVidia GeForce6 sowie ATI Radeon X800 stellt die Implementierung kein Problem mehr dar.

Tabelle 5.3 gibt einen Überblick über die erforderlichen Texturen. Es werden mehrere zweidimensionale Gleitkommatexturen für die Speicherung des Tetraedergitters verwendet. Die Größe der Texturen kann seit der Aufhebung der Beschränkung der Texturgröße auf Zweierpotenzen mittels der OpenGL-Erweiterung `NV_float_buffer` bzw. `ARB_texture_non_power_of_two` an die Anforderungen des Datensatzes angepasst werden, um möglichst effektiven Gebrauch von dem zur Verfügung stehenden Texturspeicher zu machen. Dabei sind sogar unterschiedliche Größen für die Texturen, welche die Vertizes, Normalen, Konnektivität etc. speichern, möglich.

Eine minimale Menge von Texturen für den Tetraederstreifen beinhaltet eine RGBA-Textur für die Liste der Vertizes sowie eine RGB-Textur für die als Tetraederstreifen kodierte Konnektivität. Die Vertexliste speichert dabei jeweils die Koordinaten $\mathbf{v}_k$ der Vertizes sowie die Skalarwerte s_k. Die Anordnung der Vertizes in der Textur erfolgt sequentiell entsprechend ihrem Index k.

Die RGB-Konnektivitätstextur speichert eins-zu-eins die kodierten Strings der Tetraederstreifen, so wie sie in Abschnitt 5.2 beschrieben wurden. Jeder Texel in der Textur entspricht dabei einem Tetraeder. Auch die drei zusätzlichen Vertizes je Streifen werden mit einem eigenen Index repräsentiert, genauso wie virtuelle Tetraeder in generalisierten Streifen. Ein Eintrag in dieser Textur enthält unter anderem $v_{t,0}$, den Index des ersten Vertex im korrespondierenden Tetraeder. Er wird als zwei-komponentige 16 Bit Texturkoordinate repräsentiert und verweist auf die Textur der Vertexliste. Weiterhin gibt es zwei Einträge $N_{t,1}$ und $N_{t,2}$, ebenfalls zwei-komponentige 16 Bit Texturkoordinaten, die auf die N_1- und N_2-Nachbarn des Tetraeders t in der Tetraederstreifentextur verweisen. Die Information, ob ein Tetraeder implizite N_0- und N_3-Nachbarn besitzt, wird im höchstwertigen Bit (Most Significant Bit) der u- und v-Komponente von $N_{t,1}$ gespeichert. Auf diese Weise lässt sie sich leicht mit Hilfe einer einzigen Fragmentinstruktion extrahieren und mit einer weiteren auch entfernen. Durch die Verwendung dieses einen Bits reduziert sich die maximal adressierbare Textur auf $2^{15} \times 2^{15}$, was allerdings praktisch keine Rolle spielt, da dies immer noch für die Speicherung von einer Milliarde Tetraeder ausreicht und außerdem physikalische

Tabelle 5.3: Texturkodierte Speicherung der Tetraederstreifen.

Tetraederstreifen	Texturkoordinaten		Texturdaten			
	u	v	r	g	b	α
Tetraederstreifen	t		$v_{t,0}$	$N_{t,1}$	$N_{t,2}$	—
Vertizes	k			$\mathbf{v}_k$		s_k
Normalenindizes (opt.)	t		$n_{t,0}$	$n_{t,1}$	$n_{t,2}$	$n_{t,3}$
Flächennormalen (opt.)	j			$\mathbf{n}_j$		o_j
Skalardaten (opt.)	t			$\mathbf{g}_t$		$\hat{g}_t$

Beschränkungen der maximalen Texturgröße deutlich darunter liegen.

Um mit einem vorzeichenbehafteten Offset der u-Texturkoordinate auf vorangegangene und nachfolgende Einträge eines Streifens zugreifen zu können, ist es erforderlich, dass die Streifen entlang der Zeilen der Textur ausgerichtet sind. Umbrüche innerhalb des Streifens sind dabei zu vermeiden. Um den in der Textur verfügbaren Speicherplatz optimal auszunutzen, kann ein einfacher aber effektiver Greedy-Algorithmus zum Layout der Streifen verwendet werden: Alle Streifen werden dabei nach absteigender Länge sortiert und der jeweils längste für die Platzierung innerhalb der Textur ausgewählt. Er wird in die Texturzeile eingetragen, die gerade genügend freie Einträge enthält. Zum schnellen Auffinden dieser Zeile kann eine Freispeicherliste verwendet werden, welche einen Eintrag je Texturzeile enthält und diese nach aufsteigender Anzahl von freien Plätzen sortiert vorhält.

Mit diesen beiden Texturen sind alle Informationen kodiert, die ein Fragmentprogramm benötigt, um die Strahlberechnung durchführen zu können. Die übrigen in Abschnitt 5.1.3 verwendeten Größen wie Flächennormalen lassen sich aus den Koordinaten der Vertizes ableiten. Der Skalarwert am Austrittspunkt ergibt sich durch baryzentrische Interpolation aus den Skalarwerten an den Vertizes, wobei sich die Interpolationsgewichte aus dem normalisierten Abstand des Austrittspunkts zu den jeweils gegenüberliegenden Tetraederflächen ergibt:

$$s(\mathbf{x}) = \sum_{i=0}^{3} w_i s_i \qquad \text{mit} \qquad w_i = \frac{\mathbf{n}_i \cdot (\mathbf{x} - \mathbf{v}_{3-i})}{\mathbf{n}_i \cdot (\mathbf{v}_i - \mathbf{v}_{3-i})} \tag{5.4}$$

Da jedoch lange Fragmentprogramme in geringeren Bildwiederholraten resultieren, kann es sinnvoll sein, Speicherbedarf gegen Performanz zu tauschen. Mit am aufwändigsten ist die Rekonstruktion der Normalen der Tetraeder, die sich daher als weitere optional gespeicherte Größen anbieten. Sinnvollerweise erfolgt die Speicherung dabei in zwei Texturen, einer Indexliste mit den Referenzen $n_{t,0\ldots3}$ auf die vier Tetraedernormalen und einer Textur mit den Normalenvektoren $\mathbf{n}_j$ und den Ebenen-Offsets o_j. Dies erlaubt die Wiederverwendung der Normalen für benachbarte Tetraeder. Im höchstwertigen Bit der u-Komponente der Indizes kann dabei gespeichert werden, ob die Richtung des Normalenvektors umgedreht werden muss, um eine nach außen gerichtete Orientierung zu erreichen. Zu Vereinfachung der Skalarwertinterpolation (siehe Abschnitt 5.1.3) kann darüber hinaus eine Textur mit dem Gradienten $\mathbf{g}_t$ des Tetraeders und dem Skalarwert $\hat{g}$ an einer beliebigen Position verwendet werden. Der gespeicherte Gradient lässt sich zusätzlich zur Beleuchtungsberechnung verwenden.

Der eigentliche Vorteil der vorgeschlagenen Repräsentation aus Tetraederstreifen liegt in der Speicherersparnis. Wie in [Weiler et al. 2004b] gezeigt wurde, lässt sich der Speicherbedarf von 160 Byte je Tetraeder für die im vorangegangenen Abschnitt beschriebene Listenrepräsentation auf etwa 117 Bit je Tetraeder reduzieren, wenn eine Streifenrepräsentation nur aus Konnektivitätstextur und der Textur der Vertexpositionen gewählt wird. Durch diese Datenreduktion beinahe um den Faktor 11 wächst die Größe für Tetraedergitter, die mit dem Hardware-basierten Ray Casting bearbeitet werden können, signifikant auf 17,5 Millionen Tetraeder für Graphikkarten mit 256 MB Speicher. Mit bald erscheinenden Graphikkarten mit 512 MB Speicher wächst die Kapazität weiter auf 35,8 Millionen Tetraeder, was den Ansatz für Datensätze aus der Praxis praktikabel macht.

5.5 Implementierung

In der Beschreibung des Algorithmus zum parallelen Ray Casting auf programmierbarer Graphik-Hardware in Abschnitt 5.1 wurde versucht, weitgehend auf Hardware-spezifische Details zu verzichten. Dieser Abschnitt beschäftigt sich mit zwei Implementierung auf einer ATI Radeon 9700 unter Verwendung von DirectX9 mit Pixel Shader 2.0 sowie einer nVidia GeForceFX unter Verwendung von OpenGL und der Shader-Hochsprache Cg [nVidia 2002]. Dabei werden Gemeinsamkeiten und Besonderheiten aufgezeigt. In beiden Fällen ist eine direkte Implementierung der in Abschnitt 5.1 beschriebenen Operationen möglich, weshalb hier kein Shader-Programm präsentiert wird.

5.5.1 Datenstrukturen

In beiden Implementierungen ist die Traversierung der Sehstrahlen als ein Mehrpass-Verfahren implementiert, bei dem in jedem Zeichendurchgang ein Rechteck, das die Projektion des Tetraedergitters auf die Bildebene überdeckt, mit einem geeigneten Fragmentprogramm beziehungsweise Pixel Shader gezeichnet wird. Nach jedem Zeichendurchgang werden Zwischeninformationen in eine Traversierungsdatenstruktur geschrieben. Diese besteht, abhängig davon, ob die verwendete Graphik-Hardware mehrere Render Targets anbietet, aus einer oder mehreren zweidimensionalen Texturen. Moderne Graphik-Hardware unterstützt das direkte Schreiben in Texturen beispielsweise durch das Zeichnen in Hardware-beschleunigte p-Buffer, die mittels der OpenGL-Erweiterung `NV_render_texture_rectangle` direkt als Textur dienen können.

Die Texturen werden mit der Rasterposition des korrespondierenden Pixels adressiert. Letztere steht nur bei der OpenGL-Implementierung automatisch zur Verfügung. Unter DirectX9 ist eine explizite Spezifikation der Rasterposition an den Vertizes des gezeichneten Rechtecks erforderlich, die dann für jedes Fragment interpoliert wird.

Tabelle 5.4 zeigt die Realisierung mit mehreren Texturen, wie sie von der ATI Radeon 9700 unterstützt wird. Die erste Traversierungstextur speichert für jeden Sehstrahl die zuletzt betretene Zelle einschließlich der Eintrittsfläche. Die zweite Textur spezifiziert die Position und den Skalarwert am letzten Schnittpunkt des Strahls mit einer der Flächen der Zelle. Schließlich ist auf den untersuchten Hardware-Architekturen eine weitere Textur für die Akkumulation der Farb-

Tabelle 5.4: Traversierungstexturen für mehrere Render Targets.

Traversierungsdaten	Texturkoordinaten		Texturdaten			
	u	v	r	g	b	α
Aktuelle Zelle	Rasterposition		t		j	—
Schnittpunkt	Rasterposition		—	—	λ	$s(\mathbf{e}+\lambda\mathbf{r})$
Farbe, Opazität	Rasterposition		r	g	b	α

Tabelle 5.5: Traversierungstexturen für ein einzelnes Render Target.

Traversierungsdaten	Farbkanal	Texturdaten (Bits)			
		0-7	8-15	16-23	24-31
Aktuelle Zelle	R	t			
Schnittpunkt	G	λ		$s(\mathbf{e}+\lambda\mathbf{r})$	
Akkumulierte Farbe	B	r		g	
Akk. Farbe/Opazität	A	b		α	

und Opazitätswerte erforderlich. Anders als in Abschnitt 5.1 angedeutet, kann dies nicht im Framebuffer erfolgen, weil Framebuffer mit Blending-Funktionalität und 32 Bit Gleitkommagenauigkeit momentan noch nicht unterstützt werden und die bisher übliche 8 Bit Genauigkeit, insbesondere aufgrund der oftmals sehr kleinen Strahlbeiträge, zu Blending-Artefakten führen würde. Eine nähere Betrachtung zum Einfluss der Darstellungsgenauigkeit auf die Bildqualität im Kontext von texturbasierter Volumenvisualisierung findet sich in [Bitter et al. 2004].

Für die GeForceFX-Implementierung muss berücksichtigt werden, dass diese Graphikkarte lediglich ein einziges Render Target bestehend aus vier 32 Bit Komponenten im Gleitkommaformat unterstützt. Ohne den Index der Eintrittsfläche, der bei der GeForceFX-Implementierung zugunsten eines Schnitttests mit allen vier Flächen des Tetraeders enfällt, müssen jedoch insgesamt sieben Parameter gespeichert werden. Möglich wird dies, wie Tabelle 5.5 zeigt, unter Verwendung des 16 Bit Gleitkommaformats, welches durch die OpenGL-Erweiterung `NV_fragment_program` zur Verfügung gestellt wird. Sie stellt auch Pack- und Entpack-Operationen bereit, um zwei 16 Bit Floats oder Unsigned Shorts in einem einzelnen 32 Bit Float zu speichern. Dieselben Operationen können auch zum Auspacken der Nachbarschaftsinformation aus der Gittertextur (siehe Abschnitt 5.4.2) ausgenutzt werden. Wichtig ist dabei jedoch, zu beachten, dass Graphik-Hardware und CPU unterschiedliche Byte Order verwenden. Dies muss bei der Zusammenstellung der Textur berücksichtigt werden.

In die vorgestellten Texturen werden nach jedem Zeichendurchgang die Zwischenergebnisse der Traversierung geschrieben, so dass sie als Eingabe für nachfolgende Traversierungsschritte dienen können. Im Allgemeinen erfordert dies die Verwendung von zwei Sätzen von Texturen, da aktuelle Graphikkarten offiziell gleichzeitiges Lesen und Schreiben derselben Textur nicht unterstützen (vergleiche die Spezifikation der OpenGL-Erweiterung `WGL_draw_buffer` [OpenGL ARB 2004]). Ein Textursatz wird dabei jeweils zum Lesen, ein weiterer zum Schreiben verwendet. Die Rolle der Texturen wechselt nach jedem Zeichendurchgang hin und her, weshalb diese Technik auch den Namen Ping-Pong-Rendering trägt. Als undokumentierte Funktionalität unterstützt die GeForceFX das gleichzeitige Lesen und Schreiben derselben Textur dennoch, zumindest sofern der lesende Zugriff auf die Textur nicht wahlfrei sondern lediglich auf den zur aktuellen Rasterposition korrespondierenden Texel erfolgt. Dadurch lässt sich der Mehraufwand der permanenten Texturbindungen vermeiden.

5.5.2 Strahlterminierung

Die Terminierung der Strahltraversierung erfolgt auf zwei Ebenen. Erstens muss für jedes Fragment erkannt werden, ob der korrespondierende Sehstrahl das Volumen bereits verlassen hat. Idealerweise sollten in diesem Fall keine weiteren Fragmentoperationen mehr auf das zugehörige Pixel verwendet werden. Zweitens muss ermittelt werden, ob alle Sehstrahlen das Volumen verlassen haben und somit die Traversierung beendet werden kann. Die erste Aufgabe kann mit Hilfe des vorgelagerten Tiefentests (Early Depth Test), den moderne Graphik-Hardware wie die ATI Radeon 9700 [Riguer 2002] anbieten, erreicht werden. Dieser Test erfolgt vor dem Shading des Fragments, somit wird die Bearbeitung des Fragments beendet, noch bevor irgendein Pixel Shader aufgerufen wird. Entsprechend sollte diese Lösung viel effizienter sein als die Verwendung der „Kill"-Operation im Pixel Shader.

Die Erkennung von vollständig bearbeiteten Sehstrahlen mit Hilfe des vorgelagerten Tiefentests funktioniert, indem im Pixel Shader anhand der Nachbarschaftsinformation entschieden wird, ob eine nachfolgende Zelle existiert. Ist dies nicht der Fall, so wird der Tiefenwert des Fragments auf z_{near} gesetzt und damit weitere Aktualisierungen des korrespondierenden Pixels durch den Tiefentest verhindert. Dies kann allerdings nicht im normalen Traversierungs-Shader erfolgen, weil der vorgelagerte Tiefentest für Shader, die den Tiefenwert eines Fragments verändern, deaktiviert wird [Riguer 2002]. Die Lösung dafür besteht in zusätzlichen Zeichendurchgängen zur Aktualisierung der z-Werte, welche nur den Tiefenpuffer aktualisieren. Da sie einen zusätzlichen Mehraufwand erzeugen, werden sie nur gelegentlich zwischen die Traversierungs-Zeichendurchgänge eingestreut.

Um zu entscheiden, wann alle Sehstrahlen fertig bearbeitet sind, kann bei modernen Graphikkarten eine sogenannte *Occlusion Query* ausgenutzt werden (`NV_occlusion_query` bzw. `D3DQUERYTYPE_OCCLUSION`). Wird eine Occlusion Query mit einem Zeichendurchgang initiiert, so liefert sie die Anzahl der Fragmente zurück, die den Tiefentest passiert haben. Diese entspricht der Anzahl der noch nicht vollständig bearbeiteten Sehstrahlen. Sobald eine Anfrage den Wert Null zurückliefert, kann die Traversierung beendet werden. Aus Gründen der Effizienz erfolgt das Absetzen dieser Abfragen wiederum nicht mit jedem Zeichendurchgang. Dies führt zusammen mit der Tatsache, dass die Ergebnisse aus einer Occlusion Query asynchron mit leichter Verzögerung geliefert werden, zu einem Mehraufwand durch zusätzliche Zeichendurchgänge. Im Vergleich zur Verzögerung, die durch das Warten auf das Ergebnis der jeweiligen Anfrage entsteht, ist dieser Effekt jedoch vernachlässigbar.

5.5.3 Nicht-konvexe Tetraedergitter

Wie in Abbildung 5.4 veranschaulicht wird, liefert die Strahlpropagierung von Zelle zu Zelle nur für konvexe Tetraedergitter korrekte Ergebnisse. In nicht-konvexen Tetraedergittern können Sehstrahlen, die das Tetraedergitter bereits verlassen haben, wiedereintreten. In Abschnitt 5.1.4 wurde dieses Problem durch einen Konvexifizierungsprozess gelöst, der den Raum zwischen Gitter und seiner konvexen Hülle mit virtuellen Tetraedern auffüllt. Diese erfordern eine Sonder-

behandlung im Pixel Shader. Darüber hinaus stellt die Berechnung der Konvexifizierung einen aufwändigen Vorverarbeitungsschritt dar und kann sehr viele zusätzliche Tetraeder erzeugen.

Daher wird hier eine Lösung zur intrinsischen Unterstützung nicht-konvexer Tetraedergitter vorgeschlagen, die auf einer ähnlichen Technik wie Depth Peeling [Everitt 2001] beruht und gänzlich ohne Vorverarbeitungsschritt auskommt. Eine vergleichbare Lösung, die parallel entwickelt wurde, findet sich in [Bernardon et al. 2004]. Die Grundidee basiert auf einer Traversierung in mehren Zyklen. In jedem Zyklus erfolgt eine vollständige Strahltraversierung, die so lange Traversierungsschritte durchführt, bis alle Strahlen, wie in Abschnitt 5.5.2 beschrieben, terminiert wurden. Im ersten Zyklus werden alle Sehstrahlen traversiert, die am ersten Schnittpunkt mit der Umrandung des Gitters starten. Sie entsprechen den roten Strahlen in Abbildung 5.4. Weitere Zyklen setzen die Traversierung am zweiten (blau), dritten (grün) usw. Schnittpunkt des Sehstrahls mit der Umrandung des Tetraedergitters erneut auf. Die Verwendung der immer gleichen Traversierungstexturen für alle nachfolgenden Traversierungszyklen sorgt gleichsam automatisch dafür, dass die akkumulierte Farbe und Opazität aus den vorangegangenen Zyklen korrekt mit den neuen Zellen überlagert wird.

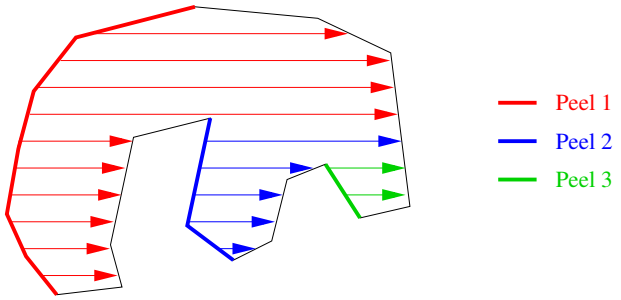


Abbildung 5.4: Ohne weitere Vorkehrungen werden die Sehstrahlen beim Hardware-basierten Ray Casting nur bis zum ersten Verlassen des Tetraedergitters traversiert, was für nicht-konvexe Tetraedergitter problematisch ist. Die Extraktion der zweiten, dritten usw. Ebene von Front Faces mittels Depth Peeling erlaubt die korrekte Verarbeitung von wiedereintretenden Sehstrahlen.

Für das Wiederaufsetzen der Strahltraversierung ist die Extraktion der zweiten, dritten, usw. Ebene von Randflächen des Tetraedergitters erforderlich. Dies wird erreicht, indem ein zweiseitiger Tiefentest mit Hilfe eines Pixel Shaders realisiert wird. Alle Grenzflächen des Tetraedergitters werden dabei mit aktiviertem Back Face Culling gezeichnet, so dass nur die Dreiecke mit Orientierung zum Betrachter (Front Faces) extrahiert werden. Durch Aktivierung des Standard-Tiefentests `GL_LESS` kann nur jeweils das vorderste Dreieck passieren. Zusätzlich vergleicht der Pixel Shader den Tiefenwert aller Fragmente mit dem Tiefenwert der vorherigen Ebene von Randflächen, die in einer Textur gespeichert sind. Fragmente mit einem Tiefenwert kleiner oder gleich dem Tiefenwert in der Textur werden verworfen, was gleichsam die vorangegangene Ebene von Randflächen „abschält" (englisch: *to peel*).

Um den vorgestellten Ray Caster um diese Technik zu erweitern, ist lediglich eine Textur mit den Tiefenwerten erforderlich, die dem Pixel Shader zur Berechnung der Strahleintrittspunkte zur Verfügung gestellt wird. Sie muss nach jedem Zeichendurchgang, der die ersten Strahlein-

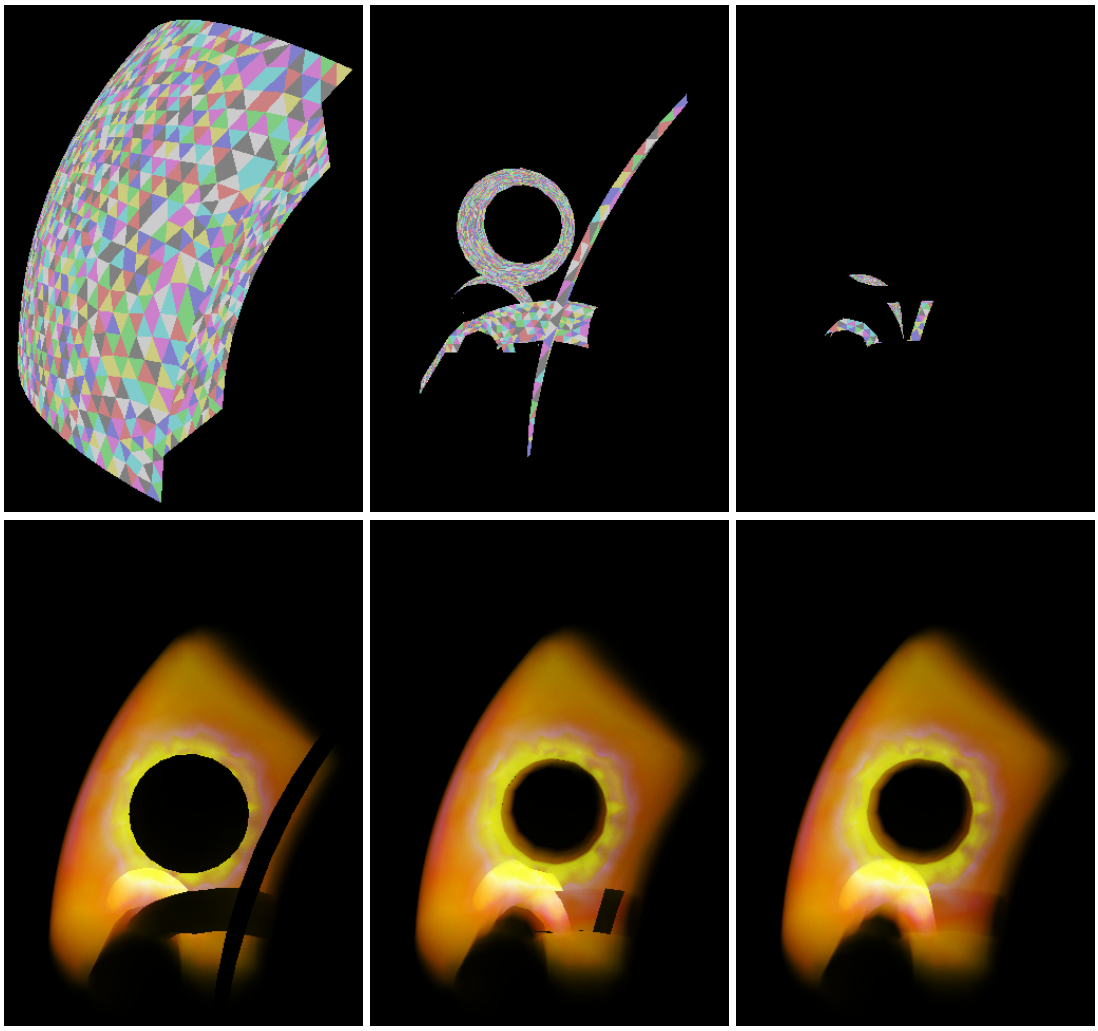


Abbildung 5.5: Volumendarstellung des Super Phoenix (Spx) Datensatzes unter Verwendung von Depth Peeling.

trittspunkte mit dem Gitter ermittelt, aktualisiert werden, was durch das Kopieren der Tiefenwerte direkt auf der Graphikkarte effizient erfolgen kann. Außerdem muss der Pixel Shader um den zusätzlichen Tiefentest erweitert werden. Ein Beispiel für einen nicht-konvexen Datensatz, der mit dem erweiterten Ray Caster gezeichnet wird, ist in Abbildung 5.5 dargestellt.

5.6 Ergebnisse

Für die Evaluierung der beiden Implementierungen werden hier die erzielten Ergebnisse auf zwei Testsystemen präsentiert: ein PC mit Athlon XP 1900+ Prozessor und 512 MB RAM, ausgestattet mit einer ATI Radeon 9700 Pro mit 128 MB Graphikspeicher und ein 2,8 GHz Intel Pentium 4

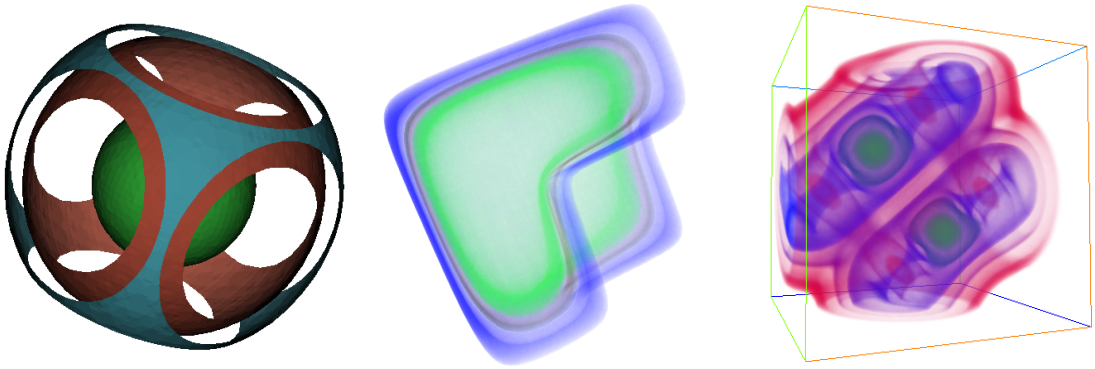


Abbildung 5.6: Alle Bilder zeigen Tetraedergitter, die mit dem Hardware-basierten Ray Caster gezeichnet wurden. Das linke Bild zeigt mehrere beleuchtete Isoflächen, die beiden übrigen Volumendarstellungen basieren auf dem Density-Emitter-Model. Für das rechte Bild wurde der Datensatz darüber hinaus in Form von Tetraederstreifen gespeichert, die von der Graphik-Hardware dekodiert werden.

System mit 2 GB Speicher und einer nVidia GeForce 6800 GT mit 350 MHz GPU-Takt und 256 MB Graphikspeicher, der mit 1000 MHz getaktet ist. Nachdem der hier vorgestellte Ansatz nahezu alle Berechnungen auf der Graphikkarte durchführt, reflektieren die präsentierten Zahlen im Wesentlichen den Performanz-Unterschied der beiden Graphikkarten; der Einfluss der CPU kann dabei als vernachlässigbar betrachtet werden.

Die Implementierung für die ATI Radeon basiert auf DirectX9 und verwendet Tetraederlisten sowie imaginäre Zellen zur Behandlung nicht-konvexer Tetraedergitter, während die Implementierung für die nVidia GeForce 6800 auf OpenGL basiert, die Tetraeder in Form von Tetraederstreifen auf der Graphikkarte speichert und im Falle von nicht-konvexen Tetraedergittern Depth Peeling einsetzt.

Alle hier präsentierten Messwerte wurden mit Hilfe einer Animation ermittelt, bei welcher der Datensatz gleichzeitig mit jeweils unterschiedlicher Winkelgeschwindigkeit um die lokale x-Achse und y-Achse rotiert wird. Soweit nicht anders erwähnt, erfolgte die Messung auf einem Viewport der Größe 512×512 Pixel. Einige Beispiele für erzielte Visualisierungen sind in Abbildung 5.6 dargestellt. In allen getesteten Fällen wurden interaktive Bildwiederholraten von mehreren Frames pro Sekunde erreicht.

5.6.1 DirectX-Implementierung

Tabelle 5.6 zeigt einige exemplarische Performanz-Werte für die ATI Radeon-Implementierung. Es handelt sich dabei um Durchschnittswerte gemessen über einen kompletten Zyklus der Animation. Die Anmerkungen in Klammern sehen dabei für das angewendete optische Modell: (Volumen) für das volle Density-Emitter Modell sowie (Iso) für beleuchtete Isoflächendarstellung. Wie erwartet und bereits in Kapitel 4 im Zusammenhang mit der Zellprojektion beobachtet, hat

die Wahl das optische Modells kaum Einfluss auf die erreichte Performanz. Lediglich das komplette Density-Emitter Modell besitzt aufgrund des erforderlichen Nachschlags in einer dreidimensionalen Textur statt einer zweidimensionaler Textur, wie für alle anderen in Abschnitt 4.4 vorgestellten Modelle, einen leichten Nachteil bezogen auf den Rasterisierungsaufwand. Einen stärkeren Einfluss besitzt die gewählte Transferfunktion, da der vorgestellte Ray Caster Early Ray Termination implementiert und somit die Traversierung stoppt, sobald die akkumulierte Opazität ausreichend hoch ist.

Die jeweilige Bildwiederholrate hängt fast direkt von der Anzahl der Shader-Aufrufe für die Strahltraversierung ab. Diese wird beeinflusst von der Anzahl an Pixeln, die von der Projektion des Volumens überdeckt werden, und von der Anzahl der Traversierungsschritte, die für jedes Pixel erforderlich sind. Letzteres hängt vom Datensatz selbst und den Parametern der Visualisierung ab, welche sich auf die Effektivität der Early Ray Termination auswirken. Durchschnittlich war die Implementierung für die ATI Radeon 9700 in der Lage, etwa 15 Millionen Traversierungs-Shader-Aufrufe pro Sekunde abzuarbeiten.

Der Effekt der Early Ray Termination wird in Abbildung 5.7 verdeutlicht. Gezeigt wird eine Isoflächendarstellung des Heatsink-Datensatzes zusammen mit einer Auflistung der Anzahl der Fragmente, die in jedem Zeichendurchgang bearbeitet werden mussten. Ohne Early Ray Termination waren für diese spezielle Wahl der Darstellungsparameter 156 Zeichendurchgänge bis zum endgültigen Bild erforderlich (grüne Linie). Dies stellt bereits eine deutliche Verbesserung zur Brute-Force-Verarbeitung aller Fragmente dar, bei der die Berechnung auf Pixeln, die das Volumen bereits verlassen haben, nicht durch den vorgelagerten Tiefentests unterbunden wird (rote Linie). Die Einsparung beträgt nahezu 50%. Durch Aktivierung der Early Ray Termination kann die Last für die Fragmenteinheit der Graphikkarte nochmals um einen Faktor 2 reduziert werden und die Bearbeitung kann bereits nach 96 Zeichendurchgängen enden (blaue Linie).

Leider führten technische Limitierungen bei der DirectX-Implementierung zu einem unnötig großen Speicherplatzbedarf von 2×48 Byte/Pixel und 160 Byte/Tetraeder. Allein die Limitierungen der Testplattform, die mehrere Ausgabe-Targets lediglich mit jeweils derselben Bittiefe anbietet, erzeugen ein Drittel Mehraufwand in Form von ungenutzten Textureinträgen oder

Tabelle 5.6: Performanz des Hardware-basierten Algorithmus zum Ray Casting für unterschiedliche Datensätze auf einer ATI Radeon 9700 Pro. Die Zellenanzahlen beinhalteten die imaginären Zellen aus der Konvexifizierung.

	Gesamtzahl der Zellen	Imaginäre Zellen	∅ Fps	∅ Tets/Sek
Bluntfin (Volumen)	190.852	3.534	3,37	643 K
Heatsink (Volumen)	124.152	2.484	2,27	282 K
Cylinder (Volumen)	203.460	1.860	1,94	395 K
Sphere (Iso)	148.955	—	5,13	764 K

Einträgen mit unnötig hoher Genauigkeit. Darüber hinaus verhindern Beschränkungen in der Länge der unterstützten Pixel Shader auf der ATI Radeon 9700 eine effektivere Speicherung der Gitterdaten mit indizierten Vertexlisten. Dies limitiert die maximale Größe für Gitter, die auf dieser Graphikkarte mit 128 MB Speicher verarbeitet werden können auf 500.000 bis 600.000 Tetraeder. Sicherlich lässt sich diese Grenze mit Detailverbesserungen, wie sie beispielsweise in [Bernardon et al. 2004] vorgestellt wurden, noch nach oben schieben. In der Praxis erreichen Tetraedergitter jedoch Größen von mehreren Millionen Zellen, was eine deutlich intelligentere Speicherung beispielsweise in Form komprimierter Tetraederstreifen erfordert, wie sie im folgenden Abschnitt untersucht wird.

5.6.2 OpenGL-Implementierung

Tabelle 5.7 zeigt die Ergebnisse, die mit der Implementierung für eine nVidia GeForce 6800 erzielt wurden. Neben der Speicherung des Tetraedergitters als Tetraederstreifen ist ein wichtiges Merkmal dieser Implementierung, dass aufgrund der von dieser Hardware unterstützten langen Pixel Shader unterschiedliche Implementierungsvarianten möglich waren. Einerseits wurde eine speicheroptimierte Variante realisiert, die versucht, mit minimalem Speicher in der Gitterdatenstruktur auszukommen und alle abgeleiteten Größen wie Normalen oder den Gradient berechnet. Die zweite Variante ist auf Performanzoptimierung ausgerichtet und erweitert die Gitterdatenstruktur mit zusätzlichen Parametern um Fragmentoperationen zu sparen.

Im Vergleich der beiden Varianten lässt sich eine fast doppelt so hohe Bildwiederholrate für die performanzoptimierte Variante feststellen, die in einer signifikanten Reduzierung der arithmetischen Operationen begründet ist. Im Bezug auf Texturoperationen sind beide Varianten vergleichbar, weil in der performanzoptimierten Variante der Nachschlag der Vertexkoordinaten

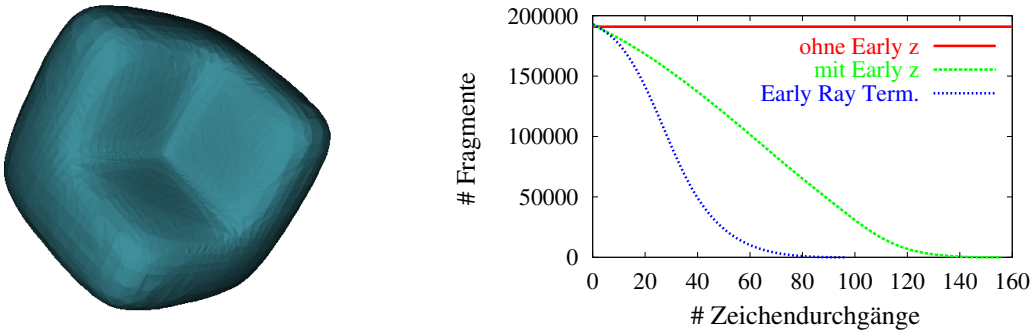


Abbildung 5.7: Die Abbildung demonstriert den Effekt von Early Ray Termination bei der Anwendung mit dem Hardware-basierten Ray Caster. Die Kurven zeigen die Anzahl der Fragmente, die in jedem Zeichendurchgang verarbeitet werden. Wie anhand der Fläche unter der Kurve ersichtlich, lässt sich durch Ausnutzung des vorgelagerten Tiefentests die Gesamtmenge der rasterisierten Fragmente deutlich reduzieren.

entfällt. Der Performanz-Gewinn geht jedoch einher mit einem signifikant höheren Speicherbedarf (≈ 608 Bit pro Tetraeder im Vergleich zu 117 Bit pro Tetraeder). Dies stellt immer noch eine Verbesserung im Vergleich zu den 1280 Bit pro Tetraeder für die DirectX-Implementierung dar und erlaubt die Verarbeitung von Datensätzen mit mehr als drei Millionen Tetraedern. Es wurde ebenfalls mit Varianten experimentiert, welche nur eine Normalentextur oder eine Gradiententextur verwenden. Die erzielten Ergebnisse sind etwas besser als die speicheroptimierte Variante, allerdings wiegen die zusätzlichen Texturoperationen die gesparten arithmetischen Operationen nahezu auf.

Weiterhin erlauben die längern Pixel Shader als Optimierung mehrere Traversierungsschritte in einem Zeichendurchgang durchzuführen. Mit den 1024 Instruktionen einer nVidia GeForce 6800 sind abhängig vom verwendeten optischen Modell 5-7 Traversierungsschritte realisierbar. Dies reduziert die teuren Lese- und Schreiboperationen in beziehungsweise aus den Traversierungstexturen und sollte daher in einer höheren Performanz resultieren. Leider sind die erzielten Ergebnisse im Vergleich zur ATI Radeon 9700 nicht so gut wie erwartet. Da ein Generationswechsel zwischen den beiden Graphikkarten vorliegt, sollte die nVidia-Karte viel schneller sein. Analysen zeigen, dass dies hauptsächlich daran liegt, dass der vorgelagerte Tiefentest auf der nVidia-Karte weitaus weniger effektiv arbeitet, was dazu führt, dass auf Pixeln, die bereits fertig bearbeitet sind, ständig weitere Berechnungen durchgeführt werden. Diese Analyse basiert auf zwei Beobachtungen: Erstens führt die Verwendung von „Kill"-Operationen anstelle von speziellen Zeichendurchgängen zur Terminierung zu einer Beschleunigung um 14%, obwohl die „Kill"-Operation dafür bekannt ist, den vorgelagerten Tiefentest zu deaktivieren [nVidia 2004], weshalb eine geringere Performanz zu erwarten ist. Zweitens konnte durch eine Unterteilung des Bildes in mehrere Tiles, wie in [Bernardon et al. 2004] vorgeschlagen, die Geschwindigkeit signifikant erhöht werden.

Beim Tiling wird das bildschirmfüllende Rechteck, das zur Strahltraversierung gezeichnet wird, in mehrere kleinere Rechtecke zerlegt, die dann hintereinander gezeichnet werden. Der Test auf Strahlterminierung kann dabei für jedes Tile individuell erfolgen und sobald ein Tile nur noch fertig berechnete Pixel überdeckt, wird es nicht mehr gezeichnet. Theoretisch sollte, wenn der

Tabelle 5.7: Vergleich der Bildwiederholraten (in Frames pro Sekunde) zwischen der speicherplatzoptimierten und der performanzoptimierten Implementierung auf einer nVidia GeForce 6800.

	# Tetraeder	Zeichen-durchgänge	speicheropt.			performanzopt.		
			∅	min	max	∅	min	max
Sphere	148.955	108	4,0	3,5	4,7	6,4	4,4	7,5
Ell (Volumen)	148.955	129	2,7	2,2	3,1	4,3	2,9	5,1
Ell (Iso)	148.955	128	2,9	2,2	3,4	4,8	3,2	5,7
Spx	103.488	346	2,7	2,2	3,2	4,0	3,3	4,7

vorgelagerte Tiefentest die Berechnungen für fertige Pixel unterbindet, mit Tiling dieselbe oder, bedingt durch den damit verbundenen Mehraufwand, sogar eine niedrigere Performanz als ohne Tiling resultieren. In den durchgeführten Tests wurde das Optimum jedoch bei einer Unterteilung in 16×16 Tiles erreicht.

Auch die Bindeoperationen für die Texturen haben sich nicht als Flaschenhals für die Implementierung herausgestellt. Die Variation bei Verwendung mehrerer Traversierungsschritte pro Zeichendurchgang lag unter 8%. Allerdings hat die Verwendung mehrerer Traversierungsschritte pro Zeichendurchgang einen Effekt auf die akkumulierte Opazität, welche die Terminierung der Sehstrahlen anstößt. Jedesmal, wenn die Opazität in den Traversierungs-Puffer geschrieben wird, erfolgt einer Quantisierung auf 16 Bit, was innerhalb der Bearbeitung mehrerer Traversierungsschritte in demselben Shader unterbleibt. Hier wird immer mit 32 Bit Genauigkeit gerechnet. Dies kann zu einer leicht unterschiedlichen Anzahl von durchgeführten Zeichendurchgängen führen, wobei allerdings kein visueller Effekt beobachtet wurde.

5.6.3 Diskussion

Trotz aller Vorteile durch die Ausnutzung von Early Ray Termination wird die Performanz des hier vorgestellte Algorithmus zum Ray Casting von Tetraedergittern gegenwärtig noch in vielen Situationen von einer optimierten Implementierung der Zellprojektion, wie sie z.B. in [Guthe et al. 2003] veröffentlicht wurde, übertroffen. Dies trifft insbesondere bei großem Viewport zu, da Ray Casting in hohem Maße ausgabeabhängig ist.

Demgegenüber sind Ansätze zur Zellprojektion insbesondere für nicht-kommutative optische Modelle durch die Notwendigkeit der Sichtbarkeitssortierung beschränkt, was bereits heutige Graphikprozessoren daran hindert, mit voller Geschwindigkeit zu arbeiten, da CPU und Graphikbus die Daten nicht ausreichend schnell liefern können. Beim Hardware-basierten Ray Casting sind alle erforderlichen Daten im lokalen Speicher der Graphikkarte abgelegt. In der Projektion der Graphikkartenentwicklung der letzten Jahre in die Zukunft kann Hardware-basiertes Ray Casting bald aufholen, da es nur durch die Rasterisierungsleistung der Graphik-Hardware beschränkt ist.

Auf der anderen Seite werden Ansätze zur Zellprojektion von höheren CPU-Geschwindigkeiten und neuen Technologien für den Graphikbus profitieren, da die Rasterisierung der Dreiecke aus der Shirly-Tuchman-Projektion weit weniger aufwändig ist als die Traversierung der Sehstrahlen im Pixel Shader. Letztere erfordert eine deutlich größere Anzahl von Texturnachschlägen, weshalb Hardware-basiertes Ray Casting auch durch die interne Anbindung des Graphikkartenspeichers limitiert ist. Auch kann trotz Ausnutzung der Early Ray Termination die Gesamtanzahl der bearbeiteten Fragmente größer sein, da Ray Casting unter Umständen mehr Fragmente erzeugt, als die Tiefenkomplexität der Gitters erfordert. Zellprojektion rasterisiert garantiert nur die Front Faces aller Zellen.

Eine weitere wichtige Beschränkung des Hardware-basierten Ray Casting liegt in der Tatsache, dass dabei alle Gitterdaten in den lokalen Speicher der Graphikkarte passen müssen, um interaktive Visualisierungen zu erreichen. Die Speicherplatzressourcen auf der Graphikkarte stei-

gen zwar kontinuierlich an, können aber vermutlich nicht mit den Anforderungen an die Größe zu visualisierender Datensätze mithalten. In diesem Zusammenhang stellt die Möglichkeit der komprimierten Datenspeicherung eine wichtige Verbesserung dar. Allerdings steigt durch die zunehmende Verbreitung der Computersimulation in Wissenschaft und Technik die Größe praktischer Datensätze dramatisch an, was zunehmend auch den Einsatz von Out-of-Core-Techniken erfordert. Diese würden die kontinuierliche Übertragung der Gitterdaten auf die Graphikkarte erfordern, so dass sich kein Vorteil mehr gegenüber der Zellprojektion ergibt.

In der Summe ist das hier vorgestellte Hardware-basierte Ray Casting besonders dann interessant, wenn mittelgroße, möglicherweise zyklische und nicht-konvexe Gitter visualisiert werden sollen und hohe Opazitäten, beispielsweise aufgrund opaquer Isoflächen einen frühen Strahlabbruch erlauben.

Kapitel 6

Volumenvisualisierung mit radialen Basisfunktionen

Die Speicherung der zu visualisierenden Daten direkt im Texturspeicher der Graphikkarte wurde bereits im vorangegangenen Kapitel als erfolgreiche Strategie präsentiert, um die Datenübertragung zwischen Zentralprozessor und Graphikprozessor als Flaschenhals für die Visualisierung zu beseitigen und damit die volle Rechenleistung der Graphik-Hardware auszunutzen zu können. Die Anforderungen an eine geeignete Datenrepräsentation sind dabei vielschichtig. So verlangen die enormen Datenmengen, die heutzutage von Scannern, Sensoren oder in Simulationen generiert werden und von Visualisierungssystemen verarbeitet werden müssen, nach einer kompakten Speicherung. Dies gilt umso mehr als der auf Graphikkarten verfügbare Texturspeicher beschränkt ist.

Weiterhin muss eine geeignete Datenstruktur an die Fähigkeiten der Graphik-Hardware angepasst sein. Dies ist für die in Kapitel 5 verwendeten Tetraederlisten oder Tetraederstreifen nur bedingt der Fall, da sie im Wesentlichen eine direkte Umsetzung der Datenstruktur darstellen, die auch eine CPU-Implementierung der Strahlverfolgung verwenden würde. Den Unterschied zu einer für die Graphik-Hardware optimierten Datenstruktur kann man am deutlichsten im Vergleich mit texturbasierter Volumenvisualisierung erkennen, deren Performanz trotz der auch in dieser Arbeit dokumentierten Verbesserungen bei der Volumendarstellung von Tetraedergittern immer noch um eine Größenordnung höher liegt.

Von Vorteil ist darüber hinaus eine Datenstruktur, die einen universellen Ansatz zur Visualisierung erlaubt, da dies weniger Aufwand erzeugt als die (Weiter-)Entwicklung individueller Verfahren für uniforme dreidimensionale Gitter, unstrukturierte Tetraeder- oder Hexaedergitter und gitterlose Daten, um nur einige gebräuchliche Organisationsformen zu nennen.

Eine Datenrepräsentation aufbauend auf radialen Basisfunktionen (RBF) erfüllt alle drei Anforderungen. Da hierbei die Kodierung der Volumendaten prozedural als Linearkombination einer Menge von Basisfunktionen erfolgt, kommt eine RBF-Repräsentation komplett ohne Konnektivitätsinformation aus. Darüber hinaus erlauben die guten Approximationseigenschaften der radialen Basisfunktionen eine Kodierung des Volumens mit relativ wenigen Koeffizienten bei ak-

zeptabler Fehlertoleranz. Dies ermöglicht die Speicherung selbst extrem großer Datensätze auf der Graphikkarte. Die RBF-Repräsentation kommt den Fähigkeiten moderner Graphik-Hardware entgegen, da die programmierbare Fragmenteinheit eine einfache und interaktive Auswertung der RBF-Kodierung on-the-fly erlaubt, wodurch die RBF-Repräsentation der Volumendaten mit einer Vielzahl von bekannten und etablierten Visualisierungsverfahren kombiniert werden kann. Schließlich ist der RBF-Ansatz universell, da Volumendaten mit unterschiedlichen Gitterstrukturen und Organisationsformen als Basis für die RBF-Kodierung dienen können.

Das vorliegende Kapitel beschreibt einen Ansatz zur interaktiven Visualisierung RBF-kodierter Volumendaten, der darüber hinaus auch die interaktive Extraktion von charakteristischen Merkmalen aus Vektordaten erlaubt. Die hier vorgestellte Arbeit entstammt einer Kooperation mit der Universität von Purdue und der Universität von Texas in Austin. Publikationen der Idee erfolgten in [Jang et al. 2004, Weiler et al. 2005] oder stehen noch aus [Jang et al. 2005].

6.1 Radiale Basisfunktionen

Radiale Basisfunktionen (RBF) sind zirkular-symmetrische Funktionen, deren Zentrum an einem definierbaren Punkt im Raum liegt. Gegenüber anderen Kompressionstechniken wie iterierten Funktionensystemen [Barnsley et al. 1988] oder Wavelets [Nguyen und Saupe 2001, Bertram et al. 2000] besitzen sie viele Vorteile, wie z.B. eine kompakte Repräsentation der Basisfunktionen, die einfache Auswertbarkeit sowie die Glättung der ersten und zweiten Ableitung im Falle von verrauschten Daten. Daher spielen sie in vielen Bereichen wie der Bildverarbeitung [Fornefett et al. 1999] oder medizinischen Anwendungen [Zhang et al. 2002] eine Rolle. In der Visualisierung werden sie zur Rekonstruktion und zur Darstellung großer Punktmengen [Hardy 1971, Franke 1982, Hardy 1990, Franke und Nielson 1991, Franke und Hagen 1999, Co et al. 2003] sowie zur Gittervereinfachung bei der Rekonstruktion mittels Laserscanner abgetasteter Oberflächen [Savchenko et al. 1995, Carr et al. 2001, Morse et al. 2001, Turk und O'Brien 2002] eingesetzt. Die Darstellung einer Funktion mittels radialer Basisfunktionen erfolgt als gewichtete Summe der Form

$$f(x) = \lambda_0 + \sum_{i=1}^{N} \lambda_i \phi_i \left(\| x - \mu_i \| \right) \quad , \tag{6.1}$$

wobei N die Anzahl der benötigten Basisfunktionen, λ_i das Gewicht und μ_i das Zentrum der Basisfunktion ϕ_i beschreibt. Gelegentlich findet sich in der Literatur auch ein Polynom niedrigen Grades als ergänzender Summand. In dieser Arbeit wird lediglich ein zusätzlicher Term λ_0 verwendet, um einen globalen Offset zu kodieren, der nicht mit kompakten Basisfunktionen beschrieben werden kann.

Abhängig von der Anwendung bieten sich unterschiedliche Typen von Basisfunktionen an. Neben der Multiquadrik-Funktion werden für die Oberflächenrekonstruktion vor allem biharmonische oder triharmonische Splines eingesetzt, da sie im Allgemeinen bessere Ergebnisse als

kompakte Basisfunktionen liefern [Carr et al. 2001]. Für die Rekonstruktion mittels programmierbarer Graphikhardware besser geeignet sind beschränkte Gauß-Funktionen, da ihr begrenzter räumlicher Einfluss dazu führt, dass für die Rekonstruktion der Ursprungsfunktion an einer gegebenen Position im Raum letztlich nur eine kleine Anzahl von Basisfunktionen ausgewertet werden muss. Der dadurch verringerte Berechnungsaufwand erlaubt interaktive Bildwiederholraten. Die RBF-Repräsentation mittels Gauß-Basis ergibt sich zu

$$f(x) = \lambda_0 + \sum_{i=1}^{N} \lambda_i \exp\left(-\frac{\|x-\mu_i\|^2}{2\sigma_i^2}\right) \quad , \tag{6.2}$$

wobei σ_i^2 die Breite der Gauß-Funktion beschreibt, und auch als Varianz bezeichnet wird.

6.2 Kodierung von Volumendaten

Die Erzeugung einer RBF-Repräsentation für einen vorgegebenen Datensatz erfordert einen mehrstufigen Kodierungsprozess. Ziel dieses Prozesses ist es, geeignete Parameter für die Gleichung (6.2) zu bestimmen, so dass die Funktion $f(x)$ eine vorgegebene Menge von Datenpunkten $p_j = (x_j, y_j), j = 1,...,n$ mit Positionen $x_j \in \mathbb{R}^3$ und Datenwerten $y_j \in \mathbb{R}^k$ innerhalb einer benutzerdefinierten Fehlerschranke approximiert. Diese p_j können die Vertizes eines strukturierten oder eines unstrukturierten Gitters sein oder auch die Datenpunkte einer punktbasierten Repräsentation. Im Detail wird dieser Kodierungsprozess in [Jang et al. 2005] beschrieben, wo auch vergleichende Betrachtungen unterschiedlicher Berechnungsmethoden und Fehlermetriken erfolgen. Grob gliedert sich die Kodierung, wie in Abbildung 6.1 skizziert, in folgende Schritte:

1. Bestimmung der RBF-Zentren: Zur Verbesserung der Kodiergenauigkeit erfolgt die Bestimmung der RBF-Zentren für den niederfrequenten beziehungsweise den hochfrequenten Anteil des Ausgangssignals in unterschiedlicher Weise. Um die groben Strukturen zu repräsentieren, erfolgt zunächst eine Tiefpass-Filterung mit einem Gauß-Filterkernel, der die 25 nächsten Nachbarn eines Datenpunktes berücksichtigt. RBF-Zentren werden dann an den Extrema des gefilterten Signals positioniert. Dies erfolgt iterativ in mehreren Stufen, wobei der Grad der Filterung in jeder Stufe um 40% reduziert wird. Eine erste Approximation der übrigen Parameter in Gleichung (6.2) mittels Least Square Fit erlaubt die Bestimmung des Residuums an jedem Datenpunkt (hochfrequenter Signalanteil).

Weitere RBF-Zentren für den hochfrequenten Signalanteil werden mittels Principal Component Analysis (PCA) [Jolliffe 1986] bestimmt, wobei die Menge der Datenpunkte p_j durch sukzessive Split-Operationen in Cluster aufgeteilt werden, die jeweils mit einer Gauß-Funktion approximiert werden. Bis zum Erreichen einer benutzerdefinierten Fehlerschwelle wird dabei jeweils der Cluster mit dem größten Approximationsfehler an der Ebene geteilt, die senkrecht zu dem Eigenvektor liegt, der zum größten Eigenwert der Kovarianzmatrix gehört.

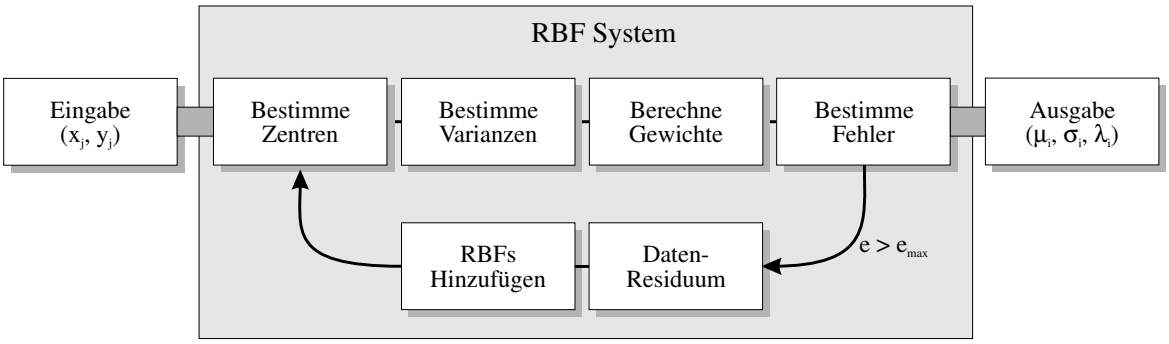


Abbildung 6.1: Volumenkodierung mittels radialer Basisfunktionen.

2. Bestimmung der Varianzen: Mit der Vorgabe der RBF-Zentren erfolgt die Bestimmung der Varianzen mittels nicht-linearer Optimierung [Branch et al. 1999], wobei alle übrigen Parameter als konstant angenommen werden. Der globale Offset λ_0 wird dazu auf 0 gesetzt und die Gewichte λ_i werden auf einen festen Wert gesetzt, der für die Zentren aus der Filterung dem Datenwert am Extremum und für das PCA Clustering dem Residuum am Cluster-Zentrum entspricht. Als Startbedingung für σ_i^2 wird für die Zentren aus der Filterung der halbe Abstand zum nächsten Zentrum angenommen. Für die Zentren aus dem PCA Clustering ergibt sich der Startwert für σ_i^2 durch lokale Least Square Optimierung über die nächsten 25 Nachbar-Cluster.

3. Berechnung der Gewichte: Mit den bereits bestimmten Gauß-Zentren und Varianzen ergibt sich ein im Allgemeinen überbestimmtes lineares Gleichungssystem für die Gewichte und den globalen Offset. Da die direkte Lösung dieses Gleichungssystems, beispielsweise mittels der Methode der Pseudo-Inversen [Albert 1972], die Invertierung einer sehr großen Matrix beinhaltet und darüber hinaus zu extremen Gewichten λ_i führen kann, kommt hier ein iteratives Verfahren zum Einsatz [MathWorks 2003].

4. Fehlerbestimmung: Nach der Kodierung wird der Fehler an den ursprünglichen Datenpunkten bestimmt. Sollte der maximale oder der mittlere quadratische Fehler oberhalb einer vorgegebenen Schwelle liegen, kann das Residuum, die Differenz zwischen dem kodierten Signal und den ursprünglichen Datenwerten, bestimmt werden, und eine weitere Kodierstufe beginnend mit Schritt 1 schließt sich an.

Trotz dieser schrittweisen Herangehensweise stellt der Least Square Fit der RBF-Parameter einen rechenaufwändigen Prozess dar, dessen Speicheranforderungen die Bearbeitung von großen Datensätzen unmöglich machen kann. Eine Lösung bietet die sogenannte *Domain Localization* [Nielson 1993]. Dabei wird das Datengebiet beispielsweise mittels eines Kd-Baums in überlappende Teilbereiche mit jeweils gleicher Menge von Datenpunkten unterteilt. Datenpunkte, die in unterschiedliche Teilbereiche fallen, werden dabei repliziert, wobei eine zum Rand des Teilbereiches abfallende Gewichtsfunktion sicherstellt, dass sich in der Summe derselbe Datenwert

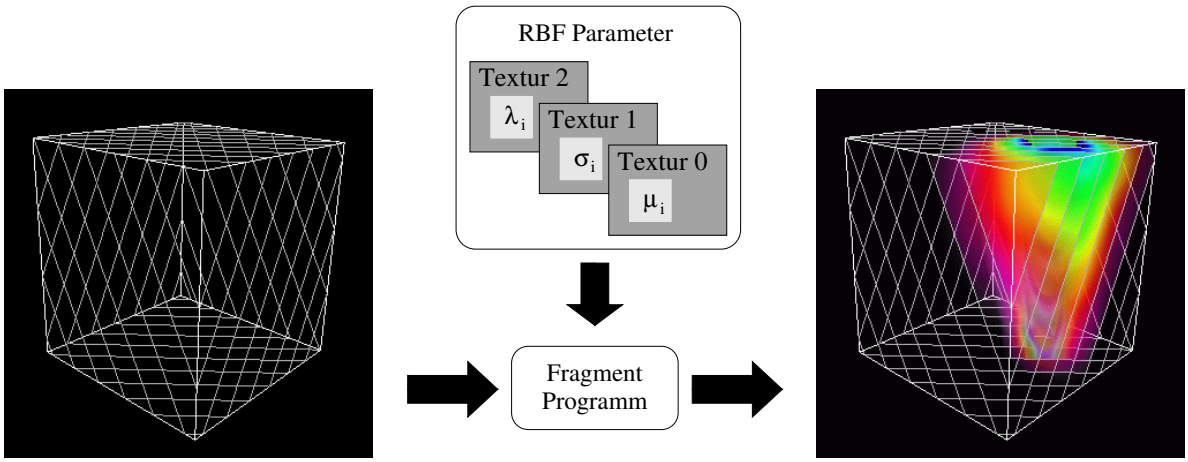


Abbildung 6.2: Das RBF-basierte Visualisierungssystem verwendet die programmierbare Fragmenteinheit, um das RBF-kodierte Datensignal während der Rasterisierung on-the-fly auszuwerten. Die RBF-Koeffizienten werden dabei aus Texturen ermittelt, welche die kompakte RBF-Repräsentation eines Datensatzes direkt auf der Graphikkarte speichern.

wie im unzerlegten Fall ergibt. Der Vorteil dieser Zerlegung ist, dass die RBF-Kodierung für jeden Teilbereich unabhängig erfolgen kann und diese aufgrund der deutlich geringeren Anzahl von Datenpunkten sehr viel effizienter ist.

6.3 RBF-basierte Visualisierung

Für die Visualisierung der RBF-kodierten Volumendaten wird eine angepasste Version der texturbasierten Volumenvisualisierung [Cabral et al. 1994] verwendet. Dabei werden Slice-Polygone durch den Schnitt von äquidistanten Ebenen parallel zur Bildebene mit den Grenzen des Volumens bestimmt, mit den Volumendaten texturiert und mittels Compositing von hinten nach vorne im Framebuffer überlagert. Der hier vorgestellte Ansatz beseitigt jedoch die Notwendigkeit, die Volumendaten als dreidimensionale Textur zu speichern, indem er die Fähigkeit moderner Graphikkarten nutzt, beliebige arithmetische Operationen in der Fragmenteinheit auszuführen. Nur die Parameter der RBF-Kodierung werden in zweidimensionalen Texturen auf der Graphikkarte vorgehalten und die Rekonstruktion der ursprünglichen Volumenwerte erfolgt, wie Abbildung 6.2 zeigt, on-the-fly während der Rasterisierung der einzelnen Slice-Polygone.

Die kompakte RBF-Repräsentation erlaubt es, sehr viel größere Datensätze direkt auf der Graphikkarte zu speichern, als dies als dreidimensionale Textur möglich wäre. Ein teurer Transfer der Volumendaten über den Graphikbus wird dadurch vermieden. Außerdem nutzt dieser Ansatz die massive Rechenleistung der Graphikkarte für die Rekonstruktion der Volumendaten. Ein weiterer wesentlicher Vorteil liegt in der Flexibilität. Die Auswertung der RBFs erfolgt durch den Graphikprozessor für jedes rasterisierte Fragment und damit transparent von der gezeichneten

Geometrie. Damit lässt sich der Ansatz auf eine Vielzahl von Visualisierungsalgorithmen erweitern. So sind sowohl beliebig orientierte Schnittebenen, als auch nicht-polygonal mittels Slicing-Technik dargestellte Isoflächen [Westermann und Ertl 1998] realisierbar. Weitere Möglichkeiten schließen die Einfärbung einer Oberfläche oder Geometrie basierend auf korrelierten physikalischen Größen ein, beispielsweise Druck auf einem Flugzeugrumpf. Für RBF-kodierte Vektordaten können anspruchsvollere Techniken wie Teilchenverfolgung oder die interaktive Berechnung von spezifischen Merkmalen eines Strömungsfeldes, wie beispielsweise der Vortizität, realisiert werden.

6.3.1 Hierarchische Strukturierung

Ein wesentliches Kriterium für die Wahl der Gauß-Funktion als Basisfunktion ist ihr eingeschränkter räumlicher Einfluss, der dazu führt, dass für einen gegebenen Raumpunkt nur relativ wenige Basisfunktionen ausgewertet werden müssen. Um der Fragmenteinheit einen effizienten Zugriff auf diese Teilmenge zu ermöglichen, erfolgt eine hierarchische Strukturierung der Basisfunktionen mit Hilfe eines adaptiven Oktalbaums. Dazu wird das Volumen nicht-uniform in einzelne Zellen unterteilt. Jede Zelle speichert eine Liste der Basisfunktionen, welche einen nicht vernachlässigbaren Beitrag innerhalb der Zelle liefern. Letzteres ist der Fall, wenn der Einflussradius r_i der RBF die Zelle schneidet. Für die Gauß-Basis ergibt sich dieser Radius für eine definierte Fehlertoleranz ε zu:

$$r = \sigma_i \cdot \sqrt{2 \cdot ln\left(\frac{|\lambda_i|}{\varepsilon}\right)} \quad . \tag{6.3}$$

Die sukzessive Unterteilung erfolgt solange, bis die Anzahl der Basisfunktionen pro Zelle kleiner ist als ein vorgegebener Schwellwert (in der Regel die Maximalzahl von Basisfunktionen, welche die Fragmenteinheit in einem Zeichendurchgang auswerten kann) oder bis eine weitere Unterteilung die Anzahl der Basisfunktionen in den acht Kindzellen nicht mehr deutlich reduziert.

6.3.2 Texturkodierung

Ein geeignetes Texturlayout für die Speicherung der RBF-Koeffizienten muss zwei Anforderungen erfüllen: Einerseits sollten die Koeffizienten aller Basisfunktionen, die Einfluss auf eine bestimmte Zelle besitzen, sequentiell entlang einer Zeile der Textur gespeichert sein. So muss das Fragmentprogramm zur Rekonstruktion lediglich die Position der ersten RBF einer Zelle innerhalb der Textur kennen und kann auf die weiteren RBF-Koeffizienten durch sukzessive Inkrementierung der x-Texturkoordinate zugreifen.

Andererseits entsteht, insbesondere auf Graphik-Hardware ohne Unterstützung für dynamische Schleifen im Fragmentprogramm[1], oftmals die Situation, dass mehr RBF-Basisfunktionen ausgewertet werden müssen, als die Fragmenteinheit in einem Zeichendurchgang berechnen kann.

[1] betrifft ATI 9700 und ATI X800 sowie die nVidia GeForceFX Chip-Serie

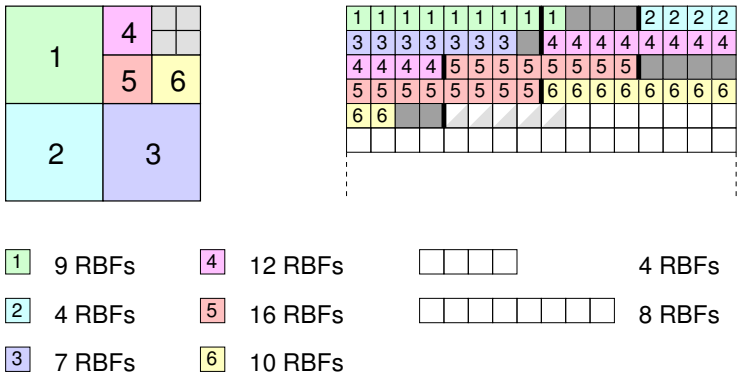


Abbildung 6.3: Die RBF-Daten aller Zellen werden dicht gepackt und in einem einzigen Satz von Texturen gespeichert. In diesem Beispiel wurde angenommen, dass zwei Fragmentprogramme, die vier beziehungsweise acht Basisfunktionen auswerten, verfügbar sind.

In diesem Fall muss ein Mehrpass-Verfahren zum Einsatz kommen, welches zwei Hardware-beschleunigte p-Buffer verwendet. In jedem Zeichendurchgang wird die maximal mögliche Anzahl von Basisfunktionen ausgewertet, aufaddiert und in einen Ausgabepuffer geschrieben. Dieser wird im nachfolgenden Zeichendurchgang als Textur gebunden und dient damit als Eingabe des bereits akkumulierten Wertes. Weitere Zeichendurchgänge können so sukzessive mehr Basisfunktionen aufaddieren.

Für das Texturlayout bedeutet dies, dass die Liste von RBF-Koeffizienten einer Zelle in mehrere Blöcke unterteilt werden muss. Jeder Block enthält maximal so viele Basisfunktionen, wie die Graphik-Hardware in einem Zeichendurchgang auswerten kann. Da die fehlende Möglichkeit von dynamischen Schleifen durch eine beschränkte Anzahl von Fragmentprogrammen ausgeglichen werden muss, welche eine feste Anzahl von RBF-Funktionen auswerten, erfolgt gegebenenfalls ein Auffüllen des Blocks mit Nullinformation bis zur nächsten unterstützten RBF-Anzahl.

Basierend auf diesen Anforderungen wird in dem hier vorgestellten Ansatz ein Packalgorithmus verwendet, welcher alle Zellen der hierarchischen Unterteilung traversiert und die jeweilige Liste der erforderlichen RBF-Koeffizienten in Blöcke maximaler Größe unterteilt. Immer dann, wenn ein Block die verbleibende Kapazität einer Texturzeile übersteigt, wird der gesamte Block am Beginn der nächste Texturzeile platziert, um Umbrüche innerhalb des Blocks zu vermeiden. Somit muss nur der letzte Block einer Zelle mit Nullinformation aufgefüllt werden, um auf eine RBF-Anzahl zu kommen, die von der Liste der verfügbaren Fragmentprogramme unterstützt wird. Abbildung 6.3 zeigt dies an einem einfachen Beispiel.

Die Speicherung der RBF-Koeffizienten erfolgt mit voller Gleitkommagenauigkeit in mehreren Texturen, die alle dasselbe Layout besitzen, um den Zugriff auf alle Koeffizienten einer Basisfunktion mit demselben Index zu erlauben. Wie Abbildung 6.2 andeutet, wird jeweils eine Textur für die Position der RBF-Zentren μ_i, die Gewichte λ_i und σ_i verwendet. Genauer gesagt wird der Term $\frac{1}{2\sigma^2}$ gespeichert, um Rechenoperationen in der Fragmenteinheit zu sparen.

6.3.3 Schnittebenen und Volumenvisualisierung

Beim Zeichnen von Schnittebenen, die mittels RBF-Rekonstruktion texturiert werden sollen, muss berücksichtigt werden, dass jede Zelle der räumlichen Unterteilung einen unterschiedlichen Satz an RBF-Koeffizienten besitzen kann und somit mit unterschiedlichem OpenGL-Zustand gezeichnet werden muss. Diese Situation ähnelt dem Bricking-Ansatz [Grzeszczuk et al. 1998], der im Kontext von texturbasierter Volumenvisualisierung verwendet wird, wenn die Größe eines Datensatzes den physikalischen Texturspeicher übersteigt. Dabei wird der Datensatz in einzelne Blöcke oder *Bricks* unterteilt, die jeweils als eigenständige dreidimensionale Textur repräsentiert werden. Das Slicing der Bricks kann unter Berücksichtigung der Verdeckung von hinten nach vorne unabhängig erfolgen, wobei zur Minimierung der OpenGL-Zustandsänderungen jeweils alle Slices eines Bricks gezeichnet werden, bevor der nächste Brick betrachtet wird.

In dem hier vorgestellten Ansatz der RBF-basierten Volumenvisualisierung ist die Umschaltung zwischen unterschiedlichen Zellen relativ kostengünstig und erfordert oftmals nicht einmal eine Texturumschaltung. Auf der anderen Seite muss die Möglichkeit einer Rekonstruktion mit mehreren Zeichendurchgängen berücksichtigt werden, da eine Zelle mehr RBFs beinhalten kann, als von der Fragmenteinheit in einem Durchlauf bearbeitet werden kann. Es wird daher eine Traversierungsreihenfolge verwendet, welche in einer äußeren Schleife über alle Slices iteriert und für jedes Slice alle getroffenen Zellen betrachtet. Slices werden dazu an den Zellgrenzen abgeschnitten und das resultierende Polygon (potentiell mehrere Male) gezeichnet. Die Verwaltung einer Liste mit den aktiven Zellen, die inkrementell für jedes neue Slice aktualisiert wird, erlaubt dabei den effizienten Zugriff auf die von einem Slice geschnittenen Zellen.

Die Rekonstruktion des RBF-kodierten Volumenwertes auf jedem gezeichneten Slice erfolgt mit einem Fragmentprogramm, wie es in Abbildung 6.4 dargestellt ist. Das in der Shader-Hochsprache Cg [nVidia 2002] formulierte Programm, setzt die Unterstützung dynamischer Schleifen in der Fragmenteinheit voraus, wie sie von dem nVidia GeForce 6800 Chip angeboten werden. Im Falle der fehlenden Unterstützung für dynamische Schleifen erlaubt das Ersetzen der Schleifenabbruchbedingung `texpos.z` durch die mittels Cg-Präprozessor definierbare Konstante `CONST_NUMFUNCS` eine einfache Generierung eines Satzes von Fragmentprogrammen für die Rekonstruktion einer unterschiedlichen Anzahl von RBF-Funktionen. Diese werden durch Ausrollen der Schleife durch den Cg-Compiler generiert. Aus Gründen der Übersichtlichkeit wurde in Abbildung 6.4 auf die Darstellung der in diesem Fall außerdem erforderlichen Erweiterung um die Fähigkeit mehrerer Zeichendurchgänge verzichtet, da diese offensichtlich ist.

Das Programm akkumuliert die RBF-Funktionen in einer lokalen Variable. Die Iteration über die einzelnen Basisfunktionen beinhaltet den Nachschlag der Koordinaten des jeweiligen RBF-Zentrums, des RBF-Gewichts und der Varianz aus den entsprechenden Texturen, die Berechnung des Abstands des aktuellen Fragments vom RBF-Zentrum und die Evaluation der RBF-Funktion. Die Exponentialfunktion zur Basis zwei wird dabei aus Performanzgründen verwendet und durch einen Korrekturterm kompensiert, der in den σ_i-Einträgen der Texturen berücksichtigt ist. Für die Ermittlung der Fragmentposition in Weltkoordinaten werden die Texturkoordinaten in `inpos` verwendet, die aus den an den Slice-Vertizes definierten Weltkoordinaten interpoliert werden. Nach der Auswertung aller RBF-Funktionen ergibt sich durch Addition des globalen Offsets der

```
// Maximale Anzahl von Basisfunktionen für das Aufrollen der Schleife
#define CONST_NUMFUNCS 36

float4 main ( // Weltkoordinaten und RBF-Texturen
              float4              inpos    : TEXCOORD0,
              uniform samplerRECT rbfcenter: TEXTUREUNIT1,
              uniform samplerRECT rbfwidth : TEXTUREUNIT2,
              uniform samplerRECT rbfweight: TEXTUREUNIT3,
                // Texturadressierung: Offset + Increment
              uniform float4      texstart : C0,
              uniform float4      texinc   : C1,
              // Globaler Offset für die RBF-Rekonstruktion
              uniform float       bias     : C2,
              // Farbtabelle mit Scale und Bias, Opazitätsskalierung
              uniform sampler1D   map      : TEXTUREUNIT0,
              uniform float4      mapSBA   : C20,
            ) : COLOR
{
  float  val    = 0.0;
  float4 texpos = texstart, output;

  for (float i = 0; i < texpos.z; i++) {
    // texpos.z gibt an, wie viele RBFs in dieser Zelle ausgewertet werden müssen
    float4 center = texRECT (rbfcenter, texpos.xy);
    float  s2_inv = texRECT (rbfwidth, texpos.xy);
    float  lambda = texRECT (rbfweight, texpos.xy);
    float3 vec    = center.xyz - inpos.xyz;
    float  expval = - dot (vec, vec) * s2_inv;
    val    += lambda * ex2 (expval);
    texpos += texinc;
  }
  // Addiere globalen Offset
  val += bias;
  // Farbtabellennachschlag nach geeignetem Scale und Bias
  output.rbga = tex1D (map, (val + mapSBA.r) * mapSBA.g);
  // Opazitätskorrektur für die Volumendarstellung
  output.a   *= mapSBA.a;
  return output;
}
```

Abbildung 6.4: Fragmentprogramm für die Rekonstruktion von radialen Gauß-Basisfunktionen.

rekonstruierte Volumenwert. Dieser wird mittels Nachschlag in einer eindimensionalen Textur, die als Transferfunktion dient, auf einen Farb- und Opazitätswert abgebildet. Eine vorgeschaltete Scale- und Bias-Operation erlaubt dabei die Einschränkung des relevanten Datenbereiches.

Neben diesem Programm für semi-transparent dargestellte Volumendaten wurde auch ein Programm für die Slice-basierte Isoflächenrekonstruktion realisiert. Dieses führt zusätzlich eine Beleuchtungsberechung durch, die auf dem Gradienten der Volumendaten basiert, welcher parallel zum Skalarwert analytisch ausgewertet werden kann. Wie der geringe Unterschied zwischen der Rekonstruktionsgleichung (6.2) und der Gleichung für die Gradientenberechnung (6.4) nahelegt, erfordert dies nur wenige zusätzliche Operationen im Fragmentprogramm.

$$\nabla f(x) = -\sum_{i=1}^{N} \frac{x-\mu_i}{\sigma_i^2} \lambda_i \exp\left(-\frac{\|x-\mu_i\|^2}{2\sigma_i^2}\right) \tag{6.4}$$

6.3.4 Splatting

Für Datensätze mit spezifischen, räumlich eng begrenzten Merkmalen wie zum Beispiel Schockwellen neigt die RBF-Kodierung dazu, sehr viele RBFs mit kleinem Einflussradius zu erzeugen. Für die Slice-basierte Rekonstruktion ist dies von Nachteil, da die meisten RBFs einer Zelle nicht zum Datenwert eines bestimmten Fragments beitragen. Eine weitere Verfeinerung der räumlichen Hierarchie stellt nur eine unbefriedigende Lösung dieses Problems dar, weil dies die Zellenanzahl dramatisch erhöht, was durch den Mehraufwand für die Zelltraversierung und das Clipping der Slice-Polygone zu einer schlechten Performanz führt. Eine alternative Darstellungstechnik kann in dieser Situation jedoch die Anzahl der je Fragment erforderlichen Operationen drastisch reduzieren.

In einem Objektraumansatz wird dabei die globale Liste der RBF-Funktionen anstelle der hierarchischen Strukturierung betrachtet und für jede dieser Funktionen ein Polygon (Rechteck) gezeichnet, welches lediglich die Einflussscheibe der Basisfunktion auf das aktuelle Slice überdeckt. Um diesen *Splat* zu rasterisieren, ist wesentlich weniger Aufwand erforderlich, weil lediglich ein Summand der RBF-Rekonstruktionsgleichung im Fragmentprogramm ausgewertet werden muss. Die Aufsummierung der einzelnen RBF-Beiträge sollte idealerweise durch Blending der Splat-Polygone mittels additiver Blending-Funktion im Framebuffer erfolgen. Jedoch unterstützt keine verfügbare Graphik-Hardware Blending für p-Buffer mit 32 Bit Genauigkeit. Dies macht einen Ansatz mit mehreren Zeichendurchgängen erforderlich, der im Sinne von Ping-Pong-Rendering jeweils das Ergebnis bis zum vorangegangenen Splat als Textur lädt und den neu akkumulierten Wert in einen p-Buffer schreibt. Konzeptionell würde dies eine Texturladeoperation je Splat erfordern, was zu sehr schlechter Performanz führt. Als undokumentierte Funktionalität erlaubt der nVidia GeForceFX Chip jedoch das simultane Binden des aktuellen p-Buffers als Textur, was die Notwendigkeit kontinuierlicher Texturladeoperationen beseitigt.

Eine weitere Reduktion des Rasterisierungsaufwands kann erreicht werden, indem das Splat-Polygon an die runde Form der RBF-Einflussscheibe angepasst wird. Dies erfolgt durch rekursive Unterteilung des initialen Rechtecks in jeweils vier gleichgroße Teile. Auf jeder Stufe werden die Teilrechtecke verworfen, die nicht von der Einflussscheibe der RBF getroffen werden oder außerhalb des Volumens liegen. Untersuchungen ergaben, dass sich so die gezeichneten Polygone selbst nach nur drei oder vier Unterteilungsschritten eng an den tatsächlich zu rasterisierenden Bereich anpassen. Ein Nachteil des Verfahrens ist, dass der Unterteilungsansatz die Anzahl der gezeichneten Polygone erhöht. Dies hat sich für wenige Unterteilungsschritte nicht als Flaschenhals erwiesen, weil die Vertexeinheit der Graphikkarte vorher nur gering beschäftigt war.

Ein weiteres Problem ist, dass die Splat-Polygone trotz der beschriebenen Unterteilung über die Grenzen des Volumens hinausragen, was zu unerwünschten visuellen Effekten führen kann. Für die hier verwendeten parallel zur Bildebene liegenden Schnittebenen kann dies jedoch durch eine geschickte Ausnutzung des Tiefentests behoben werden. Nach dem Löschen des Tiefenpuffers mit z_{near} werden die vom Betrachter weg gerichteten Flächen der Bounding Box des Volumens mit einem auf `GL_GREATER` gesetzten Tiefentest gezeichnet. Die Splat-Darstellung erfolgt mit dem normalen Tiefentest `GL_LESS`, der nur Fragmente durchlässt, die aus dem Inneren des Volumens projiziert werden. Bereiche vor dem Volumen werden zwar durch dieses Vorgehen nicht

Tabelle 6.1: Genauigkeit und Kompressionrate für RBF-kodierte Datensätze.

Datensatz	# Zellen	# RBFs	Relativer Fehler (%) ∅	Max.
X38 Kompressions-Shock	1.943.483	5.703	0,19	5,12
X38 Expansions-Shock	1.943.483	6.750	0,68	8,39
Ölreservoir	156.642	141	0,58	4,99
Tornado	32.768	1.793	1,82	6,82
Lang	32.805	2.105	1,46	4,73

beseitigt, jedoch stellt dies für die hier verwendete Slice-Ausrichtung parallel zur Bildebene kein Problem dar. Die Verwendung des Tiefentests hat darüber hinaus einen positiven Effekt auf die Performanz, weil der vorgelagerte Tiefentest moderner Graphikkarten Fragmente vor dem eigentlichen Shading verwirft.

6.4 Ergebnisse und Diskussion

Die vorgestellte Technik wurde auf einem System mit Intel Pentium 4 Prozessor mit 2,8 GHz und 2 GB Hauptspeicher getestet. Als Graphikkarten kamen eine nVidia GeForceFX 5900 Ultra als Vertreter für Hardware ohne Unterstützung für dynamische Schleifen sowie eine nVidia GeForce 6800 GT mit jeweils 256 MB Graphikspeicher zum Einsatz. Zeitmessungen erfolgten soweit nicht anders vermerkt auf einem Viewport der Größe 400^2. Unter der Vielzahl der verwendeten Testdatensätze finden sich die Simulation der Strömungsverhältnisse des X38 Transporters, eine simulierte Konvektionsströmung, der von Roger Crawfis analytisch modellierte Tornado sowie eine Wasserkanalsimulation (Lang).

Die Ergebnisse der Kodierung in Tabelle 6.1 zeigen deutlich den niedrigen Speicherbedarf der kompakten RBF-Repräsentation, der unter Einhaltung einer maximalen Fehlertoleranz von 5% bis 8% erreicht wird. Außerdem wird die Möglichkeit demonstriert, sowohl skalare als auch Vektordaten bzw. Datensätze mit mehreren Attributen je Datenwert zu repräsentieren. Der Kodierungsprozess für Vektordatensätze, wie Lang oder Tornado, unterscheidet sich primär im Fehlermaß, welches während der Kodierung verwendet wird und für Vektordaten neben dem Betrag des Vektors auch die Abweichung der Orientierung berücksichtigt. Für die Kodierung von Vektordaten wird ein System aus n-wertigen RBF-Zentren verwendet, welches n unabhängige Gewichte λ_i und Varianzen σ_i^2 für jedes RBF-Zentrum enthält. Diese Kodierung hat sich in Experimenten gegenüber n unabhängigen RBF-Systemen mit jeweils einem Gewicht und einer Varianz als überlegen herausgestellt, da sie weniger RBFs erzeugt und einen geringeren Fehler generiert.

In Abbildung 6.5 werden die unterschiedlichen Darstellungstechniken demonstriert, die mit Hilfe des präsentierten RBF-basierten Visualisierungsansatzes erzielt wurden. Gezeigt werden eine semi-transparente Volumendarstellung sowie eine Isofächendarstellung des simulierten Temperaturfeldes der natürlichen Konvektion einer nicht-Newtonschen Flüssigkeit in einem Würfel.

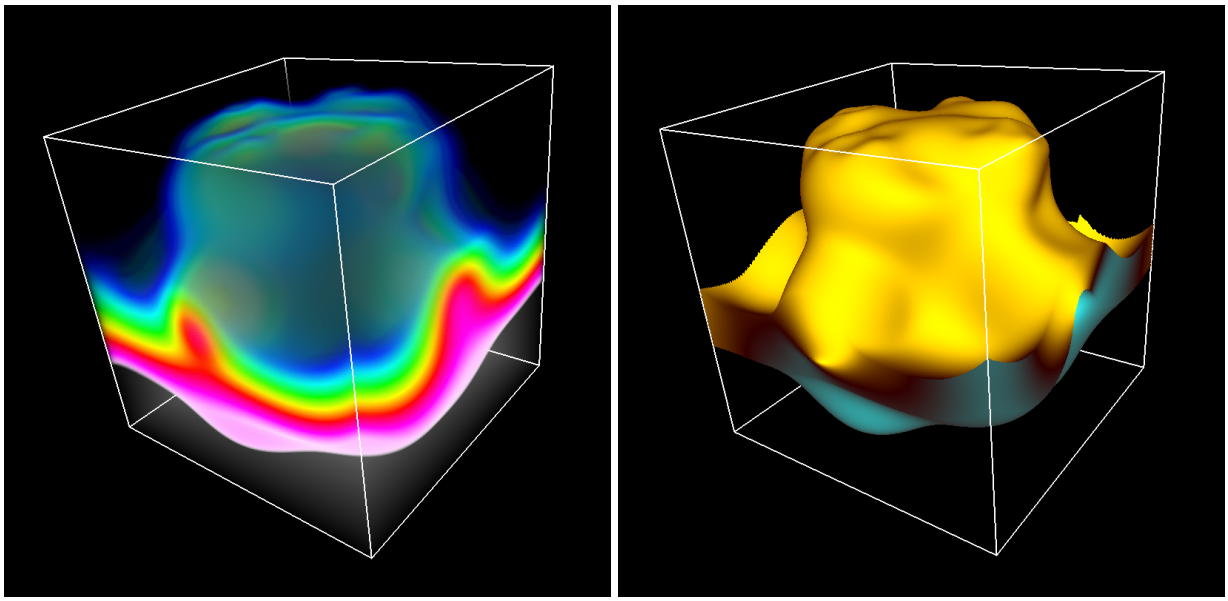


Abbildung 6.5: Volumen- und Isoflächendarstellung eines Temperaturfeldes, welches durch Simulation der natürlichen Konvektion erzeugt wurde.

Die Simulation wurde am Computational Fluid Dynamics Laboratory der Universität von Texas in Austin auf einem Gitter aus 48.000 Tetraedern berechnet. Die semi-transparente Volumendarstellung basiert auf einer Kodierung mittels 435 RBFs, die in einer Hierarchie mit maximal 100 RBFs je Zelle organisiert sind. Die in diesem Fall auf einer GeForceFX 5900 Ultra erzielbare Geschwindigkeit beträgt 1,8 fps für 32 Slices. Die verhältnismäßig schlechte Performanz ergibt sich aus der Tatsache, dass dieser Datensatz eine sehr gleichmäßige Verteilung der RBFs über den gesamten Bereich des Volumens besitzt, was die Effektivität der hierarchischen Strukturierung beschränkt und für jedes Fragment zu einer hohen Anzahl von auszuwertenden RBF-Funktionen führt. Für die anderen untersuchten Datensätze wurden auf der GeForceFX 5900 Ultra durchgehend interaktive Bildwiederholraten von mindestens 2 fps für 64 Slices erzielt. Die Isoflächendarstellung demonstriert darüber hinaus die hohe Bildqualität, die durch die Beleuchtungsberechnung mittels analytisch bestimmter Gradienten entsteht.

Die Vorzüge der Splat-basierten Darstellungstechnik zeigt Abbildung 6.6 am Beispiel des X38 Schock-Datensatzes, welcher einen einzelnen Zeitschritt im Wiedereintrittsprozess des NASA Rückkehr-Transporters X38 repräsentiert. Dazu werden die Darstellungen mittels 64 Schnittebenen auf einer GeForce 6800 GT Graphikkarte verglichen. In diesem Fall konnte die Performanz von 10,5 fps unter Verwendung einer Hierarchie von 204 Zellen mit bis zu 60 RBFs pro Zelle auf 44,3 fps bei Splat-basierter Darstellung mit vier Unterteilungsstufen für die Splat-Polygone erhöht werden. Diese Beschleunigung ergibt sich durch die vergleichsweise geringe Größe der Splat-Polygone, die links in Abbildung 6.6 zu sehen sind. Ähnliche Beschleunigungen konnten auch für andere Datensätze mit einer großen Anzahl von RBFs mit räumlich lokalem Einflussbereich erzielt werden. Ein weiterer Vorteil der Splat-basierten Darstellung ergibt sich aus der Möglichkeit der dynamisch zur Laufzeit wählbaren Fehlerschwelle für die Berechnung der

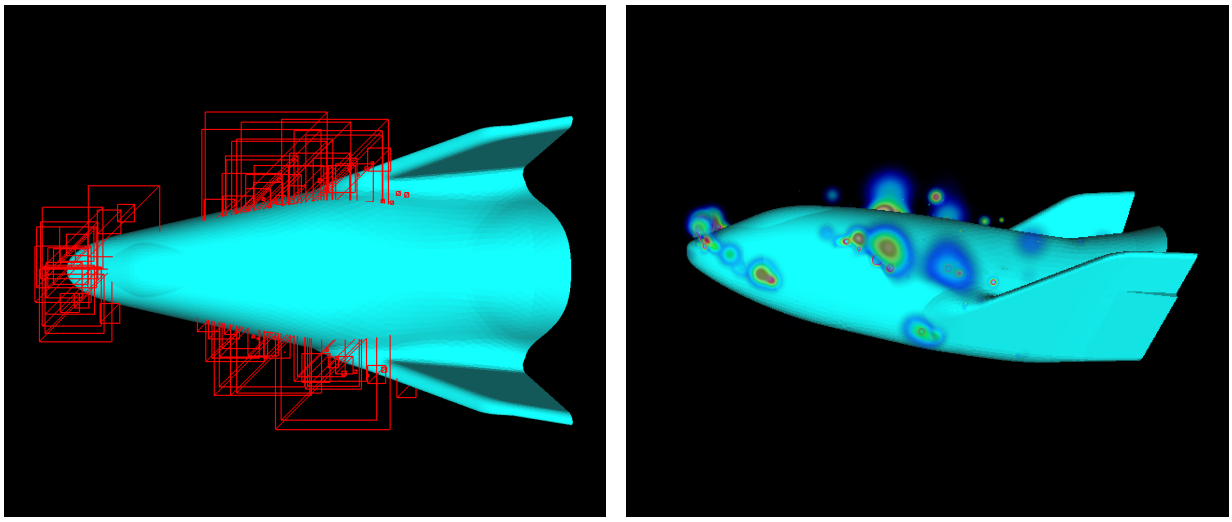


Abbildung 6.6: Volumenvisualisierung des mit 1.173 RBFs kodierten X38 Schock-Datensatzes mittels der Splat-basierten Darstellungsmethode. Im linken Bild sind die Splat-Polygone für diesen Datensatz zu sehen, welche die Einflussscheiben der RBF-Zentren mit einem absoluten Fehler von 0,0004 repräsentieren.

Einflussscheibe. Während der Interaktion kann diese Fehlerschwelle heraufgesetzt werden, was zu kleineren Splat-Polygonen und damit deutlich höheren Interaktionsraten führt, wobei die Eigenschaften des Datensatzes immer noch gut erkennbar sind. Für Datensätze mit einer großen Anzahl von RBFs mit globalem Einfluss ist die Splat-basierte Darstellung in der Regel langsamer, da in diesem Fall der Mehraufwand für die zusätzlichen Polygone und das Zeichnen mit aktiviertem Tiefentest durch den gesparten Rasterisierungsaufwand nicht ausgeglichen wird.

Abbildung 6.7 demonstriert spezielle Visualisierungstechniken für RBF-kodierte Vektordaten. Hierbei wird ausgenutzt, dass programmierbare Fragmentoperationen in der Regel auf einem vier-komponentigen Vektor arbeiten und mithin die Auswertung einer vektoriellen Größe in der RBF-Rekonstruktion nahezu keinen Mehraufwand im Vergleich zu der Rekonstruktion von Skalardaten erfordert. Für die Darstellung mittels Volumenvisualisierung muss eine skalare Größe aus dem Vektorfeld abgeleitet werden. Dies kann beispielsweise ein Geschwindigkeitsbetrag sein oder eine einzelne Komponente des Vektorfeldes.

Darüber hinaus erlaubt die RBF-Repräsentation auch die Extraktion von charakteristischen Merkmalen des Vektorfeldes zur Laufzeit. Unter Ausnutzung der in einer RBF-Repräsentation analytisch definierten Ableitung lässt sich ein Fragmentprogramm konstruieren, welches den Gradiententensor (Jacobimatrix) des Vektorfelds bestimmt und in einem zweiten Schritt charakteristische Größen, wie die Vortizität als Rotationsanteil der Jacobimatrix [Haimes und Kenwright 1999] oder das in der Strömungsmechanik weit verbreitete λ_2-Wirbelkriterium [Jeong und Hussain 1995, Müller et al. 1998], ableitet und als Farbwert auf die gezeichneten Slice-Polygone abbildet. Die Möglichkeit zur Berechnung des λ_2-Kriteriums mittels Graphik-Hardware wurde erstmals in [Stegmaier und Ertl 2004] demonstriert.

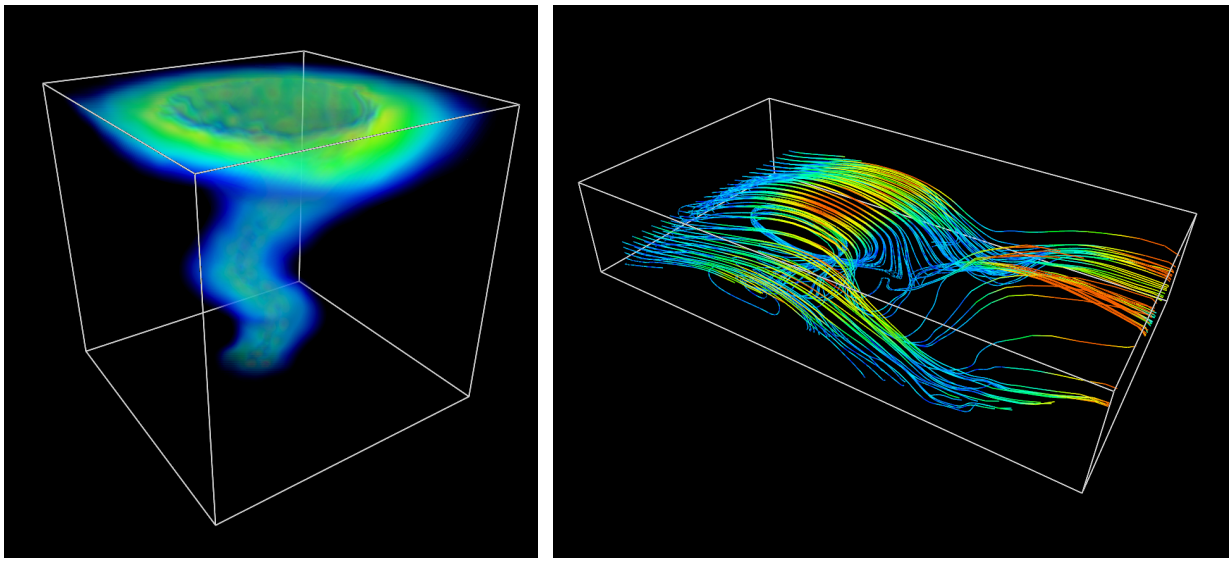


Abbildung 6.7: Visualisierungen RBF-kodierter Vektordaten. Das Bild des mit 1.793 RBFs kodierten Tornado-Datensatzes links zeigt den auf Farbwerte abgebildeten Geschwindigkeitsbetrag. Die Volumendarstellung verwendet 150 Slices. Rechts sind 110 Partikelbahnen zu sehen, welche mittels Graphik-Hardware direkt auf einer RBF-Repräsentation des Lang-Datenensatzes mit 2.105 RBFs berechnet wurden.

Die Berechnung von Strömungslinien auf der Graphik-Hardware, die in Abbildung 6.7 rechts dargestellt ist, nutzt deren universelle Berechnungseigenschaften, indem eine Liste mit Partikelpositionen als Textur auf die Graphikkarte geladen wird. Das Zeichnen eines Rechtecks in der Größe dieser Textur initiiert dabei die Berechnung der jeweils nächsten Partikelposition, die in einen p-Buffer geschrieben wird und durch Binden des p-Buffers als Textur (nach dem Ping-Pong-Rendering-Schema) als Eingabe für die Berechnung des nächsten Zeitschrittes zur Verfügung steht. Das dabei zum Einsatz kommende Fragmentprogramm rekonstruiert das Geschwindigkeitsfeld aus der RBF-Repräsentation und führt darauf aufbauend einen Euler-Integrationsschritt aus.

Die effiziente Darstellung dieser Partikelbahnen ist momentan jedoch nur unter Ausnutzung der sogenannten *uberbuffer*-Erweiterung [Mace 2004] von ATI möglich. Diese erlaubt die direkte Definition von Vertex Arrays aus dem Bildspeicher. Ohne diese Erweiterung ist für jeden Zeitschritt ein teures Auslesen der neuen Partikelpositionen mittels `glReadPixels` notwendig. Auf einer nVidia-Karte wie der GeForce 6800 GT, die diese Erweiterung nicht unterstützt, benötigt die Berechung für 400 Partikel, die über 400 Zeitschritte verfolgt werden, daher etwa 5 Sekunden. Eine Möglichkeit zur Beschleunigung könnten die von der nVidia GeForce6 Chip-Serie unterstützten Texturnachschläge auf Vertexebene bieten. Beispielsweise könnten die Partikelbahnen in einzelnen Segmenten gezeichnet werden. Ein Segment entspricht dabei der Verbindungslinie zwischen der Partikelposition zum Zeitpunkt t_i und der Position zum Zeitpunkt t_{i+1} und wird als Liniensegment mit beliebigen Vertexkoordinaten gezeichnet. Beim Rendering dieses Liniensegments könnte ein Vertexprogramm durch den Nachschlag in zwei Texturen, wel-

che die Position aller Partikel zum Zeitpunkt t_i beziehungsweise t_{i+1} enthalten, die jeweils korrekte Vertexposition ermitteln und entsprechend anpassen. Dazu muss jeder gezeichnete Vertex lediglich die Information besitzen, zu welcher Partikelbahn er gehört und ob er der Start- oder der Endpunkt des Liniensegments ist. Beides kann in einfacher Weise über geeignete Vertexattribute erreicht werden. Nachdem sich diese Information nicht ändert, kann zum Zeichnen ein statisches Vertex Array verwendet werden, welches ein Liniensegment für jede darzustellende Partikelbahn enthält. Es wird einmal für jeden Zeitschritt gezeichnet. Der wesentliche Vorteil dieser Idee liegt darin, dass als Texturen für den Nachschlag der Partikelpositionen im Vertexprogramm dieselben Texturen verwendet werden können, die auch für die Berechnung der Partikelbahnen vom Fragmentprogramm benutzt werden.

Im Vergleich der Implementierungen der RBF-Rekonstruktion für Graphik-Hardware mit und ohne die Möglichkeit für dynamische Schleifen zeigen sich nur bedingt Vorteile für die Lösung mit dynamischen Schleifen. Eigentlich stellen dynamische Schleifen die effizientere Implementierung dar, weil hier das Fragmentprogramm immer genau die richtige Anzahl von RBFs je Zelle auswerten kann und keine Null-RBFs berücksichtigen muss, die im Falle von statisch ausgerollten Schleifen als Füllinformation erforderlich sind. Für die einfache skalare RBF-Rekonstruktion, die nur acht Fragmentprogramminstruktionen innerhalb der Schleife benötigt, stellen die durch eine `REP`-Anweisung implizierten drei zusätzlichen Instruktionen je Schleifendurchlauf jedoch einen signifikanten Mehraufwand dar, der zu bis zu 30% niedrigeren Bildwiederholraten führt. Vorteile ergeben sich vor allem da, wo durch die Verwendung dynamischer Schleifen zusätzliche Zeichendurchgänge eingespart werden können. So erfordert beispielsweise die Rekonstruktion der Jacobimatrix auf einer nVidia GeForceFX drei Zeichendurchgänge, da diese Graphikkarte keine mehrfachen Render Targets anbietet und die neun Komponenten der Matrix nicht in einem einzelnen p-Buffer akkumuliert werden können. Potentiell erfordert jeder dieser drei Durchgänge, abhängig von der maximalen Anzahl der RBFs je Zelle, wiederum mehrere Zeichendurchgänge, was einen signifikanten Mehraufwand durch das Auslesen der Zwischenergebnisse sowie die Aktualisierung der Texturen erzeugt. Die Verwendung dynamischer Schleifen erlaubt demgegenüber die komplette Berechnung der Jacobimatrix inklusive der Berechnung abgeleiteter Größen wie Vortizität oder λ_2 in einem einzigen Zeichendurchgang. Dadurch ergibt sich eine Beschleunigung bis zu einem Faktor 3.

In der Summe stellt die hier präsentierte neuartige Technik einen interessanten universellen Ansatz zur interaktiven Rekonstruktion und Visualisierung von beliebigen dreidimensionalen Skalar- und Vektorfeldern dar. Er ist weitgehend unabängig von der Gitterstruktur der Daten und kann damit sowohl auf uniforme Voxeldaten wie auch auf unstrukturierte Gitter oder punktbasierte Repräsentationen angewendet werden. Durch die Speicherung der Daten als kompakte Liste von RBF-Koeffizienten direkt auf der Graphikkarte beseitigt der Ansatz die Datenübertragung als Flaschenhals und kann damit den vollen Vorteil aus dem Geschwindigkeitszuwachs neuer Graphikkartengenerationen ziehen. Die Flexibilität und Erweiterbarkeit der funktionalen Kodierung und der Rekonstruktion erlaubt die interaktive Erforschung sehr großer Datensätze aus einer Vielzahl von Quellen, was auch die interaktive Erkennung von charakteristischen Merkmalen von Vektordaten mit einbezieht.

Ein wesentlicher Nachteil des Verfahrens liegt allerdings in dem hohen Rasterisierungsaufwand, der durch die Auswertung der RBF-Rekonstruktion für jedes Fragment entsteht. Da dieser Aufwand signifikant größer ist als ein schlichter Texturnachschlag, erreicht das Verfahren deutlich niedrigere Interaktionsraten als die Algorithmen nach dem klassischen Muster der texturbasierten Volumenvisualisierung. Letztere waren jedoch bisher auf uniforme Gitter als Datenrepräsentation beschränkt. In den nächsten Kapiteln wird untersucht, wie die klassische texturbasierte Volumenvisualisierung mit adaptiven Datenstrukturen so kombiniert werden kann, dass sowohl die Vorzüge der adaptiven Repräsentation, wie geringer Speicherbedarf bei hoher Genauigkeit, als auch die hohen Interaktionsraten der texturbasierten Volumenvisualisierung erhalten bleiben.

Kapitel 7

Hardware-Software-balanciertes Resampling

In den letzten Jahren hat es große Verbesserungen bei der interaktiven Visualisierung unstrukturierter Gitter gegeben. Einige Beispiele markieren die in den Kapiteln 4 und 5 vorgestellten Verfahren, die versuchen, unter Ausnutzung der Fähigkeiten neuester Graphik-Hardware die Darstellung zu beschleunigen. Alle diese Algorithmen erreichen jedoch immer noch nicht die interaktiven Bildwiederholraten, die bei der Visualisierung von regulären Gittern mit texturbasierter Volumenvisualisierung [Cabral et al. 1994, Cullip und Neumann 1994] auf Standard-Graphik-Hardware oder mit Hilfe spezieller Volume Rendering Hardware [Pfister et al. 1999] erzielt werden. Ein Resampling der unstrukturierten Volumendaten auf kartesische Gitter erscheint daher als vielversprechender Ansatz für das Erreichen interaktiver Bildwiederholraten.

Für ein genaues Resampling muss jedoch die Auflösung des regulären Gitters an die Größe der kleinsten Gitterzelle angepasst werden, was für Datensätze praktischer Größe zu einer enormen Datenmenge führt. Gleichzeitig können die Zellen im unstrukturierten Gitter sehr unterschiedliche Größe besitzen, wobei große Zellen Bereiche mit geringer beziehungsweise linearer Variation der Datenwerte erzeugen, die auch mit geringerer Auflösung repräsentierbar wären. Die Hardware-unterstützte Darstellung solch großer Datensätze wird durch den limitierten Texturspeicher erschwert. Sobald Teile des Datensatzes nachgeladen werden müssen, sinkt die Bildwiederholrate signifikant.

Ein adaptives Resampling, basierend auf einer Oktalbaum-Hierarchie oder einer Datenstruktur nach dem Muster der adaptiven Gitterverfeinerung, ist daher dringend angezeigt. Statt eines einzelnen regulären Gitters wird dabei eine Hierarchie von Gittern mit unterschiedlicher Auflösung verwendet, um ein unstrukturiertes Gitter zu rekonstruieren. Die Aufösung für eine bestimmte Region kann dabei entweder statisch, abhängig von Charakteristiken des unstrukturierten Gitters, oder dynamisch, z.B. abhängig vom Fokus des Betrachters, gewählt werden. Da Letzterer im Allgemeinen nicht im Voraus bekannt ist und sich während der Visualisierung ändern kann, ist es wichtig, über einen interaktiven Resampling-Algorithmus zu verfügen, der benötigte Auflösungsstufen auf Anforderung erzeugt. Anderenfalls muss die volle Auflösungsstufe für den gesamten Datenbereich gespeichert werden, was zu einem gigantischen Speichermehraufwand und langen Ladezeiten führt.

Die schnellsten Resampling-Algorithmen, die in der Literatur zu finden sind, nutzen die Graphik-Hardware und basieren auf Slicing. Da solche Ansätze eine sehr große Menge von Polygonen zeichnen, sind sie massiv durch den limitierten Datentransfer zur Graphikkarte beschränkt. Ein lohnenswertes Ziel ist daher, wie bereits in Kapitel 4 thematisiert, die Reduzierung der Anzahl der Polygone, die für das Resampling erforderlich sind. Auf der anderen Seite besitzt Hardware-beschleunigtes Resampling den Nachteil, dass die Kapazität der immer schneller werdenden Hauptprozessoren nicht genutzt wird. Die CPU verkommt zu einem reinen Lieferanten von Dreiecken. Für eine möglichst hohe Resampling-Geschwindigkeit ist es erforderlich, eine geeignete Balance zwischen der CPU und dem Graphikprozessor (Graphics Processing Unit – GPU) zu finden.

Dieses Kapitel beschäftigt sich mit dem Problem der Lastverteilung zwischen CPU und GPU im Rahmen der Erstellung einer Resampling-Hierarchie. Dazu werden drei Algorithmen vorgestellt und untersucht. Einer davon verwendet ausschließlich die CPU, während die anderen beiden versuchen, die Graphik-Hardware auszunutzen. Der Unterschied zwischen den beiden liegt in der Verteilung der Rechenlast zwischen CPU und GPU. Die Evaluierung dieser Algorithmen erlaubt eine Aussage über die optimale Lastverteilung. Dieser Teil der Arbeit wurde auch in [Weiler und Ertl 2001] veröffentlicht.

7.1 Vorarbeiten

Bei den Techniken zum Resampling unstrukturierter Gitter können Bildraum- (Image Order) und Objektraumverfahren (Object Order) unterschieden werden. Algorithmen, die im Bildraum arbeiten, iterieren über alle Punkte des regulären Zielgitters und versuchen, für diese Gitterpunkte den Datenwert zu ermitteln. Dies beinhaltet zwei Aufgaben. Zunächst muss untersucht werden, welche Zelle die gegebene Gitterposition umschließt. Dies wird auch als Problem der Point Location bezeichnet. Der zweite Schritt ist die Interpolation des Datenwertes innerhalb der gefundenen Zelle. Der wesentliche Nachteil dieses Ansatzes liegt in der aufwändigen Zellsuche. Diese kann zwar mit Hilfe eines Oktalbaums [Wilhelms und Van Gelder 1992] deutlich beschleunigt werden, dominiert aber immer noch gegenüber der billigen Dateninterpolation den algorithmischen Aufwand. Eine andere Möglichkeit der Beschleunigung besteht in der Ausnutzung der Konnektivität innerhalb des unstrukturierten Gitters. Die Konnektivitätsinformationen erzeugen jedoch einen Mehraufwand an Speicher und ihre Initialisierung benötigt Rechenzeit. Bei zeitlich veränderlichen Gittern kann die Konnektivität für jeden Zeitschritt unterschiedlich sein.

Algorithmen, die im Objektraum arbeiten, iterieren über alle Zellen des unstrukturierten Gitters und ermitteln alle regulären Gitterpunkte, die von dieser Zelle umschlossen werden. Jeder dieser Gitterpunkte erhält den aus den Knoten der Zelle interpolierten Datenwert. In dieser Hinsicht ist Resampling im Ortsraum eng mit dem Problem der 3D-Scan-Konvertierung verwandt, die auch unter dem Begriff Voxelisierung bekannt ist und relativ gut erforscht ist. Neben Software-Lösungen [Kaufmann und Shimony 1986], die im Wesentlichen die 2D-Scan-Konvertierung auf den 3D-Fall verallgemeinern, haben in letzter Zeit mehrere Autoren versucht, Graphik-Hardware für die schnelle Voxelisierung einzusetzen.

Ähnlich wie bei Techniken zur direkten Darstellung unstrukturierter Gitter [Yagel et al. 1996, Wilhelms et al. 1996] wird dabei Slicing verwendet, um von der Hardware-beschleunigten Polygonrasterisierung der Graphik-Hardware zu profitieren. Das reguläre Gitter, auf welchem das unstrukturierte Gitter abgetastet werden soll, wird dabei in Schichten (Slices) senkrecht zur z-Achse betrachtet. Jeweils ein Slice steht für eine Schicht von Voxeln. Für jeden Slice wird der Schnitt mit allen Zellen des unstrukturierten Gitters bestimmt. Während dem anschließenden Zeichnen der Menge von Schnittpolygonen erfolgt die Dateninterpolation durch die Graphik-Hardware im Rahmen der Rasterisierung ohne Eingriff durch den Hauptprozessor. Das Ergebnis kann aus dem Framebuffer zurückgelesen werden oder direkt in eine Textur geladen werden, um für die Visualisierung zur Verfügung zu stehen.

Die Berechnung der Schnittpolygone kann entweder explizit wie in [Yagel et al. 1996, Wilhelms et al. 1996] erfolgen oder implizit pro Pixel unter Ausnutzung von Fragmentoperationen und Fragmenttests der Graphik-Hardware [Westermann und Ertl 1998]. In [Westermann 2001] findet sich ein aktuellerer Ansatz dazu, der jedoch die Berechnung der Unterteilung der projizierten Tetraeder entsprechend der Shirley-Tuchman-Klassifikation [Shirley und Tuchman 1990] voraussetzt. Letzteres erzeugt einen Mehraufwand in der Berechnung und durch die Notwendigkeit, diese Unterteilung zweimal je Slice zu zeichnen, leidet dieser Ansatz auch unter einer hohen Polygonanzahl, welche die Anzahl der Zellen im unstrukturierten Gitter bei weitem überschreitet.

7.2 Hardware-unterstütztes Slicing

Die folgende Darstellung geht davon aus, dass das Resampling eines Tetraedergitters in eine Hierarchie von regulären Gittern erfolgen soll. Nicht-Tetraederzellen allgemeiner unstrukturierter Gitter müssen dazu zunächst tesseliert, d.h. in Tetraeder zerlegt werden. Als Kern dieses Resampling-Prozesses wird das Abtasten (von Teilen) des Tetraedergitters in ein einzelnes reguläres Gitter betrachtet. Der erste Algorithmus, der hier vorgestellt werden soll, macht exzessiv von der Graphik-Hardware Gebrauch, indem er eine Slicing-Technik, ähnlich wie in [Yagel et al. 1996, Wilhelms et al. 1996] beschrieben, einsetzt.

Allerdings wird hier weder das Schnittpolygon explizit berechnet noch müssen mehr als zwei Polygone pro Tetraeder gezeichnet werden. Dies stellt einen Vorteil gegenüber der Lösung in [Westermann 2001] dar. Tatsächlich besteht die Grundidee des Algorithmus darin, einfach ein etwas größeres Polygon zu zeichnen und den Schnitt mit dem Tetraeder durch Fragmentoperationen zu bestimmen. Das gezeichnete Polygon muss nicht notwendigerweise aus Flächen des Tetraeders bestehen und wird im Folgenden als Ersatzgeometrie bezeichnet.

Durch Ausnutzung von Texturierung und programmierbarer Fragmentoperationen wird sichergestellt, dass nur Fragmente der Ersatzgeometrie gezeichnet werden, die auf dem Schnittpolygon mit dem Tetraeder liegen. Weiterhin muss die korrekte Interpolation der Datenwerte für die gezeichneten Fragmente erfolgen. Beides kann durch die Verwendung baryzentrischer Koordinaten als Texturkoordinaten erreicht werden (siehe Abbildung 7.1).

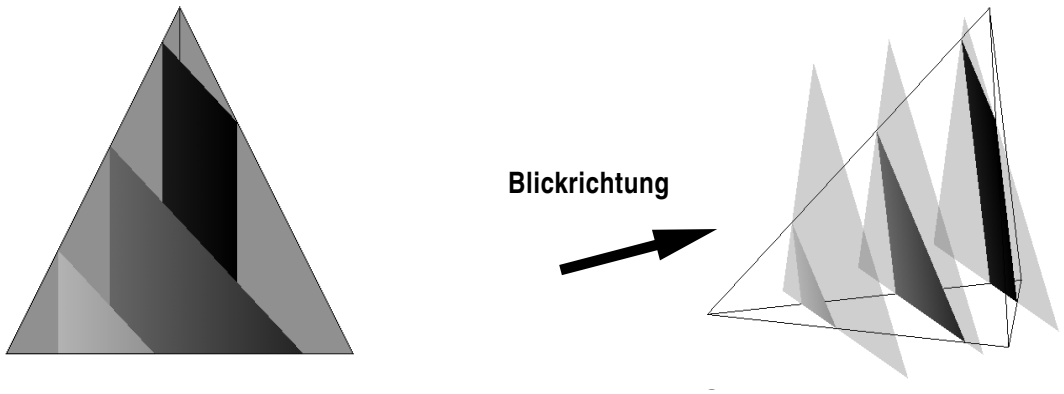


Abbildung 7.1: Demonstriert wird das Grundprinzip für das Hardware-unterstützte Slicing. Wie links zu sehen, erfolgt das Slicing senkrecht zur Blickrichtung mit Hilfe einer Ersatzgeometrie (Silhouette des Tetraeders). Texturierung und Alpha-Test sorgen dafür, dass nur Fragmente, die Teil des jeweiligen Schnittpolygons sind, tatsächlich gezeichnet werden. Rechts ist die zugehörige Situation im Texturraum dargestellt. Die Texturkoordinaten entstehen durch Projektion der Vertizes der Ersatzgeometrie auf die Slice-Ebene und sind als baryzentrische Koordinaten kodiert.

7.2.1 Ersatzgeometrie

Für optimale Resampling-Performanz muss das Polygon, welches für jeden Tetraeder gezeichnet wird, sorgfältig gewählt werden. Dies erfolgt unter dem Gesichtspunkt des Aufwands für seine Berechnung und der Größe seiner Projektion in die Bildebene. Einerseits muss es mindestens die Silhouette des betrachteten Tetraeders überdecken. Andererseits resultiert ein größeres Polygon in höheren Kosten für die Rasterisierung, was sich negativ auf die Geschwindigkeit auswirkt.

Daher wird ein Dreiecksstreifen als Ersatzgeometrie verwendet, der entweder die Front Faces oder die Back Faces des Tetraeders beschreibt. Die Wahl zwischen Front Faces und Back Faces erfolgt auf der Basis der minimalen Polygonanzahl. In der Regel werden die Front Faces verwendet, es sei denn ihre Anzahl ist gößer als die der Back Faces. Durch dieses Schema wird sichergestellt, dass die Projektion der Ersatzgeometrie auf die Bildebene mit der Silhouette des Tetraeders übereinstimmt und entweder ein oder zwei Dreiecke gezeichnet werden, die – kodiert als Dreiecksstreifen – drei oder vier Vertizes besitzen.

Weiterhin braucht kein Vertex des Dreiecksstreifen berechnet zu werden, da sie alle durch die Tetraeder-Datenstruktur definiert sind. Die Reihenfolge der Vertizes für den Dreiecksstreifen kann durch eine Tabelle ermittelt werden, die alle möglichen Vertexreihenfolgen enthält und auf die mit der Klassifikation der Tetraederseiten nach Front und Back Face zugegriffen wird.

Zum Vergleich sei angemerkt, dass die Shirley-Tuchman-Unterteilung im Durchschnitt zwei Dreiecke und zwei Vertizes mehr benötigt. Sie erzeugt entweder drei oder vier Dreiecke, die als Streifen aus fünf oder sechs Vertizes gezeichnet werden. Einer dieser Vertizes, der innere Ver-

tex der Shirley-Tuchman-Unterteilung, muss als Schnittpunkt der Projektion zweier ausgewählter Kanten des Tetraeders auf die Bildebene berechnet werden. Dies, obgleich nicht sonderlich aufwändig, stellt einen signifikanten Mehraufwand im Vergleich zu dem oben erwähnten Tabellennachschlag dar.

7.2.2 Identifizierung des Schnittbereichs

Wie im vorigen Abschnitt beschrieben, werden durch die Rasterisierung der Ersatzgeometrie Fragmente für alle Pixel, die von der Projektion der Tetraedersilhouette überdeckt werden, erzeugt. An dieser Stelle muss gewährleistet werden, dass nur für solche Pixel Datenwerte in den Framebuffer geschrieben werden, die vom Schnittpolygon des Tetraeders mit dem Slice überdeckt werden. Dies erfordert eine geeignete Charakterisierung für Punkte innerhalb eines Tetraeders, was hier mit Hilfe baryzentrischer Koordinaten gelingt. Es gilt, dass ein 3D-Punkt genau dann im Inneren eines Tetraeders liegt, wenn alle vier Komponenten seiner baryzentrischen Koordinaten bezogen auf den betrachteten Tetraeder im Intervall $[0,1]$ liegen.

Dies führt zu einer eleganten Methode, um zu verhindern, dass Fragmente, die außerhalb des Schnittpolygons liegen, in den Framebuffer geschrieben werden. Für die Beschreibung wird zunächst angenommen, dass jedes Fragment mit seinen baryzentrischen Koordinaten bezogen auf den betrachteten Tetraeder ausgestattet werden könnte. In diesem Fall erlaubt der Einsatz geeigneter Fragmentoperationen, den Alpha-Wert für Fragmente im Inneren des Tetraeders auf 1 zu setzten und auf Werte kleiner als 1 für alle übrigen Fragmente. Letztere werden dann automatisch durch Aktivierung des OpenGL Alpha-Tests entfernt. Die erforderlichen Fragmentoperationen lassen sich heutzutage einfach mit Hilfe von Fragmentprogrammen realisieren. Hier soll jedoch eine Alternative mit Hilfe von Texturen vorgestellt werden, die lediglich die Möglichkeiten von OpenGL 1.2 und der OpenGL-Erweiterung `NV_register_combiners` [OpenGL ARB 2004] voraussetzt und damit bereits auf einem nVidia GeForce256 Chip realisierbar ist.

Der Ansatz verwendet eine Textur, die über die baryzentrischen Koordinaten, die als Texturkoordinaten dienen, parametrisiert wird und im Alpha-Kanal die Klassifikation für die Fragmente, wie sie gerade beschrieben wurde, enthält. Eine naive Implementierung würde jedoch eine vierdimensionale Textur erfordern, da die baryzentrischen Koordinaten im Dreidimensionalen aus vier Komponenten bestehen. Solche hochdimensionalen Texturen werden allerdings nur auf High-End-Graphiksystemen unterstützt. Außerdem stellt eine vierdimensionale Textur eine unnötige Verschwendung von Texturspeicher dar, was aus folgender Überlegung klar wird:

Betrachten wir die einzelnen Komponenten der baryzentrischen Koordinaten unabhängig voneinander: Ist zunächst der Wert der ersten Komponente im Intervall $[0,1]$ enthalten, so liegt das Fragment möglicherweise innerhalb des Tetraeders, im anderen Fall definitiv nicht. Dasselbe gilt für die übrigen Komponenten. Unter Berücksichtigung der oben erfolgten Definition für die Alpha-Werte kann damit eine individuelle Abbildung jeder Komponente b_i $(i \in \{0,1,2,3\})$ der baryzentrischen Koordinaten auf entsprechende Werte α_i erfolgen. Die vollständige Klassifikation ergibt sich aus der Multiplikation der α_i. Nur wenn jeder Faktor gleich 1 ist, ergibt sich als Produkt ebenfalls 1, was voraussetzt, dass alle Komponenten der baryzentrischen Koordinaten

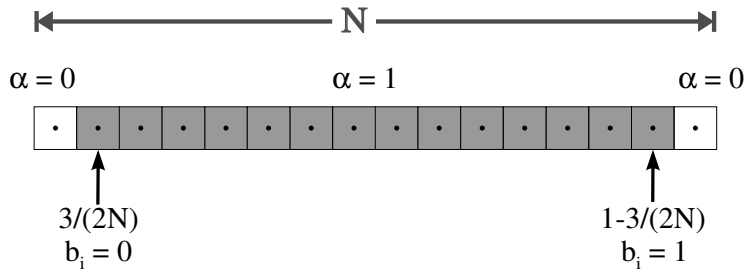


Abbildung 7.2: Textur, welche eine einzelne Komponente der baryzentrischen Koordinaten auf Alpha-Werte entsprechend der Klassifikation für innerhalb des Tetraeders liegende Fragmente abbildet.

im Intervall $[0,1]$ liegen. Dies kann ausgenutzt werden, um die Klassifikation der Fragmente mit Hilfe von Multi-Texturen und der Texturumgebung `GL_MODULATE` auszuwerten. Dabei wird die in Abbildung 7.2 dargestellte eindimensionale Textur verwendet, was konzeptionell allerdings die Verfügbarkeit von vier Texturstufen voraussetzt.

Nur das jeweils erste und letzte Element der Textur besitzt einen Alpha-Wert von 0. Alle anderen Elemente werden auf $\alpha = 1$ gesetzt. Eine einfache lineare Abbildung der baryzentrischen Koordinaten b_i auf Texturkoordinaten t_i stellt sicher, dass $b_i = 0$ und $b_i = 1$ auf den Mittelpunkt des ersten beziehungsweise letzten Texturelements mit $\alpha = 1$ gesetzt werden:

$$t_i = b_i \left(1 - \frac{3}{N}\right) + \frac{3}{2N} \tag{7.1}$$

Sie wird unter Verwendung der OpenGL-Texturmatrix, mit der alle Texturkoordinaten multipliziert werden, realisiert. Anstelle der Definition eines Rahmens aus transparenten Texeln wäre auch die Verwendung der in OpenGL definierten Texture Border möglich. Dies führt jedoch zu unnötigen Hardware-Einschränkungen, da diese Funktionalität nicht auf jeder Graphik-Hardware unterstützt wird.

Ein weiteres Problem ergibt sich aus den erforderlichen vier Texturstufen. Erste Graphikkarten mit Unterstützung für Multi-Texturen boten lediglich zwei Texturstufen beziehungsweise waren für die Verwendung von zwei Texturstufen optimiert. Auch auf aktueller Graphik-Hardware sollte zur Vermeidung unnötiger Latenz die Anzahl der Texturnachschläge minimal sein. Für die Implementierung bietet es sich daher an, zweidimensionale Texturen zu verwenden. Die vier Komponenten der baryzentrischen Koordinaten werden dabei auf zwei zweidimensionale Texturen verteilt, wobei die erste Textur die Komponenten (b_0, b_1) und die zweite Textur (b_2, b_3) als Texturkoordinaten verwendet. Der Inhalt beider Texturen ist identisch und entsteht durch das Tensorprodukt der vorher beschriebenen eindimensionalen Funktion mit sich selbst. Es entsteht eine kompakte Textur aus Elementen mit $\alpha = 1$ und einem ein Texel breiten transparenten Rand.

7.2.3 Rasterisierung des Schnittpolygons

Zur Rasterisierung des Schnittpolygons muss lediglich für jeden Slice die Ersatzgeometrie mit dem soeben beschriebenen Texturen gezeichnet werden. Jedem Vertex der Ersatzgeometrie werden dabei die vier baryzentrischen Koordinaten in Form von zwei Sätzen von zweidimensionalen Texturkoordinaten zugewiesen, jeweils ein Satz für die erste und zweite Texturstufe. Allerdings wird nicht die Position der Vertizes der gezeichneten Dreiecke als Basis für die Ermittlung der baryzentrischen Koordinaten verwendet. Stattdessen erfolgt die Berechnung basierend auf der Projektion dieser Punkte auf die aktuelle Slice-Ebene. Dies wird in Abbildung 7.3 verdeutlicht und führt zur korrekten Darstellung des gesuchten Schnittpolygons.

Die graue Linie rechts außen in Abbildung 7.3 entspricht der gezeichneten Ersatzgeometrie. Die leicht versetzte Linie links davon entspricht den Fragmenten, die den Alpha-Test passieren. Dieser nicht durch den Alpha-Test entfernte Bereich der Ersatzgeometrie überdeckt, projiziert auf die Bildebene, exakt dieselben Pixel wie das Schnittpolygon zwischen Slice-Ebene und Tetraeder. Wichtig dabei ist, darauf hinzuweisen, dass dies nur funktioniert, da die baryzentrischen Koordinaten genauso wie die Texturkoordinaten linear entlang der gezeichneten Dreiecke variieren und dass für die korrekte Behandlung von negativen baryzentrischen Koordinaten eine Clamp-Texturumgebung verwendet werden muss.

Es sei weiterhin bemerkt, dass, obwohl das Resampling auf einem regulären Gitter mit orthographischer Projektion erfolgen muss, das Zeichnen der Schnittpolygone im Prinzip auch mit perspektivischer Projektion erfolgen kann und dass auch die Orientierung der Slice-Ebene im Prinzip beliebig sein kann. Damit kann die Methode, wie in [Speray und Kennon 1990] vorgeschlagen, auch zum Zeichnen von beliebig orientierten Slice-Proben innerhalb eines unstrukturierten Gitters verwendet werden. Dies ist in Abbildung 7.4 dargestellt.

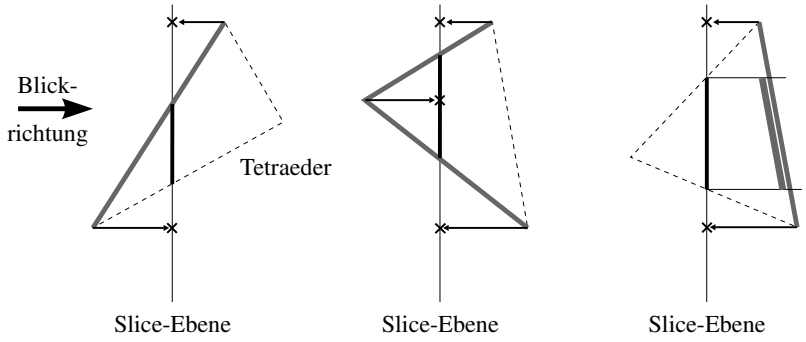


Abbildung 7.3: Seitenansicht der drei unterschiedlichen Fälle von Ersatzgeometrie. Die Fälle sind von links nach rechts entsprechend der Anzahl von Front Faces sortiert. Die grauen Linien entsprechen den Tetraederflächen, die als Ersatzgeometrie gezeichnet werden. Durch geeignete Wahl der Texturkoordinaten, die konzeptionell durch Projektion der Vertizes auf die Slice-Ebene erfolgt, werden durch Texturabbildung und Alpha-Test nur die Fragmente in den Framebuffer geschrieben, die dem Schnittpolygon des Tetraeders mit der Slice-Ebene entsprechen.

7.2.4 Hardware-unterstützte Dateninterpolation

In den letzten beiden Abschnitten wurde gezeigt, dass sich bei der Rasterisierung der Ersatzgeometrie die Beschränkung auf Fragmente aus dem Schnitt zwischen Tetraeder und Slice-Ebene lediglich unter Ausnutzung der Standardfunktionalität von OpenGL 1.2 erreichen lässt. Dies trifft auf die Interpolation der Datenwerte nicht zu. Die hier vorgestellte Lösung nutzt die parametrisierbare Fragmentarithmetik, wie sie mit der nVidia GeForce256 Chip-Familie eingeführt wurden. Dieser Chip erlaubte erstmals mit Hilfe der OpenGL-Erweiterung `NV_register_combiner` [OpenGL ARB 2004] eine flexible Kombination von Fragmentfarbe und dem Ergebnis von Texturnachschlägen. Natürlich kann dasselbe Ergebnis auf aktueller Graphik-Hardware auch unter Verwendung programmierbarer Shader erreicht werden.

Für die korrekte Dateninterpolation werden an den Vertizes der Ersatzgeometrie, zusätzlich zu den bereits beschriebenen Attributen, noch die primäre und sekundäre Zeichenfarbe gesetzt. Die Farben werden jeweils pro Tetraeder definiert, wobei jedem Vertex, der zu einem Tetraeder gehörenden Ersatzgeometrie dieselbe Farbe zugewiesen wird. Im R- bzw. G-Kanal der primären Zeichenfarbe wird der Skalarwert am ersten bzw. zweiten Knoten des Tetraeders gespeichert. Entsprechend enthält der R- und G-Kanal der sekundären Zeichenfarbe die Skalarwerte des dritten und vierten Knotens. Die B- und Alpha-Kanäle beider Farben bleiben unbenutzt und werden mit Null initialisiert. Unter diesen Voraussetzungen kann die Interpolation der Datenwerte innerhalb des Tetraeders mit Hilfe des Skalarproduktes der baryzentrischen Koordinaten mit den nicht-verschwindenden Anteilen der primären und sekundären Zeichenfarbe bestimmt werden.

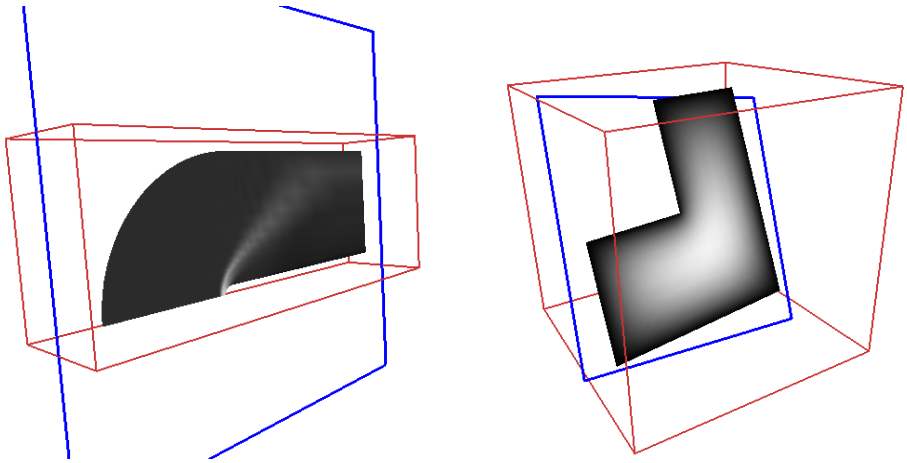


Abbildung 7.4: Beispiele für eine beliebig orientierbare Schnittebene, die mit Hilfe des Hardware-unterstützten Slicing-Algorithmus gezeichnet wird. Ein Oktalbaum erlaubt das schnelle Auffinden der von der Schnittebene getroffenen Tetraeder. Nur die Front Faces dieser Tetraeder werden gezeichnet, wobei die Graphik-Hardware über Fragmentoperationen die eigentlichen Schnittpolygone bestimmt und die Datenwerte interpoliert. In beiden Fällen werden auf einem AMD Athlon 900 Rechner mit GeForce2 GTS Graphikkarte interaktive Bildwiederholraten von mehr als 8 fps erreicht.

$$s(\vec{b}) = \sum_{i=0}^{3} b_i s_i = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \cdot \begin{pmatrix} R_P \\ G_P \end{pmatrix} + \begin{pmatrix} b_2 \\ b_3 \end{pmatrix} \cdot \begin{pmatrix} R_S \\ G_S \end{pmatrix} \tag{7.2}$$

Bei der Versorgung der Fragmentarithmetik-Einheit mit den baryzentrischen Koordinaten, die in den Texturkoordinaten der ersten und zweiten Texturstufe gespeichert sind, stellt sich das Problem, dass die Register Combiners keinen direkten Zugriff auf Texturkoordinaten vorsehen. Stattdessen stehen lediglich die Ergebnisse des Texturnachschlags zur Verfügung. Die oben beschriebenen Texturen werden daher derart modifiziert, dass im Ergebnis des Texturnachschlags der Wert der s- bzw. t-Texturkoordinate im R- bzw. G-Kanal ausgegeben wird. Dabei muss keine Rücksicht auf Koordinaten außerhalb des Tetraeders genommen werden. Zwar kann das Abschneiden der Texturwerte auf den Bereich $[0,1]$ zu falsch interpolierten Datenwerten führen, die Belegung des Alpha-Kanals stellt allerdings in Verbindung mit dem Alpha-Test sicher, dass kein falsch interpolierter Datenwert in den Framebuffer geschrieben wird. Die erforderliche Einstellung der Register Combiners ist in Abbildung 7.5 dargestellt.

7.2.5 Implementierungsdetails

Um eine optimale Resampling-Performanz zu erreichen, sollte die Anzahl der je Slice zu betrachtenden Tetraeder minimal sein. Dies kann, ähnlich wie in [Yagel et al. 1996] beschrieben, durch die Verwendung einer Liste der aktiven Tetraeder erreicht werden. Sobald die Orientierung der Slice-Ebene zum Tetraedergitter feststeht, wird der Datenbereich parallel zur Slice-Richtung in eine vorgegebene Anzahl von numerierten Streifen zerlegt. Jeder Tetraeder wird in maximal zwei Streifen eingetragen. Dabei handelt es sich um die beiden Streifen mit der höchsten bzw. niedrigsten Nummer, die noch Knoten des Tetraeders enthalten. Ein Tetraeder kann auch nur in einen Streifen eingetragen sein, wenn alle seine Knoten in diesen Streifen fallen.

Jedesmal, bevor ein neuer Slice bearbeitet wird, erfolgt zunächst eine Überprüfung, ob der Slice noch im aktuellen Streifen enthalten ist. Sollte das nicht der Fall sein, werden die Tetraeder des

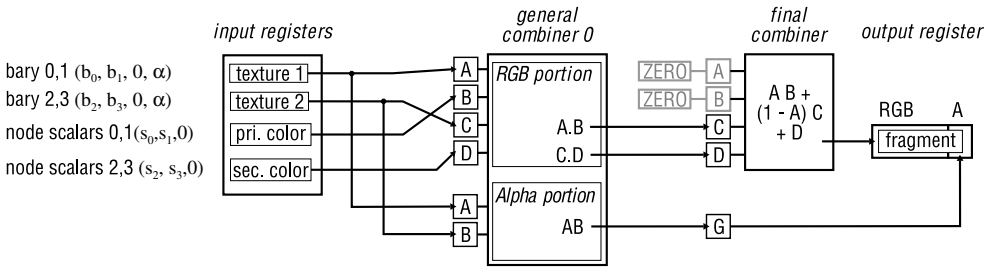


Abbildung 7.5: Einstellung der Register Combiners für die korrekte Dateninterpolation. Die RGB-Portion berechnet die interpolierten Skalarwerte, die Alpha-Portion simuliert die Modulate-Texturumgebung.

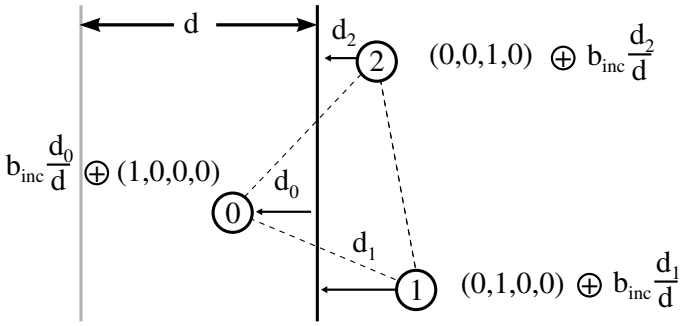


Abbildung 7.6: Berechnung der baryzentrischen Koordinaten eines auf die Slice-Ebene projizierten Vertex des Tetraeders. Sie kann ohne die eigentliche Projektion erfolgen, indem die skalierten Inkremente der baryzentrischen Koordinaten zu den implizit definierten baryzentrischen Koordinaten der Tetraedervertizes addiert werden.

neuen Streifens in die Liste der aktiven Tetraeder aufgenommen und solche, die bereits enthalten waren, stattdessen aus der Liste gelöscht. Nach diesem Schritt enthält die Liste der aktiven Tetraeder die Teilmenge von Tetraedern des gesamten Gitters, die potentiell vom aktuellen Slice getroffen werden. Nur diese müssen bearbeitet werden. Für Tetraeder, die nur in einem Streifen enthalten sind, ist eine Sonderbehandlung erforderlich, da sie durch das beschriebene Schema nicht mehr aus der Liste der aktiven Tetraeder entfernt werden.

Eine weitere Beschleunigung ergibt sich durch die inkrementelle Berechnung der baryzentrischen Koordinaten. Für jeden Tetraeder, der in die Liste der aktiven Tetraeder eingefügt wird, werden dabei die Inkremente der baryzentrischen Koordinaten entlang der Blickrichtung berechnet. Diese Inkremente spezifizieren, wie sich die baryzentrischen Koordinaten eines Punktes ändern, wenn er parallel zur Blickrichtung auf die nächste Slice-Ebene verschoben wird. Für einen definierten Slice-Abstand und einen vorgegebenen Tetraeder sind diese Werte konstant. Sie können aus den partiellen Ableitungen der Transformation von kartesischen in baryzentrische Koordinaten berechnet werden. Mit Hilfe dieser Inkremente ist die Berechnung der baryzentrischen Koordinaten eines Vertex der Ersatzgeometrie sehr einfach möglich. Dazu muss lediglich der inkrementelle Wert zu den Koordinaten, die im vorangegangenen Slice verwendet wurden, addiert werden.

Selbst für das erste Slice, das einen Tetraeder trifft, kann die explizite Berechnung der baryzentrischen Koordinaten für die projizierten Vertizes der Ersatzgeometrie vermieden werden. Stattdessen werden, wie in Abbildung 7.6 dargestellt, die baryzentrischen Koordinaten der Vertizes des Tetraeders als Ausgangspunkt verwendet und die mit dem Abstand des Vertex von der Slice-Ebene skalierten Inkremente addiert. Letztere müssen dazu noch durch dem Slice-Abstand dividiert werden. Die baryzentrischen Koordinaten des Vertex i eines Tetraeders sind implizit als Vektor gegeben, der an der Stelle i eine 1 besitzt und sonst 0. Somit können die baryzentrischen Koordinaten der auf die Slice-Ebene projizierten Vertizes ohne Kenntnis der Position dieser Punkte berechnet werden. Tatsächlich findet bei dem beschriebenen Ansatz niemals eine Projektion statt, was erheblich Rechenzeit spart.

7.3 Software-Resampling

Hardware-beschleunigtes Resampling, wie im vorangegangenen Abschnitt vorgestellt, ist sowohl durch die Rasterisierungsleistung der Graphikkarte als auch durch die Datentransferrate des Graphikbus beschränkt. Ein weiteres Problem ergibt sich aus der beschränkten Framebuffer-Auflösung sowie der internen Auflösung, die für das Compositing der Alpha-Werte auf der Graphikkarte verwendet wird. Beide beeinflussen die Genauigkeit des Resampling-Ergebnisses. Zwar sind mittlerweile PC-Graphikkarten, die mit Gleitkommagenauigkeit arbeiten, verfügbar, allerdings führt die Verwendung von Gleitkommawerten als Ausgabeformat zu deutlich geringerer Performanz im Vergleich zu der jahrelang üblichen Standardauflösung von 8 Bit je Farbkanal. Die interne Genauigkeit von aktuellen ATI-Graphikkarten ist immer noch auf 24 Bit beschränkt. Auf der anderen Seite kann Software-Resampling mit einfacher oder doppelter Gleitkommagenauigkeit durchgeführt werden. Zur näheren Untersuchung dieser Umstände soll das Hardware-beschleunigte Resampling mit einer reinen Software-Lösung verglichen werden.

Die hier vorgestellte Lösung verwendet einen Objektraumansatz, da dieser sich im Vergleich zu Bildraumverfahren als überlegen herausgestellt hat und bereits von vielen Autoren verwendet wurde [Prakash und Manohar 1995, Kaufmann und Shimony 1986]. Für jeden Tetraeder des unstrukturierten Gitters werden dabei diejenigen Voxel des Resampling-Gitters ermittelt, die von dem Tetraeder überdeckt werden. Für jeden überdeckten Voxel wird der Datenwert basierend auf den Skalarwerten der Vertizes dieses Tetraeders interpoliert und das Ergebnis an die richtige Stelle des Resampling-Gitters geschrieben. Die wesentlichen Vorteile dieses Ansatzes sind, dass Voxel des regulären Gitters, die von keinem Tetraeder getroffen werden, auch nicht angefasst werden müssen und dass die Voxelverarbeitung innerhalb eines Tetraeders sowohl räumliche Kohärenz als auch Kohärenz zwischen den Datenwerten ausnutzen kann.

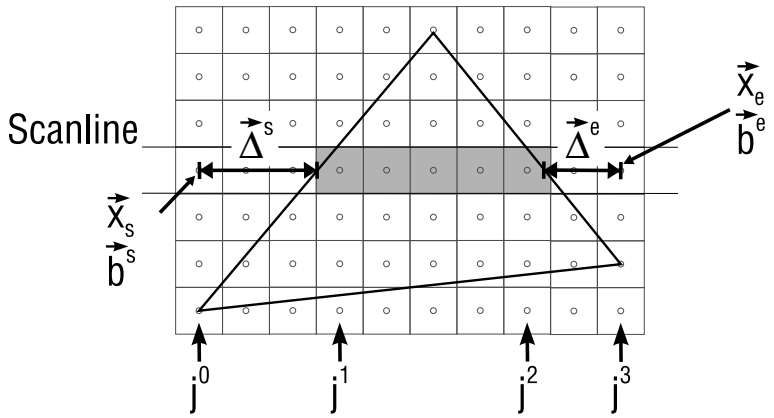


Abbildung 7.7: Berechnung der Indizes von Start- und Endvoxel innerhalb einer Scanline der Bounding Box des Tetraeders.

Entsprechend dieser Idee wird das Resampling durchgeführt, indem jeder Tetraeder des unstrukturierten Gitters individuell bearbeitet wird. Zunächst wird die Bounding Box des Tetraeders innerhalb des Resampling-Gitters bestimmt, die dann Scanline-basiert bearbeitet wird. Dabei wird die Orientierung der Scanline so gewählt, dass ihre Richtung mit der längsten Ausdehnung der Bounding Box übereinstimmt. Dies vergrößert die durchschnittliche Scanline-Länge und ermöglicht die bestmögliche Ausnutzung der Datenkohärenz. Für den verbleibenden Teil dieses Abschnitts wird ohne Beschränkung der Allgemeinheit von einer Scanline-Orientierung parallel zur x-Achse ausgegangen.

Wird eine einzelne Scanline innerhalb der Bounding Box eines Tetraeders betrachtet, so fällt auf, dass nur ein Teil der Linie tatsächlich innerhalb des Tetraeders liegt. Die Scanline startet typischerweise außerhalb, tritt an einem bestimmten Voxel (Startvoxel) in den Tetraeder ein, verlässt ihn wieder am Endvoxel und endet außerhalb des Tetraeders. Nachdem außen liegende Voxel nicht zum Resampling-Ergebnis beitragen, ist für ein schnelles Resampling ein effizientes Überspringen dieser Voxel erforderlich. Dies ist umso wichtiger, da die Anzahl der Voxel der Bounding Box außerhalb des Tetraeders die der Voxel im Inneren um den Faktor $5-6$ übersteigt. Deshalb werden hier die Indizes von Start- und Endvoxel explizit bestimmt.

Wie Abbildung 7.7 verdeutlicht, werden für jede Scanline Startvoxel j^0 und Endvoxel j^3 betrachtet. Zunächst erfolgt die Berechnung der Positionen $\vec{x}_s$ und $\vec{x}_e$ der Zentren dieser Voxel im Koordinatensystem des Tetraedergitters. Unter Verwendung der korrespondierenden baryzentrischen Koordinaten dieser Punkte $\vec{b^s}$ und $\vec{b^e}$ können nun komponentenweise Distanzvektoren $\vec{\Delta^s}$ und $\vec{\Delta^e}$ folgendermaßen bestimmt werden:

$$\Delta_i^s = \begin{cases} b_i^s - 1 & \text{wenn} \quad b_i^s > 1, \\ 0 & \text{wenn} \quad 0 \leq b_i^s \leq 1 \\ b_i^s & \text{sonst} \end{cases} \tag{7.3}$$

Der Vektor $\vec{\Delta^e}$ ergibt sich analog. Die Distanzvektoren bestehen aus vier Komponenten. Eine Komponente wird auf Null gesetzt, wenn die korrespondierende Komponente der baryzentrischen Koordinaten innerhalb des Intervalls $[0,1]$ liegt. Anderenfalls misst diese Komponente den Abstand zu diesem Intervall.

Mit Hilfe der partiellen Ableitung der Koordinatentransformation von kartesischen Koordinaten in baryzentrische Koordinaten bezogen auf den aktuellen Tetraeder gelingt es, auf einfache Weise die Indizes j^1 und j^2 des Start- und Endvoxels zu bestimmen.

$$j^1 = j^0 + \left\lceil \max_i \frac{-\Delta_i^s}{\frac{\partial b_i^s}{\partial x}} \right\rceil \qquad j^2 = j^3 - \left\lceil \max_i \frac{\Delta_i^e}{\frac{\partial b_i^e}{\partial x}} \right\rceil \tag{7.4}$$

Gleichung (7.4) setzt dabei voraus, dass die partiellen Ableitungen entsprechend dem Gitterabstand entlang der Scanline skaliert sind. Auf diese Weise lässt sich die Bearbeitung jeder Scanline auf die Voxel zwischen j^1 und j^2, einschließlich der Randvoxel, reduzieren. Zur Rechtfertigung der Vorzeichenwahl für die Argumente der Maximumfunktionen in Gleichung (7.4) sei bemerkt,

dass sich für Punkte auf der Scanline links vom Tetraeder folgendes zeigen lässt: Wenn eine Komponente Δ_i^s des Distanzvektors negativ ist, dann ist die korrespondierende Komponente der partiellen Ableitung positiv und umgekehrt. Für Punkte rechts vom Tetraeder besitzt jede Komponente Δ_i^e des Distanzvektors dasselbe Vorzeichen wie die partielle Ableitung.

Die Dateninterpolation in jedem Voxel entlang der Scanline kann von der linearen Variation der Datenwerte innerhalb eines Tetraeders profitieren. Aufgrund dieser Eigenschaft unterscheiden sich zwei benachbarte Voxel der Scanline lediglich um einen konstanten Term δ. Dieser kann für jeden Tetraeder vorberechnet werden und ergibt sich zu:

$$\delta = \left(\frac{\partial b_0}{\partial x}, \frac{\partial b_1}{\partial x}, \frac{\partial b_2}{\partial x}, \frac{\partial b_3}{\partial x} \right) \cdot (s_0, s_1, s_2, s_3) \tag{7.5}$$

Eine vollständige Interpolation des Datenwertes ist daher nur einmal am Beginn jeder Scanline notwendig. Für alle anderen Voxel der Scanline ist nur eine einzelne Addition erforderlich. In analoger Argumentation für die partiellen Ableitungen nach y und z ergibt sich, dass die Interpolation sogar nur einmal für die erste Scanline der Bounding Box ausgeführt werden muss und sich der Startwert für jede weitere Scanline jeweils durch einfache Addition ergibt.

Weitere Optimierungsmöglichkeiten liegen bei der Berechnung der baryzentrischen Koordinaten von Start- und Endpunkt der Scanline. Hier kann ebenfalls auf inkrementelle Berechnung zurückgegriffen werden, um ausgehend von den baryzentrischen Koordinaten des Startpunktes der letzten Scanline die Koordinaten für die aktuelle Scanline durch Addition der partiellen Ableitungen $\frac{\partial b_i}{\partial y}$ beziehungsweise $\frac{\partial b_i}{\partial z}$ zu ermitteln. Analoges gilt für die Endpunkte. Somit ist eine explizite Berechnung der baryzentrischen Koordinaten nur ein einziges Mal für die erste Scanline der Bounding Box des Tetraeders erforderlich.

7.4 Hybrides Resampling

Während das vorgestellte Hardware-unterstützte Resampling versucht, möglichst den gesamten Aufwand für die Berechnung auf die Graphikkarte zu verlagern, vertritt der Software-Ansatz das andere Extrem. Beide Ansätze lassen jeweils Kapazitäten der GPU oder der CPU ungenutzt. Es wird daher hier eine hybride Technik vorgeschlagen, die in optimaler Weise die Rechenleistung von CPU und GPU verbindet, um die beste Resampling-Performanz zu erzielen.

In Abschnitt 7.2 wurde darauf hingewiesen, dass der Hardware-unterstützte Ansatz hauptsächlich durch die Rasterisierungsleistung der Graphikkarte beschränkt ist, die unter anderem von der internen Bandbreite zum Graphikkartenspeicher bestimmt wird. Dies motiviert zwei Optimierungen:

1. Reduktion der für die Rasterisierung erforderlichen Speicherbandbreite durch Verzicht auf Texturnachschläge und das Zeichnen einfacher farbschattierter Dreiecke. Dies reduziert deutlich die Komplexität der Rasterisierung.

2. Reduktion der Anzahl verarbeiteter Fragmente durch die Beschränkung der Fläche der gezeichneten Polygone auf die Fläche der tatsächlich in den Framebuffer geschriebenen Pixel.

Letzteres induziert die explizite Berechnung der Schnittpolygone jedes Tetraeders mit der betrachteten Slice-Ebene. Dieses Schnittpolygon lässt sich als Isofläche eines Skalarfeldes betrachten, welches durch den Abstand von der aktuellen Slice-Ebene definiert wird. Dies erlaubt die Verwendung eines dem Marching-Cubes-Algorithmus [Lorensen und Cline 1987] verwandten Klassifikationsschemas, um ausgehend vom Vorzeichen des orientierten Abstands aller vier Tetraedervertizes die Anzahl und Lage der Vertizes des Schnittpolygons zu ermitteln. Für jede Kante des Tetraeders, die als von der Slice-Ebene geschnitten identifiziert wird, wird die tatsächliche Position des Schnittpunktes aus den Positionen der Endpunkte der Kante linear interpoliert.

Um die Anzahl der zu bearbeitenden Tetraeder je Slice zu reduzieren, wird beim hybriden Slicing ebenfalls eine Liste aktiver Tetraeder, so wie in Abschnitt 7.2.5 beschrieben, verwendet. Der Abstand jedes Vertex von der Basis-Ebene der Slices, der zum Eintragen der Tetraeder in diese Liste verwendet wird, kann für Optimierungszwecke wiederverwendet werden. Wird von diesem Wert der Abstand der aktuellen Slice-Ebene von der Slice-Basisebene subtrahiert, so ergibt sich direkt der orientierte Abstand jedes Knotens von der aktuellen Slice-Ebene. Das Vorzeichen dieses Abstands dient der Ermittlung der Dreieckskonfiguration aus der Marching-Cubes-Tabelle. Weiterhin definieren diese Werte auch die korrekten Interpolationsgewichte für die Bestimmung der Schnittpunkte der Isofläche mit den Kanten des Tetraeders und für die Interpolation der Datenwerte an diesen Punkten. Das Schnittpolygon wird mit den Datenwerten, die als Farben an den Vertizes definiert sind, gezeichnet.

7.5 Ergebnisse und Analyse

In diesem Abschnitt wird die Performanz der unterschiedlichen vorgestellten Resampling-Algorithmen analysiert. Die Tests fanden auf einem PC mit AMD Athlon 900 MHz Prozessor und 256 MB Hauptspeicher sowie einer nVidia GeForce2 GTS Graphikkarte mit 32 MB DDR Speicher statt. Für die Experimente wurde der Bluntfin-Datensatz, der in 220.000 Tetraeder zerlegt wurde, sowie ein Finite-Elemente-Datensatz bestehend aus 120.000 Tetraedern verwendet. Volumendarstellungen exemplarischer Resampling-Ergebnisse für beide Datensätze sind in Abbildung 7.8 zu sehen.

Tabelle 7.1 enthält einen direkten Vergleich zwischen dem Hardware-unterstützten Slicing (vgl. Abschnitt 7.2), welches Multi-Texturen und Register Combiners verwendet (MultiComp) sowie dem hybriden Resampling (Hybrid) aus Abschnitt 7.4. Sie werden mit einer Nachimplementierung des von Westermann [Westermann 2001] veröffentlichten Ansatzes (WestM/C) verglichen. Die Tabelle enthält die Gesamtzeit (Total) für das Resampling, die zusätzlich in die für das Rendering benötigte Zeit (Rnd) und die Zeit für die Initialisierung (Init) aufgeteilt ist. Letztere besteht hauptsächlich aus dem Aufbau der Liste der aktuellen Tetraeder.

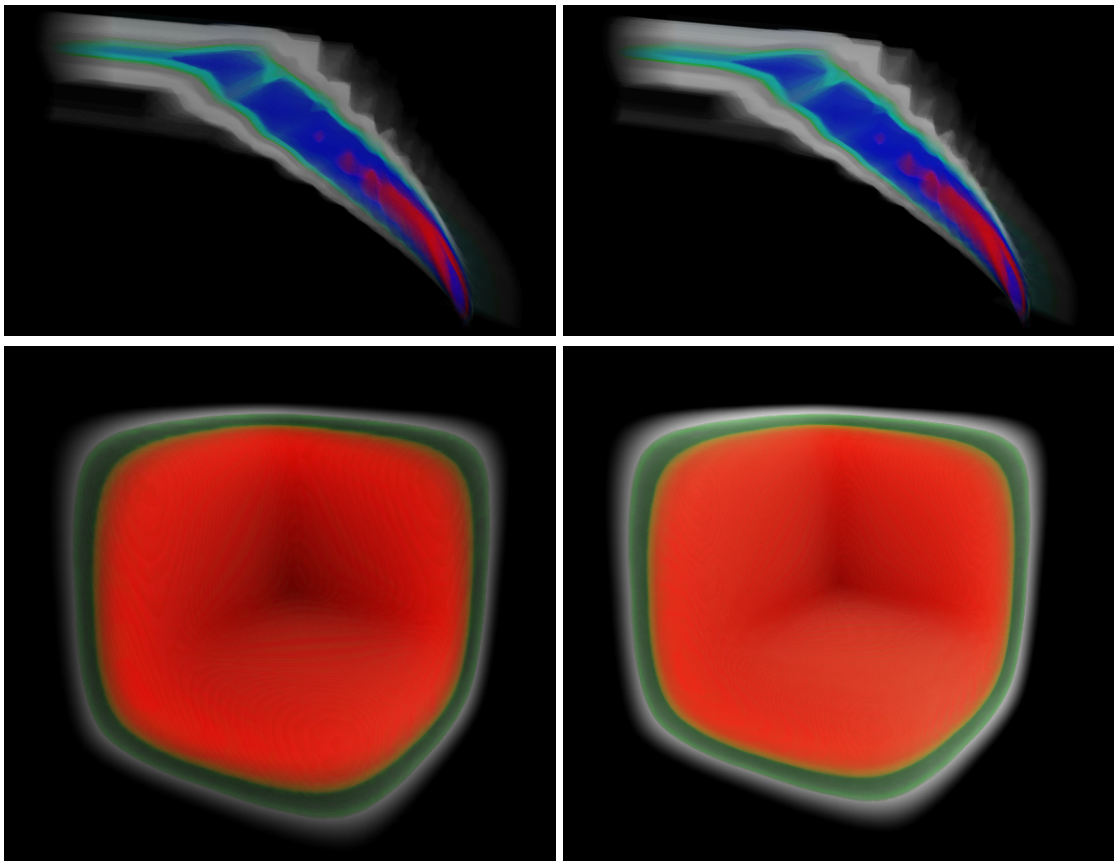


Abbildung 7.8: Die obere Reihe zeigt Volumendarstellungen des Bluntfin-Datensatzes, der auf einem regulären Gitter der Auflösung $256 \times 128 \times 64$ (links) beziehungsweise $512 \times 256 \times 128$ (rechts) Pixel rekonstruiert wurde. Die untere Reihe zeigt den Heatsink-Datensatz jeweils abgetastet auf einem regulären Gitter der Auflösung 128^3 (links) beziehungsweise 256^3 (rechts) Pixel. Die gewählte nicht kontinuierliche Transferfunktion betont die Unterschiede.

Nicht enthalten ist die Zeit, die für das Zurücklesen der interpolierten Slices aus dem Framebuffer benötigt wird, bzw. die Zeit für das Kopieren der Daten aus dem Framebuffer in eine zweidimensionale oder dreidimensionale Textur, die direkt zur Visualisierung verwendet werden kann. Diese Zeiten sind prinzipbedingt für alle Hardware-basierten Verfahren identisch und können daher für den Vergleich vernachlässigt werden. Für den Heatsink-Datensatz benötigte das Zurücklesen von 64 Slices in der Auflösung 64^2 lediglich 0,12 s und 0,72 s für 256 Slices in der Auflösung 256^2, so dass für die untersuchten Fälle die Zeit für das Resampling hauptsächlich durch das Rendering bestimmt wird. Das direkte Kopieren der Daten aus dem Framebuffer in einen Stapel von zweidimensionalen Texturen war auf der untersuchten Graphik-Hardware etwa dreimal schneller.

Aus der Tabelle wird deutlich, dass der Hardware-untersützte Algorithmus (MultiComp) im Vergleich zu WestM/C nur etwa ein Drittel der Polygone und die Hälfte der Vertizes für dasselbe Resampling-Ergebnis benötigt. Dies spiegelt die Tatsache wider, dass bei dem hier vorgeschla-

Tabelle 7.1: Vergleich der Rechenzeit (in Sekunden) bei Anwendung unterschiedlicher Hardware-unterstützter Resampling-Algorithmen auf den Heatsink-Datensatz und den Bluntfin-Datensatz bei variierender Auflösung.

Heatsink	Init	$64 \times 64 \times 64$			$128 \times 128 \times 128$			$256 \times 256 \times 256$		
		Rnd	Total	#Poly	Rnd	Total	#Poly	Rnd	Total	#Poly
MultiComb	0,17	0,40	0,57	286 K	0,82	0,99	578 K	3,75	3,92	1161 K
Hybrid	0,07	0,36	0,43	337 K	0,72	0,78	680 K	1,47	1,53	1367 K
WestM/C	0,29	0,64	0,93	810 K	1,29	1,59	1580 K	2,54	2,83	3191 K

Bluntfin	Init	$128 \times 64 \times 32$			$256 \times 128 \times 64$			$512 \times 256 \times 128$		
		Rnd	Total	#Poly	Rnd	Total	#Poly	Rnd	Total	#Poly
MultiComb	0,30	0,39	0,69	239 K	0,82	1,12	457 K	3,28	3,57	921 K
Hybrid	0,10	0,37	0,41	319 K	0,74	0,84	457 K	1,56	1,67	921 K
WestM/C	0,49	0,54	1,03	696 K	1,03	1,52	1327 K	2,08	2,57	2720 K

genen Verfahren lediglich 1,5 Polygone pro Slice und Tetraeder gezeichnet werden, wohingegen die Shirley-Tuchman-Unterteilung, die von Westermann verwendet wird, im Schnitt 3,5 Polygone produziert. Für kleine Auflösungen ist MultiComp deutlich schneller als WestM/C. Der Geschwindigkeitsvorteil resultiert einerseits, wie aus der Spalte (Init) ersichtlich, aus der gesparten Zeit für die Shirley-Tuchman-Unterteilung, und andererseits aus der schnelleren Darstellung.

Für hohe Auflösungen des Resampling-Gitters sinkt die Performanz von MultiComp deutlich. Offensichtlich ist bei diesem Verfahren die Rasterisierung aufwändiger als bei WestM/C, was von den verhältnismäßig teueren Fragmentoperationen herrührt. Verifizieren ließ sich dies durch eine Reduktion der x- und y-Auflösung des Resampling-Gitters unter Beibehaltung der Slice-Anzahl, wobei die Geschwindigkeit deutlich anstieg. Somit liegt der Flaschenhals weder in der Slice-Verarbeitung noch in der Zusammenstellung der Dreiecke oder der Transformation der Vertizes durch die Graphik-Hardware. Dies stellt allerdings für die hier skizzierte Anwendung, die Konstruktion einer Hierarchie von regulären Gittern, keine echte Einschränkung dar, da eine solche Hierarchie in der Regel aus vielen verhältnismäßig kleinen Teil-Gittern besteht.

Der hybride Resampling-Algorithmus erreicht auf den getesteten Datensätzen im Vergleich zu WestM/C eine Beschleunigung zwischen 1,5 und 2,5. Dabei ist die eigentliche Darstellung um den Faktor 1,4 bis 1,8 schneller. Der Rest ergibt sich aus der deutlich schnelleren Initialisierung, da in diesem Fall die Shirley-Tuchman-Unterteilung komplett vermieden werden kann. Gegenüber dem Hardware-unterstützten Verfahren fällt die Berechnung der partiellen Ableitungen der baryzentrischen Koordinatentransformation weg. In der Summe erreicht der hybride Ansatz eine ausgewogene Balance zwischen Graphik-Hardware und CPU und ist der schnellste der untersuchten Algorithmen.

Tabelle 7.2: Zeiten (in Sekunden) für das Software-Resampling des Heatsink- und Bluntfin-Datensatzes bei unterschiedlichen Auflösungen. Die Doppel- und Vierfach-Bluntfin-Datensätze wurden als Testfälle für Datensätze mit einer großen Anzahl an Tetraedern konstruiert. Sie entstanden durch Spiegelung des Bluntfin-Datensatz an der y-Achse beziehungsweise an der y- und z-Achse.

Heatsink

Auflösung	Zeit
$32 \times 32 \times 32$	0,45
$64 \times 64 \times 64$	0,69
$128 \times 128 \times 128$	1,58
$256 \times 256 \times 256$	5,03
$512 \times 512 \times 512$	20,93

Bluntfin

Auflösung	Zeit
$128 \times 64 \times 32$	0,80
$256 \times 128 \times 64$	1,23
$512 \times 256 \times 128$	3,01
$1024 \times 512 \times 256$	12,13

Doppel-Bluntfin (≈450.000 Tetraeder)

Auflösung	Zeit
$128 \times 64 \times 32$	1,50
$256 \times 128 \times 64$	1,96
$512 \times 256 \times 128$	4,08
$128 \times 128 \times 32$	1,66
$256 \times 256 \times 64$	2,54
$512 \times 512 \times 128$	6,07

Vierfach-Bluntfin (≈900.000 Tetraeder)

Auflösung	Zeit
$128 \times 64 \times 32$	2,79
$256 \times 128 \times 64$	3,52
$512 \times 256 \times 128$	6,07
$128 \times 128 \times 64$	3,17
$256 \times 256 \times 128$	5,09
$512 \times 512 \times 256$	12,11

Obwohl die Implementierung der reinen Software-Lösung sicherlich noch Raum für Optimierungen lässt, ist die erzielte Geschwindigkeit, die in Tabelle 7.2 dokumentiert ist, bemerkenswert. Wie erwartet, ist die Software-Lösung langsamer als das hybride Resampling. Allerdings schlägt sie in vielen Fällen, besonders für kleine Resampling-Gitter, WestM/C. Ein fairer Vergleich mit einer Hardware-Lösung muss allerdings je nach Anwendungsfall noch den Datentransfer von bzw. zur Graphikkarte berücksichtigen. Gerade in den jüngsten Publikationen zum Einsatz der GPU als General Purpose (Co-)Prozessor wird dieser Punkt gerne vernachlässigt. Für die Speicherung der Daten als Datei kommt bei den Hardware-Lösungen noch die Zeit für das Zurücklesen der Slice-Daten in den Hauptspeicher hinzu. Für die direkte Visualisierung der Daten mit Hilfe der Graphik-Hardware müssen die in Software berechneten regulären Gitterdaten auf die Graphikkarte transferiert werden und die Texturen müssen aufgebaut werden.

Unerwarteter Weise hat sich die dynamische Wahl der Scanline-Richtung entsprechend der größten Dimension der Bounding Box des Tetraeders als Schüssel für gute Resampling-Performanz herausgestellt. Speziell im Falle des Bluntfin-Datensatzes konnte dadurch die durchschnittliche Länge des Bereichs der Scanline, der inkrementell bearbeitet werden kann, dramatisch erhöht werden. Für das Resampling auf einem 256^3 Gitter beispielsweise von 2,95 auf 7,87 Voxel und von 5,00 auf 14,41 Voxel für ein 512^3 Gitter. Eigentlich wäre zu erwarten gewesen, dass die Bearbeitung des regulären Gitters in y- oder z-Richtung deutlich langsamer ist. Das in diesen Fällen ungeschicktere Zugriffsschema auf den Speicher erhöht die Wahrscheinlichkeit für Cache-Misses. In der hier untersuchten Implementierung scheint jedoch die deutlich längere inkrementelle Scanline-Verarbeitung diesen Nachteil, zumindest für den AMD-Prozessor, mehr als auszugleichen.

Weitere Tests haben gezeigt, dass sich die Laufzeit des Software-Algorithmus für kleine Auflösungen des Resampling-Gitters eher linear mit der Anzahl der Tetraeder verhält, da in diesem Fall die Zeit für die für jeden Tetraeder durchzuführende Initialisierung die Scanline-Bearbeitung dominiert. Für Resampling-Gitter mit einer Auflösung von mehr als 256^3 ist die Laufzeit eher linear mit der Anzahl der Voxel. Als wesentlicher Vorteil des Software-Resampling verbleibt jedoch, dass es Interpolationsfunktionen höherer Ordnung erlaubt und eine höhere Genauigkeit bietet.

Kapitel 8

Level-of-Detail-Volumenvisualisierung mittels 3D-Texturen

Um von den hohen Interaktionsraten, die bei texturbasierter Volumenvisualisierung [Cabral et al. 1994] erzielt werden, zu profitieren, wurde in Kapitel 7 ein Resampling des Tetraedergitters auf einem regulären Gitter vorgeschlagen. Dabei wurde ebenfalls die Notwendigkeit für adaptives Resampling motiviert, da die uniforme Abtastung eines Tetraedergitters in der Praxis sehr große Datenmengen erzeugt. Letzteres stellt in Anbetracht des limitierten Texturspeichers auf der Graphikkarte ein ernstzunehmendes Problem für die texturbasierte Volumenvisualisierung dar. Überschreitet die Größe eines Datensatzes die Menge an physikalischem Texturspeicher, ist eine Unterteilung des Volumens in mehrere Blöcke oder Bricks [Grzeszczuk et al. 1998] notwendig, was die Interaktivität der Darstellung deutlich reduziert, da die Bricks in jedem Frame zwischen dem Hauptspeicher und der Graphikkarte transferiert werden müssen.

Dasselbe Problem tritt bei Volumendaten aus der medizinischen Bildgebung oder der Computersimulation auf, die nativ als reguläres Gitter gegeben sind. Auch hier überschreiten die erzeugten Datensätze mit ihrer zunehmenden Auflösung den auf Graphikkarten verfügbaren Texturspeicher, was die interaktive Visualisierung dieser Daten zu einer großen Herausforderung macht. Glücklicherweise besitzen auch diese Datensätze meistens große Regionen ohne interessante Daten, was eine adaptive Repräsentation nahelegt. In dieser Repräsentation können interessante Bereiche sehr viel feiner aufgelöst gespeichert werden als die übrigen Teile des Volumens, was zu einer signifikanten Datenreduktion führt. Ein weiterer Vorteil der adaptiven Repräsentation ergibt sich aus der Tatsache, dass gröbere Auflösungsstufen eine niedrigere Abtastung erlauben. Die daraus resultierende Verringerung der Anzahl an Slice-Polygonen reduziert die Menge an teuren trilinearen Interpolationsoperationen und damit den Rasterisierungsaufwand für die Graphik-Hardware.

Das Problem dieser mit unterschiedlichen Auflösungen arbeitenden Methoden ist, dass sie Artefakte an den Rändern zwischen Blöcken mit unterschiedlichem Level-of-Detail erzeugen. Um diese zu vermeiden, muss eine konsistente Interpolation zwischen den unterschiedlichen

Auflösungsstufen erreicht werden, wobei die Menge von replizierten Daten klein bleiben soll. Außerdem können durch das adaptive Sampling selbst bei konsistenter Interpolation visuelle Artefakte an den Grenzen zwischen unterschiedlich aufgelösten Blöcken auftreten.

Dieser Abschnitt beschreibt einen adaptiven Ansatz zur interaktiven Darstellung großer Volumendatensätze mit unterschiedlichem Level-of-Detail, der beide Probleme löst. Er wurde auch in [Weiler et al. 2000] veröffentlicht. Eine Hierarchie von Texturen wird so erzeugt, dass die Menge an Texturspeicher, insbesondere unter Berücksichtigung der für 3D-Texturen auch auf aktuellen Graphikkarten von OpenGL geforderten Beschränkung von Texturgrößen auf Zweierpotenzen, minimiert wird. Außerdem garantiert die Repräsentation eine konsistente Interpolation zwischen Blöcken mit unterschiedlichem Level-of-Detail. Das Füllen dieser Hierarchie kann entweder durch Downsampling eines Volumendatensatzes, der als reguläres Gitter gegeben ist, erfolgen oder durch Resampling eines Tetraedergitters, wie in Kapitel 7 beschrieben. Für adaptive Slice-Abstände wird ein Verfahren vorgeschlagen, das die aufgrund der nicht angepassten Opazitätswerte auftretenden Artefakte beseitigt.

Ähnliche Arbeiten zur Volumendarstellung mit unterschiedlichen Auflösungsstufen wurden unter anderem in [Boada et al. 2001] und [LaMar et al. 1999] veröffentlicht. Bei LaMar et al. wird ein Ansatz basierend auf einer Oktalbaum-Hierarchie beschrieben, die mit gefilterten Versionen der Volumendaten initialisiert wird. Für die Darstellung werden die jeweiligen Auflösungsstufen entsprechend ihrer Wichtigkeit gewählt. Allerdings wird dort keine Rücksicht auf eine konsistente Interpolation genommen. Artefakte an den Grenzen unterschiedlich fein abgetasteter Blätter des Oktalbaums werden durch die Wahl von Kugelschalen als Slicing-Geometrie minimiert, nicht aber beseitigt.

8.1 Level-of-Detail-Texturrepräsentation

Die hier vorgeschlagene Volumenrepräsentation lehnt sich an die Bricking-Idee [Grzeszczuk et al. 1998] an, welche oftmals verwendet wird, um Datensätze zu visualisieren, die den physikalischen Texturspeicher überschreiten. Der Datensatz wird dabei in Subvolumina oder Bricks zerlegt, die jeweils klein genug sind, um in den Texturspeicher zu passen. Diese können dann individuell von vorne nach hinten oder von hinten nach vorne gezeichnet werden. Im Prinzip müssen dabei alle Bricks je Frame einmal auf die Graphikkarte geladen werden, was zu deutlich schlechterer Performanz führt.

Um dies zu vermeiden, speichert jeder Brick lokal Approximationen des ursprünglichen Datensatzes bei immer gröberer Auflösung. Diese Kopien werden dann verwendet, um beliebige Regionen adaptiv mit niedrigerem Detaillierungsgrad und niedrigerem Speicherplatzbedarf zeichnen zu können. Im Idealfall führt dies dazu, dass alle zur Darstellung benötigen Bricks nun gleichzeitig im Texturspeicher vorliegen und kein Datentransfer stattfinden muss. Selbst wenn aufgrund der gewählten Auflösungsstufen der gesamte Datensatz nicht in den Texturspeicher passt, so ergibt sich immer noch ein Vorteil durch das schnellere Hochladen der Texturen. Weiterhin kann der Slice-Abstand an die jeweilige Brick-Auflösungsstufe angepasst werden, was Rasterisierungsleistung einspart. Summiert führt dies zu deutlich höheren Bildwiederholraten.

Im Weiteren sei angenommen, dass der gegebene Datensatz bereits initial in eine bestimmte Anzahl von Bricks unterteilt wurde. Ausgehend von dieser Unterteilung wird zunächst der Aufbau der Texturhierarchie beschrieben, die es erlaubt, diese Bricks mit unterschiedlichen Auflösungen zu zeichnen. Wie im Weiteren ausgeführt wird, ist zusätzlicher Aufwand erforderlich, um einen kontinuierlichen Übergang zwischen den Datenwerten benachbarter Bricks auf unterschiedlichen Auflösungsstufen zu garantieren. Kriterien für die initial gewählte Brick-Anzahl sowie die Auflösungsstufe, die für die Darstellung verwendet wird, werden in Abschnitt 8.1.4 diskutiert.

8.1.1 Texturunterteilung

Vor der Darstellung wird der Volumendatensatz in mehrere Bricks unterteilt. Jeder Brick bekommt zunächst eine Textur zugewiesen, die groß genug ist, um den Brick in der Originalauflösung darzustellen. Darauf aufbauend erzeugt jeder Brick seine eigene lokale Texturhierarchie, indem er Kopien der Originaltextur mit immer gröberer Auflösung erzeugt. Diese unterschiedlichen Levels-of-Detail werden, wie in Abbildung 8.1 dargestellt, in zusätzlichen Texturen gespeichert.

Der Originaldatensatz wird mit den Texturen auf Level 0 repräsentiert. Höhere Indizes bezeichnen gröbere Texturauflösungen. Die Breite der Texturelemente auf Level k ist doppelt so groß wie die Breite eines Elements auf Level $k-1$, wohingegen sich die Größe der Textur in jeder Dimension jeweils um den Faktor zwei reduziert. Texturelemente auf unterschiedlichen Auflösungsstufen werden so ausgerichtet, dass die Mittelpunkte der Elemente auf einem bestimmten Level mit den Mittelpunkten von Elementen auf dem nächst feineren Level, die einen geradzahligen Index besitzen, übereinstimmen.

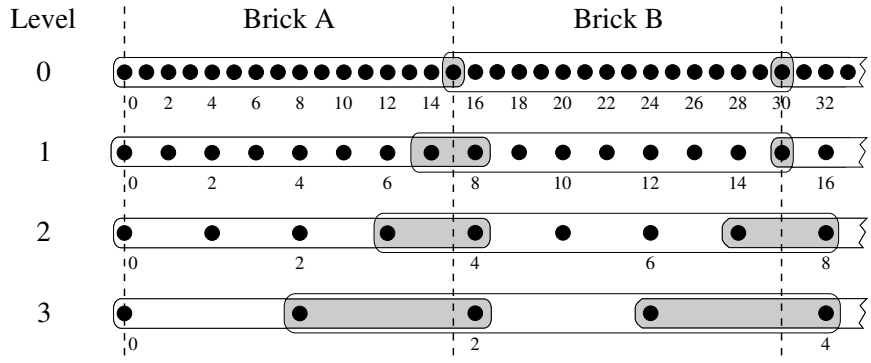


Abbildung 8.1: Eindimensionales Beispiel für einen Datensatz mit vier Levels-of-Detail. Auf jedem Level wächst die Breite der Texturelemente um den Faktor zwei. An Brick-Grenzen auf demselben Level müssen die Randtexel in beiden benachbarten Bricks enthalten sein, um konsistente Texturinterpolation zu garantieren (graue Bereiche). Um der Beschränkung der Texturgröße auf Zweierpotenzen zu entsprechen, müssen die Kopien jedes Bricks ggf. erweitert und mit Nullen aufgefüllt werden.

Da die Geometrie bzw. die Form eines Bricks durch alle Hierarchiestufen unverändert bleibt, muss die Berechnung der Texturkoordinaten entsprechend angepasst werden. Auf dem ersten Level verläuft der Wertebereich für die Texturkoordinaten vom Mittelpunkt des ersten Texturelements bis zum Mittelpunkt des letzten Texturelements. Der Wertebereich für nachfolgende Level ergibt sich aus dem Offset der Texturen zueinander sowie der Breite eines Texturelements der entsprechenden Stufe. Aufgrund der Ausrichtung der Texel auf unterschiedlichen Stufen fallen die Grenzen des Wertebereiches für die Texturkoordinaten oftmals zwischen zwei benachbarte Texel. In diesem Fall sind zusätzliche Texel für eine korrekte Interpolation erforderlich.

Die Kopien des ursprünglichen Datensatzes mit gröberer Auflösung werden durch sukzessive Filterung der Daten konstruiert. Im Allgemeinen kann dieser Filter beliebig gewählt werden, obwohl einfaches Subsampling der Originaldaten mit Nearest Neighbor Interpolation bereits nach wenigen Schritten zu unbefriedigenden Ergebnissen führt. Gute Resultate wurden durch die Verwendung eines quadratischen Splines als Filter-Kernel erzielt. Im Falle eines zugrundeliegenden Tetraedergitters kann auch für jede Stufe ein Resampling aus den ursprünglichen Gitterdaten angestoßen werden.

An dieser Stelle muss darauf hingewiesen werden, dass die von OpenGL induzierte Beschränkung der Texturgröße auf Zweierpotenzen eine Erweiterung der Texturen notwendig macht. Diese Beschränkung lässt sich zwar auf der jüngsten Generation von PC-Graphikkarten unter Verwendung spezieller OpenGL-Erweiterungen umgehen, allgemein lassen sich unnötige Texel aber auch durch eine hierarchische Speicherung, wie sie im Abschnitt 8.1.3 beschrieben wird, vermeiden.

8.1.2 Kontinuierliche Level-Übergänge

Wird die lokale Texturhierarchie wie beschrieben konstruiert, so treten an den Grenzen zwischen benachbarten Bricks, die auf der gleichen Auflösungsstufe gezeichnet werden, keine Interpolations-Artefakte auf. Dies rührt daher, dass benachbarte Bricks dieselben Randtexel besitzen. Für benachbarte Bricks, die auf unterschiedlichen Auflösungsstufen gezeichnet werden, trifft dies nicht zu.

Um hier Kontinuität zu erreichen, ist eine leichte Modifikation der Behandlung von Level-Übergängen notwendig: Der Skalarwert für feinere Texel links und rechts von der Brick-Grenze wird nicht aus der aktuellen Auflösungsstufe ermittelt sondern stattdessen aus der nächst gröberen Stufe interpoliert (Abbildung 8.2). Für Texel mit geradzahligem Index in jeder Dimension ist dies äquivalent mit einer Kopieroperation, da sie direkt zu einem Texel auf der nächst gröberen Stufe korrespondieren. Die Prozedur adaptiert *nur* Brick-Texturen auf feinerer Stufe. Damit die Level-Kontinuität funktioniert, darf der Unterschied zwischen den Auflösungen benachbarten Bricks maximal *einen* Level betragen. Dies stellt jedoch keine echte Einschränkung dar, weil höhere Übergänge durch aufeinanderfolgende Übergänge mit jeweils einem Level Unterschied erreicht werden können.

Die Anzahl der Texel, die angepasst werden müssen, hängt von der Position der Brick-Grenze relativ zu den Texel-Koordinaten ab. Entweder liegt die Grenze genau in einem Texel-Mittelpunkt,

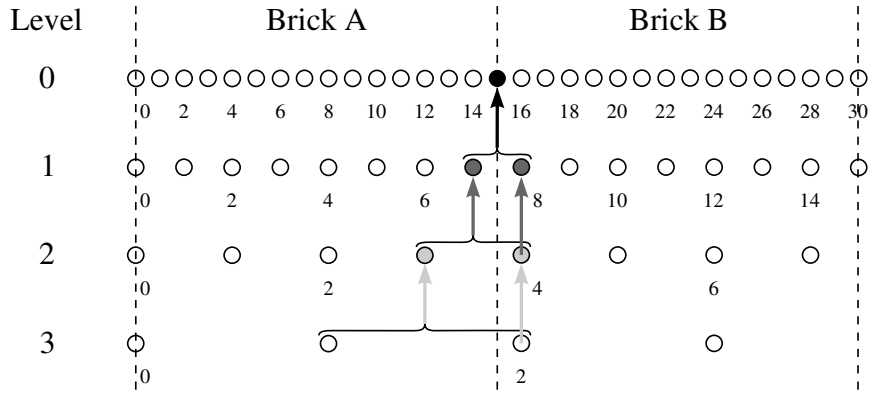


Abbildung 8.2: Benachbarte Bricks, die mit unterschiedlicher Auflösung dargestellt werden, müssen für eine konsistente Texturinterpolation angepasst werden. Nur Texturen für feinere Auflösungsstufen müssen angepasst werden, wobei Texel links und rechts der Brick-Grenzen entweder von gröberen Stufen kopiert oder interpoliert werden. Für den dargestellten Brick auf Level 0 muss der schwarze Texel interpoliert werden. Die Texel, die für die Level 1 und 2 angepasst werden müssen, sind hellgrau bzw. dunkelgrau gezeichnet.

wie es beispielsweise auf Level 0 der Fall ist, oder im Zwischenraum zweier Texel, wie es für alle übrigen Level der Fall ist. Im ersten Fall muss nur ein Texel angepasst werden, während im zweiten Fall beide Texel modifiziert werden müssen. Durch den Überlapp zwischen den Bricks muss eine Adaption immer dann erfolgen, wenn ein Brick an einen gröberen Brick grenzt. Im Dreidimensionalen bedeutet dies, dass bis zu 26 Nachbarn betrachtet werden müssen.

Die Anpassung der Brick-Texturen erfolgt nicht statisch während des Aufbaus der Level-of-Detail-Repräsentation, da dies für den Fall, dass benachbarte Bricks auf derselben Auflösungsstufe dargestellt werden, zu Genauigkeitsverlust führt. Stattdessen erfolgt eine dynamische Anpassung, sobald die Auflösungsstufe für alle Bricks feststeht. Während der Darstellung kann sie sich ändern, was eine kontinuierliche Anpassung der Texturen erforderlich macht. In typischen Szenarios werden in einem Frame nur wenige Bricks eine neue Auflösung wählen. Außerdem ist die Anzahl der Texel, die je Brick angepasst werden müssen, klein, wodurch der Mehraufwand durch die dynamische Anpassung gering ist.

8.1.3 Verschmelzen der Texturen

Die Texturerweiterung, die in Abschnitt 8.1.1 eingeführt wurde, um kontinuierliche Übergänge zwischen Bricks auf unterschiedlichen Auflösungsstufen zu garantieren, führt, insbesondere aufgrund der Beschränkung von Texturgrößen auf Zweierpotenzen, zu einem signifikanten Mehraufwand bezüglich des verwendeten Texturspeichers (siehe Abbildung 8.1). Um die durch die ersten 31 Texel auf Level 0 repräsentierte Information zu zeichnen, werden auf der feinsten Auflösungsstufe zwei Texturen der Größe 16 benötigt und ebenfalls zwei Texturen der Größe 16 auf Level 1, wobei hiervon nur jeweils 9 Texel für die Speicherung der Information nötig sind.

Effektiv betrachtet ergibt sich also in diesem Fall durch die Wahl von Level 1 statt Level 0 keine Speicherplatzersparnis. Die Betrachtung höherer Level führt zu ähnlichen Ergebnissen, da nur etwa die Hälfte der Texel jeder Textur – zuzüglich einer Überlappung von ein oder zwei Texel – nicht-redundante Informationen enthält.

Dieser Mehraufwand kann reduziert werden, indem die Texturen benachbarter Bricks, wie in Abbildung 8.3 gezeigt, zu einer einzigen Textur verschmolzen werden. Die Abbildung verdeutlicht die Situation im Eindimensionalen. Für alle Levels-of-Detail wird nun dieselbe Texturgröße verwendet. Auf Level 0 belegt jeder Brick eine eigene Textur. Auf dem nächst gröberen Level werden in jeder Richtung nur jeweils die Hälfte der Texel je Brick benötigt. Dies erlaubt die Repräsentation von acht benachbarten Bricks mit nur einer Textur. Die Texturkoordinaten der jeweiligen Bricks müssen dann entsprechend angepasst werden. Das Schema lässt sich für höhere Levels fortführen, wobei die Anzahl von Bricks, die jeweils eine gemeinsame Textur verwenden, mit jeder Stufe um den Faktor acht steigt. Insgesamt ergibt sich damit am Ende eine Hierarchie von Texturen, die einem Oktalbaum ähnelt.

Die Verschmelzung von Texturen funktioniert am besten, wenn initial mit einer Brick-Anzahl gestartet wird, die einer Zweierpotenz entspricht. Wenn in der initialen Unterteilung des Datensatzes mehr als 2^{l-1} Bricks erzeugt werden, wobei l für die Anzahl der Levels steht, die zur Darstellung verwendet werden soll, so werden die überzähligen Bricks mit einem eigenen Satz von Texturen, so wie in Abschnitt 8.1.1 beschrieben, repräsentiert. Stehen am Anfang zu wenige Bricks zur Verfügung, um eine komplette Hierarchie mit l Stufen aufbauen zu können, so wird die Tiefe der Texturhierarchie reduziert. In diesem Fall werden die gröbsten Auflösungsstufen je Brick in individuellen Texturen gespeichert.

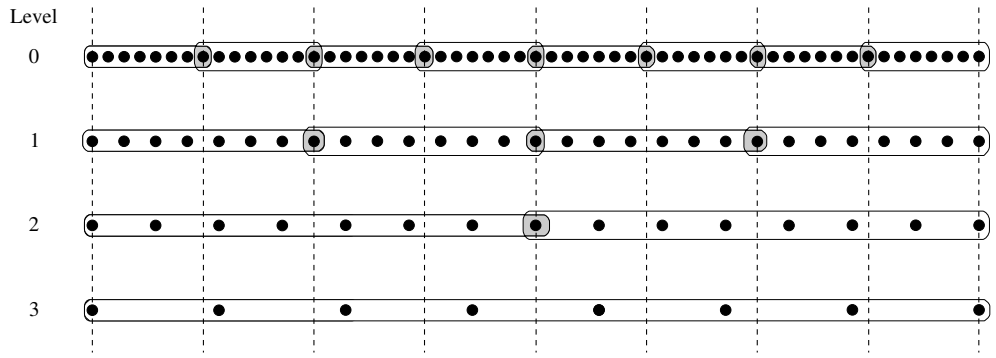


Abbildung 8.3: Eine Hierarchie wird zur effizienten Speicherung der Brick-Texturen verwendet. Die Texturen besitzen auf jeder Stufe dieselbe Größe. Während auf Level 0 jeder Brick seine eigene Textur enthält, nutzen 8 benachbarte Bricks auf Level 1 jeweils eine gemeinsame Textur. Mit jedem weiteren Level vergrößert sich die Zahl der Bricks, die eine gemeinsame Textur verwenden, um den Faktor acht. Für eine möglichst effiziente Nutzung der Hierarchie muss die Gesamtanzahl von Bricks eine Zweierpotenz sein.

8.1.4 Adaptive Darstellung

Ziel des gewählten Level-of-Detail-Ansatzes ist die Reduktion des benötigen Speicherbedarfs durch die Repräsentation von Bereichen geringerer Wichtigkeit mit niedrigeren Auflösungen. In der hier vorgestellten Implementierung erfolgt die Wahl der Auflösung je Brick mit Hilfe eines Fokuspunkt-Orakels. Abhängig vom Abstand des Brick-Mittelpunktes von einem benutzerspezifizierten Fokuspunkt wählt jeder Brick eine geeignete Auflösung. Immer dann, wenn ein Brick seine Auflösung neu wählt, muss gegebenenfalls eine Anpassung der Texturhierarchie erfolgen, um kontinuierliche Übergänge zu gewährleisten.

Die Auflösung für jeden Brick kann bei dem hier vorgestellten Ansatz beliebig gewählt werden, solange keine Level-Übergänge größer als eins auftreten. Dies kann durch die in [LaMar et al. 1999] präsentierte Technik aufgrund einer strikten Korrespondenz zwischen den Texturen und der Brick-Geometrie nicht erreicht werden. Abbildung 8.4 zeigt beispielsweise eine Verfeinerung, wie sie für Fokus-basierte Auflösungswahl typisch ist. Diese ist mit einem Verfeinerungsschema basierend auf einem Oktalbaum, wie es LaMar et al. verwenden, nicht möglich.

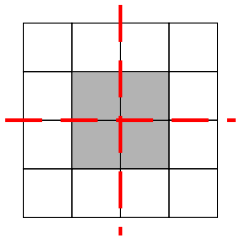


Abbildung 8.4: Beispiel für eine Auflösungswahl, die basierend auf einer Verfeinerung mittels Oktalbaum nicht erreicht werden kann. Die dunklen Blöcke stellen eine um einen Level feinere Auflösung als die weißen Blöcke dar. Basierend auf dem mit gestrichelten Linien skizzierten Oktalbaum würden in diesem Fall alle Blöcke fein dargestellt.

Ein wesentlicher Nachteil der Fokus-basierten Auflösungswahl ist, dass dabei der Approximationsfehler, der durch das Sampling der Daten auf einer gewissen Auflösungsstufe entsteht, nicht berücksichtigt wird. In der Tat wird sich, insbesondere wenn die Daten hohe Frequenzen enthalten, die grobe Approximation signifikant vom Originalsignal unterscheiden. Die Folge sind visuelle Artefakte an den Brick-Grenzen, selbst wenn kontinuierliche Interpolation garantiert wird.

Um dieses Problem zu lösen, müssen andere Kriterien für die Wahl der Auflösungsstufe eines Bricks verwendet werden. Eine Evaluierung unterschiedlicher hierarchischer Repräsentationen für das fehlergetriebene Volume Ray Casting wurde in [Danskin und Hanrahan 1992] präsentiert. Auch eine Wavelet-Analyse, wie sie in [Muraki 1993, Westermann 1994] für verbesserte Volumedarstellung eingeführt wurde, kann ein effektives Werkzeug für die Berechnung der Abweichung zwischen Originaldaten und grober Approximation sein. Dieses Maß kann dann für eine automatische Wahl der geeigneten Auflösungsstufe verwendet werden.

Eine weitere Option stellt die blickpunktabhängige Level-of-Detail-Auswahl dar. Mögliche Kriterien für eine gröbere Darstellung eines Bricks sind das Verhältnis der auf die Bildebene projizierten Texel-Größe im Vergleich zur Größe eines Pixels oder die Verdeckung eines Bricks durch davorliegende Bricks. Entsprechend der vor dem Brick entlang der Blickrichtung aufgesammelten Opazität kann der visuelle Fehler, den die Vergröberung eines Bricks erzeugt, deutlich geringer sein, als die analytische Abweichung von den Originaldaten erwarten lässt. Eine Abschätzung des im Bildraum entstehenden Fehlers a-priori ist allerdings schwierig und hängt von der jeweils gewählten Transferfunktion ab.

8.2 Adaptives Slicing

Nachdem die Volumendaten, wie in Abschnitt 8.1 beschrieben, in eine hierarchische Repräsentation zerlegt wurden, erfolgt die Darstellung. Die Bricks werden dabei unabhängig voneinander mit geeignetem Level-of-Detail von hinten nach vorne gezeichnet. Dies erfolgt mit Hilfe der in [Cabral et al. 1994, Cullip und Neumann 1994] beschriebenen Methode für Hardwarebeschleunigte Volumenvisualisierung basierend auf dreidimensionalen Texturen. Die Grundidee dabei ist, die dreidimensionale Textur mit Hilfe von Schnittebenen parallel zur Bildebene, auf denen die Volumendaten trilinear interpoliert werden, abzutasten und die resultierenden Fragmente mittels Compositing in den Framebuffer zu blenden.

In der Regel wird dabei über das ganze Volumen von einem konstanten Abstand zwischen benachbarten Slices ausgegangen. Hier soll das Sampling jedoch an die je Brick gewählte Auflösungsstufe angepasst werden. Da für niedriger aufgelöste Bricks aufgrund steigenden Größe der Texel weniger Slices verwendet werden müssen, resultiert dies in eingesparter Rasterisierungsleistung und damit potentiell höherer Performanz. Eine korrekte Darstellung mit adaptiertem Slice-Abstand erfordert allerdings eine entsprechende Korrektur der Opazitätswerte. Außerdem treten dabei an Grenzen zwischen benachbarten Bricks bedingt durch teilweise überlappende Slice-Polygone Artefakte auf, die behoben werden müssen.

8.2.1 Anpassung der Opazität

Für die weitere Beschreibung der vorgestellten Lösung zur texturbasierten Level-of-Detail-Volumenvisualisierung ist es wichtig zu bemerken, dass hier Post-Klassifikation verwendet wird. Das bedeutet, dass in den Texturen Skalar- bzw. Indexwerte gespeichert sind, die mit Hilfe der in OpenGL definierten Texture Lookup Tables beim Texturnachschlag entsprechend der Transferfunktion auf Farb- und Opazitätswerte abgebildet werden.

Initial wird bei der Abtastung des Volumens der Abstand zwischen benachbarten Slices an die Größe eines Texels angepasst. Angenommen dieser Basis-Abstand sei d_0 und der Opazitätswert α_0, der für den Skalarwert s in der Texture Lookup Table gespeichert wird, ergibt sich als $1-\exp(-\tau(s)d_0)$, wobei τ die Transferfunktion für die Absorptionskoeffizienten beschreibt (vgl. Kapitel 3). Wird nun der Slice-Abstand bei gleichem Skalarwert auf d_k verändert, so müssen die

Einträge in der Texture Lookup Table entsprechend korrigiert werden. Die neuen Werte berechnen sich wie folgt:

$$\begin{aligned} \alpha_k &= 1-\exp(\tau(s)d_k) \\ &= 1-\exp(\tau(s)d_0)^{\frac{d_k}{d_0}} \\ &= 1-[1-\alpha_0]^{\frac{d_k}{d_0}} \end{aligned} \tag{8.1}$$

Damit die Bricks mit unterschiedlichen Slice-Abständen gezeichnet werden können, werden Texture Lookup Tables verwendet, welche die korrigierten Opazitätswerte für alle möglichen Auflösungsstufen enthalten. Für l unterschiedliche Auflösungsstufen werden somit l unterschiedliche Tabellen vorgehalten, welche die passend korrigierten Opazitätswerte zu den korrespondierenden Slice-Abständen enthalten. Soll ein Brick nun auf einer gewissen Auflösungsstufe gezeichnet werden, so wird der Slice-Abstand für diesen Brick entsprechend gesetzt und die geeignete Tabelle aktiviert. Slices auf Level k werden dabei so positioniert, dass sie immer einen Abstand $n \cdot 2^k d_0$, $n \in Z$ von der Bildebene besitzen. Dies garantiert die korrekte Ausrichtung zwischen den Slices benachbarter Bricks. Im Folgenden werden, wann immer ein Brick auf Level k an einen Brick auf Level $k+1$ grenzt, die Slices an der Position $n \cdot 2^{k+1} d_0$ als *gerade* Slices und die an Position $(n \cdot 2^{k+1}+1)d_0$ als *ungerade* Slices bezeichnet.

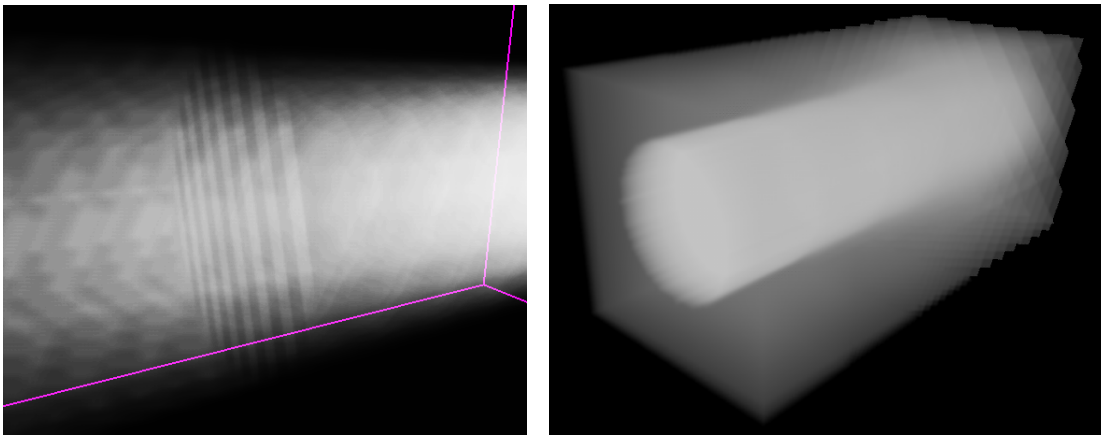


Abbildung 8.5: Beide Bilder zeigen denselben künstlichen Datensatz, der mit vier unterschiedlichen Aufösungsstufen gezeichnet wurde. Die Detailvergrößerung einer Brick-Grenze (links) zeigt deutliche Streifen-Artefakte, die aus der fehlenden Korrektur der partiellen Slice-Überlappung resultieren. Das Bild wurde zur Verdeutlichung des Effekts kontrastverstärkt. Wie in der Totalen rechts zu sehen, lassen sie sich durch eine Anpassung der Polygone vollständig beheben. Die Artefakte rechts am Rand des Volumens resultieren aus einer deutlichen Unterabtastung. Sie sind typisch für texturbasierte Volumenvisualisierung, treten aber durch das übertrieben grobe Sampling deutlicher zu Tage.

8.2.2 Opazitätskorrektur durch Polygon-Clipping

Selbst mit der prinzipiellen Korrektur der Opazität, wie sie gerade beschrieben wurde, treten an den Grenzen zwischen unterschiedlichen Auflösungsstufen visuelle Artefakte auf, wie sie in Abbildung 8.5 links zu sehen sind. Diese Artefakte manifestieren sich als helle und dunkle Linien entlang der Grenze zwischen den Bricks. Die fehlerhaften Bereiche können durch eine Projektion der Ränder der Slice-Geometrie entlang der Blickrichtung ermittelt werden. Dabei treten folgende zwei Fälle auf, die in Abbildung 8.6 verdeutlicht sind:

- **Fall 1:** Ein Slice, der einem Abstand von d_{k+1} entspricht und ein Slice entsprechend dem Abstand d_k werden gezeichnet, obwohl insgesamt nur eine Dicke von d_{k+1} überdeckt werden sollte. Ein Beispiel ist auf der linken Seite von Abbildung 8.6 dargestellt. Dort wird beim Zeichnen von hinten nach vorne zunächst das *gerade* Slice i und dann in Region A das *ungerade* Slice $i+1$ auf feinerer Stufe gezeichnet. In Region B wird ausschließlich das *gerade* Slice i mit angepasster Opazität auf grobem Level gezeichnet. In Region C wird jedoch das *gerade* Slice i auf grobem Level und danach das *ungerade* Slice $i+1$ auf feinerem Level gezeichnet bevor mit Slice $i+2$ fortgefahren wird. Insgesamt entspricht dies einem überdeckten Abstand von $d_k + d_{k+1}$.

- **Fall 2:** Es wird lediglich ein Slice entsprechend dem Abstand d_k gezeichnet, obwohl eine Dicke von d_{k+1} überdeckt werden sollte. Dies wird in der rechten Hälfte von Abbildung 8.6 dargestellt. In Region B wird hier beim Zeichnen von hinten nach vorne zunächst das *gerade* Slice i und dann das Slice $i+1$ jeweils auf feinerer Stufe gezeichnet. In Region A wird ausschließlich das *gerade* Slice i mit der Opazitätskorrektur für die grobe Stufe gezeichnet. Das Problem tritt in Region C auf, wo ebenfalls nur Slice i gezeichnet wird, hier aber auf feiner Stufe, d.h. mit der Opazität, die nur einem Abstand von d_k entspricht.

Generell können jedoch die inkorrekten Opazitäten vermieden werden, indem die fehlerhaft gezeichneten Bereiche – dicke Linien in Abbildung 8.6 – durch Clipping bestimmt werden und danach mit der richtigen Opazität gezeichnet werden. Die Anpassung der Slice-Geometrie muss nur für Bricks erfolgen, die mindestens einen Nachbarn besitzen, der auf einer feineren Auflösungsstufe dargestellt wird. Angepasst wird dabei nur der grobe Brick. Im Fall 1 muss das *gerade* Slice i im groben Brick mit der perspektivischen Projektion des *ungeraden* Slice $i+1$ aus dem feinen Brick auf die Ebene von i geschnitten werden. Das resultierende Polygon wird mit derselben Opazität wie der feinere Brick gezeichnet. Im Fall 2 muss auch im groben Brick das *ungerade* Slice $i+1$ konstruiert werden und mit der Projektion des *geraden* Slice i des feinen Bricks auf die Ebene von $i+1$ geschnitten werden. Das resultierende Polygon wird mit der korrigierten Opazität α_{k+1} gezeichnet. Dies funktioniert ebenso für orthographische Projektion.

Die gesamte Prozedur kann einfach für den dreidimensionalen Fall verallgemeinert werden. Für jedes Paar von *geraden/ungeraden* Slices in einem groben Brick wird sukzessive das *gerade* Slice i mit den Projektionen aller *ungeraden* Slices $i+1$ im selben Brick und allen Nachbar-Bricks auf demselben oder einem feineren Level geschnitten. *Ungerade* Slices $i+1$ werden nur mit

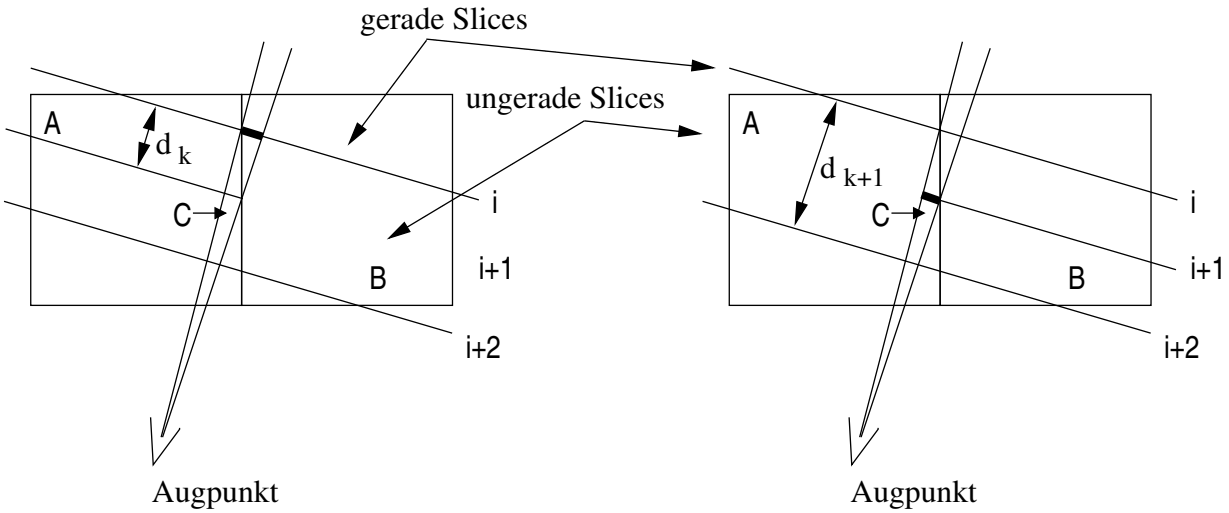


Abbildung 8.6: Die Opazitätskorrekturen an Level-Übergängen für orthographische und perspektivische Projektion fallen in zwei Klassen, die davon abhängen, ob der Brick auf grobem oder feinen Level in der Sichtbarkeitssortierung vorne liegt.

geraden Slices in Nachbar-Bricks auf feinerem Level geschnitten. Die Opazität wird für die resultierenden Schnittpolygone entsprechend dem Level des Nachbar-Bricks angepasst. Auf diese Weise werden Diskontinuitäten bei Level-Übergängen an Brick-Grenzen vollständig vermieden, so lange die Übergänge maximal eine Auflösungsstufe betragen. Eine Anwendung dieser Technik demonstriert das rechte Bild in Abbildung 8.5, wo die Artefakte an den Grenzen benachbarter Bricks vollständig verschwunden sind.

Da hierbei offensichtlich sehr viele Clipping-Operationen erforderlich sind, ist die Implementierung eines effizienten Clipping-Algorithmus von entscheidender Bedeutung. Hier wurde ein modifizierter dreidimensionaler Sutherland-Hodgman [Sutherland und Hodgman 1974] Clipping-Algorithmus verwendet, der sowohl das beschnittene Polygon, als auch die Bereiche, die vom Originalpolygon weggeschnitten wurden, zurückliefert, wobei für diese zusätzliche Information nur sehr wenig Mehraufwand entsteht. Dadurch lässt sich die Zahl der Clipping-Operationen deutlich reduzieren, weil beide Bereiche für die korrekte Darstellung erforderlich sind.

8.3 Ergebnisse und Analyse

Die beschriebene Technik zur Volumenvisualisierung mit unterschiedlichem Level-of-Detail wurde auf einer SGI Octane mit einem 250 MHz R10000 Prozessor, 256 MB Hauptspeicher und 4 MB Texturspeicher getestet. Abbildung 8.7 zeigt die Anwendung der Technik auf einen Magnetresonanztomographie-Datensatz mit einer Auflösung von 256^3 Voxel. Die Bilder wurden mit 8^3 Bricks der Größe 32^3 Voxel erzeugt. Zusätzlich wurde eine Clipping-Ebene verwendet, um die Auflösungshierarchie deutlicher zeigen zu können. Das linke Bild wurde mit voller

Auflösung gezeichnet, im mittleren Bild wurden nur die vier oberen Brick-Reihen und im rechten Bild nur die zwei oberen Brick-Reihen mit voller Auflösung gezeichnet. Die übrigen Reihen wurden nach unten hin sukzessive vergröbert.

Trotz der Korrektur der Opazität zeigen diese Bilder deutlich den potentiellen Nachteil einer Vergröberung, die keine Rücksicht auf die Topologie der Daten nimmt. Dies ist bei dem hier verwendeten Fokuspunkt-basierten Ansatz der Fall. Die Vergröberung basiert ausschließlich auf dem Abstand zum Fokuspunkt und nimmt keine Rücksicht auf den entstehenden Approximationsfehler. Wird dieser zu groß, können an den Grenzen zwischen den Bricks visuelle Artefakte auftreten. Derselbe Effekt wird auch in Abbildung 8.8 deutlich. Viele der relevanten Strukturen verschwinden durch die Tiefpassfilterung.

Auf der anderen Seite führt die hierarchische Repräsentation zu einer deutlichen Ersparnis von Texturspeicher. Bei dem in Abbildung 8.9 dargestellten Computertomogramm eines Maschinenteils beträgt die Ersparnis zwischen 43% und 71% bei immer noch akzeptabler Bildqualität. Im mittleren Bild von Abbildung 8.7 wurden nur 64% des ursprünglichen Texturspeicher verwendet und im rechten Bild lediglich 37%. Die Anzahl der Texturnachschläge wurde durch das adaptive Slicing auf 69% beziehungsweise 40% verringert. Dies führt zu verbesserter Performanz. Während für die volle Auflösung jeder Frame etwa 2 Sekunden benötigte, konnte durch die Level-of-Detail-Repräsentation das rechte Bild in nur 0,5 Sekunden berechnet werden.

Besondere Sorgfalt wurde bei der Implementierung darauf verwendet, die geschnittenen Polygone zu sortieren, um die Häufigkeit des Nachladens der Texture Color Table zu reduzieren. Anderenfalls reduziert sich die Performanz auf der getesteten Plattform deutlich. Dies liegt an der Tatsache, dass für die Opazitätskorrektur des dargestellten Kopf-Datensatzes im Schnitt etwa

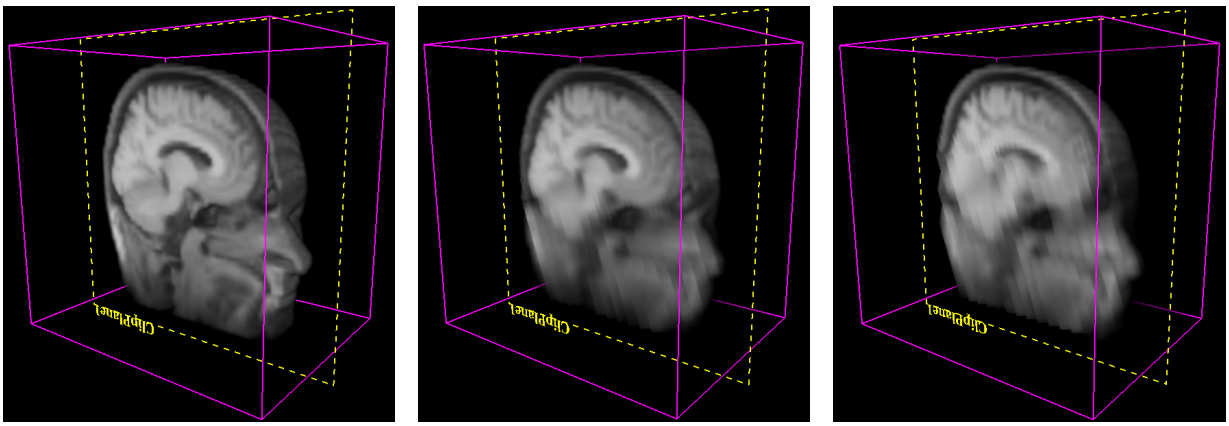


Abbildung 8.7: Beispiel für die Level-of-Detail-Darstellung eines Magnetresonanztomographie-Datensatzes. Das linke Bild zeigt den Datensatz mit der Originalaufösung. Dann wurde der Datensatz in 8^3 Bricks der Größe 32^3 unterteilt. Im mittleren und rechten Bild sind nur mehr die obersten vier beziehungsweise die obersten beiden Brick-Reihen mit der vollen Auflösung dargestellt. Nach unten hin nimmt die Auflösung mit jeder Reihe kontinuierlich ab.

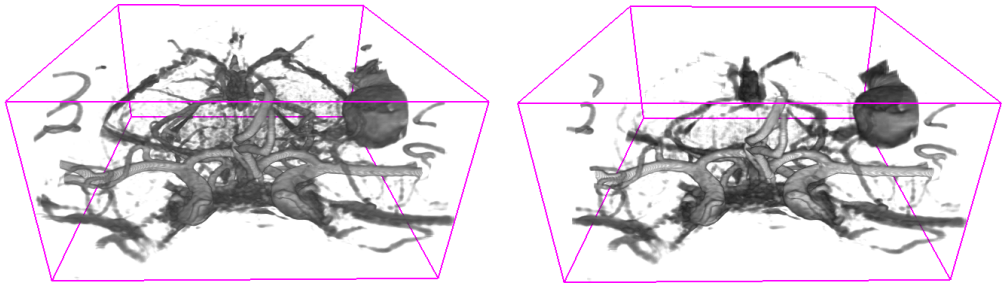


Abbildung 8.8: Medizinische Volumenvisualisierung eines Aneurysmas mit und ohne adaptive Repräsentation. Die Level-of-Detail-Darstellung auf der rechten Seite, bei der Bricks im Hintergrund gröber repräsentiert werden, erfordert lediglich 38% des ursprünglichen Texturspeichers und etwa 41% der für den Originaldatensatz erforderlichen Texturnachschläge.

2.500 Clipping-Operationen durchgeführt werden müssen und jede resultierende Region potentiell ein Umschalten der Texture Color Table erfordert. Schlimmstenfalls, bei einer Blickrichtung entlang der Diagonalen des Datensatzes, müssen sogar bis zu 5.500 Polygone geschnitten werden. Die Effizienz des vorgestellten Clipping-Algorithmus kann daran gesehen werden, dass der Mehraufwand unter 0,1 Sekunde liegt. Für aktuelle Graphik-Hardware mit programmierbaren Fragmentoperationen stellt das Umschalten der Texturtabelle keinen Flaschenhals mehr dar. Stattdessen kann hier mit nur einer Tabelle gearbeitet werden und die Opazitätsanpassung on-the-fly nach erfolgtem Nachschlag von Farb- und Opazitätswert erfolgen. Das Umschalten der Opazitätskorrektur zwischen unterschiedlichen Auflösungsstufen erfordert damit nicht mehr als das Setzen eines einzelnen Shader-Parameters.

Um einen Eindruck zu geben, wie lange die in Abschnitt 8.1.2 beschriebene Anpassung der Texturen zum Erreichen einer kontinuierlichen Interpolation der Datenwerte dauert, sei hier ein Beispiel gegeben. Für das rechte Bild in Abbildung 8.7 dauerte die Anpassung aller Bricks zusammen weniger als 0,2 Sekunden. Dies muss jedoch nur erfolgen, wenn sich der Fokuspunkt ändert, und selbst dann müssen nur einige der Bricks angepasst werden. Eine interaktive Änderung im Level-of-Detail ist damit möglich.

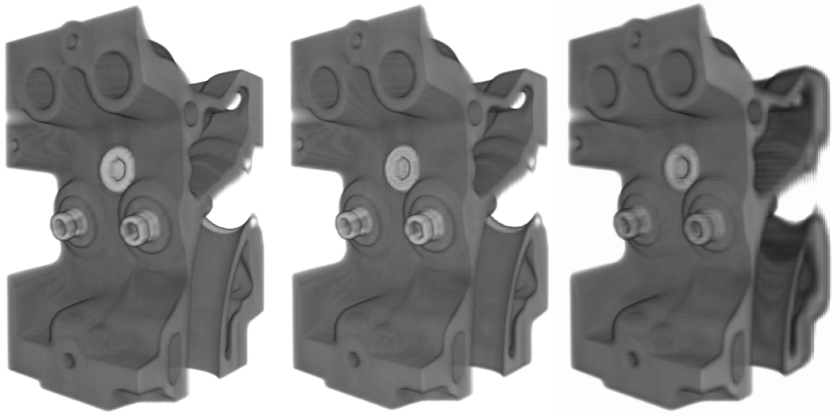


Abbildung 8.9: Level-of-Detail-Volumendarstellung des Engine-Datensatzes mittels dreidimensionaler Texturen. Links ist der Datensatz mit der vollen Auflösung dargestellt. In der Mitte wurden zwei unterschiedliche Auflösungsstufen verwendet, wobei die niedrigere im Hintergrund zum Einsatz kommt. Dies reduziert den benötigten Texturspeicher auf 57%. Die adaptive Darstellung rechts verwendet vier Levels-of-Detail von vorne nach hinten und benötigt lediglich 29% des ursprünglichen Texturspeichers. In diesem Fall wurde der Datensatz mit lediglich 32% der ursprünglich verwendeten Texturnachschläge gezeichnet.

Kapitel 9

Adaptive texturbasierte Volumenvisualisierung

Schon vor der Einführung von flexibel parametrisierbaren und programmierbaren Einheiten auf der Graphikkarte hat sich die texturbasierte Volumenvisualisierung [Cabral et al. 1994, Cullip und Neumann 1994] von regulären Volumengittern als eine der wichtigsten Techniken zur Volumenvisualisierung herauskristallisiert, da sie unter Ausnutzung der Textur-Hardware moderner Graphikkarten vergleichsweise hohe Bildwiederholraten erreicht. Letzteres ermöglicht eine interaktive Exploration der Volumendaten und stellt somit die Basis für eine effektive Analyse dar. Diese offensichtlichen Vorteile haben zu der Entwicklung einer Vielzahl von Anwendungen z.B. in Medizin und Technik geführt.

Vom Aufkommen programmierbarer Graphik-Hardware konnten Verfahren basierend auf texturbasierter Volumenvisualisierung im besonderen Maße profitieren, weil hier das Shading des Volumens ausschließlich von der Graphikkarte erledigt wird. Große Verbesserungen wurden sowohl im Hinblick auf die Geschwindigkeit der Darstellung – z.B. durch die Verwendung mehrerer Texturnachschläge in einen Schritt [Rezk-Salama et al. 2000] – als auch im Hinblick auf die Bildqualität erzielt. Beispiele sind der Einsatz multidimensionaler Transferfunktionen [Kniss et al. 2001], das Zeichnen beleuchteter Isoflächen [Westermann und Ertl 1998] und beleuchteter Volumen [Rezk-Salama et al. 2000] bis hin zu durchscheinenden (*translucent*) Volumenobjekten mit mehrfachem Scattering [Kniss et al. 2002]. Der Einsatz von Prä-Integrierter Klassifikation [Engel et al. 2001] beseitigt Artefakte aufgrund der diskreten Abtastung.

Für den Anwendungsprogrammierer stellt sich das Problem, dass all diese Techniken in hohem Maße von der Graphik-Hardware Gebrauch machen, was sehr detaillierte Kenntnisse der Zielplattform erfordert. Darüber hinaus existieren unterschiedliche Möglichkeiten, um abhängig von den Fähigkeiten der Graphik-Hardware eine bestimmte Funktionalität zu realisieren. Transferfunktionen können beispielsweise durch Hardware-unterstützte Farbtabellen oder abhängige Texturnachschläge mittels Texture Shader oder Fragmentprogramm implementiert werden. All dies erschwert die Implementierung portabler Programme, die auf unterschiedlichen Graphikplattformen lauffähig sind.

Der Anwendungsprogrammierer wünscht sich eine abstrakte Programmierschnittstelle (Application Programming Interface – API), mit der es auf einfache Weise möglich ist, Volumenvisualisierungsanwendungen zu realisieren, ohne dabei Rücksicht auf die spezifischen Eigenheiten der Graphik-Hardware nehmen zu müssen. Stattdessen erfolgt zur Laufzeit eine automatische Adaption an die Fähigkeiten der Graphikkarte. Eine geeignete Volumenvisualisierungsbibliothek soll gleichzeitig flexibel und erweiterbar sein, in dem Sinne, dass neue Varianten der texturbasierten Volumenvisualisierung leicht realisierbar sind und auch zukünftige Graphik-Hardware mit neuer Funktionalität integrierbar ist. In diesem Kapitel soll eine Programmierbibliothek basierend auf dem Open-Source Szenengraph OpenSG [OpenSG 2004] vorgestellt werden, mit der es sehr einfach möglich ist, Volumenvisualisierungsanwendungen zu realisieren und auch volumetrische Effekte wie Nebel oder Wolken in beliebige Graphikanwendungen zu integrieren. Entsprechende Veröffentlichungen finden sich in [Weiler und Ertl 2002, Weiler et al. 2004a].

9.1 OpenSG

Bei OpenSG handelt es sich um ein *Szenengraph*-API, ähnlich wie OpenInventor, Java3D, Cosmo3D/Optimizer oder OpenScenegraph, bei dem die darzustellende Szene in Form eines gerichteten, azyklischen Graphen (Directed Acyclic Graph – DAG) modelliert wird. Die Darstellung der Szene erfolgt durch die Traversierung dieser Datenstruktur, wobei die im Szenengraph gespeicherte Geometrie zusammen mit Materialinformation an die Graphik-Hardware zum Zeichnen übergeben wird.

Damit bieten Szenengraphen für den Anwender den Vorteil, dass er auf einer abstrakteren Ebene programmieren kann und sich weniger mit der Datenhaltung oder den Low-Level-Schnittstellen der Graphik-Hardware auseinandersetzen muss. Letzteres ist ein entscheidender Punkt für die effiziente Entwicklung komplexer Anwendungen mit dynamischen, eventuell animierten Szenen, in denen ggf. interaktiv navigiert werden soll. Auf der anderen Seite verliert er die Möglichkeit, direkt auf die Verarbeitungsschritte der Rendering-Pipeline einzuwirken, was bei der Verwendung von Low-Level-Graphikprogrammierschnittstellen wie OpenGL oder DirectX möglich ist. Viele Szenengraph-APIs erlauben jedoch den Durchgriff auf die darunterliegende Low-Level-Graphikschnittstelle.

Die Modellierung der Szene als explizite Datenstruktur erlaubt es dem Szenengraph-API, zusätzliche Funktionalität zur Verfügung zu stellen oder Performanzoptimierungen durchzuführen. So bietet OpenInventor beispielsweise einen integrierten Satz von 3D-Manipulator-Objekten, mit denen sich Teile der Szene in intuitiver Weise mit Hilfe der 2D-Maus im Raum bewegen lassen. Cosmo3D/OpenGL Optimizer stellt integrierte Tesselierer zur Umwandlung von NURBS-Flächenbeschreibungen in Dreiecksnetze zur Verfügung. Zur Beschleunigung der Darstellung werden oftmals Culling-Techniken zur Verfügung gestellt, die z.B. im Falle von Occlusion Culling verdeckte Teile der Szene erkennen. Die Teilbäume verdeckter Geometrie können von der Traversierung ausgeschlossen werden, was zur Reduktion der gezeichneten Geometrie führt und damit die Darstellung signifikant beschleunigen kann.

OpenSG wurde 1999 am Fraunhofer Institut für Graphische Datenverarbeitung aus der Taufe gehoben. Vorrangiges Ziel war dabei, VR-Anwendungen zu unterstützen, was durch konsequente Multi-Thread/Multi-Pipe-Unterstützung erreicht werden soll. Daneben stehen Flexibilität, Portablilität und die Erweiterbarkeit im Vordergrund, um eine große Verbreitung und damit die Weiterentwicklung im Open-Source-Umfeld zu sichern. Die langjährigen Ankündigungen verschiedener Firmen zu neuen Szenengraphbibliotheken, deren Entwicklung teilweise schon vor der ersten Implementierung wieder eingestellt wurde, motivierten zehn deutsche Forschungsinstitute beziehungsweise Universitäten, sich zwischen 2001 und 2003 im Rahmen des BMBF Projektes „OpenSG PLUS" an der Weiterentwicklung zu beteiligen. Mit den im Rahmen dieses Projektes entwickelten Erweiterungen, wie beispielsweise Multi-Pipe-/Cluster-Funktionalität, Occlusion Culling, Tesselierung von NURBS-Flächen on-the-fly oder hochqualitativen, Hardware-beschleunigten Shading-Modellen, die zum großen Teil in die aktuelle Version 1.4 von OpenSG eingeflossen sind[1], steht nun eine interessante und offene Plattform für interaktive Graphikanwendungen zur Verfügung. Im Folgenden soll die Integration von Volumenobjekten in das OpenSG-Szenengraph-API beschrieben werden.

9.2 Hardware-Abstraktion

Die Grundidee der hier vorgestellten flexiblen und erweiterbaren Volumenvisualisierungsbibliothek für das Szenengraph-API OpenSG nach dem Prinzip der texturbasierten Volumenvisualisierung lehnt sich an OpenGL Volumizer [Eckel 1998] an und beruht auf der Beobachtung, dass alle bekannten Algorithmen auf demselben Grundalgorithmus basieren, so wie er in Abbildung 9.1 dargestellt ist. Dies gilt unabhängig davon, ob zur Darstellung zweidimensionale oder dreidimensionale Texturen verwendet werden.

```
1    Sortiere Bricks Back-to-Front
2
3    for (alle Bricks) {
4
5        Lade Brick-Textur in den Texturspeicher
6
7        for (alle Slices)
8            Zeichne alle Polygone des Slice
9                innerhalb des Bricks Back-to-Front
10   }
```

Abbildung 9.1: Pseudocode der texturbasierten Volumenvisualisierung mit Bricking.

[1] Einen interessanten Überblick gibt die Artikelserie zum OpenSG-Symposium 2003, erschienen in Computers & Graphics 28 (2004), Seiten 59-112.

In beiden Situationen erfolgt die Darstellung des Volumens, wie bereits in Abschnitt 3.3.2 beschrieben, durch das Betrachten von Schnittebenen, sogenannter Slices, die an den Grenzen des Volumens geschnitten werden. Die resultierenden Schnittpolygone werden dann sortiert von hinten nach vorne gezeichnet und dabei mittels Blending in den Framebuffer geschrieben.

Bei Verfügbarkeit von dreidimensionalen Texturen wird das Volumen als 3D-Textur definiert und die Schnittebenen sind senkrecht zur Blickrichtung ausgerichtet. Für Volumen, welche die Größe des verfügbaren Texturspeichers überschreiten, erfolgt zusätzlich eine Unterteilung des Volumens in *Bricks* oder *Tiles*, welche unter Berücksichtigung der Reihenfolge, die sich aus der Sichtbarkeit ergibt, unabhängig gezeichnet werden können. In diesem Fall müssen die Schnittpolygone zusätzlich noch an den Brick-Grenzen geschnitten werden.

Stehen nur zweidimensionale Texturen zur Verfügung, wird das Volumen als Stapel von 2D-Texturen definiert und die Schnittebenen sind senkrecht zu der Hauptachse des Volumens ausgerichtet, welche den kleinsten Winkel zur Blickrichtung bildet. Konzeptionell lässt sich, wie in Abbildung 9.2 zu sehen, im zweidimensionalen Fall eine einzelne 2D-Textur als ein Brick interpretieren, womit die Darstellung des Volumens auch in diesem Fall auf denselben Grundalgorithmus führt. Im Gegensatz zu den Bricks im dreidimensionalen Fall wird hier allerdings nur jeweils ein Slice je Brick gezeichnet. Auch sind, aufgrund der Definition der 2D-Schichten in jeder der drei möglichen Dimensionen, drei Sätze von Bricks erforderlich.

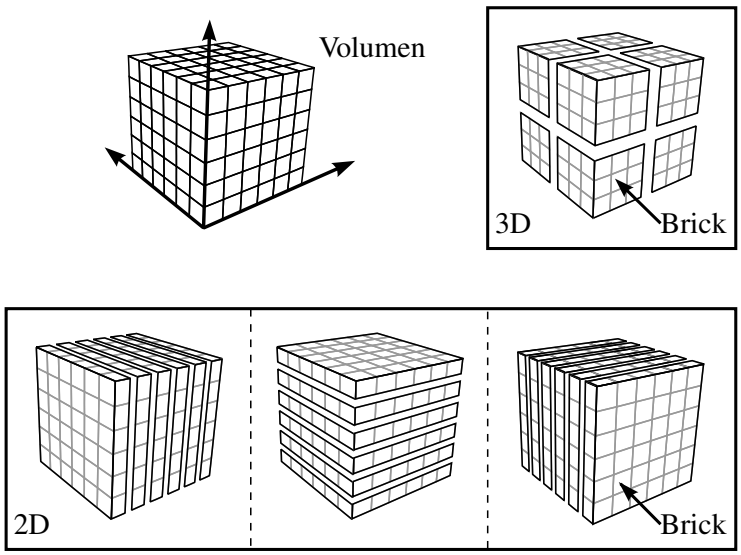


Abbildung 9.2: Die Interpretation von 2D-Texturen als Bricks erlaubt eine Vereinheitlichung der texturbasierten Volumenvisualisierung mittels zweidimensionaler oder dreidimensionaler Texturen.

Da der wesentliche Unterschied zwischen den Variationen texturbasierter Volumenvisualisierung folglich nicht in der gezeichneten Geometrie sondern lediglich im OpenGL-Zustand besteht, mit dem die Slice-Geometrie gezeichnet wird, sollte der Anwender folglich die gemeinsamen Teile wie Texturverwaltung und Slicing nicht immer wieder programmieren müssen. Diese Grundfunktionalität wird von der vorgestellten Bibliothek erledigt. Um dennoch die Möglichkeit zu

eröffnen, den OpenGL-Zustand beeinflussen zu können, wird hier ein Callback-Mechanismus verwendet, wobei ein Shader-Objekt als Callback-Einsprungpunkt dient. Somit kann ein neuer Darstellungsalgorithmus oder eine Hardware-spezifische Implementierung sehr einfach durch Implementierung eines spezialisierten Shader-Objektes integriert werden.

9.3 Shader-Schnittstellen

Die Architektur der vorgestellten Bibliothek besteht aus den vier Komponenten, die in Abbildung 9.3 dargestellt sind: einem Renderer, einem Texture Manager, einem Slicer und dem Shader. Die zentrale Steuerinstanz ist der Renderer, der die übrigen Komponenten initialisiert. Der Shader entscheidet bei der Initialisierung abhängig vom gewünschten Visualisierungseffekt, wie viele Texturen und welche Art von Texturen er benötigt. So kann er beispielsweise für einfaches Slicing eine Luminanz-Textur mit den Volumenskalarwerten definieren, für beleuchtete Isoflächen, die mittels Slicing gezeichnet werden, können RGBA-Texturen mit Gradienteninformation in RGB und dem Volumenskalarwert im Alpha-Kanal definiert werden. Die Texturen werden beim Texture Manager registriert, der auch mehrere Texturen unterstützt, wie es zur Darstellung von interpolierten Zwischenschichten zwischen 2D-Texturen [Rezk-Salama et al. 2000], der Verwendung von abhängigen Texturnachschlägen zur Realisierung von Transferfunktionen oder der Prä-Integrierten Klassifikation [Engel et al. 2001] erforderlich ist. Eine weitere Anwendung von Multi-Texturen sind Tag-Texturen oder Stencil-Texturen, die mit Hilfe des OpenGL-Alpha-Tests das Ausblenden bestimmter Voxel des Volumens durch Multiplikation mit einem binären Klassifikationswert erlauben. Für alle benötigten Texturen wählt der Shader auch geeignete OpenGL-Texturformate und übergibt die Texturen an den Texture Manager.

Der Texture Manager realisiert das Ressourcen-Management für die Texturen inklusive Bricking. Er untersucht die Hardware in Hinblick auf den verfügbaren Texturspeicher und die Anzahl der unterstützten Textureinheiten und lässt den Shader nur so viele Texturen registrieren, wie Textureinheiten verfügbar sind. Schließlich erfolgt die Zerlegung in einzelne Bricks. Dabei unterscheidet der Manager Texturen, die nicht zerlegt werden dürfen, wie abhängige Texturen zur Realisierung einer Transferfunktion oder zum Nachschlagen im Rahmen der Prä-Integrierten Klassifikation. Diese werden zunächst vom verfügbaren Texturspeicher abgezogen. Anschließend werden

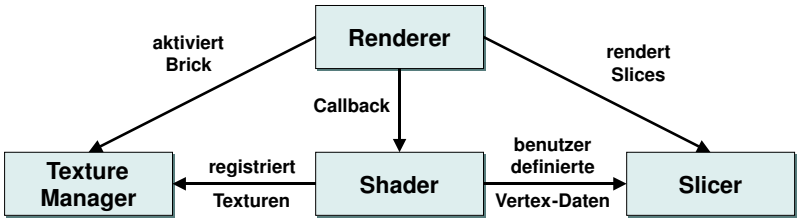


Abbildung 9.3: Im Zusammenspiel zwischen Shader, Textur-Management und Slicer lassen sich unterschiedliche Visualisierungseffekte mit texturbasierter Volumenvisualisierung erreichen.

die übrigen Texturen, die Attribute wie Datenwerte oder Gradienten für jeden Voxel beschreiben, so unterteilt, dass jeweils ein Brick von jeder Textur gleichzeitig in den Texturspeicher geladen werden kann.

Nach der Initialisierung steuert der Renderer den eigentlichen Slicing-Algorithmus wie in Abbildung 9.1 angedeutet. Er sortiert die Bricks von hinten nach vorne und aktiviert über den Texture Manager alle erforderlichen Brick-Texturen. Für jeden Brick ruft er einen entsprechenden Callback des Shaders auf, der den korrekten OpenGL-Zustand initialisiert. Danach zeichnet er über den Slicer alle Slice-Polygone des Bricks. Dies erfolgt direkt aus dem Code zur Slice-Berechnung durch den Aufruf der entsprechenden `glVertex`-Aufrufe. In dieser Hinsicht ist die hier vorgestellte Lösung effizienter als die von OpenGL Volumizer, der erst alle Slice-Polygone für das Volumen in einer Zwischendatenstruktur speichert, was einen signifikanten Mehraufwand durch Lesen und Schreiben sowie die dynamische Vergrößerung und Verkleinerung dieser Datenstruktur erzeugt.

Für manche Visualisierungseffekte kann es erforderlich sein, zusätzliche Parameter je Vertex zu definieren. Für Deformation des Volumens sind beispielsweise explizite Texturkoordinaten erforderlich, weil hierbei die Texturkoordinate bei Verschieben der Vertexposition konstant bleibt. Im Allgemeinen lässt sich keine Aussage darüber machen, welche Parameter zukünftige Shader erfordern werden. Um zu vermeiden, den Algorithmus für die Slice-Berechnung jeweils an unterschiedliche benutzerdefinierte Knotendaten anpassen zu müssen, besitzt der Shader die Möglichkeit, beliebige Daten pro Vertex zu definieren. Diese werden bei der Berechnung der Slice-Polygone linear auf den Kanten der Geometrie interpoliert. Statt unterschiedliche Knotenformate zu definieren, wie in Volumizer realisiert, wird hier ein generisches Sammelformat vorgeschlagen. Als User-Daten je Knoten kann dadurch eine beliebige Anzahl von Gleitkommawerten definiert werden.

Dies erfordert jedoch eine eigene Routine zum Zeichnen der Slice-Polygone durch den Shader. Sie muss die benutzerdefinierten Knotenparameter in geeigneter Weise interpretieren und in die richtigen OpenGL-Kommandos umsetzen. Shader-spezifische Render-Methoden werden ebenfalls über einen Callback angesteuert, der als Parameter das jeweilige Slice-Polygon mit allen interpolierten Knotendaten erhält. Da dieser Mechanismus jedoch wiederum zu einem Mehraufwand durch Schreiben, Lesen und Kopieren der Daten führt, wird er nicht standardmäßig verwendet, sondern wird nur wenn der Shader explizit eine eigene Render-Methode beim Renderer registriert.

9.4 Volumenmodellierung in OpenSG

Für die Integration der volumetrischen Objekte in eine OpenSG-Szene wurde eine Menge von neuen Szenengraphobjekten entworfen, die in Abbildung 9.4 dargestellt ist. Nach außen bietet die Volumenvisualisierungsbibliothek nur einen einzigen neuen Szenengraphknoten `osg:DVRVolume` an. Alle weiteren Objekte wurden als sogenannte *Fieldcontainer* realisiert, die an den Volumenknoten angehängt werden. Diese Struktur bietet gegenüber einem Teilgraphen aus mehreren Szenengraphknoten den Vorteil, dass die Traversierung nicht durch die

Standardmechanismen von OpenSG erfolgt, die für die hier vorgestellten Zwecke zu restriktiv sind. Darüber hinaus könnte es leicht zu Problemen kommen, wenn ein Anwender „normale" OpenSG-Knoten in den Volumenteilbaum einhängt. Der Aufbau der Volumendatenstrukturen aus Feldcontainern entkoppelt auf elegante Weise das Zeichnen des Volumens von der OpenSG-Szenengraphtraversierung, ohne die Multi-Threading- und Cluster-Fähigkeit von OpenSG zu beeinflussen. Der Volumenknoten arbeitet somit, wie in Abschnitt 9.7 gezeigt wird, auch in OpenSG-basierten virtuellen Umgebungen.

Die *Appearance* ist abgeleitet von der Klasse `osg::Material` und beschreibt das Erscheinungsbild, also die visuellen Attribute des Volumens. In erster Linie sind dies die Volumendaten, die Skalarwerte in den Voxeln. Sie werden als 3D-Bild (`osg::Image`) gespeichert, wodurch die allgemeine Ladefunktionalität von OpenSG ausgenutzt werden kann. Hinzu kommen weitere Parameter, die das Erscheinungsbild der Visualisierung beeinflussen: Eine Transferfunktion bildet die Skalarwerte auf Farbe und Transparenz ab, wobei Transferfunktionen, wie in [Kniss et al. 2001] vorgeschlagen, mehr als eine Dimension besitzen können. Ambiente und spekulare Farbe beschreiben Beleuchtungsparameter, die beispielsweise für beleuchtete Isoflächen verwendet werden. Weitere Parameter wie Tag-Volumen oder eine Environment Map sind denkbar. Diese Volumenparameter verwendet der Shader, um die erforderlichen Texturen aufzubauen und den passenden OpenGL-Zustand zu setzen.

Jeder Shader kann einen unterschiedlichen Satz an Parametern benötigen. Diese müssen als Member in der Appearance enthalten sein. Im Sinne einer Erweiterung des Volumenvisualisierungs-APIs soll es möglich sein, neue Shader zu implementieren. Dies erfordert die nachträgliche Erweiterung der Appearance-Klasse. Eine Ableitung der Basisklasse zur Erweiterung wäre jedoch ein zu großer Mehraufwand und führt zu Typ-Problemen sowie zur Notwendigkeit der Verwendung von Mehrfachvererbungen. Deshalb wird der Attachment-Mechanismus von OpenSG verwendet, der prinzipiell eine beliebige Anzahl benutzerdefinierter Daten im Sinne von User-Data-Zeigern realisiert. Alle Parameter in der Appearance sind von der Klasse `osg::Attachment` abgeleitet. Werden für neue Shader weitere Parameter erforderlich, so sind diese ebenfalls von `osg::Attachment` abzuleiten. Jedes Attachment erhält

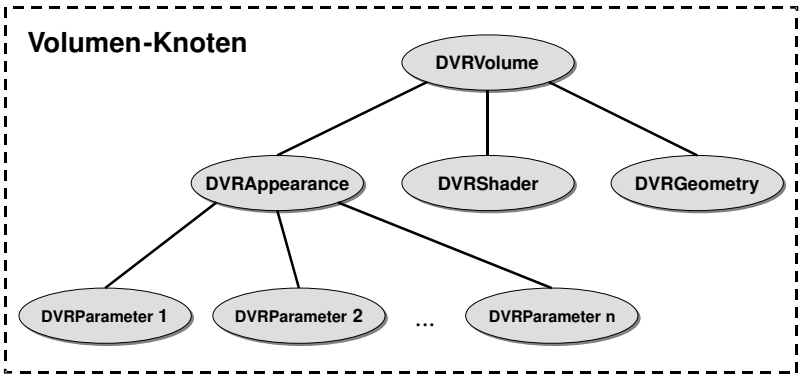


Abbildung 9.4: Aufbau des OpenSG-Volumenknotens aus einzelnen Feldcontainern.

einen eindeutigen Namen, das seine Identifizierung und damit die Referenzierung durch die Shader erlaubt. Bei der Initialisierung überprüft der Shader, ob die Appearance die erforderlichen Attachments enthält, und erzeugt gegebenenfalls einen Fehler.

Die *Geometrie* ist eine Spezialisierung von `osg::Geometry` und beschreibt die geometrische Ausdehnung des Volumens. Sie kann als beliebiges geschlossenes Dreiecksnetz definiert sein. Aus diesem Dreiecksnetz erfolgt die Berechnung der Slices durch Methoden der Klasse `osg::DVRGeometry`. Dazu wird ein iterativer Algorithmus verwendet, der den zuletzt berechneten Slice heranzieht, um den nächsten Slice sehr schnell zu ermitteln. Das Slice-Polygon wird als geschlossener Linienzug zurückgeliefert, was nachträglich ein Clipping an den Brick-Grenzen oder einer zusätzlich verwendeten Clip-Geometrie ermöglicht. Neben diesem allgemeinen Dreiecksnetz ist auch ein `osg::DVRBlock` vorgesehen, der eine einfache Würfelgeometrie repräsentiert. Eine Sonderbehandlung dieser Geometrie ist gerechtfertigt, da sie den häufigsten Fall darstellen wird und sich die Schnittpolygone in einem Block im Vergleich zum Dreiecksnetz sehr effizient berechnen lassen.

Der *Shader* wurde bereits in Abschnitt 9.3 beschrieben.

9.5 Volumen-Shader

Im Folgenden sollen einige exemplarisch implementierte Shader zusammen mit ihren Hardware-spezifischen Implementierungen vorgestellt werden, die es dem Shader erlauben, sich an die Fähigkeiten der aktuellen Graphikplattform anzupassen. Letzteres ist von entscheidender Bedeutung, nachdem OpenSG IRIX, Linux, Windows und MacOS unterstützt und die Volumenbibliothek damit mit einer Vielzahl unterschiedlicher Hardware von High-End-Graphik-Workstations bis zu der großen Palette von PC-Graphikkarten zurechtkommen muss.

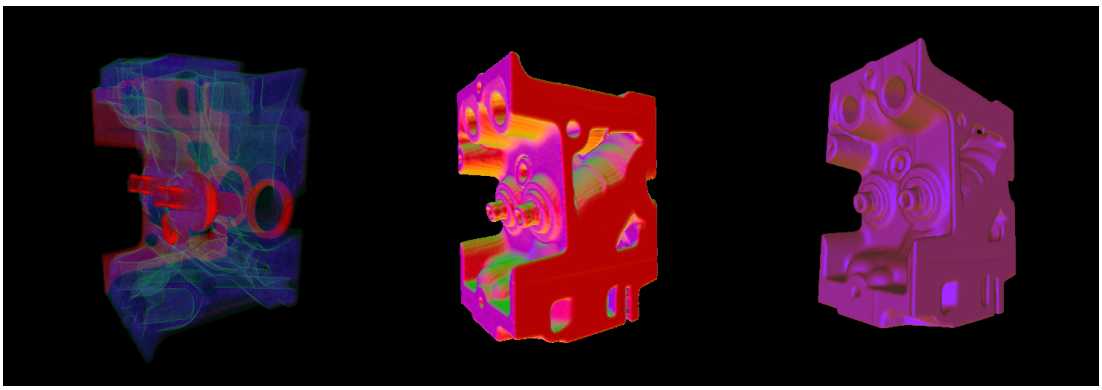


Abbildung 9.5: Verschiedene Darstellungsmodi, die über Shader-Objekte in OpenSG erzielt werden. In der linken Abbildung wird eine geeignete Transferfunktion verwendet, das mittlere Bild zeigt eine Isofläche, die diffus von sechs verschiedenfarbigen Lichtquellen beleuchtet wird. Die rechte Abbildung zeigt dieselbe Isofläche, die von drei Lichtquellen mit jeweils diffusem und spekularem Anteil beleuchtet werden.

9.5.1 Color-Table-Shader

Die Möglichkeit, die Transferfunktion – die Abbildung von Volumenwerten auf Farbe und Opazität – zu modifizieren, ist vielleicht das grundlegendste Werkzeug für die effiziente Analyse eines Volumendatensatzes. Da die Auswahl einer geeigneten Transferfunktion oftmals Benutzerinteraktion erfordert, ist dabei die Modifikation der Transferfunktion in Echtzeit eine wichtige Eigenschaft.

Post-Klassifikation

Der hier vorgestellte `osg::DVRSimpleLUTShader` erreicht diese Interaktivität durch die Ausnutzung unterschiedlicher Hardware-Funktionalität. Eine Möglichkeit besteht in der Ausnutzung der OpenGL-Erweiterung `SGI_texture_color_table`, die von allen IRIX-basierten SGI Graphik-Workstations und Graphik-Servern sowie einigen ATI Graphikkarten unterstützt wird. Das auf diese Weise realisierte Klassifikationsmodell ist Post-Klassifikation (vgl. Abschnitt 3.1.3), da hierbei für jedes Pixel zunächst der Texturnachschlag erfolgt. Der linear interpolierte Texturwert wird anschließend über eine eindimensionale Tabelle auf Farbe und Opazität abgebildet.

Eine alternative Möglichkeit besteht darin, den nachgeschlagenen Volumenwert als Texturkoordinate zu verwenden und die Farbe und Opazität über einen zweiten, abhängigen Texturnachschlag zu ermitteln. Diese Implementierung kommt bei Graphik-Hardware, die programmierbare Fragmenteinheiten (`ARB_fragment_program`) oder die OpenGL-Erweiterung Texture Shader von nVidia (`NV_texture_shader`) unterstüzt, zur Anwendung. Alle drei Methoden führen zu derselben Bildqualität bei vergleichbarer Performanz, weshalb der Shader über die entsprechenden Zugriffsmethoden von OpenSG (`osg::Window::hasExtension`) der Reihe nach die Verfügbarkeit dieser drei OpenGL-Erweiterungen abprüft und die Methode auswählt, welche der ersten gefundenen Erweiterung entspricht. Ein Beispiel für einen mittels Transferfunktion semi-transparent dargestellten Volumendatensatz ist in Abbildung 9.5 links dargestellt.

Prä-Klassifikation

Zwei weitere Methoden wurden als Fallback-Lösung realisiert, da sie im Gegensatz zu den oben erwähnten Techniken Prä-Klassifikation verwenden. Diese wird von vielen Autoren als qualitativ schlechter eingeschätzt, wenngleich die Frage, ob Prä- oder Post-Klassifikation „korrekte" Bilder liefert, immer noch kontrovers diskutiert wird.

Die erste Methode verwendet die OpenGL-Erweiterung `NV_paletted_textures`[2] und erlaubt immer noch eine interaktive Modifikation der Transferfunktion, da bei Änderungen an der Transferfunktion nur eine verhältnismäßig kleine Tabelle ausgetauscht werden muss. Im Kontrast dazu lädt die zweite Methode die Volumentextur erneut auf die Graphikkarte und verändert

[2] In neueren Versionen der nVidia-Treiber wird diese Erweiterung nicht mehr unterstützt, da nVidia diese Funktionalität aufgrund der in modernen Graphikkarten verfügbaren Fragmentprogramme für verzichtbar hält.

dabei die Farbe und Opazität jedes Voxels abhängig von der spezifizierten Transferfunktion. Besonders für große Volumen kann dies nicht mehr interaktiv erfolgen, obwohl mit der OpenGL-Erweiterung `GL_SGI_color_table` auch Hardware-Funktionalität verwendet werden kann, um die Volumenwerte auf Farbe und Opazität abzubilden. Weiterhin verwendet diese Methode RGBA-Texturen anstelle von Luminanz-Texturen mit nur einem Wert je Voxel, was eine Vervierfachung des Speichers zur Folge hat. Der große Vorteil dieser Methode liegt jedoch darin, dass sie mit der Standardfunktionalität von OpenGL auskommt.

Alle diese Methoden unterstützt der `osg::DVRSimpleLUTShader` sowohl für zweidimensionale als auch für dreidimensionale Texturen. Allerdings erfordert dies im Falle von abhängigen Texturnachschlägen mittels Texture Shader oder Fragmentprogrammen jeweils eigenständige Shader-Programme, da die Operationen zum Texturnachschlag die Dimensionalität der Textur enthalten. Dies ist bei älteren Programmiermodellen nicht der Fall. Weder `NV_register_combiners` noch `ATI_fragment_shader` kodieren die Dimensionalität der Quelltextur.

Ähnliche Spezial-Shader benögtigt auch die Darstellung von Zwischenschichten mittels 2D-Texturen, da hier der Nachschlag der Volumenwerte in zwei nebeneinanderliegenden Texturen erfolgen muss und die nachgeschlagenen Werte mittels linearer Interpolation entsprechend der Lage des Slice-Polygons zu den Texturschichten kombiniert werden müssen. Da hierbei auch modifizierte Textureinstellungen inklusive zweier Sätze von Texturkoordinaten an den Vertizes der Slice-Polygone erforderlich sind, wurde dieser Modus über einen eigenständigen Shader (`osg::DVRMtexLUTShader`) realisiert, der von `osg::DVRSimpleLUTShader` abgeleitet ist.

9.5.2 Isoflächen-Shader

Der in OpenSG realisierte Isoflächen-Shader `osg::DVRIsoShader` zeichnet nicht-polygonale Isoflächen mittels eines Slicing-Ansatzes. Die Idee wurde erstmals von Westermann und Ertl [Westermann und Ertl 1998] vorgeschlagen und beruht, ähnlich wie die texturbasierte Volumenvisualisierung darauf, texturierte Slice-Polygone zu zeichnen. Der Volumenwert wird dabei auf den Alpha-Kanal der generierten Fragmente abgebildet. Durch geschickte Ausnutzung des Alpha-Tests lassen sich nun diejenigen Fragmente aussortieren, die einen Volumenwert größer als ein vorgegebener Isowert besitzen, was in Kombination mit dem Tiefentest die First-Hit-Semantik beim Ray Casting von Isoflächen simuliert. Das Shading der Isofläche erfolgt basierend auf den Gradienten des Volumens, die im RGB-Kanal der Volumentextur gespeichert werden.

Um die Anpassung an unterschiedliche Graphik-Hardware zu erlauben, wurden wiederum mehrere Implementierungen realisiert, die der Shader abhängig von den Fähigkeiten der Graphik-Hardware automatisch auswählt. Die erste Version entspricht der in [Westermann und Ertl 1998] vorgeschlagen Methode und verwendet mit der Color Matrix eine weit verbreitete OpenGL-Funktionalität. Nachdem alle Slices gezeichnet wurden, enthält der Framebuffer zunächst ein Bild der Gradientenwerte. Dieses wird auf sich selbst kopiert, wobei eine geeignete Definition der

Color Matrix die diffuse Beleuchtung auf der Isofläche berechnet. Die Ausnutzung von Stencil-Buffer und Stencil-Test garantiert dabei, dass nur Pixel, die zum Volumen gehören, tatsächlich kopiert werden. Letzteres erlaubt eine kombinierte Darstellung von Volumen mit polygonalen Szenengraphobjekten. Neben dem Mehraufwand für den verhältnismäßig aufwändigen Kopierprozess hat diese Methode den Nachteil, dass lediglich diffuse Beleuchtung möglich ist und nur eine einzelne Lichtquelle unterstützt wird. Daher wird diese Methode als Fallback betrachtet.

Die zweite Implementierung beseitigt diese beiden Schwächen durch Ausnutzung der OpenGL-Erweiterung `NV_register_combiners`, die z.B. vom nVidia GeForce3 Chip und neueren Varianten unterstützt wird. Eine geeignete Einstellung der Kombination der Farbwerte mittels Register Combiners erlaubt die Berechnung der diffusen Beleuchtung mit bis zu sechs direktionalen Lichtquellen oder der diffusen und spekularen Beleuchtungsbeiträge von bis zu drei Lichtquellen on-the-fly. Der Zeichendurchgang zum Kopieren entfällt dadurch komplett. Abbildung 9.6 demonstriert die nötigen Einstellungen für sechs diffuse Lichtquellen. Sie werden algorithmisch erzeugt, was eine Erweiterung auf zusätzliche Lichtquellen erlaubt, sofern die Graphik-Hardware mehr General Combiners unterstützt. Jeweils zwei weitere Lichtquellen benötigen dabei einen weiteren Funktionsblock aus drei General Combiners, so wie er in der rechten oberen Ecke der Abbildung dargestellt ist. Für ältere Versionen des nVidia-Chips, respektive GeForce256 und GeForce2, die lediglich zwei General Combiners anbieten, wurde eine genügsamere Version implementiert, die maximal zwei diffuse Beleuchtungsterme unterstüzt oder alternativ dazu einen diffusen und einen spekularen Beleuchtungsterm nur mit Intensität, d.h. weißer Lichtquelle.

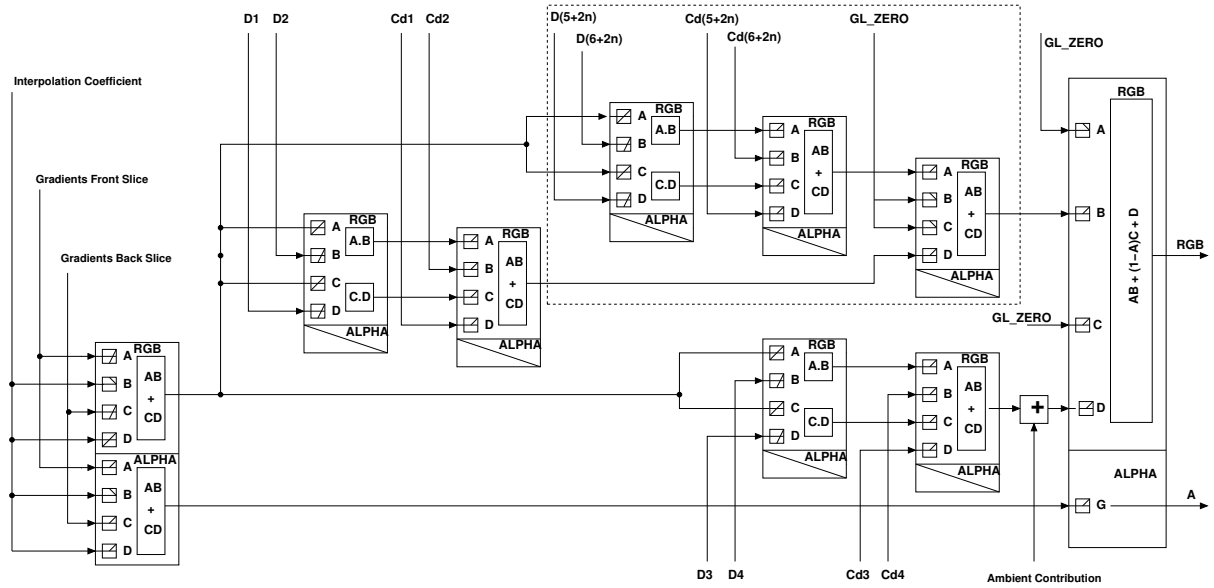


Abbildung 9.6: Einstellung der Register Combiners für die diffusen Beleuchtungsbeiträge von bis zu sechs direktionalen Lichtquellen. Demonstriert werden die Einstellungen für zweidimensionale Texturen mit Zwischenschichten.

Darüber hinaus bietet der Shader eine Implementierung mit Hilfe von Fragmentprogrammen an. Die hierbei ebenfalls algorithmisch erzeugten Shader-Programme unterstützen eine virtuell beliebige Anzahl von diffusen und spekularen Beleuchtungstermen, die nur durch die Anzahl der verfügbaren Fragmentprogramm-Instruktionen sowie die Anzahl der für den Shader definierbaren Parameter beschränkt ist, was in der Praxis vermutlich niemals erreicht wird.

Alle beschriebenen Implementierungen lassen sich wiederum sowohl mit dreidimensionalen Texturen als auch mit zweidimensionalen Texturen inklusive interpolierter Zwischenschichten kombinieren. Beispiele für beleuchtete Isoflächen, die mittels Slicing eines $256 \times 256 \times 110$ Voxel großen Volumens gezeichnet wurden, finden sich im mittleren und rechten Bild von Abbildung 9.4.

9.5.3 Prä-Integrations-Shader

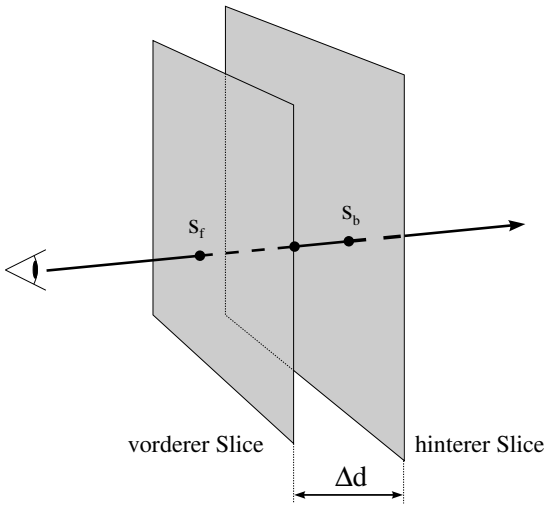
Die Volumendarstellung basierend auf Prä-Integrierter Klassifikation (vgl. Abschnitt 3.1.5), welche zuerst im Kontext von Zellprojektion auf Tetraedergittern veröffentlicht wurde, hat sich auch im Umfeld der texturbasierten Volumenvisualisierung seit ihrer Erstveröffentlichung [Engel et al. 2001] als wichtige Standardtechnik etabliert. Wesentliche Gründe dafür liegen in der höheren Darstellungsqualität, die aus der weitgehenden Vermeidung von Sampling-Artefakten herrührt. Insbesondere bei hohen Frequenzen in der Transferfunktion, wie sie z.B. durch Simulation einer Isofläche mittels einer Spitze in der Transferfunktion erzielt werden kann, erreicht Prä-Integriertes Volume Rendering bessere Ergebnisse als Post-Klassifikation bei gleicher Abtastrate, d.h. gleicher Anzahl von Schnittebenen. Alternativ kann bei Prä-Integrierter Klassifikation die Anzahl der Schnittebenen bei vergleichbaren Ergebnissen reduziert werden, was zu deutlichen Geschwindigkeitssteigerungen führt. Weitere Vorteile liegen darin, dass durch einfachen Austausch der Prä-Integrationstabelle unterschiedliche optische Modelle erreicht werden können und auch Beleuchtungseffekte relativ einfach integrierbar sind [Lum et al. 2004].

Für OpenSG wurde nur eine Lösung mittels dreidimensionaler Texturen realisiert, da diese gegenüber der Implementierung mittels zweidimensionaler Texturen den Vorteil bietet, dass zweidimensionale Tabellen für den Nachschlag der integrierten Strahlstücke verwendet werden können. Bis vor kurzem war dies die einzige Möglichkeit, interaktive Modifikation der Transferfunktion zu erlauben. Außerdem verlangt der Nachschlag der Strahlsegmente in der Prä-Integrationstextur nach der Unterstützung für abhängige Texturnachschläge und mit Ausnahme vom nVidia GeForce256 Chip unterstützen alle bisher veröffentlichten Graphikkarten mit der Fähigkeit abhängiger Texturnachschläge auch dreidimensionale Texturen.

Der realisierte `osg::DVRPreIntShader` führt anstelle einer einfachen Abtastung der Volumentextur an der Position des aktuellen Slices einen weiteren Texturnachschlag an der Position des nächsten Slices durch, was, wie in Abbildung 9.7(a) gezeigt wird, einer Betrachtung des Volumens in sogenannten *Slabs* entspricht. Für jeden dieser Slabs kann nun der Farb- und Opazitätsbeitrag durch den Nachschlag in einer als Textur realisierten Tabelle ermittelt und mittels Blending in den Framebuffer geschrieben werden. Da im Falle von senkrecht zur Blickrichtung

ausgerichteten Slice-Polygonen der Abstand zwischen zwei Slices Δd konstant ist, dienen als Koordinaten für diesen Lookup lediglich der Volumenwert am vorderen Slice s_f und der Volumenwert am hinteren Slice s_b. Für die Berechnung der korrekten Texturkoordinaten für den zweiten Nachschlag verwendet der Prä-Integrations-Shader den in Abschnitt 9.3 vorgestellten Callback-Mechanismus. An jedem Vertex des Slice-Polygons werden die vom Renderer berechneten Texturkoordinaten dazu entlang eines Strahls vom Auge zu diesem Punkt nach hinten auf den nächsten Slice projiziert.

Um auch für diesen Shader weitestgehende Adaption an die aktuell verwendete Graphikplattform ermöglichen zu können, wurden wiederum zwei Implementierungen vorgesehen, die der Shader abhängig von der Verfügbarkeit der benötigen OpenGL-Erweiterungen automatisch auswählt. Die erste Methode verwendet dabei das in Abbildung 9.7(b) dargestellte Fragmentprogramm und setzt somit die Erweiterung `ARB_fragment_program` voraus. Die zweite Implementierung entspricht der in der Originalveröffentlichung beschriebenen Realisierung mittels `NV_register_combiners` und `NV_texture_shaders` (siehe [Engel et al. 2001]). Abbildung 3.2 zeigt ein Beispiel für ein mittels Prä-Integrations-Shader gezeichnetes Volumen.



(a)

```
1    !!ARBfp1.0
2    TEMP sf; # front scalar
3    TEMP sb; # back scalar
4    TEMP depCoor;
5
6    PARAM bias  = {0.001953125}; # 1/512
7    PARAM scale = {0.99609375}; # 255/256
8    PARAM const = {0.0};
9
10   TEX sf, fragment.texcoord[0], texture[0], 3D;
11   TEX sb, fragment.texcoord[1], texture[0], 3D;
12
13   MOV depCoor, const.x;
14   MOV depCoor.x, sfront;
15   MOV depCoor.y, sback;
16   MAD depCoor, depCoor, scale.x, bias.x;
17   TEX result.color, depCoor, texture[2], 2D;
18
19   END;
```

(b)

Abbildung 9.7: (a) Slab-basierte Repräsentation für texturbasierte Volumenvisualisierung mit Prä-Integration. Die Skalarwerte am Eintritts- und Austrittspunkt des Sehstrahls werden mit s_f beziehungsweise s_b bezeichnet. Der Slice-Abstand beträgt Δd. (b) Fragmentprogramm zur Realisierung von Prä-Integriertem Volume Rendering mittels dreidimensionaler Texturen. Scale und Bias sorgen für eine korrekte Abbildung auf den gültigen Wertebereich der Prä-Integrationstextur.

9.6 Volumen-Clipping

Die Verdeckung ist ein wesentliches Problem bei der Volumenvisualisierung, da interessante Merkmale vielfach im Inneren eines Volumendatensatzes verborgen sind. Mit Transferfunktionen alleine ist es oftmals nicht möglich, ungewollte Bereiche auszublenden, weil aufgrund ähnlicher Volumenwerte eventuell auch relevante Teile des Volumens transparent dargestellt werden. Zusätzlich zur Datenwert-basierten Filterung ist daher die Geometrie-basierte Filterung der Volumendaten ein wichtiges Werkzeug für eine effektive Volumenvisualisierung. In OpenSG sind daher zwei Methoden realisiert, um ungewollte Volumenbereiche aufgrund ihrer Geometrie auszublenden.

Die erste Methode basiert auf frei positionierbaren Clip-Ebenen, die mit Hilfe der von OpenGL zur Verfügung gestellten Clip-Ebenen realisiert werden (siehe Abbildung 9.8 links). In OpenSG wurden sie in Form eines *MaterialChunk* integriert, welcher sich in jedes beliebige Material einfügen lässt. MaterialChunks sind die in OpenSG verwendete Abstraktion von Sammlungen von Attributen einer Geometrie. Jede Geometrie besitzt ein Material, das aus einer Menge von MaterialChunks besteht. Texturen werden in MaterialChunks gespeichert, ebenso wie die Menge der diffusen, spekularen usw. Oberflächenfarben aber auch Füllstil und Linienbreite. Effektiv verbirgt sich hinter den MaterialChunks eine Abstraktion für Teile des OpenGL-Zustands. OpenSG verwendet sie unter anderem für Performanz-optimierende Zustands-Sortierung.

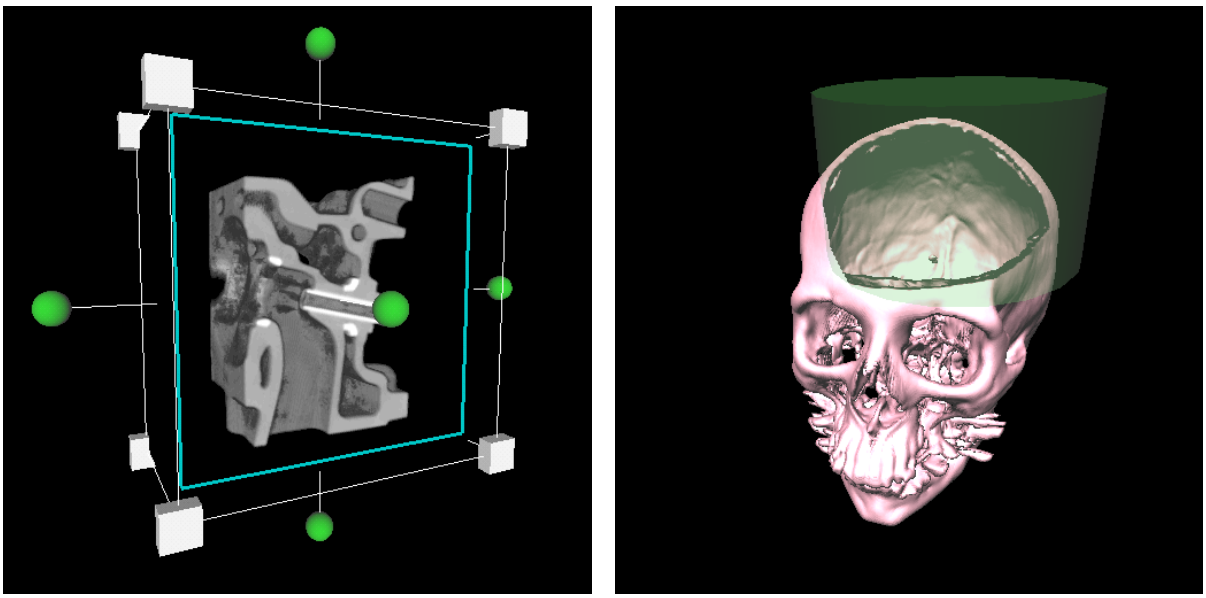


Abbildung 9.8: Demonstration der beiden unterschiedlichen Clipping-Mechanismen, die von OpenSG-Volumenobjekten unterstützt werden. Die Clip-Ebenen auf der linken Seite wurden mit Standardfunktionalität von OpenGL erzielt. Darüber hinaus besteht die Möglichkeit das Volumen mit Hilfe eines geometrisch definierten Clip-Objekts zu modifizieren.

Die Position und Orientierung der Clip-Ebene wird durch das lokale Koordinatensystem eines beliebigen Szenengraphknoten spezifiziert, der dem Clip-Chunk als sogenannter *Beacon* dient. Die Möglichkeit der Modifikation der Clip-Ebene über die Rotation und Verschiebung eines Szenengraphknotens erlaubt die Verwendung der Manipulatorprogrammierbibliothek OpenManip [Braitmaier et al. 2003] als elegante Benutzerschnittstelle. Sie lehnt sich an die Manipulatorfunktionalität von OpenInventor an, ermöglicht aber aufgrund einer abstrakten Szenengraphschnittstelle den plattformübergreifenden Einsatz mit unterschiedlichen Szenengraph-APIs. OpenManip entstand ebenfalls im Rahmen der hier vorgestellten Arbeit.

Ein wesentlicher Vorteil der hier vorgeschlagenen Implementierung ist, dass sich die Clip-Ebenen nicht nur auf ein Volumenobjekt auswirken, sondern vielmehr mit jeder beliebigen Geometrie kombiniert werden können. Da MaterialChunks darüber hinaus lokal auf die Geometrie wirken und sich nicht im Szenengraph vererben, sind Anwendungen wie das selektive Clipping einer einzelnen Geometrie möglich. Komplexere Szenarios lassen sich durch eine Kombination mehrere Clip-Ebenen erreichen.

Viele sinnvolle und wichtige Geometrien lassen sich jedoch nicht durch eine Kombination von Ebenen erzielen. Daher wurde als zweites Konzept (vgl. Abbildung 9.8 rechts) Clipping des Volumens an einem oder mehreren polygonal beschriebenen Objekten realisiert. Dies wird erreicht, indem beim Volume Rendering nicht das gesamte Slice-Polygon gezeichnet wird, sondern, abhängig vom gewählten Clipping-Modus, lediglich die Polygonteile innerhalb beziehungsweise außerhalb des Clip-Objektes. Diese Teile werden prinzipiell als Schnittkonturen zwischen der triangulierten Clip-Geometrie und dem Slice-Polygon konstruiert. Die so berechneten Linienzüge müssen anschließend noch tesseliert werden, um die zu zeichnenden Polygone zu erhalten. Letzteres kann durch die Ausnutzung der Tesselator-Funktionalität, welche die freie OpenGL-Hilfsbibliothek GLUT [GLUT 2004] mitbringt, erreicht werden. Da eine schnelle Berechnung der Schnittpolygone für hohe Interaktionsraten extrem wichtig ist, wird für die Berechnung der Schnittkonturen ein schneller inkrementeller Algorithmus, angelehnt an den Polygon-Clipping-Algorithmus nach Sutherland und Hodgman [Sutherland und Hodgman 1974], verwendet.

Das hier vorgestellte Clipping wird ausschließlich von der CPU durchgeführt, was den Vorteil bietet, dass es orthogonal zu dem in Abschnitt 9.3 vorgestellten Shader-Konzept ist und somit mit jedem der vorgestellten Shader, mit Ausnahme des Prä-Integrations-Shaders, verwendet werden kann. Clipping kann aber ebenfalls unter Ausnutzung der Möglichkeiten moderner Graphik-Hardware erfolgen. Beispiele dafür sind Stencil-Masken, die durch Zeichnen der Clip-Geometrie im Framebuffer erzeugt werden [Westermann und Ertl 1998], texturbasiertes Clipping unter Ausnutzung einer zweiten Textur mit binärer Klassifikation für jedes Voxel [Weiskopf et al. 2003] oder Methoden, die mit einer Tiefenmaske des Clip-Objektes im Framebuffer arbeiten. Die erfolgreiche Kombination von Clipping und Prä-Integriertem Volume Rendering wurde in [Röttger et al. 2003] beschrieben.

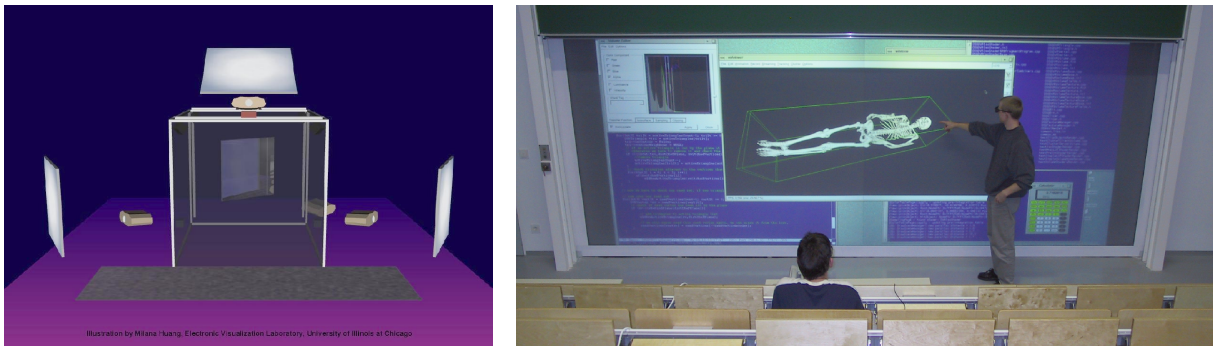


Abbildung 9.9: CAVE (CAVE Automatic Virtual Environment) links [NCSA 2004] und Powerwall rechts stellen nur zwei Beispiele für Virtual-Reality-Installationen dar.

9.7 Virtuelle Realität und Volumenvisualisierung

Im folgenden Abschnitt sollen zwei wesentliche Aspekte für den Einsatz der Volumenvisualisierungsbibliothek im Umfeld von Virtual-Reality-Anwendungen beleuchtet werden. Wie bereits in Abschnitt 9.1 ausgeführt, wurde OpenSG speziell für die Anforderungen von Virtual-Reality-Anwendungen entwickelt. Anzeigesysteme für virtuelle Umgebungen, wie sie beispielsweise in Abbildung 9.9 zu sehen sind, bestehen dabei in der Regel aus mehreren Monitoren oder Projektoren, um die für die Immersion erforderliche weite Sichtfeldabdeckung bei gleichzeitiger stereoskopischer Darstellung zu erreichen [Arthur 2000, Riedel und Deisinger 1995].

Bei einer *CAVE* [Cruz-Neira et al. 1992] wird die Szene z.B. aus vier bis sechs unterschiedlichen Richtungen – entsprechend der Lage der Projektionsflächen zum Betrachter – gezeichnet, und mittels aktiv-stereoskopischer Rückprojektion auf den Seiten eines mehrseitigen Projektionsraumes dargestellt. Eine sogenannte *Powerwall* [Woodward 2004] kombiniert mehrere Projektionsflächen nebeneinander zu einer breitflächigen Darstellung. Früher wurden solche Installationen in der Regel durch große und teure Graphik-Server betrieben, die mit ihrer Rechenkapazität die Szenen gleichzeitig aus mehreren Blickwinkeln zeichnen können und mittels mehrerer Monitorausgänge auf die Ausgabegeräte ausgeben können. Die Single-System-Architektur solcher Lösungen erlaubt dabei eine Realisierung als einzelne Anwendung.

Heute findet zunehmend auch billige PC-Hardware Einsatz, die aufgrund der rapiden Entwicklung vor allem der PC-Graphik-Hardware einen adäquaten Ersatz darstellen. Dabei wird jeder Projektor von einem eigenen PC angesteuert, in dem ein eigenständiger Client als Teil einer verteilten Anwendung abläuft. Diese müssen sowohl zeitlich als auch hinsichtlich der Szenendaten synchronisiert werden, was insbesondere im Zusammenhang mit den in VR-Umgebungen auftretenden dynamischen Szenen einen erheblichen (Programmier-)Aufwand erfordert. OpenSG wurde speziell für die Anwendung in solchen Cluster-Umgebungen entworfen und unterstützt dieses Szenario durch einen einfachen Mechanismus zur Datenverteilung. Basierend auf diesem Mechanismus werden im folgenden Abschnitt verteilte Systeme vorgestellt, welche die Volumenvisualisierung in Cluster-Umgebungen ermöglichen.

Für komplexere Szenen wird darüber hinaus oftmals auch Parallelisierung eingesetzt, um die für virtuelle Umgebungen maximal akzeptierbare System-Latenz von 100 ms [Kowalsky 1993] zu unterschreiten. Mehrere Clients berechnen dabei jeweils Teildarstellungen der fertigen Szene, die im Anschluss daran als Bilder zu den Front-End-Rechnern übertragen werden und dort zum fertigen Ausgabebild kombiniert werden. Um die durch die Bildübertragung verursachte Latenz zu reduzieren, werden im Folgenden Möglichkeiten zur schnellen Bildkompression und -dekompression mittels Graphik-Hardware beschrieben. In Anpassung an die Fähigkeiten unterschiedlicher Graphik-Hardware werden drei verschiedene Realisierungen vorgeschlagen.

9.7.1 Transparente Szenengraph-Replikation

Um die Realisierung einer verteilten Graphikanwendung basierend auf OpenSG besser zu verstehen, lohnt es sich, zunächst einen Blick auf die Datenverwaltung von OpenSG zu werfen. Die Datenverwaltung von OpenSG basiert auf einer Multi-Thread-sicheren Datenarchitektur, bei der jeder Thread den Eindruck hat, auf einer eigenständigen Kopie der Daten zu arbeiten [Roth et al. 2004]. Tatsächlich werden aus Gründen der Datenersparnis jedoch nur die Datenelemente repliziert, die ein Thread tatsächlich verändert. An definierten Synchronisationspunkten, z.B. zu Beginn eines jeden Frames, erfolgt eine Synchronisation der Daten.

OpenSG speichert alle Daten in sogenannten Feldcontainern. Sie gruppieren Datenelemente (Fields), die als Einzelwerte (SingleFields) oder Arrays von Werten (MultiFields) definiert werden können. Zugriff auf die Elemente eines Feldcontainers sind nur über einen abstrakten Zeiger (FCPtr) erlaubt, was die Möglichkeit bietet, durch überladene Zugriffsfunktionen mehrere Kopien (Aspects) der Daten zu verwalten. Dabei bleibt die Replikation für die einzelnen Threads der Anwendung transparent. Schreiboperationen müssen explizit durch Aufruf der Methode `beginEditCP` angekündigt werden, wodurch ggf. eine Kopie des Feldcontainer-Inhalts angestoßen wird. Durch Aufruf von `endEditCP` beendet ein Thread die Schreiboperation. Die Veränderungen an den Datenwerten werden in einer sogenannten ChangeList gespeichert. Sie enthält den veränderten Datenwert zusammen mit einem Zeiger auf das veränderte Feld.

Die explizit aufgezeichneten Veränderungen erlauben eine effiziente Synchronisation der verschiedenen Datenkopien. Gleichzeitig stellen sie die Basis für eine Synchronisation der Szenengraphen verschiedener Anwendungen über eine Netzwerkverbindung dar. Da die ChangeList alle Informationen über die Änderungen im Szenengraph enthält, reicht es, die ChangeList zwischen den verschiedenen Client-Anwendungen auszutauschen. Dazu muss lediglich der Zeiger auf das veränderte Feld durch eine netzwerkeindeutige Kennzahl ersetzt werden. Auf der anderen Seite des Netzwerkes wird die Kennzahl durch den korrespondierenden Zeiger im lokalen Adressraum ersetzt und die veränderten Daten werden in das Feld kopiert. Danach besitzen beide Maschinen identische Szenengraphen, da alle Szenengraphknoten in OpenSG aus Feldcontainern aufgebaut sind. Durch Ausnutzung des Multicast-Protokolls kann die Synchronisation dabei mit einer beliebigen Anzahl von Clients erfolgen, ohne die übertragene Datenmenge zu erhöhen.

Aufgrund dieses für die Anwendung weitgehend transparenten Mechanismus, sind für die Implementierung einer verteilten Volumenvisualisierungsanwendung keine speziellen Maßnahmen erforderlich. Eine konsequente Realisierung der in den Volumenobjekten verwendeten Datenelemente als Felder bzw. Feldcontainer ist dafür ausreichend.

9.7.2 Client-Server-Anwendung

Mit Hilfe der gerade beschriebenen transparenten Datenreplikation ist es sehr einfach möglich, eine Anwendung im Sinne einer verteilten Darstellung (Split Display) zu realisieren, so wie dies bei CAVE- oder Powerwall-Konfigurationen der Fall ist. Die erforderliche Architektur besteht aus einer Server-Anwendung je Rechnerknoten, der einen der Projektoren betreibt, sowie einer Client-Anwendung die auf einem weiteren Knoten läuft. OpenSG verwendet die vom X-Window-System bekannte Nomenklatur. Der Client führt die eigentliche Anwendung aus, d.h. er lädt die Szene, behandelt Benutzereingaben, modifiziert den Szenengraph usw. Die Server stellen lediglich die Szene dar, d.h. sie traversieren den vom Client empfangenen Szenengraph in einem einfachen Graphikkontext. Dabei verwendet jeder Server eine entsprechend der Projektorkonfiguration leicht veränderte Kameraposition.

Für die Realisierung einer solchen Anwendung kann weitgehend auf Funktionalität zurückgegriffen werden, die bereits in OpenSG vorhanden ist. Eine komplette Client-Anwendung ist beispielsweise in Abbildung 9.10 dargestellt. Dabei übernimmt ein spezielles Graphikfenster (`osg::MultiDisplayWindow`) die komplette Netzwerkinteraktion sowie die Einstellung der korrekten Kameraparameter für die Teilanzeigen, die abhängig von der Anzahl der Bildkacheln je Reihe und Spalte sowie der Bildüberlappung in horizontaler beziehungsweise vertikaler Richtung berechnet wird. Letzteres ist für den Einsatz von sogenanntem Edge Blending erforderlich, das bei geteilten Projektionsflächen (Split Display) eingesetzt wird, um harte Kanten zwischen den Teilprojektionen zu vermeiden. Die teilweise überlappenden Projektionen werden dabei optisch jeweils zum Rand hin ausgeblendet, wodurch in der Überlagerung der beiden Projektionen ein glatter Übergang entsteht.

In der Server-Anwendung wird als Gegenstück der `osg::ClusterServer` verwendet, der die über die ChangeList übertragenen Änderungen in der lokalen Kopie des Szenengraphen nachführt und die Darstellung in einem einfachen Fenster übernimmt. Eine geeignete Anwendung findet sich mit der Datei `12ClusterServer.cpp` im Tutorial zu OpenSG, weshalb eine detailliertere Beschreibung hier unterbleibt.

Die im Rahmen dieser Arbeit realisierte Server-Anwendung besteht aus weniger als 150 Zeilen Quellcode inklusive ausführlicher Kommentare und die Client-Anwendung benötigt weniger als 500 Zeilen. Große Teile davon dienen ausschließlich der Benutzerinteraktion. Diese Konfiguration konnte erfolgreich auf Systemen mit vier Knoten (Stereo-Projektion mit zwei Projektionsflächen) bis 48 Knoten getestet werden (siehe Abbildung 9.11). Bei der letzten Konfiguration handelt es sich um die HEyeWall[3] des Fraunhofer Instituts für Graphische Datenverarbeitung, einem Feld aus 6×4 Bildkacheln mit Stereo-Rückprojektion.

[3] http://www.heyewall.de/

```
// all needed include files
#include <OpenSG/OSGGLUT.h>
#include <OpenSG/OSGConfig.h>
#include <OpenSG/OSGSimpleGeometry.h>
#include <OpenSG/OSGGLUTWindow.h>
#include <OpenSG/OSGSimpleSceneManager.h>

#include <OpenSG/OSGMultiDisplayWindow.h>
#include <OpenSG/OSGSceneFileHandler.h>

OSG_USING_NAMESPACE

SimpleSceneManager *mgr;
NodePtr scene;

int main(int argc, char **argv) {
    ChangeList::setReadWriteDefault();

    osgInit(argc,argv);

    int winid = setupGLUT(&argc, argv);

    // create the split window
    MultiDisplayWindowPtr multiWindow =
                          MultiDisplayWindow::create();

    // set some necessary values
    beginEditCP(multiWindow);
        // we connect via multicast
        multiWindow->setConnectionType("Multicast");
        // we want two rendering servers...
        multiWindow->getServers().push_back("Server1");
        multiWindow->getServers().push_back("Server2");
    endEditCP(multiWindow);

    // any scene here
    scene = makeTorus(.5, 2, 16, 16);

    // simple scene manager for user interaction
    mgr = new SimpleSceneManager;

    mgr->setWindow(multiWindow);
    mgr->setRoot  (scene);
    mgr->showAll();

    multiWindow->init();

    glutMainLoop();

    return 0;
}
```

```
void display(void) {
    //redrawing as usual
    mgr->redraw();

    //the changelist should be cleared - else things
    //could be copied multiple times
    OSG::Thread::getCurrentChangeList()->clearAll();

    // to ensure a black navigation window
    glClear(GL_COLOR_BUFFER_BIT);
    glutSwapBuffers();
}


void reshape(int w, int h)
{
    glutPostRedisplay();
}


void mouse(int button, int state, int x, int y)
{
    if (state)
        mgr->mouseButtonRelease(button, x, y);
    else
        mgr->mouseButtonPress(button, x, y);
    glutPostRedisplay();
}


void motion(int x, int y)
{
    mgr->mouseMove(x, y);
    glutPostRedisplay();
}


int setupGLUT(int *argc, char *argv[])
{
    glutInit(argc, argv);
    glutInitDisplayMode(GLUT_RGB |GLUT_DEPTH| GLUT_DOUBLE);

    int winid = glutCreateWindow("OpenSG");

    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutMotionFunc(motion);

    return winid;
}
```

Abbildung 9.10: Client-Anwendung zur Ansteuerung geteilter Anzeigesysteme in OpenSG.

Für das zweite skizzierte Szenario, bei dem mehrere Knoten verteilt an einem einzigen Ausgabebild rechnen, wurde eine klassische Sort-First-Architektur [Molnar et al. 1994] entworfen, bei der jedes einzelne Bild weiter in unabhängige Regionen unterteilt wird. Diese Unterteilung bietet sich für Volumenvisualisierung aufgrund der hohen Rasterisierungsanforderungen an. Um diese Konfiguration zu erreichen, ist lediglich eine minimale Änderung der Architektur erforderlich. Für die Client- und Server-Anwendung spielt es keine Rolle, ob die Server das Bild direkt auf den Ausgang der Graphikkarte geben oder ihre Teilbilder an einen weiteren Front-End-Rechner verschicken, der die eigentliche Anzeige übernimmt. Daher können dieselben oben beschriebenen Programme verwendet werden. In den Einstellungen des `osg::MultiDisplayWindows` muss dazu lediglich die Anzahl n_x der Kacheln je Zeile auf $2n_x$ beziehungsweise $4n_x$ erhöht werden, wenn zwei, beziehungsweise vier Knoten an einem Bild rechnen sollen.

Die Zusammenführung der Teilbilder erfolgt mit einer einfachen GLUT-Anwendung, welche die Bilder von den Server-Knoten entgegennimmt und mittels OpenGL in ein bildschirmfüllendes Graphikfenster zeichnet. Zum Empfang der Bilder horcht das Programm auf

einem UDP-Port. Entsprechend muss die Server-Anwendung angepasst werden, so dass sie die Bilder nach dem Zeichnen aus dem Framebuffer ausliest und danach über eine UDP-Socket-Verbindung an den Front-End-Rechner verschickt. Letzteres kann in OpenSG durch einen `GrabForeground` geschehen, der die Funktionalität zum Auslesen der gezeichneten Szene bereits zur Verfügung stellt. Im Rahmen dieser Arbeit wurde basierend auf dieser Klasse ein `osg::ImageStreamForeground` implementiert, der das ausgelesene Bild via UDP an einen Host verschickt, in diesem Fall den jeweiligen Front-End-Knoten.

9.7.3 Bildkompression auf Graphik-Hardware

Mit der Notwendigkeit zur Bildübertragung in einer parallelen Graphikanwendung ergibt sich ein weiterer potentieller Flaschenhals, der die Performanz des Gesamtsystems beschränkt: die Netzwerkverbindung zwischen den Knoten. Schon bei einer Ausgabegröße von 512×512 Pixeln übersteigt bei 20 Frames pro Sekunde das Datenaufkommen mit 15 MByte/s die Nettobitrate einer 100 MBit Fast-Ethernet-Verbindung. Dabei ist die Datensynchronisation der Knoten noch nicht einmal berücksichtigt. Die einzige Lösung für dieses Problem neben der Verwendung einer teuren Netzwerkinfrastruktur wie Gigabit-Ethernet, Myrinet oder Infiniband liegt in der Verwendung von Bildkompression, was daher von vielen Autoren verfolgt wird.

Ein weiteres Problem stellt die Latenz dar, d.h. die Zeit, die zwischen dem Start der Bildberechnung und der Anzeige des fertigen Bildes auf dem Monitor oder Projektor vergeht. Gerade

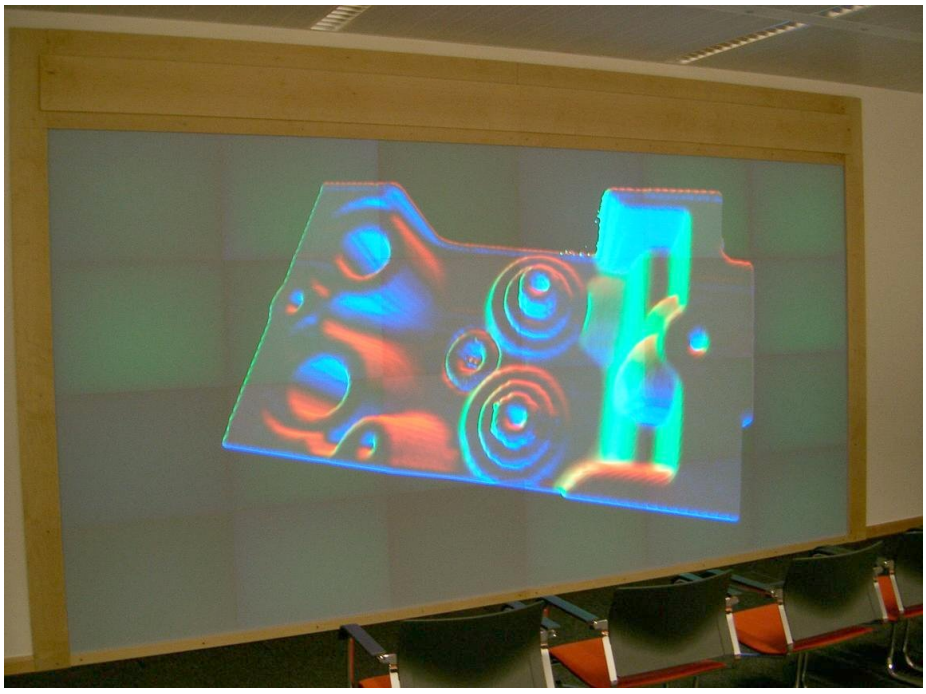


Abbildung 9.11: Verteilte Volumendarstellung mit 24 Cluster-Knoten, von denen jeder eine der 6×4 Kacheln einer geteilten Projektionsfläche berechnet.

für virtuelle Umgebungen muss diese unter 100 ms bleiben, um den Eindruck der Immersion zu bewahren. Diese Latenz wird durch die Zeit, welche die Kompression und Dekompression der Bilder benötigen, sogar noch erhöht. Eine Möglichkeit, wie sowohl Bandbreite als auch Latenz reduziert werden können, ist der Einsatz von Graphik-Hardware zur Bildkompression beziehungsweise -dekompression.

Einerseits sind dadurch geringere Kompressionszeiten zu erwarten, da die reine Rechenleistung moderner Graphik-Hardware aufgrund ihrer intrinsischen Parallelität die moderner Universalprozessoren oftmals übertrifft. Auf der anderen Seite wird durch eine Kompression in Graphik-Hardware die Datenmenge, die in den Hauptspeicher gelesen werden muss, reduziert, was den Einfluss der immer noch sehr teuren Pixelleseoperationen auf die Gesamtperformanz reduziert.

Durch die Dekompression der Bilddaten in der Graphik-Hardware kann auch die Darstellung auf dem Front-End-Rechner von der reduzierten Datenmenge profitieren. Ohne Performanz-Verlust im Vergleich mit dem Zeichnen eines texturierten Polygons lassen sich die komprimierten Bilddaten als Texturen auf die Graphikkarte laden und dort on-the-fly entpacken. Letzteres lindert den Flaschenhals, der bei der Übertragung von Daten zwischen der CPU und der GPU entsteht.

Im Folgenden soll ein solcher Algorithmus zu Bildkompression vorgestellt werden und seine Umsetzung auf die Graphik-Hardware diskutiert werden. Dabei werden unterschiedliche Realisierungen vorgeschlagen, die eine Anpassung an die Fähigkeiten der Graphik-Hardware in den Front-End-Rechnern erlauben. Der Hardware-basierte Algorithmus wurde im Umfeld von Fernvisualisierungssystemen (Remote Visualization Systems) in [Stegmaier et al. 2003] vorgestellt.

Color Cell Compression

Da Graphik-Hardware ihren Geschwindigkeitsvorteil hauptsächlich aus der intrinsischen Parallelverarbeitung zieht, ist bei der Auswahl eines geeigneten Kompressionsalgorithmus auf eine einfache Parallelisierbarkeit zu achten. Ein Algorithmus, der dieses Kriterium erfüllt, ist die von Campbell vorgeschlagene *Color Cell Compression* (CCC) [Campbell et al. 1986]. Dabei wird das Quellbild, das als RGB-Bild mit 3 Byte je Pixel gegeben ist, zunächst in Zellen von jeweils 4×4 Pixel zerlegt, die im Anschluss unabhängig bearbeitet werden. Zunächst wird für jede Zelle die mittlere Luminanz $\mathscr{L}_{mean}$ basierend auf der bekannten Luminanzgleichung $\mathscr{L} = 0,3R + 0,59G + 0,11B$ bestimmt. Danach werden alle Pixel einer Zelle in zwei Gruppen sortiert; Pixel mit einer Luminanz kleiner oder gleich $\mathscr{L}_{mean}$ und Pixel mit einer Luminanz größer als $\mathscr{L}_{mean}$. Eine 16 Bit Bitmaske wird erzeugt, in der Pixel mit einer Luminanz größer als $\mathscr{L}_{mean}$ mit einer 1 gekennzeichnet werden. Darüber hinaus werden je Zelle zwei Farbwerte $C_{\leq}$ und $C_{>}$ gespeichert. Sie repräsentieren die Mittelwerte der Farben beider Pixelgruppen.

Die Rekonstruktion des Bildes aus der Bitmaske und den zwei Farbwerten erfolgt, indem Pixel, deren korrespondierendes Bit in der Maske auf 0 steht, auf den ersten Farbwert gesetzt werden und auf den zweiten Farbwert, wenn das Bit auf 1 steht. Mit diesem Ansatz kann eine Zelle mit $2+3+3=8$ Byte kodiert werden. Darüber hinaus erfolgt eine Quantisierung der Gruppenfarben, bei der nur die 5, 6 und 5 signifikantesten Bits des Rot-, Grün- und Blau-Kanals gespeichert werden. Damit ergibt sich eine Gesamtkompressionsrate von 1 : 8.

Offensichtlich erfüllt dieser Algorithmus die Anforderung, dass alle Zellen unabhängig voneinander verarbeitet werden können, was der Architektur moderner Graphikkarten mit ihren parallelen Pixel- und Vertexeinheiten entgegenkommt. Im Folgenden soll zunächst die Kodierung betrachtet werden. Die vorgestellte Implementierung wurde für eine nVidia GeForceFX Graphikkarte entworfen und später mit leichten Veränderungen auch für die ATI Radeon 9700 adaptiert.

Encoder

Der in Graphik-Hardware implementierte CCC-Algorithmus geht davon aus, dass das Quellbild als Textur auf der Graphikkarte vorliegt. In interaktiven Anwendungen muss diese Textur mit jedem dargestellten Frame aktualisiert werden, was sehr effizient durch direkte Kopie aus dem Framebuffer in den Texturspeicher erfolgen kann, ohne über den Hauptspeicher gehen zu müssen. Die Kodierung erfolgt durch das Zeichnen von Quadrilateralen, deren Kantenlänge ein Viertel der Kantenlänge des Quellbildes betägt. Jedes dabei generierte Fragment entspricht mithin einer Zelle des Quellbildes.

Um die gezeichneten Pixel auf eine Zelle abzubilden wird ein Fragmentprogramm verwendet, welches 16 Texturnachschläge in der Textur des Quellbildes durchführt, um die benötigte Farbinformation zu ermitteln. Aus ihnen berechnet es, wie im vorangegangenen Abschnitt erläutert, die Bitmaske und die beiden Gruppenfarben jeder Zelle. Idealerweise sollten diese mit 16 Bit Genauigkeit in den Rot-, Grün- und Blau-Kanal des Framebuffers geschrieben werden. Leider unterstützt dieser momentan nur 8 Bit Präzision. Darüber hinaus sind mehrere Render Targets auf der GeForceFX nicht verfügbar und die Verwendung Hardware-beschleunigter p-Buffer verbietet sich aufgrund der dadurch erforderlichen extrem teuren Umschaltungen des Graphikkontextes.

Daher wird hier ein Verfahren mit drei Zeichendurchgängen vorgeschlagen, welches im ersten Zeichendurchgang die Bitmaske und im zweiten und dritten Zeichendurchgang jeweils die beiden Gruppenfarben berechnet. Jeder Wert wird dabei mit 16 Bit Genauigkeit berechnet und in zweimal 8 Bit zerlegt, die in den Rot- beziehungsweise Alpha-Kanal geschrieben werden. Nach jedem Durchlauf wird der von dem Quadrilateral überdeckte Bereich aus dem Framebuffer zurückgelesen.

Eine mögliche Optimierung besteht darin, nur einen Zeichendurchgang für die beiden Gruppenfarben zu verwenden und diese im Rot- und Grün-Kanal beziehungsweise im Blau- und Alpha-Kanal desselben Pixels zu speichern. Dies führt jedoch beim Auslesen der Daten zu einem Interleaving der Farbwerte was auf Decoder-Seite nur durch ein Fragmentprogramm entpackt werden kann. Im Vergleich ergibt sich auf einem AMD Athlon XP 2200+ PC bei 512×512 Pixel mit 20,6 ms gegenüber 14,2 ms nur ein leichter Geschwindigkeitsvorteil der Lösung mit zwei Zeichendurchgängen gegenüber der Lösung mit drei Zeichendurchgängen.

Decoder

Während der Encoder in jedem Fall die programmierbare Fragmentverarbeitung moderner Graphikkarten benötigt, kann die Dekodierung im Wesentlichen mit Funktionalität erreicht werden, wie sie in OpenGL 1.0 definiert ist. Die Rekonstruktion läuft dabei in drei Schritten. Zunächst wird die Bitmaske in den Stencil-Buffer geschrieben. Im zweiten Schritt wird ein bildschirmfüllendes Rechteck gezeichnet und mit den Farben $C_>$ der Bildzellen texturiert. Der Stencil-Test wird verwendet, um nur dort Farbwerte zu schreiben, wo in der Bitmaske eine 1 steht. Im dritten Schritt wird der Stencil-Test invertiert und das bildschirmfüllende Rechteck mit den Farbwerten $C_\leq$ gezeichnet. Sie werden nur dort geschrieben, wo die Bitmaske den Wert 0 enthält. Diese Lösung benötigt $5,4$ ms bei einem 512×512 Bild.

Die Rekonstruktion kann in einem einzigen Schritt erfolgen, wenn der Front-End-Rechner Multi-Texturen und Register Combiners anbietet, so wie sie mit dem GeForce256 Chip von nVidia eingeführt wurden. In diesem Fall können die Bitmaske und die beiden Gruppenfarben in jeweils einer eigenen Textur gespeichert werden und stehen in der Fragmenteinheit damit gleichzeitig zur Verfügung. Eine geeignete Combiner-Konfiguration wählt nun dynamisch für jedes Pixel entweder $C_\leq$ oder $C_>$ abhängig vom Inhalt der Bitmaskentextur. Erstaunlicherweise benötigt diese Lösung mit $5,3$ ms kaum weniger Zeit, was den Schluss nahe legt, dass die Definition der Texturen mehr Zeit als das eigentliche Zeichnen benötigt.

Die Texturen werden aus dem vom Encoder gesendeten Datenstrom erzeugt, welche dieser in drei Schritten – Bitmaske, $C_>$ und $C_\leq$ – aus dem Framebuffer liest und hintereinander in den Datenstrom schreibt. Dies stellt eine Variation des ursprünglichen CCC-Algorithmus dar, bei dem Bitmaske und die beiden Farbwerte je Zelle interleaved gespeichert werden. Da dies bei dem hier vorgestellten Encoder nicht der Fall ist, gelingt die Definition der Farbtexturen direkt aus dem empfangenen Datenblock durch Vorgabe des korrekten Offsets. Auch die Quantisierung der Farbe stellt kein Problem dar, weil OpenGL das $5/6/5$-Farbformat nativ unterstützt.

Leider erfordert die Präparierung der Bitmaske mehr Aufwand. Bitfelder sind kein gültiges Datenformat, um in OpenGL eine Textur zu definieren. Daher muss die Bitmaske zunächst in einen Byte-Strom konvertiert werden, bei dem jedes Bit in der Maske als ein Byte repräsentiert wird. Unter Verwendung eines Fragmentprogramms zur Rekonstruktion ist es ebenfalls möglich, die empfangene Bitmaske direkt in eine Textur zu schreiben und die Rekonstruktion für jedes Pixel on-the-fly aus der Bitkodierung zu extrahieren. Dies erfordert aber Graphik-Hardware mit Unterstützung für frei programmierbare Fragmenteinheiten. Die realisierte Fragmentprogramm-Implementierung benötigt $6,8$ ms, was etwas langsamer ist als die oben beschriebene Darstellung mit einer von der CPU entpackten Bitmaske. Auf Systemen mit langsameren Prozessoren konnte jedoch eine Beschleunigung von $20,3$ ms für die Implementierung mit Register Combiners auf $8,9$ ms mittels Fragmentprogrammen erzielt werden.

Kapitel 10

Zusammenfassung und Ausblick

Durch die treibende Kraft vor allem der Spieleindustrie besitzen moderne PCs neben dem Hauptprozessor (CPU) weitere leistungsstarke Recheneinheiten auf der Graphik-Hardware (GPU), deren Ausnutzung sich in den letzten Jahren als wichtiges Instrument für interaktive Visualisierungsalgorithmen herauskristallisiert hat. In Zukunft wird die Bedeutung der Graphik-Hardware vermutlich noch zunehmen, da sich die Leistungsschere zwischen CPU und GPU weiter öffnet.

Vor der Einführung programmierbarer Einheiten auf der Graphik-Hardware war deren Einsetzbarkeit für Visualisierungszwecke stark eingeschränkt. Effiziente Hardware-Umsetzungen entstanden dort, wo sich die Struktur der Daten besonders für die Mechanismen, die bereits auf der Graphik-Hardware vorhanden sind, eignen, wie dies z.B. im Falle der Volumenvisualisierung auf uniformen 3D-Gittern zutrifft, die mit Hilfe von 3D-Texturen implementiert werden kann. Die Hardware-beschleunigte Zellprojektion nach Shirly und Tuchman basiert auf der Ausnutzung der effizienten Dreieck-Rasterisierung durch die Graphik-Hardware.

Die Programmierbarkeit moderner Graphik-Hardware erlaubt seit kurzem eine sehr viel breitere Ausnutzung der GPU zum Zwecke der Beschleunigung von Visualisierungsverfahren. Dies bezieht sich sowohl auf bisher nicht umsetzbare Visualisierungsalgorithmen als auch auf die Verwendung adaptiver und damit anspruchsvollerer Datenstrukturen. Beides wurde in der vorliegenden Arbeit demonstriert.

Bei der Bilderzeugung sind auf modernen PC-Systemen unterschiedliche Prozessoren beteiligt. Die CPU erzeugt Geometrie, die anschließend von der Vertexeinheit transformiert und von der Rasterisierungseinheit in Fragmente zerlegt wird. Die Berechnung der Fragmentfarbe erfolgt durch die Fragmenteinheit. Da in diesem Zusammenspiel eine einzige überlastete Einheit die Leistung des Gesamtsystems beschränkt, muss für eine möglichst hohe Performanz eine geeignete Balance zwischen den unterschiedlichen Einheiten gefunden werden. Dies muss unter der Einbeziehung der maximalen Übertragungsraten und der Latenz für die Datenkommunikation zwischen den Einheiten erfolgen. Für die Balancierung wurden in dieser Arbeit unterschiedliche Strategien aus dem Umfeld der Hardware-Software-Synthese adaptiert. Diese sollen zunächst allgemein betrachtet werden, bevor auf die spezifische Umsetzung für Tetraedergitter als exemplarische adaptive Datenstruktur eingegangen wird.

10.1 Hardware-Software-Synthese

Allgemein ist die Suche nach geeigneten Schnittstellen für eine kombinierte Hardware-Software-Umsetzung eines gegebenen Algorithmus eines der fundamentalen Probleme der Informationstechnik, welches insbesondere unter dem Schlagwort Hardware-Software-Co-Design im Umfeld der Eingebetteten Systeme (Embedded Systems) in den letzten Jahren große Aufmerksamkeit genossen hat. Lösungen in diesem Umfeld setzen auf formale Methoden, welche in einem mehrstufigen Verfeinerungsprozess eine Spezifikation oder einen Algorithmus sukzessive in kleinere Funktionsblöcke zerlegen, die dann auf einen Universalprozessor bzw. unterstützende Co-Prozessoren, Signalprozessoren oder FPGAs verteilt werden. Oft kommt für diese sogenannte Partitionierung ein Optimierungsprozess basierend auf gewissen Metriken wie Latenz oder Datendurchsatz aber auch Chipfläche oder Produktionskosten zum Einsatz.

Für die Realisierung von balancierten Hardware-Software-Umsetzungen im Zusammenhang mit Graphik-Hardware sind diese formalen Methoden jedoch nur beschränkt einsetzbar. Das Problem liegt in der fehlenden Spezifikation der Komponenten der Graphik-Hardware. Oftmals sind nur grundlegende Informationen über die Anzahl der parallelen Vertex- oder Fragmenteinheiten oder die Breite der Speicheranbindung bekannt. Über die detaillierte Architektur, wie z.B. die genaue Anzahl der parallel ausführbaren Vertex- und Fragmentoperationen oder Texturzugriffe hüllen sich die Hardware-Hersteller verständlicherweise in Schweigen, was ein systematisches Vorgehen erschwert. Daher wurde in dieser Arbeit auch nicht versucht, ein formales Modell aktueller Graphik-Hardware zu erstellen, was zweifellos den Rahmen gesprengt hätte. Stattdessen soll eine Reihe von Heuristiken vorgestellt werden, die sich als erfolgreich erwiesen haben und somit als Leitlinie für zukünftige Hardware-Umsetzungen dienen können.

10.1.1 Algorithmische Transformation

Am Anfang einer jeden Umsetzung sollten ganz allgemeine Prinzipien der Software-Optimierung stehen, wie sie beispielsweise in [Bentley 1982] beschrieben werden. Vor jeder Überlegung hinsichtlich einer Parallelisierung oder möglichen Hardware-Unterstützung haben sie das Potential einen Algorithmus um mehrere Größenordnungen zu beschleunigen. Optimierungsanstrengungen sind dabei am sinnvollsten an den Code-Teilen, die am häufigsten aufgerufen werden oder welche die meiste Zeit beanspruchen.

Die in dieser Arbeit eingesetzten algorithmische Optimierungen basieren in der Regel auf der *Space-For-Time Rule*, der Idee, Rechenzeit durch vorberechnete Ergebnisse zu sparen. Der Einsatz von Prä-Integration zur Auswertung des Volume Rendering Integrals gehört zu dieser Kategorie ebenso wie die inkrementelle Berechnung der Prä-Integrationstabelle, die zusätzlich auf der Idee der Eliminierung gemeinsamer Teilausdrücke (*Common Subexpression Elimination*) basiert. Inkrementelle Berechnungen sind außerdem die Basis für schnelles CPU-basiertes Resampling oder die effiziente Berechnung der Strahlaustrittspunkte beim Hardware-basierten Ray Casting. Als Randbemerkung sei darauf hingewiesen, dass auch die schnelle Rasterisierung von Dreiecken in der Graphik-Hardware auf der Ausnutzung inkrementeller Prinzipien beruht.

Auf der Ebene der Fragmentprogramme wird darüber hinaus die Erweiterung von Datenstrukturen (*Data Structure Augmentation*) eingesetzt. Größen wie Normale oder Gradient, die sich aufgrund von Limitierungen der Graphik-Hardware nicht berechnen lassen, werden bei der Hardware-beschleunigten Zellprojektion als zusätzliche Parameter übergeben beziehungsweise zur Beschleunigung beim Hardware-basierten Ray Casting als zusätzliche Texturen definiert.

Eine weitere Möglichkeit besteht in der Vereinfachung des Problems. Bei der Konvertierung von unstrukturierten Tetraedergittern in eine funktionale RBF-Repräsentation wird beispielsweise die aufwändige Traversierung des Tetraedergitters vermieden und durch eine relativ einfache Auswertung der Volumendaten pro Fragment ersetzt. Der Konvertierungsprozess kann dabei als Vorverarbeitungsschritt vernachlässigt werden. Alternativ erfolgt durch das Resampling eine Konvertierung in ein uniformes dreidimensionales Gitter, eine Datenstruktur, die sich besonders zur Verarbeitung durch die Graphik-Hardware eignet und die Darstellung deutlich beschleunigt. Die adaptive Struktur des Tetraedergitters bleibt dabei unter Verwendung von Level-of-Detail-Volumenvisualisierung erhalten.

10.1.2 Beseitigung von Kommunikationsflaschenhälsen

Graphik-Hardware erreicht ihre hohe nominelle Rechengeschwindigkeit nur unter der Voraussetzung einer kontinuierlichen Datenversorgung, was aufgrund der limitierten Architektur des AGP Busses und die kontinuierlich steigende Rasterisierungsleistung der Graphik-Hardware zunehmend schwierig wird. Die effiziente Datenübertragung zur Graphik-Hardware wurde schon von vielen Arbeiten im Kontext von Visualisierung und Rendering thematisiert. Einige Beispiele finden sich in [Chidlow und Möller 2003, Schneider und Westermann 2003, Jang et al. 2004, Rossignac 1997]. Zwar sind mit neu aufkommenden Busarchitekturen wie PCI Express Verbesserungen zu erwarten, als Grundproblem bleibt jedoch, dass viele Algorithmen aufgrund der Datenübertragung in kleinen Portionen nicht die volle Bandbreite der Busse nutzen können, die nur im Burst-Modus erreicht wird. In dieser Arbeit wurden daher grundsätzlichere Möglichkeiten untersucht, das Problem zu umgehen.

Soweit möglich erfolgt die Datenhaltung im lokalen Speicher der Graphikkarte. Dies kann in Form von Texturen geschehen, die im Fall des Hardware-basierten Ray Casting Tetraederlisten und Tetraederstreifen speichern oder die kompakten Parameter der Basisfunktionen im Fall der RBF-basierten Visualisierung. Auch die Vertexlisten, die bei der Hardware-basierten Zellprojektion verwendet werden, können auf der Graphikkarte lagern, sofern sie eine gewisse Größe nicht überschreiten. Somit wird eine Datenübertragung von der CPU zur GPU jenseits der Initialisierung komplett vermieden. Diese Ansätze bieten daher ein hohes Potential für die Zukunft, weil sie in vollem Maße an zukünftigen Beschleunigungen der Graphik-Hardware partizipieren.

Nicht unter allen Umständen kann eine Speicherung der Daten auf der Graphikkarte erreicht werden. Eine zweite Möglichkeit besteht darin, die Übertragung zu optimieren, indem insbesondere der Burst-Modus des Graphikbusses ausgenutzt wird. Bei der Hardware-basierten Zellprojektion wird dies durch die Definition einer blickpunktunabhängigen und damit statischen Datenstruktur erreicht, die anstelle der dynamisch durch Zellprojektion berechneten Dreiecke tritt.

Schließlich kann der Flaschenhals der Datenübertragung durch die Verwendung komprimierter Datenstrukturen beseitigt oder zumindest gelindert werden. Beispiele für diese Strategie finden sich in der Kompression von Tetraedergittern als Tetraederstreifen, der Transformation des Tetraedergitters in eine kompakte funktionale RBF-Repäsentation sowie der komprimierten Bildübertragung mittels CCC-Kompression. Neben der Beschleunigung der Datenübertragung erreicht die Kompression auch, dass größere Datensätze lokal auf der Graphikkarte gespeichert werden können. Im Falle der Tetraedergitter beträgt der Kapazitätsgewinn eine Größenordnung.

10.1.3 Parallelisierung

Die im Vergleich mit Universalprozessoren hohe Rechenleistung moderner Graphikprozessoren beruht neben hoher Taktung und optimierten Speicher-Schnittstellen im Wesentlichen auf Parallelverarbeitung. Moderne Graphikprozessoren setzen auf Vertex- und auf Fragmentebene mehrere Verarbeitungseinheiten ein. Die Anzahl dieser Einheiten wurde in den letzten Jahren sukzessive gesteigert. Aktueller Stand der Technik sind sechs Vertexeinheiten und 16 Fragmenteinheiten (nVidia GeForce 6800 Ultra und ATI X800). Für die Zukunft ist zu erwarten, dass der Grad der Parallelität aufgrund der einfachen Skalierbarkeit der parallelen Graphikeinheiten weiter steigt. Im Vergleich mit Universalprozessoren ist die Funktionalität der Graphikeinheiten überschaubar und die vorherrschende datengetriebene Streaming-Architektur vermeidet Abhängigkeiten zwischen den parallelen Vertex- und Fragmenteinheiten. Parallele Verarbeitung findet darüber hinaus auch auf Instruktionsebene statt. Die SIMD-Befehle der programmierbaren Vertex- und Fragmenteinheiten arbeiten in der Regel parallel auf vier-komponentigen Vektoren und können daher im Idealfall vier arithmetische Berechnungen pro Taktzyklus durchführen.

In dieser Arbeit wurde Parallelität auf allen drei Ebenen zur Beschleunigung der betrachteten Visualisierungsverfahren ausgenutzt. Im Kontext von Hardware-basierter Zellprojektion erfolgt eine parallele Berechnung der Strahlschnittpunkte mit den Austrittsflächen eines Tetraeders in der Vertexeinheit. Beim Hardware-basierten Ray Casting wurde ein bewusst einfach parallelisierbarer Ansatz gewählt, um die für jedes Pixel erforderlichen Berechnungen parallel durch die programmierbare Fragmenteinheit erledigen zu lassen. Auch die CCC-Bildkompression arbeitet unabhängig und parallel auf den Makroblöcken in der Fragmenteinheit. Durch diese Ausnutzung der intrinsischen Parallelität der Graphik-Hardware konnte in dieser Arbeit eine deutliche Beschleunigung im Vergleich mit vorherigen Ansätzen erzielt werden. Im Falle von Hardware-basiertem Ray Casting wird ein Beschleunigungsfaktor von 4 erreicht.

Vorteil der Ausnutzung paralleler Vertex- und Fragmenteinheiten ist, dass neben dem Design der Berechnungs-Kernel für Vertizes und Fragmente kein weiteres Zutun erforderlich ist, weil die Verteilung der Daten auf die parallelen Einheiten automatisch durch das interne Scheduling der Graphikkarte erfolgt. Anders verhält es sich auf der Ebene der Instruktionsparallelisierung. Wie im Kontext von CCC-Kompression gezeigt wurde, ist hier eine sorgfältige Implementierung und Hand-Optimierung der Fragmentprogramme erforderlich, da die Erzeugung optimal parallelisierter Shader von den untersuchten Shader-Compilern noch nicht zufriedenstellend gelöst wird. Auch hat sich gezeigt, dass oftmals der Effekt der Optimierungen hinter den theoretischen Erwartungen zurückbleibt.

10.1.4 Hardware-Selektion

Die Frage, welcher Hardware-Baustein zur Realisierung einer projektierten Funktionalität in einer kombinierten Hardware-Software-Realisierung eingesetzt werden soll, wird im Umfeld von Eingebetteten Systemen in der Regel getrieben von externen Randbedingungen wie Taktrate, Latenz, Kommunikationsschnittstellen, aber auch Stromverbrauch, Chipfläche und Kosten. Im Zusammenhang mit dem Einsatz von Graphik-Hardware zum Zwecke der Visualisierung hat man es einerseits oftmals mit einer heterogenen Umgebung zu tun, da mehrere Kartenhersteller mit unterschiedlichen Produkten am Markt sind, und andererseits unterschiedliche Graphikkartengenerationen parallel im Einsatz sind. Darüber hinaus können auf ein und demselben Baustein unterschiedliche Mechanismen zur Verfügung stehen, um eine bestimmte Funktionalität zu realisieren.

Um dieses Problem anzugehen, wurden in dieser Arbeit am Beispiel der texturbasierten Volumenvisualisierung Möglichkeiten der Abstraktion von den Fähigkeiten der Graphik-Hardware demonstriert. Dabei wurde unter anderem deutlich, dass trotz anderslautenden Erklärungen der Graphikkarten-Hersteller die Implementierung der Fixed-Function-Pipeline unabhängig von den programmierbaren Einheiten realisiert ist und eine höhere Performanz erreicht. Im Zweifelsfall ist daher eine Implementierung mit Hilfe der Standardpipeline, auch unter Ausnutzung von OpenGL-Erweiterungen wie `NV_texture_shader` oder `NV_register_combiners` zur flexiblen Parametrisierung, der programmierbaren Pipeline vorzuziehen. Nicht auszuschließen ist jedoch, dass in Zukunft die Graphikkartenhersteller, um Chipfläche zu sparen, zunehmend die Hardware-Funktionsblöcke der parametrisierten Pipeline einsparen und durch kleine Vertexprogramme und Fragmentprogramme ersetzen, die automatisch vom Graphikkartentreiber geladen werden und von den programmierbaren Einheiten ausgeführt werden. Der Geschwindigkeitsvorteil der parametrisierbaren Einheiten geht damit in Zukunft vermutlich verloren.

10.1.5 Hardware-Software-Co-Design

Das kooperative Design eines Algorithmus basierend auf Hardware- und Software-Komponenten wurde im Speziellen für das Problem des Hardware-beschleunigten Resampling untersucht. Dabei hat sich gezeigt, dass schon für mittelgroße Zielgitter selbst optimierte Software-Lösungen zum Resampling nicht mit der schnellen trilinearen Interpolation auf der Graphikkarte mithalten können. Ein Transfer der Berechnungen auf die Graphikkarte um jeden Preis ist jedoch eher kontraproduktiv, insbesondere wenn er mit einer höheren Rasterisierungsfläche einhergeht. Die höchste Performanz wird wiederum unter Ausnutzung der Standardpipeline der Graphhik-Hardware mittels eines Marching-Cubes-ähnlichen Ansatzes erreicht. Die CPU übernimmt dabei die Aufgabe der Datenaufbereitung.

Resampling direkt auf der Graphikkarte hat darüber hinaus den Vorzug, dass die Daten bereits im lokalen Graphikspeicher vorliegen, was zu einer deutlichen Beschleunigung führt, da eine Übertragung der Resampling-Gitter entfällt. Neueste Graphik-Hardware beseitigt sogar die Notwendigkeit des Texturladens, indem Teile des Texturspeichers direkt als Ziel für das Zeichnen definiert werden können.

10.2 Partitionierungsstrategien für Tetraedergitter

Performanz kann nicht immer das ausschlaggebende Kriterium für die Auswahl einer geeigneten Verteilung des Darstellungsalgorithmus auf Hardware und Software sein. Auch Limitierungen und Einschränkungen müssen berücksichtigt werden. Daher sollen in diesem Abschnitt die Vorteile und Beschränkungen der in dieser Arbeit behandelten Methoden überblicksartig dargestellt werden. Dies erfolgt anhand der Betrachtung der Visualisierung von Volumendaten auf Tetraedergittern, die den Hauptanteil dieser Arbeit ausmacht. Die Darstellung erfolgt systematisch von CPU-zentrierten Ansätzen zu GPU-zentrierten Ansätzen mit Schwerpunkt auf der Fragmentverarbeitung.

10.2.1 CPU-zentrierte Partitionierung

Trotz der in dieser Arbeit erzielten Verbesserungen erreichen Visualisierungsalgorithmen für uniforme Gitter basierend auf 3D-Texturen deutlich höhere Bildwiederholraten als alle bekannten Ansätze, die auf Tetraedergittern arbeiten. Das Resampling eines Tetraedergitters auf ein uniformes Gitter ist somit nach wie vor sinnvoll. Es verschiebt die Rechenlast von der GPU zur CPU. Für zeitlich nicht veränderliche Daten kann das Resampling dabei in einem Vorverarbeitungsschritt erfolgen.

Nachteile sind der in der Regel durch das Resampling auftretende Informationsverlust sowie Artefakte durch die Diskretisierung und die darauf aufbauende trilineare Interpolation, die nicht mit der ursprünglichen Repräsentation konsistent ist. Limitierungen dieses Ansatzes sind durch den auf der Graphikkarte vorhandenen Speicherplatz gegeben, der für große Datensätze in der Praxis oft keine ausreichend genaue Abtastung erlaubt. Letzteres kann durch Resampling mit einer lokal verfeinerten Hierarchie von uniformen Gittern und zugehöriger adaptiver Darstellung beseitigt werden. Ein effizientes Resampling ist insbesondere für zeitlich veränderliche Daten wichtig, speziell wenn eine zeitlich veränderte Topologie des Gitters den Einsatz statischer Beschleunigungsdatenstrukturen unmöglich macht.

Alternativ zu uniformen Gittern kann in einem Vorverarbeitungsschritt eine funktionale Repräsentation der Daten gefunden werden. Die in Kapitel 6 vorgestellten radialen Basisfunktionen sollen nur ein Beispiel sein. Gegenüber uniformen Gittern ist eine RBF-Repräsentation adaptiv und erlaubt eine analytische Auswertung sowohl des Skalarwertes als auch des Skalarwert-Gradienten ohne Diskretisierungsprobleme. Die Darstellung basierend auf radialen Basisfunktionen ist aufwändiger und mit ausreichender Interaktivität nur bei begrenzter Anzahl von Basisfunktionen möglich. Die Anwendung für Tetraedergitter ist daher auf Datensätze mit geringer Informationsdichte beschränkt.

10.2.2 Vertexzentrierte Partitionierung

Zellprojektion basiert wesentlich auf dem Zeichnen von Dreiecken und ist damit vertexzentriert, obwohl, wie bereits thematisiert, die CPU einen erheblichen Einfluss auf die Performanz von

Zellprojektionsalgorithmen hat. Grund dafür ist die erforderliche Sortierung der Tetraeder, die insbesondere im Falle von nicht-konvexen Gittern aufwändig ist und zu Problemen bei der Behandlung von Sichtbarkeitszyklen führt. Daher sind Standard-Zellprojektionsalgorithmen auch nicht in der Lage, die Graphik-Hardware voll auszulasten.

Um dies zu verbessern wurde in dieser Arbeit die blickpunktunabhängige Zellprojektion entwickelt. Die Idee besteht in einer blickpunktunabhängigen Datenstruktur, die unter Ausnutzung von Optimierungsmechanismen effizient auf die Graphikkarte übertragen werden kann oder lokal auf der Graphikkarte vorliegt. Dies führt jedoch bei Implementierung der Projektion in der Vertexeinheit zu einer signifikant höheren Datenmenge und der Einschränkung, dass nur orthographische Projektionen erlaubt sind. Die Lösung besteht in der Berechnung der Projektion in der Fragmenteinheit, was sowohl die Menge an Hilfsdaten reduziert als auch perspektivische Projektion zulässt. Die Umsetzung erfordert allerdings bereits frei programmierbare Fragmenteinheiten, wie sie erst seit der vorletzten Graphikkartengeneration verfügbar sind. Die Verwendung der blickpunktunabhängigen Datenstruktur verhindert darüber hinaus die Sortierung der Tetraeder, was die Wahl der optischen Modelle einschränkt.

Kraus et al. [Kraus et al. 2004] haben gezeigt, dass zumindest ohne die Verwendung der blickpunktunabhängigen Datenstruktur unter Ausnutzung von Prä-Integration und mindestens 16 Bitgenauem Hardware-Blending die Darstellung eines Tetraedergitters mittels Zellprojektion ohne die sonst üblichen Artefakte möglich ist.

10.2.3 Fragmentzentrierte Partitionierung

Eine Lösung zur Visualisierung von Tetraedergittern, die fast ausschließlich auf Fragmentebene in der Graphik-Hardware arbeitet, wurde mit dem Hardware-basierten Ray Casting vorgestellt. Sie verwendet ebenfalls die Idee der Prä-Integration, berechnet das Volume Rendering Integral mit mindestens 16 Bit Gleitkommagenauigkeit und leidet daher nicht unter Artefakten. Zyklische Strukturen innerhalb eines Tetraedergitters stellen kein Problem dar, weil die Sortierung auf Pixelebene erfolgt und auch nicht-konvexe Gitter lassen sich durch Konvexifizierung oder die Behandlung von Wiedereintritten mittels Depth Peeling korrekt handhaben. Die wesentliche Beschränkung des Ansatzes liegt in der Größe der unterstützten Datensätze, die durch den verfügbaren Texturspeicher limitiert ist.

Eine weitere Schwäche des Ray Casting Ansatzes liegt in der hohen Berechungskomplexität je Pixel. Diese führt insbesondere bei großen Ausgabebildern zu einer schlechteren Performanz verglichen mit den vorher beschriebenen Methoden. Vorteilhaft erweist sich beim Hardware-basierten Ray Casting jedoch seine Ausgabe-Sensitivität. Der Rechenaufwand ist direkt proportional zur Anzahl der Ausgabe-Pixel. Da dies bei den vorher beschriebenen Methoden nicht oder nur eingeschränkt der Fall ist, verschwindet der Performanznachteil bei kleinen Ausgabebildern.

10.3 Ausblick

Die Entwicklung der Graphik-Hardware schreitet rasant voran und bringt stetig neue Funktionalität, von der nicht nur die vorgestellten Algorithmen sondern auch zukünftige Visualisierungsverfahren profitieren können. So könnte die vorgeschlagene Hardware-basierte Zellprojektion von der bereits angekündigten Möglichkeit von Texturnachschlägen auf der Vertexebene profitieren. Die Speicherung des kompletten Tetraedergitters in Vertextexturen macht die Datenübertragung zwischen CPU und Graphikkarte bis auf wenige Indexdaten obsolet, so dass in Kombination mit einem schnellen Sortierverfahren eine interessante Alternative zum Hardware-basierten Ray Casting, insbesondere für hohe Bildschirmauflösungen, zu erwarten ist. Darüber hinaus erlaubt die bereits jetzt sehr flexible Rasterisierungspipeline eine Reihe interessanter Erweiterungen des Algorithmus, wie beispielsweise Clipping in Tetraedergittern.

Auf der Fragmentebene bringt die jüngste Graphikkartengeneration von nVidia dynamische Schleifen und Verzweigungen in Fragmentprogrammen mit sich, die zusammen mit der Möglichkeit von linearer Interpolation und Blending, eingeschränkt auf 16 Bit Gleitkommadaten, interessante Optimierungen des Hardware-basierten Ray Casting Algorithmus erwarten lassen. Die Implementierung der kompletten Strahltraversierung als einzelne Schleife im Fragmentprogramm würde die Notwendigkeit einer Traversierungsdatenstruktur beseitigen und damit einhergehend die meisten der teuren Lese- und Schreiboperationen in den Framebuffer.

Im Hinblick auf die Kompression der Tetraedergitter könnte für die Streifenerzeugung ein Algorithmus mit höherer Lokalität die Fragmentierung des Datensatzes vermeiden und damit zu einer deutlich effizienteren Speicherung kommen. Dies könnte durch Verfolgen der Streifen entlang bereits existierender Streifen erfolgen. Aus Gründen der besseren Cache-Kohärenz sind weiterhin Speicherschemata interessant, die lokale Zugriffsmuster selbst für unterschiedliche Blickrichtungen erzeugen. In Ergänzung zu den universellen Algorithmen zur Streifenerzeugung sind statische Muster zur Streifenerzeugung interessant, die für spezielle Gittertypen wie kurvilineare oder reguläre Gitter eine signifikant bessere Komprimierung erlauben.

Die Verarbeitung zeitlich veränderlicher Daten stellt eine Herausforderung für alle Techniken dar, welche den Datensatz komplett auf der Graphikkarte abspeichern. Ihr könnte durch die Ausnutzung zeitlicher Kohärenz in der Kodierung begegnet werden. Alternativ sind effiziente Methoden untersuchenswert, welche die Darstellung des aktuellen Zeitschrittes bei gleichzeitigem Laden oder versetztem Laden des neuen Zeitschrittes erlauben.

Die Parallelisierung auf der Ebene der CPU ist ein weiteres Thema, das in dieser Arbeit nicht berücksichtigt werden konnte. SIMD-Befehle stehen mit Intels MMX- beziehungsweise SSE-Befehlssatzerweiterungen seit dem Pentium Chip zur Verfügung und können ebenso wie die entsprechenden Befehle auf der GPU mehrere Instruktionen parallel ausführen. Darüber hinaus geht die Entwicklung zu mehreren logischen oder physikalischen Rechenkernen in einem Prozessorgehäuse, so dass in naher Zukunft jeder Standard-PC in der Lage sein wird, mehrere Berechnungsstränge parallel auszuführen. Gerade für rechenaufwändige Techniken wie das in dieser Arbeit vorgestellte Hardware-basierte Resampling könnte die Ausnutzung dieser Techniken die optimale Balance zwischen GPU und CPU in Richtung Hauptprozessor verschieben.

Dies verdient eine weitere Betrachtung, die insbesondere im Kontext unterschiedlicher Prozessortypen erfolgen muss, da diese sich deutlich in ihren Hardware-Charakteristika wie Integer- und Gleitkommarechenleistung, Cache-Größe und Cache-Strategie unterscheiden.

Ein weiteres Problem, das behandelt werden sollte, ist die Genauigkeit für das Hardware-beschleunigte Resampling. Die ursprüngliche Implementierung nutzte lediglich 8 Bit, was mit der gewählten Visualisierungstechnik konsistent ist. Moderne Graphikkarten erlauben die interne Berechnung mit 24 bis 32 Bit Gleitkommagenauigkeit. Die Reduzierung auf Fragmentoperationen, die mit 16 Bit Genauigkeit arbeiten, erlaubt jedoch immer noch auf vielen Hardware-Architekturen eine deutlich schnellere Berechnung. Dies gilt in analoger Weise für eine Vielzahl anderer Visualisierungsalgorithmen.

Für die Level-of-Detail-Volumendarstellung ist die Integration und Evaluation verschiedener Fehlertechniken interessant, die eine automatische Selektion eines geeigneten Level-of-Detail erlauben. Diese würde zu einer Topologie-erhaltenden Vergröberung der Darstellung beitragen. Ein offenes Forschungsfeld ist ebenfalls, in wieweit die visuellen Artefakte, die bei dem Verfahren auftreten, durch die höhere Textur- und Framebuffer-Genauigkeit moderner Graphikkarten beseitigt werden können.

Zu einer weiteren Verbreitung des Open-Source-Volumenknotens für OpenSG könnte die Integration zusätzlicher anspruchsvoller Darstellungsalgorithmen, wie direktional oder mittels Environment Map beleuchtete Volumen, als Volumen-Shader beitragen. Shader für höherdimensionale Transferfunktionen sind vor allem für medizinische Anwendungen relevant. Im Sinne einer besseren Zukunftsperspektive ist die Vereinheitlichung der Shader-Implementierungen mittels Shader-Hochsprachen interessant.

Natürlich konnte im Rahmen dieser Arbeit nur ein Ausschnitt aus der Vielzahl von adaptiven Datenstrukturen behandelt werden. Die vorgestellten Techniken und Prinzipien können jedoch als allgemeine Gestaltungskriterien für die Implementierung weiterer Visualisierungsalgorithmen auf Graphik-Hardware dienen. Gerade durch die Kombination beschleunigender adaptiver Datenstrukturen und der enormen Rechenleistung der Graphik-Hardware ergibt sich ein großes Potential für die Realisierung verbesserter Graphik- und Visualsierungsverfahren. So besteht die begründete Hoffnung, dass diese Arbeit noch viele Forscher zu weiteren Hardware-beschleungiten Algorithmen inspiriert.

Anhang

Abbildungsverzeichnis

2.1 Die Visualisierungspipeline. 22
2.2 Klassifikation unterschiedlicher Gittertypen am zweidimensionalen Beispiel. . . 23
2.3 Klassifikation unterschiedlicher Multiblock-Gitter. 25
2.4 Standard-Rendering-Pipeline. 28
2.5 Geometrieverarbeitung als Teil der Standard-Rendering-Pipeline. 29
2.6 Rasterisierung als Teil der Standard-Rendering-Pipeline. 30
2.7 Fragmentoperationen als Teil der Standard-Rendering-Pipeline. 31
2.8 Vertex- und Fragmentprogramme als Ersatz für die statische Transformations- und Beleuchtungsberechnung beziehungsweise die Berechnung der Fragmentfarbe. 34

3.1 Gegenüberstellung der stückweise konstanten und stückweise linearen Approximation der Skalarfunktion entlang des Sehstrahls. 42
3.2 Vergleich von Volumendarstellungen mit Post-Klassifikation und Prä-Integration. 44
3.3 Unterteilung eines Strahlsegments nach der Länge. 46
3.4 Unterteilung des Strahlsegments nach dem überdeckten Skalarwertebereich. . . . 48
3.5 Organisation der Strahlteilintegrale der Länge N. 48
3.6 Gegenüberstellung von Ray Casting und Zellprojektion. 49
3.7 Gegenüberstellung der texturbasierten Volumenvisualisierung mittels zweidimensionaler und dreidimensionaler Texturen. 50
3.8 Klassifikation der nicht-degenerierten Fälle bei Shirly-Tuchman-Zellprojektion sowie deren Zerlegung in Dreiecke. 54

4.1 Schnitt eines Tetraeders mit einem Sehstrahl. 59
4.2 Veranschaulichung des fragmentbasierten Ray Casting im Zweidimensionalen und Elimination potentieller Eintrittspunkte. 60

4.3 Vertexprogramm zur blickpunktunabhängigen Zellprojektion mit emissivem optischen Modell. 63
4.4 Nomenklatur für Tetraeder. 64
4.5 Umlenken des berechneten Strahlparameters mittels SLT und SGE Instruktion sowie indirekter Adressierung. 65
4.6 Umlenken des berechneten Strahlparameters auf eine Komponente der Texturkoordinaten. Implementierung mittels Condition Code. 66
4.7 Artefakte an den Kanten zwischen zwei sichtbaren Tetraederflächen bei der Hardware-basierten Zellprojektion. 67
4.8 Fragment-Shader für fragmentorientierte Zellprojektion mit Isoflächendarstellung. 70
4.9 Einstellung für den Texture Shader bei einem abhängigen 3D-Texturnachschlag basierend auf dem minimalen Strahlparameter aus der Minimumtextur. 71
4.10 Unterschiedliche optische Modelle bei der blickpunktunabhänigen Zellprojektion. 72
4.11 Blickpunktunabhängige Isoflächendarstellung. 73

5.1 Prinzip des Hardware-basierten Ray Casting. 78
5.2 Nomenklatur für Tetraeder. 80
5.3 Beispiel für generalisierte Tetraederstreifen. 84
5.4 Prinzip des um Depth Peeling erweiterten Hardware-basierten Algorithmus zum Ray Casting von Tetraedergittern. 95
5.5 Volumendarstellung des Super Phoenix (Spx) Datensatzes unter Verwendung von Depth Peeling. 96
5.6 Beispielvisualisierungen mit dem Hardware-basierten Algorithmus zum Ray Casting von Tetraedergittern. 97
5.7 Der Effekt von Early Ray Termination bei der Anwendung von Hardware-basiertem Ray Casting. 99

6.1 Volumenkodierung mittels radialer Basisfunktionen. 106
6.2 Grundprinzip des RBF-basierten Visualisierungssystems. 107
6.3 Texturlayout für die Speicherung der RBF-Koeffizienten. 109
6.4 Fragmentprogramm für die Rekonstruktion von radialen Gauß-Basisfunktionen. . 111
6.5 Volumen- und Isoflächendarstellung eines Temperaturfeldes, welches durch Simulation der natürlichen Konvektion erzeugt wurde. 114
6.6 Splat-basierte Volumenvisualisierung des X38 Schock-Datensatzes. 115
6.7 RBF-basierte Visualisierungstechniken für Vektordaten. 116

7.1 Prinzip für das Hardware-unterstützte Slicing. 122
7.2 Textur zur Klassifikation innerhalb des Tetraeders liegender Fragmente. 124
7.3 Seitenansicht der drei unterschiedlichen Fälle von Ersatzgeometrie. 125
7.4 Beispiele für eine beliebig orientierbare Schnittebene, die mit Hilfe des Hardware-unterstützten Slicing-Algorithmus gezeichnet wird. 126
7.5 Einstellung der Register Combiners für die korrekte Dateninterpolation. 127
7.6 Berechnung der baryzentrischen Koordinaten eines auf die Slice-Ebene projizierten Vertex des Tetraeders. 128
7.7 Berechnung der Indizes von Start- und Endvoxel innerhalb einer Scanline der Bounding Box des Tetraeders. 129
7.8 Resampling-Ergebnisse für den Bluntfin- und den Heatsink-Datensatz. 133

8.1 Eindimensionales Beispiel für einen Datensatz mit vier Levels-of-Detail. 139
8.2 Anpassung benachbarter Bricks zur Erziehlung konsistenter Übergänge zwischen unterschiedlichen Auflösungsstufen. 141
8.3 Effiziente Speicherung der Brick-Daten als Hierarchie von Texturen. 142
8.4 Beispiel für eine Auflösungswahl, die basierend auf einer Verfeinerung mittels Oktalbaum nicht erreicht werden kann. 143
8.5 Typische Artefakte bei Darstellungen mit unterschiedlichem Level-of-Detail ohne Korrektur der partiellen Slice-Überlappungen. 145
8.6 Die zwei Klassen von Opazitätskorrektur an Level-Übergängen zwischen Bricks auf grobem und feinem Level. 147
8.7 Beispiel für die Level-of-Detail-Darstellung eines Magnetresonanztomographie-Datensatzes. 148
8.8 Medizinische Volumenvisualisierung eines Aneurysmas mit und ohne adaptive Repräsentation. 149
8.9 Level-of-Detail-Volumendarstellung des Engine-Datensatzes mittels dreidimensionaler Texturen. 150

9.1 Pseudocode der texturbasierten Volumenvisualisierung mit Bricking. 153
9.2 Volumenunterteilung in Bricks aus zweidimensionalen bzw. dreidimensionalen Texturen. 154
9.3 Funktionales Diagramm für flexible Hardware-unterstützte Volumendarstellung mit OpenSG. 155
9.4 Aufbau des OpenSG-Volumenknotens aus einzelnen Feldcontainern. 157
9.5 Verschiedene Darstellungsmodi, die über Shader-Objekte in OpenSG erzielt werden. 158

9.6 Einstellung der Register Combiners für die Isoflächendarstellung mit bis zu sechs direktionalen Lichtquellen. 161

9.7 Fragmentprogramm zur Realisierung von Prä-Integriertem Volume Rendering mittels dreidimensionaler Texturen. 163

9.8 Demonstration der zwei von OpenSG unterstützten Clipping-Mechanismen. . . . 164

9.9 Beispiele für Virtual Reality Installationen. 166

9.10 Client-Anwendung zur Ansteuerung geteilter Anzeigesysteme in OpenSG. 169

9.11 Verteilte Volumendarstellung mit 24 Cluster-Knoten, von denen jeder eine der 6×4 Kacheln einer geteilten Projektionsfläche berechnet. 170

Tabellenverzeichnis

4.1 Vergleich der Performanz des vertexorientierten Ansatzes zur Zellprojektion zwischen dem GeForce3 und GeForce4 Chip von nVidia. 76

5.1 Vergleich der Ergebnisse für die Erzeugung sequentieller und generalisierter Tetraederstreifen. 87

5.2 Texturen zur Speicherung des Gitters in Form von Tetraederlisten. 88

5.3 Texturkodierte Speicherung der Tetraederstreifen. 90

5.4 Traversierungstexturen für mehrere Render Targets. 92

5.5 Traversierungstexturen für ein einzelnes Render Target. 93

5.6 Performanz des Hardware-basierten Algorithmus zum Ray Casting für unterschiedliche Datensätze auf einer ATI Radeon 9700 Pro. 98

5.7 Vergleich der Bildwiederholraten zwischen der speicherplatzoptimierten und der performanzoptimierten Implementierung auf einer nVidia GeForce 6800. 100

6.1 Genauigkeit und Kompressionrate für RBF-kodierte Datensätze. 113

7.1 Vergleich der Rechenzeit bei Anwendung unterschiedlicher Hardware-unterstützter Resampling-Algorithmen. 134

7.2 Zeiten für das Software-Resampling des Heatsink- und Bluntfin-Datensatzes bei unterschiedlichen Auflösungen. 135

Literaturverzeichnis

[3Dlabs 2002] 3Dlabs, 2002. OpenGL 2.0 Shading Language White Paper. Verfügbar unter http://www.3dlabs.com/support/developer/ogl2/whitepapers.

[Akeley et al. 1990] K. Akeley, P. Haeberli, und D. Burns. 1990. The tomesh.c program. Tech. rep., Sillicon Graphics. Verfügbar auf SGI Developer's Toolbox CD.

[Albert 1972] A. Albert. 1972. Regression and the Moore-Penrose pseudoinverse. *Mathematics in Science and Engineering vol. 94*, S. xiii,180ff, Academic Press, New York-London.

[Arthur 2000] K. Arthur. 2000. *Effects of Field of View on Performance with Head-Mounted-Displays*. Doktorarbeit, University of North Carolina.

[Barnsley et al. 1988] M. F. Barnsley, A. Jacquin, F. Malassenet, L. Reuter, und A. D. Sloan. 1988. Harnessing Chaos for Image Synthesis. In *Computer Graphics (Proceedings of SIGGRAPH '88)*, vol. 22, S. 131–140.

[Beeson und Demer 2002] C. Beeson und J. Demer, 2002. Nvtristrip, library version. Software verfügbar via Internet Webseite, http://developer.nvidia.com/.

[Bentley 1982] J. L. Bentley. 1982. *Writing efficient programs*. Prentice-Hall software series. Prentice-Hall, Eaglewood Cliffs, N.J. ISBN 0-13-970251-2.

[Berger und Colella 1989] M. Berger und P. Colella. 1989. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics 82:64-84 (May)*.

[Berger und Oliger 1984] M. Berger und J. Oliger. 1984. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics 53 (March)*, S. 484–512.

[Bernardon et al. 2004] F. F. Bernardon, C. A. Pagot, J. L. D. Comba, und C. T. Silva. 2004. GPU-based Tiled Ray Casting using Depth Peeling. Tech. Rep. UUSCI-2004-006, SCI Institute, University of Utah.

[Bertram et al. 2000] M. Bertram, M. A. Duchaineau, B. Hamann, und K. I. Joy. 2000. Bicubic subdivision-surface wavelets for large-scale isosurface representation and visualization. In *Proc. of IEEE Visualization 2000*, S. 389–396.

[Bitter et al. 2004] I. Bitter, N. Neophytou, K. Mueller, und A. E. Kaufman. 2004. Squeeze: Numerical-Precision-Optimized Volume Rendering. In *Proceedings of Graphics Hardware 2004*, T. Akenine-Möller und M. McCool, Eds., S. 25–34.

[Boada et al. 2001] I. Boada, I. Navazo, und R. Scopigno. 2001. Multiresolution volume visualization with a texture-based octree. *The Visual Computer 17:3*, S. 185–197.

[Braitmaier et al. 2003] M. Braitmaier, M. Weiler, und T. Ertl. 2003. OpenManip: An Extensible Cross-Scene-Graph Framework for Direct Object Manipulation. In *Procceedings of OpenSG Symposium '03*, Dieter Fellner, Ed., S. 65–72, Eurographics.

[Branch et al. 1999] M. A. Branch, T. F. Coleman, und Y. A. Li. 1999. A subspace, interior, and conjugate gradient method for large-scale bound-constrained minimization problems. *SIAM Journal on Scientific Computing 21:1*, S. 1–23.

[Cabral et al. 1994] B. Cabral, N. Cam, und J. Foran. 1994. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. *1994 Symposium on Volume Visualization (October)*, S. 91–98, ACM SIGGRAPH. ISBN 0-89791-741-3.

[Callahan et al. 2004] S. Callahan, M. Ikits, J. Comba, und C. Silva. 2004. Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering. Tech. Rep. UUSCI-2004-003, SCI Institute, University of Utah.

[Campbell et al. 1986] G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, und L. A. Leske. 1986. Two bit/pixel full color encoding. In *Proceedings of SIGGRAPH '86*, S. 215–223, ACM Press.

[Carr et al. 2001] J. Carr, R. Beatson, J. Cherrie, T. Mitchell, W. Fright, B. McCallum, und T. Evans. 2001. Reconstruction and Representation of 3D Objects With Radial Basis Functions. In *Proc. of SIGGRAPH '01*, S. 67–76, ACM.

[Chidlow und Möller 2003] K. Chidlow und T. Möller. 2003. Rapid Emission Tomography Reconstruction. In *Workshop on Volume Graphics 2003 (VG03)*, S. 15–26.

[Cignoni et al. 1995] P. Cignoni, C. Montani, D. Sarti, und R. Scopigno. 1995. On the Optimization of Projective Volume Rendering. In *Visualization in Scientific Computing '95*, R. Scanteni, J. van Wijk, und P. Zanarini, Eds. S. 58–71, Springer-Verlag, Wien.

[Co et al. 2003] C. S. Co, B. Heckel, H. Hagen, B. Hamann, und K. I. Joy. 2003. Hierarchical Clustering for Unstructured Volumetric Scalar Fields. In *Proc. of IEEE Visualization '03*, S. 325–332, IEEE.

[Comba et al. 1999] J. Comba, J. T. Klosowski, N. Max, J. S. B. Mitchell, C. T. Silva, und P. L. Williams. 1999. Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids. *Computer Graphics Forum (Proceedings of Eurographics '99) 18:3*, S. 369–376.

[Cruz-Neira et al. 1992] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, und J. C. Hart. 1992. The CAVE: audio visual experience automatic virtual environment. *Commun. ACM 35:6*, S. 64–72, ACM Press.

[Cullip und Neumann 1994] T. J. Cullip und U. Neumann. 1994. Accelerating volume reconstruction with 3D texture mapping hardware. Tech. Rep. TR93-027, Department of Computer Science, University of North Carolina, Chapel Hill.

[Cyrus und Beck 1978] M. Cyrus und J. Beck. 1978. Generalized Two- and Three-Dimensional Clipping. In *Computers and Graphics 3(1)*, S. 23–28.

[Danskin und Hanrahan 1992] J. Danskin und P. Hanrahan. 1992. Fast Algorithms for Volume Ray Tracing. *1992 Workshop on Volume Visualization*, S. 91–98.

[Drebin et al. 1988] R. Drebin, L. Carpenter, und P. Hanrahan. 1988. Volume rendering. In *Proc. of SIGGRAPH '88*, S. 65–74.

[Eckel 1998] G. Eckel. 1998. *OpenGL Volumizer Programmer's Guide*. Silicon Graphics Computer Systems, Mountain View, CA, USA.

[Engel et al. 2001] K. Engel, M. Kraus, und T. Ertl. 2001. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, Annual Conference Series, S. 9–16.

[Evans et al. 1996] F. Evans, S. S. Skiena, und A. Varshney. 1996. Optimizing Triangle Strips for Fast Rendering. *Proc. of IEEE Visualization '96*, S. 319–326.

[Everitt 2001] C. Everitt. 2001. Interactive Order-Independent Transparency. Tech. rep., nVidia Whitepaper, Verfügbar unter www.nvidia.com/object/Interactive_Order_Transparency.html.

[Foley et al. 1990] J. D. Foley, A. van Dam, S. K. Feiner, und J. F. Hughes. 1990. *Computer Graphics: Principles and Practice*, second ed. Addison-Wesley, Reading, MA.

[Fornefett et al. 1999] M. Fornefett, K. Rohr, und H. S. Stiehl. 1999. Elastic Registration of Medical Images Using Radial Basis Functions with Compact Support. In *Proc. of Computer Vision and Pattern Recognition '99*, S. 402–409.

[Franke und Hagen 1999] R. Franke und H. Hagen. 1999. Least squares surface approximation using multiquadrics and parametric domain distortion. *Computer Aided Geometric Design 16:3 (March)*, S. 177–196.

[Franke und Nielson 1991] R. Franke und G. M. Nielson. 1991. Scattered data interpolation and applications: A tutorial and survey. In *Geometric Modelling, Methods and Applications*, H. Hagen und D. Roller, Eds. S. 131–160, Springer-Verlag.

[Franke 1982] R. Franke. 1982. Scattered Data Interpolation: Test of Some Methods. *Mathematics of Computation 38:157*, S. 181–200.

[Garrity 1990] M. P. Garrity. 1990. Raytracing Irregular Volume Data. In *Proceedings of the 1990 Workshop on Volume Visualization*, S. 35–40, ACM Press.

[GLUT 2004] GLUT, 2004. GLUT - The OpenGL Utility Toolkit. Verfügbar unter http://www.opengl.org/resources/libraries/glut/glut_downloads.html.

[Grzeszczuk et al. 1998] R. Grzeszczuk, C. Henn, und R. Yagel. 1998. Advanced Geometric Techniques for Ray Casting Volumes. In *ACM SIGGRAPH '98, Course 4 Notes*, ACM.

[Gumhold et al. 1999] S. Gumhold, S. Guthe, und W. Straßer. 1999. Tetrahedral Mesh Compression with the Cut–Border Machine. In *IEEE Visualization '99*, S. 51–58.

[Gumhold und Straßer 1998] S. Gumhold und W. Straßer. 1998. Real Time Compression of Triangle Mesh Connectivity. In *Proc. of SIGGRAPH '98*, S. 133–140.

[Guthe et al. 2002] S. Guthe, M. Wand, J. Gonser, und W. Straßer. 2002. Interactive Rendering of Large Volume Data Sets. In *Proc. of IEEE Visualization 2002*, S. 53–60.

[Guthe et al. 2003] S. Guthe, S. Röttger, A. Schieber, W. Straßer, und T. Ertl. 2003. High-Quality Unstructured Volume Rendering on the PC Platform. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '02*, S. 119–125.

[Guthe und Straßer 2004] S. Guthe und W. Straßer. 2004. Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. *Computers & Graphics 28:1 (Feb.)*, S. 51–58.

[Guthe 2002] S. Guthe, 2002. Persönliche Kommunikation.

[Haimes und Kenwright 1999] R. Haimes und D. Kenwright. 1999. On the Velocity Gradient Tensor and Fluid Feature Extraction. *AIAA:99-3288*.

[Hanrahan und Lawson 1990] P. Hanrahan und J. Lawson. 1990. A Language for Shading and Lighting Calculations. In *Comp. Graphics*, vol. 24, S. 289–298.

[Hardy 1971] R. L. Hardy. 1971. Multiquadric equations of topography and other irregular surfaces. *Journal of Geophysical Research 76:8*, S. 1905–1915.

[Hardy 1990] R. L. Hardy. 1990. Theory and applications of the multiquadric-biharmonic method: 20 years of discovery 1968-1988. *Computers and Mathematics with Applications 19:8-9*, S. 163–208.

[Hege et al. 1993] H.-C. Hege, T. Höllerer, und D. Stalling. 1993. Volume Rendering, Mathematical Foundations and Algorithmic Aspects. Tech. Rep. 93-7, Konrad-Zuse-Zentrum für Informationstechnik Berlin.

[Hopf und Ertl 2000] M. Hopf und T. Ertl. 2000. Hardware Accelerated Wavelet Transformations. In *Proc. of EG/IEEE TCVG Symposium on Visualization VisSym '00*, S. 93–103, EG/IEEE.

[Hopf 2004] M. Hopf. 2004. *Hierarchical Methods for Filtering and Visualization Based on Graphics Hardware*. Doktorarbeit, Universität Stuttgart.

[Ihm und Park 1998] I. Ihm und S. Park. 1998. Wavelet-Based 3D Compression Scheme for Very Large Volume Data. In *Graphics Interface*, S. 107–116.

[Jang et al. 2004] Y. Jang, M. Weiler, M. Hopf, J. Huang, D. S. Ebert, K. P. Gaither, und T. Ertl. 2004. Interactively Visualizing Procedurally Encoded Scalar Fields. In *Proc. of EG/IEEE TCVG Symposium on Visualization VisSym '04*, S. 35–44, EG/IEEE.

[Jang et al. 2005] Y. Jang, J. Huang, D. S. Ebert, K. P. Gaither, M. Weiler, S. Stegmaier, und T. Ertl. 2005. Procedural Encoding of Volume Data. *IEEE Transactions on Visualization and Computer Graphics (eingereicht)*.

[Jeong und Hussain 1995] J. Jeong und F. Hussain. 1995. On the Identification of a Vortex. *Journal of Fluid Mechanics (285)*, S. 69–94, Cambridge University Press.

[Jolliffe 1986] I. T. Jolliffe. 1986. *Principal Component Analysis*. Springer-Verlag.

[Kaufmann und Shimony 1986] A. Kaufmann und E. Shimony. 1986. 3D scan-conversion algorithms for voxel-based graphics. In *Proc. of ACM Workshop on Interactive 3D Graphics, Computer Graphics*, S. 45–75.

[Kessenich et al. 2004] J. Kessenich, D. Baldwin, und R. Rost, 2004. The OpenGL Shading Language. Verfügbar unter: http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf. 3Dlabs, Inc.

[Kim und Shin 1999] T. Kim und Y. Shin. 1999. An Efficient Wavelet-based Compression Method for Volume Rendering. In *Proc. of Pacific Graphics '99*, S. 147–157.

[King et al. 2001] D. King, C. M. Wittenbrink, und H. J. Wolters. 2001. An Architecture for Interactive Tetrahedral Volume Rendering. In *Proceedings of the International Workshop on Volume Graphics 2001*, K. Mueller und A. Kaufman, Eds., S. 163–180, Springer-Verlag.

[Kniss et al. 2001] J. Kniss, G. Kindlmann, und C. Hansen. 2001. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. In *Proc. of IEEE Visualization '01*, S. 255–262, IEEE.

[Kniss et al. 2002] J. Kniss, S. Premoze, C. Hansen, und D. Ebert. 2002. Interactive translucent volume rendering and procedural modeling. In *Proceedings of IEEE Visualization 2002*, S. 109–116.

[Kniss et al. 2003] J. Kniss, S. Premoze, C. Hansen, P. Shirley, und A. McPherson. 2003. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics 9:2*, S. 150–162.

[Knittel 2002] G. Knittel. 2002. Using Pre-Integrated Transfer Functions in an Interactive Software System for Volume Rendering. In *Proceedings of Short Presentations EUROGRAPHICS 2002*, S. 119–123.

[Kowalsky 1993] R. Kowalsky. 1993. *The Science of Virtual Reality and Virtual Environments*. Addison-Wesley, Wokingham, England.

[Kraus et al. 2004] M. Kraus, W. Qiao, und D. S. Ebert. 2004. Projecting Tetrahedra without Rendering Artifacts. In *Proceedings of IEEE Visualization 2004*, S. 27–34, IEEE.

[Kraus und Ertl 2001] M. Kraus und T. Ertl. 2001. Cell-Projection of Cyclic Meshes. In *Proceedings of IEEE Visualization 2001*, T. Ertl, K. Joy, und A. Varshney, Eds., S. 215–222, ACM Press.

[Kraus und Ertl 2002] M. Kraus und T. Ertl. 2002. Implementing Ray Casting in Tetrahedral Meshes with Programmable Graphics Hardware. Tech. Rep. 1, Visualization and Interactive Systems Group at the University of Stuttgart.

[Krüger und Westermann 2003] J. Krüger und R. Westermann. 2003. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization 2003*, S. 287–292.

[Krüger 1990] W. Krüger. 1990. The Application of Transport Theory to the Visualization of 3-D Scalar Data Fields. *Proc. of IEEE Visualization '90*, S. 273–280, IEEE Computer Society Press., IEEE.

[Lacroute und Levoy 1994] P. Lacroute und M. Levoy. 1994. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Proc. of SIGGRAPH '94*, S. 451–457, ACM.

[LaMar et al. 1999] E. LaMar, B. Hamann, und K. Joy. 1999. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *Proc. of IEEE Visualization '99*, S. 355–361, IEEE CS Press.

[Levoy 1988] M. Levoy. 1988. Display of Surfaces from Volume Data. *Computer Graphics & Applications 8:3 (May)*, S. 29–37, ACM.

[Levoy 1990] M. Levoy. 1990. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics 9:3*, S. 245–261.

[Lorensen und Cline 1987] W. E. Lorensen und H. E. Cline. 1987. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics 21:4 (July)*, S. 163–169. Proc. of ACM SIGGRAPH '87.

[Lum et al. 2004] E. B. Lum, B. Wilson, und K.-L. Ma. 2004. High-Quality Lighting and Efficient Pre-Integration for Volume Rendering. In *Proc. of EG/IEEE TVCG Symposium on Visualization 2004*, S. 25–34.

[Mace 2004] R. Mace, 2004. OpenGL ARB Superbuffers. Verfügbar unter: http://www.ati.com/developer/gdc/SuperBuffers.pdf. ATI Research, Inc.

[Malzbender 1993] T. Malzbender. 1993. Fourier-Volume-Rendering. *Transactions on Graphics 12:3 (July)*, S. 233–250, ACM.

[Mark et al. 2003] W. R. Mark, S. Glanville, und K. Akeley. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proc. of SIGGRAPH '03*, S. 896–907, ACM.

[MathWorks 2003] MathWorks, 2003. Matlab Optimization Toolbox: fsolve. Matlab documentation, available at http://www.mathworks.com/access/helpdesk/help/toolbox/optim/fsolve.html.

[Max et al. 1990] N. Max, P. Hanrahan, und R. Crawfis. 1990. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *ACM Computer Graphics (Proceedings of San Diego Workshop on Volume Visualization 1990) 24:5*, S. 27–33, ACM Press.

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

[Meißner et al. 2002] M. Meißner, U. K. A. G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, und P. F. R. Proksa. 2002. VIZARD II: A Reconfigurable Interactive Volume Rendering System. In *Proceedings of Graphics Hardware 2002*, S. 137–146.

[Microsoft 2004] Microsoft, 2004. Microsoft DirectX. Verfügbar unter http://www.microsoft.com/ windows/directx. Microsoft Corporation.

[Molnar et al. 1994] S. Molnar, M. Cox, D. Ellsworth, und H. Fuchs. 1994. A Sorting Classification of Parallel Rendering. *Computer Graphics and Applications: Special Issue on Rendering 14:4 (July)*, S. 23–32.

[Morse et al. 2001] B. S. Morse, T. S. Yoo, P. Rheingans, D. T. Chen, und K. R. Subramanian. 2001. Interpolating Implicit Surfaces From Scattered Surface Data Using Compactly Supported Radial Basis Functions. In *Proc. of Shape Modeling International*, S. 89–98, IEEE.

[Müller et al. 1998] K. Müller, U. Rist, und S. Wagner. 1998. Enhanced Visualization of Late-Stage Transitional Structures using Vortex Identification and Automatic Feature Extraction. In *Computational Fluid Dynamics*, S. 786–791, John Wiley & Sons.

[Muraki 1993] S. Muraki. 1993. Approximation and Rendering of Volume Data Using Wavelet Transforms. *Computer Graphics and Applications 13:4*, S. 50–56.

[NCSA 2004] NCSA, 2004. CAVERNUS Image Gallery. National Center for Supercomputing Applications, http://www.ncsa.uiuc.edu/VR/cavernus/GALLERY/Gallery.html.

[Nguyen und Saupe 2001] K. G. Nguyen und D. Saupe. 2001. Rapid High Quality Compression of Volume Data for Visualization. *Computer Graphics Forum 20:3*, S. 49–56.

[Nielson 1993] G. M. Nielson. 1993. Scattered Data Modeling. *IEEE Comput. Graph. Appl. 13:1*, S. 60–70.

[NLM 1987] NLM, 1987. The National Library of Medicine. The Visible Human Project. http://www.nlm.nih.gov/research/visible/visible_human.html.

[nVidia 2002] nVidia, 2002. Cg Language Specification. Verfügbar unter http://developer.nvidia.com/cg.

[nVidia 2004] nVidia, 2004. NVIDIA GPU Programming Guide. Verfügbar unter http://download.nvidia.com/developer/GPU_Programming_Guide/GPU_Programming_Guide.pdf.

[OpenGL ARB 2004] OpenGL ARB, 2004. OpenGL Extension Registry. Verfügbar unter http://oss.sgi.com/projects/ogl-sample/registry/.

[OpenSG 2004] OpenSG, 2004. www.opensg.org.

[Pfister et al. 1999] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, und L. Seiler. 1999. The VolumePro Real-Time Ray-casting System. *Proceedings of SIGGRAPH '99 (August)*, S. 251–260, Addison Wesley Longman. ISBN 0-20148-560-5. Held in Los Angeles, California.

[Phong 1975] B. T. Phong. 1975. Illumination for Computer Generated Pictures. *Communications of the ACM 18:6*, S. 311–317.

[Porter und Duff 1984] T. Porter und T. Duff. 1984. Compositing Digital Images. *Proc. of SIGGRAPH '84 18:3*, S. 253–259.

[Prakash und Manohar 1995] C. E. Prakash und S. Manohar. 1995. Volume rendering of unstructured grids–a voxelization approach. *Computers & Graphics 19:5 (Sept.)*, S. 711–726, Pergamon Press / Elsevier Science. ISSN 0097-8493.

[Rezk-Salama et al. 2000] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, und T. Ertl. 2000. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Proc. of Workshop on Graphics Hardware '00*, S. 109–118,147, EG/ACM SIGGRAPH.

[Rezk-Salama 2002] C. Rezk-Salama. 2002. *Volume Rendering Techniques for General Purpose Graphics Hardware*. Doktorarbeit, Lehrstuhl für Graphische Datenverarbeitung, Universität Erlangen-Nürnberg.

[Riedel und Deisinger 1995] O. Riedel und J. Deisinger. 1995. Displays für Virtual Reality. In *Proceedings of the 10. Electrical Displays 95*, S. 81–92, Network GmbH.

[Riguer 2002] G. Riguer, 2002. Performance Optimization Techniques for ATI Graphics Hardware with DirectX ® 9.0. verfügbar unter http://mirror.ati.com/developer/techpapers.html.

[Rodler 1999] F. Rodler. 1999. Wavelet based 3D Compression with Fast Random Access for Very Large Volume Data. In *Proc. of Pacific Graphics '99*, S. 108–117.

[Rossignac 1997] J. Rossignac, 1997. Geometric Simplification and Compression. ACM SIGGRAPH '97, Course 25 Notes.

[Rossignac 1999] J. Rossignac. 1999. Edgebreaker: Connectivity Compression for Triangle Meshes. *IEEE Transactions on Visualization and Computer Graphics 5:1*, S. 47–61.

[Roth et al. 2004] M. Roth, G. Voss, und D. Reiners. 2004. Multi-threading and clustering for scene graph systems. *Computers and Graphics 28:1 (Feb.)*, S. 63–66.

[Röttger et al. 2000] S. Röttger, M. Kraus, und T. Ertl. 2000. Hardware-Accelerated Volume and Isosurface Rendering Based On Cell-Projection. In *Proceedings of IEEE Visualization 2000*, T. Ertl, B. Hamann, und A. Varshney, Eds., S. 109–116, ACM Press.

[Röttger et al. 2003] S. Röttger, S. Guthe, D. Weiskopf, und T. Ertl. 2003. Smart Hardware-Accelerated Volume Rendering. In *Proc. of EG/IEEE TCVG Symposium on Visualization VisSym '03*, S. 231–238.

[Röttger und Ertl 2002] S. Röttger und T. Ertl. 2002. A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids. In *Proc. of IEEE Volume Visualization and Graphics Symp. 2002*, S. 23–28.

[Savchenko et al. 1995] V. V. Savchenko, A. A. Pasko, O. G. Okunev, und T. L. Kunii. 1995. Function Representation of Solids Reconstructed from Scattered Surface Points and Contours. *Computer Graphics Forum 14:4*, S. 181–188, ACM.

[Schneider und Westermann 2003] J. Schneider und R. Westermann. 2003. Compression Domain Volume Rendering. In *Proceedings of IEEE Visualization 2003*, S. 293–300.

[Schulze et al. 2003] J. Schulze, M. Kraus, U. Lang, und T. Ertl. 2003. Integrating Pre-Integration into the Shear-Warp Algorithm. In *Proceedings of Third International Workshop on Volume Graphics*, S. 109–118.

[Segal und Akeley 2004] M. Segal und K. Akeley, 2004. The OpenGL Graphics System: A Specification (Version 2.0 - September 7, 2004). Verfügbar unter http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf, September. Silicon Graphics, Inc.

[Shirley und Tuchman 1990] P. Shirley und A. Tuchman. 1990. A Polygonal Approximation to Direct Scalar Volume Rendering. *ACM Computer Graphics (Proceedings of San Diego Workshop on Volume Visualization 1990) 24:5*, S. 63–70, ACM Press.

[Shreiner et al. 2004] D. Shreiner, M. Woo, J. Neider, und T. Davis. 2004. *OpenGL Programming Guide*, 1.4 ed. Addison-Wesley.

[Speckmann und Snoeyink 1997] B. Speckmann und J. Snoeyink. 1997. Easy triangle for TIN terrain models. *In Canadian Conference on Computational Geometry*, S. 239–244.

[Speray und Kennon 1990] D. Speray und S. Kennon. 1990. Volume Probes: Interactive Data Exploration on Arbitrary Grids. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, S. 5–12.

[Stegmaier et al. 2003] S. Stegmaier, J. Diepstraten, M. Weiler, und T. Ertl. 2003. Widening the Remote Visualization Bottleneck. In *Procceedings of ISPA '03*, S. 174–179, IEEE.

[Stegmaier und Ertl 2004] S. Stegmaier und T. Ertl. 2004. A Graphics Hardware-based Vortex Detection and Visualization System. In *Proceedings of IEEE Visualization 2004*, S. 195–202.

[Stein et al. 1994] C. M. Stein, B. G. Becker, und N. L. Max. 1994. Sorting and Hardware Assisted Rendering for Volume Visualization. In *Proceedings of the 1994 Symposium on Volume Visualization*, A. Kaufman und W. Krueger, Eds., S. 83–89, ACM Press.

[Strengert et al. 2004] M. Strengert, M. Magallon, D. Weiskopf, S. Guthe, und T. Ertl. 2004. Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters. *Parallel Graphics and Visualization*, S. 41–48.

[Sutherland und Hodgman 1974] B. Sutherland und G. Hodgman. 1974. Reentrant Polygon Clipping. *CACM 17(1) (January)*, S. 32–42.

[Szymczak und Rossignac 1999] A. Szymczak und J. Rossignac. 1999. Grow & Fold: Compression of Tetrahedral Meshes. *In Proc. ACM Symp. on Solid Modeling and Applications*, S. 54–64.

[Totsuka und Levoy 1993] T. Totsuka und M. Levoy. 1993. Frequency Domain Volume Rendering. *Computer Graphics 27:4 (August)*, S. 271–278, ACM.

[Touma und Gotsman 1998] C. Touma und C. Gotsman. 1998. Triangle Mesh Compression. *GI98 Conference Proceedings*, S. 26–34.

[Turk und O'Brien 2002] G. Turk und J. F. O'Brien. 2002. Modelling with implicit surfaces that interpolate. *Transactions on Graphics 21:4*, S. 855–873, ACM.

[Weiler et al. 2000] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, und T. Ertl. 2000. Level-Of-Detail Volume Rendering via 3D Textures. In *Volume Visualization and Graphics Symposium 2000*, S. 7–13, IEEE.

[Weiler et al. 2002] M. Weiler, M. Kraus, und T. Ertl. 2002. Hardware-Based View-Independent Cell Projection. In *Procceedings of IEEE Symposium on Volume Visualization 2002*, S. 13–22, IEEE.

[Weiler et al. 2003a] M. Weiler, M. Kraus, M. Merz, und T. Ertl. 2003. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proceedings of IEEE Visualization '03*, S. 333–340, IEEE.

[Weiler et al. 2003b] M. Weiler, M. Kraus, M. Merz, und T. Ertl. 2003. Hardware-Based View-Independent Cell Projection. *IEEE Transactions on Visualization and Computer Graphics (Special Issue on IEEE Visualization 2002) 9:2 (April-June)*, S. 163–175.

[Weiler et al. 2004a] M. Weiler, T. Klein, und T. Ertl. 2004. Direct volume rendering in OpenSG. *Computers and Graphics 28:1 (Feb.)*, S. 93 – 98.

[Weiler et al. 2004b] M. Weiler, P. N. Mallón, M. Kraus, und T. Ertl. 2004. Texture-Encoded Tetrahedral Strips. In *Proceedings of Symposium on Volume Visualization 2004*, S. 71–78, IEEE.

[Weiler et al. 2005] M. Weiler, J. Huang, Y. Jang, S. Stegmaier, K. P. Gaither, D. S. Ebert, und T. Ertl. 2005. Hardware-Assisted Feature Analysis and Visualization of Procedurally Encoded Multifield Volumentric Data. *IEEE Computer Graphics and Application 25:5*, S. 72–81.

[Weiler et al. 2006] M. Weiler, M. Kraus, S. Guthe, T. Ertl, und W. Straßer. 2006. Ray Casting with Programmable Graphics Hardware. In *Proceedings of Scientific Visualization - Dagstuhl '03*, G.-P. Bonneau, T. Ertl, und G. M. Nielson, Eds., S. 115–130, Springer-Verlag.

[Weiler und Ertl 2001] M. Weiler und T. Ertl. 2001. Hardware-Software-Balanced Resampling for the Interactive Visualization of Unstructured Grids. In *Proc. of IEEE Visualization '01*, S. 199–206, IEEE.

[Weiler und Ertl 2002] M. Weiler und T. Ertl. 2002. Ein Volume-Rendering-Framework für OpenSG. In *Proceedings of OpenSG Symposium 2002*. Verfügbar unter http://www.opensg.org/OpenSGPLUS/symposium/Papers2002/.

[Weiskopf et al. 2003] D. Weiskopf, K. Engel, und T. Ertl. 2003. Interactive Clipping Techniques for Texture-Based Volume Visualization and Volume Shading. *IEEE Transactions on Visualization and Computer Graphics 9:3*, S. 298–312.

[Westermann und Ertl 1997] R. Westermann und T. Ertl. 1997. The VSBUFFER: Visibility Ordering unstructured Volume Primitives by Polygon Drawing. In *Proceedings of IEEE Visualization '97*, S. 35–43, IEEE Computer Society Press, IEEE Computer Society.

[Westermann und Ertl 1998] R. Westermann und T. Ertl. 1998. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (ACM SIGGRAPH '98 Proceedings)*, S. 169–177.

[Westermann 1994] R. Westermann. 1994. A Multiresolution Framework for Volume Rendering. In *ACM Symposium on Volume Visualization'94*, S. 51–58.

[Westermann 2001] R. Westermann. 2001. The Rendering of Unstructured Grids Revisited. In *Eurographics/IEEE Symposium on Visualization 2001 (Proceedings)*, S. 65–74.

[Westover 1990] L. Westover. 1990. Footprint Evaluation for Volume rendering. *Computer Graphics 24:4 (August)*, S. 367–376. ACM SIGGRAPH '90 Conference Proceedings.

[Wilhelms et al. 1996] J. Wilhelms, P. van Gelder, P. Tarantino, und G. J. 1996. Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids. In *Proc. of IEEE Visualization '96*, S. 57–65.

[Wilhelms und Van Gelder 1992] J. Wilhelms und A. Van Gelder. 1992. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics 11:3 (July)*, S. 201–227.

[Williams und Max 1992] P. L. Williams und N. Max. 1992. A volume density optical model. In *ACM Computer Graphics (1992 Workshop on Volume Visualization)*, S. 61–68.

[Williams 1992] P. L. Williams. 1992. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics 11:2*, S. 103–126, ACM Press.

[Wittenbrink 1999] C. M. Wittenbrink. 1999. CellFast: Interactive unstructured Volume Rendering. In *IEEE Visualization '99 Late Breaking Hot Topics*, S. 21–24.

[Wittenbrink 2001] C. M. Wittenbrink. 2001. R-Buffer: A Pointerless A-Buffer Hardware Architecture. In *Proceedings of Graphics Hardware 2001*, S. 73–80, ACM Press.

[Woodward 2004] P. Woodward, 2004. The Powerwall. http://www.lcse.umn.edu/research/powerwall/powerwall.html.

[Wylie et al. 2002] B. Wylie, M. Kenneth, L. A. Fisk, und P. Crossno. 2002. Tetrahedral Projection using Vertex Shaders. In *Procceedings of IEEE Symposium on Volume Visualization 2002*, S. 7–12, IEEE.

[Xiang et al. 1999] X. Xiang, M. Held, und J. S. B. Mitchell. 1999. Fast and effective stripification of polygonal surface models. *In ACM Sympos. Interactive 3D Graphics*, S. 71–78.

[Yagel et al. 1996] R. Yagel, D. Reed, A. Law, P. Shih, und N. Shareef. 1996. Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing. In *ACM Symposium on Volume Visualization '96*, S. 55–63.

[Zhang et al. 2002] Y. Zhang, R. Rohling, und D. K. Pai. 2002. Direct surface extraction from 3D freehand ultrasound images. In *Proc. of IEEE Visualization '02*, S. 45–52, IEEE.