Compression and Visualization of Large and Animated Volume Data

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Stefan Guthe
aus Castrop-Rauxel

Tübingen
2004

Tag der mündlichen Qualifikation:	12.05.2004
Dekan:	Prof. Dr. U. Güntzer
1. Berichterstatter:	Prof. Dr. W. Straßer
2. Berichterstatter:	Prof. Dr. E. Gröller
	(Technische Universität Wien)
3. Berichterstatter:	Prof. Dr. A. Schilling

Zusammenfassung

Die Volumenvisualisierung beschäftigt sich mit der Darstellung von skalaren Feldern, die als Funktion des Raumes gegeben sind, in einer Weise, dass sie vom Benutzer interpretiert werden können. Bei der Visualisierung dieser Felder gibt es drei zentrale Probleme, die alle in dieser Arbeit behandelt werden. Das erste Problem ist eine adäquate Darstellung des Volumens zu erhalten um alle interessanten Details so schnell wie möglich finden und analysieren zu können. Das zweite Problem ist die Repräsentation der Daten und damit auch die Datenmenge, die schnell die Größe des Arbeitsspeichers überschreiten kann. Das dritte Problem tritt bei zeitveränderlichen oder animierten Daten auf, da hier oftmals nur sehr wenige oder sogar nur ein einzelner Zeitschritt im Arbeitsspeicher gehalten werden kann.

Die interaktive Visualisierung großer und animierter Volumen-Datensätze ist, vor allem im Bereich der medizinischen und physikalischen Anwendungen, ein sehr wichtiges Problem, das sich nur durch speziell angepasste Algorithmen lösen lässt. Während die Daten in der Medizin für gewöhnlich als reguläre Gitter gegeben sind, werden bei physikalischen Simulationen oft strukturierte oder unstrukturierte Gitter verwendet um sich besser an die Problemstellung anzupassen.

Bei der Darstellung von Volumendaten auf regulären Gittern gab es in den vergangenen Jahren große Fortschritte, vor allem im Bereich der Bildqualität. Ein zentrales Problem bleibt allerdings weiterhin die große Datenmenge, die für jedes einzelne Bild bearbeitet werden muss. Als Lösungsansätze bieten sich zum einen Mehrfachgitterverfahren an, die Teile des Datensatzes nicht oder nur in geringer Auflösung betrachten, und Kompressionsverfahren, die die Datenmenge an sich reduzieren. Auf beide Verfahren wird in dieser Arbeit eingegangen und es wird untersucht, wie sie sich effizient kombinieren lassen.

Im Fall von strukturierten oder unstrukturierten Gittern kommt noch ein weiteres Problem bei der Darstellung hinzu. Im Gegensatz zu der festen Reihenfolge, in der die Daten zur Darstellung von regulären Gittern behandelt werden, muss hier vorab entschieden werden, in welcher Reihenfolge die Daten zu bearbeiten sind. Die kann, je nachdem, ob das Gitter konvex oder konkav ist, bzw. ob zyklische Überdeckung zwischen Zellen besteht, zusätzliche Zellen zum Datensatz hinzu fügen, die für jedes Bild neu berechnet werden müssen, da sie von der Position des Betrachters abhängen. Nachdem die Sortierung der Zellen fest steht, muss nun jede Zelle einzeln dargestellt werden. Diese Darstellung kann dabei ähnlich wie bei regulären Gittern erfolgen, um eine gleich hohe Qualität zu gewährleisten. Ein weiterer Punkt bei strukturierten oder unstrukturierten Gittern ist die Kompression, da die Datenmengen hier auch sehr große Ausmaße annehmen können.

Eine weitere Anwendung der Darstellung unstrukturierter Gitter ist das so genannte Displacement Mapping, bei dem ein Höhenfeld über einem beliebigen Dreiecksnetz dargestellt wird. Um eine gute Bildqualität zu erhalten, muss der bestehende Algorithmus allerdings an einigen Stellen modifiziert werden. Diese Modifikationen sind zum Teil Vereinfachungen, da man im allgemeinen nur an einer einzigen Iso-Fläche interessiert ist, aber auch neue Fähigkeiten, denn Beleuchtung und Farbgebung sind beim Displacement Mapping wesentlich komplexer.

Abstract

Volume visualization is the rendering of scalar fields in a way that it can be interpreted. The scalar field is given as a function of space. Visualizing these fields, three common problems are faced that will be discussed in this work. The first problem is to find a representation that allows for fast access and analysis of interesting parts within the volume. The second problem is to reduce the size of this representation since it can easily exceed the size of the main memory. The third problem is that for time dependent or animated volume data only very few or even a single time step can be held in main memory.

Interactive visualization of large and animated volume data is, especially in the area of medical and physical applications, a very important problem. This problem has to be solved with special algorithms. While the data in medical applications is usually sampled on a regular grid, physical simulations use structured or unstructured grids to better adapt to the details within the volume.

There have been a lot of publications on very good research for rendering volume data sampled on regular grids. The main focus of the research was to improve the image quality of the final renderings. Reducing the amount of data to be processed for each image still remains as a very important problem that has to be solved. There are two existing solutions to this problem. One is to use multi-resolution algorithms in order to process parts of the volume data at lower resolutions or not to process them at all. The other one is to use compression in order to reduce the size of the data set in the first place. Both approaches will be discussed and it will be investigated how to combine them efficiently.

For rendering structured or unstructured grids an additional problem arises. In contrast to the fixed rendering order of regular grids, the visibility order of all cells has to be determined prior to rendering. This can also introduce new cells depending on whether the grid is convex or concave or if it contains visibility cycles. These additional cells have to be computed for each image since they depend on the location of the viewer. After the visibility order has been determined each cell is rendered individually. The rendering works similar to the rendering of regular grids in order to achieve the same high quality results. Finally, compression for structured and unstructured grids is considered since the amount of data can also be very high for this representation.

A further application of rendering unstructured grids is displacement mapping. Here a simple height field over any kind of triangle mesh is rendered. To achieve a good image quality however, the previous rendering approaches need to be modified. These modifications include simplifications since the rendering is only interested in a single iso-surface but also new features since lighting and coloring are much more complex for displacement mapping.

Acknowledgements

This work would not have been possible without the help and support of a lot of people. Special thanks to my advisor, Prof. Dr.-Ing. Dr.-Ing. E.h. Wolfgang Straßer, for his support and confidence in my ideas. Only his connections to researchers in industry and academia enabled me to always work with both, the cutting edge graphics hardware and to communicate with those who know how to handle its capabilities.

I would like to thank David Kirk, Paul Heckbert and Jonah Alben at NVIDIA for an invaluable 3 month Internship in 2003. Thanks must also go to David Kirk at NVIDIA and Michael Dogget at ATI for always supplying me with the latest graphics hardware, drivers and bug fixes.

Since working alone is impossible nowadays, I would like to thank my colleagues and other people I had the opportunity to work with. Stefan Gumhold for suggesting wavelet compression for animated volume data and his support at the very beginning of my work, Michael Meissner for asking me if I would like to do volume rendering on the graphics card (a GeForce3 at that time), Michael Wand for helping me with the ideas of multiresolution rendering and writing the paper together with me, Martin Kraus for his ideas on rendering of tetrahedral meshes, Stefan Roettger for further ideas on that topic, Johannes Hirche for his displacement mapping algorithm and other valuable discussions, Armin Kanitsar for the Christmas Tree data set, the Christmas Tree case study and the best case study award, and Gunter Knittel for always telling me what I still can't do on the graphics card. I would also like to thank two students that did some work on this thesis. Thanks to Julius Gonser for his efficient cache implementation for multiresolution volume rendering and Andreas Schieber for his tetrahedral sorting algorithm.

Most of this work was funded by the Sonderforschungsbereich (SFB) 382 of the German Research Council (DFG). During the last time period, this work was sponsored by the NVIDIA Fellowship Program 2003-2004.

I would like to thank Melanie Künzel for always believing in me, Stefan Kimmerle, Sven Fleck and Johannes Hirche for proof reading, and finally my friends and my family for their never ending support.

Contents

1 Introduction **1**
1.1 Volume Representation 2
1.1.1 Cartesian Grids 2
1.1.2 Regular Grids 2
1.1.3 Rectangular Grids 3
1.1.4 Structured Grids 3
1.1.5 Unstructured Grids 4
1.2 Optical Model 5
1.2.1 Absorption 5
1.2.2 Emission 6
1.2.3 Emission & Absorption 8
1.2.4 Lighting, Scattering & General Light Transportation 9
1.2.5 Numeric Integration 10
1.3 Transfer Functions 13
1.3.1 Gradient Magnitude Modulation 14
1.3.2 Multi-Dimensional Transfer Functions 15
1.3.3 Pre-Integration Technique 16
1.3.4 Pre-Integrated Lighting 18

2 Regular Grids **23**
2.1 Hardware Accelerated Rendering 23
2.1.1 Simple Shading 24
2.1.2 Pre-Classification 26
2.1.3 Post-Classification 28
2.1.4 Pre-Integration 31
2.1.5 Pre-Integrated Lighting 37
2.1.6 Ray-casting 41
2.1.7 Results 50
2.2 Compression 51
2.2.1 Wavelet Transformation 51
2.2.2 Entropy Coding 51
2.2.3 Animated Volume Data 52
2.2.4 Large Volume Data 55
2.2.5 Results 55
2.3 Multi-Resolution Rendering 58
2.3.1 Multi-Resolution Hierarchy 59
2.3.2 Projective Classification 61
2.3.3 View-dependent Priority Schedule 61

2.3.4 Error Estimation 62
2.3.5 Visibility Testing 63
2.3.6 Rendering of Blocks 65
2.3.7 Caching Strategy 66
2.3.8 Results 66
2.4 Conclusion & Future Work 71

3 Unstructured Grids **73**
3.1 Visibility Sorting 75
3.1.1 Tetrahedral Convexification 76
3.1.2 Handling Cycles 80
3.2 Hardware Accelerated Rendering 81
3.2.1 Simple Shading 81
3.2.2 Pre-Integration 82
3.2.3 Pre-Integrated Lighting 86
3.2.4 Results 94
3.3 Compression 95
3.3.1 Cut-Border Machine 99
3.3.2 Results 107
3.4 Conclusion & Future Work 109

4 Displacement Mapping **111**
4.1 General Purpose Algorithm 112
4.2 Hardware Acceleration 113
4.2.1 Single pass prism Renderer 113
4.2.2 Tetrahedral Renderer 115
4.2.3 Mesh Construction 115
4.2.4 Rendering 116
4.2.5 Performance Optimizations 123
4.2.6 Results 124
4.3 Conclusion & Future Work 125

List of Figures

1.1 Visualization of ultrasonic data set in the 25.th week of pregnancy (left) and photograph taken 24 hours after birth (right) [SW95]. 1
1.2 Slices of a regular sampled volume. 3
1.3 Data sampled on a structured grid. 4
1.4 Data sampled on an unstructured grid. 4
1.5 Positive (a) and negative (b) absorption-only rendering of aneurism data set. 6
1.6 Emission-only rendering of aneurism data set with saturation (a) and re-normalization (b). 7
1.7 Maximum intensity projection of aneurism data set 7
1.8 Uniform (a) and non-uniform (b) emission κ and absorption τ rendering of aneurism data set. 8
1.9 Local lighting (a) and global lighting (b) of aneurism data set. 9
1.10 Approximation of an integral (grey) using Riemann-sums (dashed). . 10
1.11 Simple transfer function used for rendering the lobster data set in Figure 1.12. The two color bars represent κ (top) and τ (bottom). 13
1.12 Rendering of lobster data set without (a) and with (b) transfer function applied. 14
1.13 Engine data set rendered with normal transfer function (a) and gradient magnitude modulation (b). 15
1.14 Tooth data set (a) and head of visible human data set (b) rendered with multi-dimensional transfer function [KKH01]. 16
1.15 High frequency transfer function applied to Neghip data set rendered without (a) and with pre-integration (b). 17
1.16 High frequency transfer function and Phong lighting applied to Neghip data set rendered without (a) and with pre-integrated lighting (b) compared against ray-casting using 5 times more samples (c) (negative difference image (d) 8 times amplified). 19

2.1 Slices for shear-warp (a) and 3D texture mapping based algorithm (b). 24
2.2 Simple shading using two-dimensional textures and object aligned slices without (a) or with lighting (b). 25
2.3 Rendering pipeline for simple shading with (red, green and grey) and without (green and grey only) lighting. The green portion is the rendering itself. 25
2.4 Comparison of simple shading (a) and pre-classified rendering with three-dimensional textures (b). 27

2.5 Rendering pipeline for pre-classification rendering with lighting. The green portion is the rendering itself. 27
2.6 Comparison of pre-classified (a) and slice based post-classified (b) rendering on the graphics card. 28
2.7 Rendering pipeline for post-classification rendering with lighting. The green portion is the rendering itself. 29
2.8 Comparison of simple post-classified (a) and slab based pre-integrated rendering (b). 31
2.9 Rendering pipeline for pre-integrated rendering with lighting. The green portion is the rendering itself. 32
2.10 Comparison of pre-integrated (a) with material pre-integrated rendering (b) for higher quality lighting of the volume. 37
2.11 Rendering pipeline for material pre-integrated rendering with lighting. The green portion is the rendering itself. 38
2.12 Comparison of pre-integrated rendering (a) with pre-integrated ray-casting (b) for highest quality images. 41
2.13 Rendering pipeline for ray-casting with pre-integration and lighting. The green portion is the rendering itself. 42
2.14 Previous image and current image (upper row). Differential image, standard motion compensation and windowed motion compensation (lower row). 53
2.15 Scaling function used for the cosine windowed motion compensation with neighboring windows. 54
2.16 Reconstruction and storage order for different types of volumes and sequences, similar to MPEG. a) motion compensated differences b) motion compensation only, c) popular MPEG order 54
2.17 Original engine data set (a) and compressed data set at ratio 200:1 using the Haar wavelet (b) and a CDF wavelet (c). 56
2.18 Original lobster data set (a) and compressed data set at ratio 100:1 using the CDF wavelet (b). 56
2.19 Average PSNR of each different kind of volume against bits per voxel (CDF wavelet with support 2/6 and popular MPEG like sequence). . . 57
2.20 PSNR of each kind of volume of the first 500 volumes of a 2000 volume animation (CDF wavelet with support 2/6, maximum quality and popular MPEG like sequence). 58
2.21 Original volume 1000 of an animation of 2000 volumes (a). I-only sequence, quality 63, compression ratio 31:1, I-Volume (b); P-only sequence, quality 63, compression ratio 50:1, P-Volume (c); MPEG sequence, quality 63, compression ratio 110:1, B-Volume(d) of the same animation 58
2.22 Representation of a large volume data set as a wavelet tree. 59
2.23 Analysis of the projective classification strategy. 60
2.24 Depth complexity for a given image (a) without optimizations (b), with block based empty space leaping (c), alpha test (d) and occlusion culling (e) with corresponding occlusion map (f). About 30% of all sample points have been removed with each optimization. 64
2.25 Copying data from neighbors for the 3D-texture blocks. 65
2.26 Texture interpolation, the blocks overlap each other by half a voxel. . 66
2.27 PSNR for Christmas tree data set, and the visible human data set. . . . 68
2.28 SNR for Christmas tree data set, and the visible human data set. . . . 68

2.29 Quality comparison for different compression ratios for the Christmas Tree data set. 69
2.30 Christmas Tree data set (a) and visible human male data set (b) rendered at interactive frame rates. 69
2.31 Visible human female data set (a) and visible human female RGB data set (b) rendered at interactive frame rates. 70

3.1 Classification of non-degenerated projected tetrahedra (top row) and the corresponding decomposition (bottom row) according to [ST90]. . 74
3.2 Artifacts produced by incorrect visibility sorting using the MPVO algorithm. Correct image (a) and difference image (b). 76
3.3 Determining the exterior volume (depicted in light green). **Left**: minimal volume using convex hull. **Right**: easy setup with bounding box. 77
3.4 Selection of cutting plane. **Left**: bad BSP strategy (no concavity cut away). **Center**: elimination of one concavity. **Right**: elimination of two concavities (but total number of auxiliary cells is the same, since the bottom sub-volume is disconnected and needs to be split by another cut). 79
3.5 2D convexification example with resulting sorting graph. The light red balls depict cells with multiple dependencies for a face. 80
3.6 To convexify the BluntFin data set (a) only one auxiliary cell needs to be added (b). 81
3.7 a) A cycle generated by three cuts ($A > B > C > D > A$ with $>$ meaning ”in front of”). b) A different cut selection resolves the cycle ($A > B > C < D > A$). 81
3.8 Simple shading using no textures and per vertex coloring. 82
3.9 Rendering pipeline for simple shading. The green portion is the rendering itself. 82
3.10 Pre-integrated rendering of the Blunt Fin (a) and Bucky Ball (b) data sets. 83
3.11 Rendering pipeline for pre-integrated rendering. The green portion is the rendering itself. 84
3.12 Blunt Fin (a, b) and Bucky Ball (c, d) data set using per-ray lighting. With ambient only (a, c) and diffuse lighting (b, d). 87
3.13 Specular highlight on different iso-surfaces within a single tetrahedron (a), approximation for non-iso-surface setting (b) and correct solution (c). 88
3.14 Specular highlight on different iso-surfaces for a data set containing 60 tetrahedra (a) and highlights for non-iso-surface setting (b). 88
3.15 Rendering pipeline for pre-integrated rendering. The green portion is the rendering itself. 89
3.16 Convexification of two interleaving gears. The top row (a, b) shows the data set from the side and from above. For the final convexified mesh (f) a total of 42 cuts are necessary for a data set consisting of 202 tetrahedra. 96
3.17 tetrahedral meshes with a) minimum and b) maximum vertex tetrahedron-order $\frac{4t}{v}$. 97
3.18 The different manifold cut-border situations. The gate is shown as green triangle and the newly added tetrahedron with green edges and blue transparent faces. 101

3.19 The different types of non manifold cut-border situations. 102
3.20 Vertex enumeration. 103
3.21 Edge-adjacency of cut-border triangles around non manifold edge. . . 105
3.22 The measured tetrahedral meshes Random (a), Proto (b), Bubble (c), Torso I (d), Torso II (e) and Blunt Fin (f). The transparent meshes were rendered with projected tetrahedra. To the tetrahedra of the ”Torso I”-mesh a material identifier is attached. The ”Blunt Fin”-mesh was rendered with false colors. 108

4.1 The prism with its resulting triangles used for rendering. 112
4.2 The vectors used to define the second local coordinate system for simpler calculation of the ray exit point. 114
4.3 Sampling within the extruded prism with a slice of the displacement map shown. 115
4.4 Subdivision of prism into three tetrahedra (v0-v1-v2-v5, v0-v1-v4-v5, v0-v3-v4-v5). 116
4.5 One possible decomposition of tetrahedron into triangles. Intersection of front and back edge at point S in the viewplane. 117
4.6 Optimized base mesh. 123
4.7 Flat base mesh with a half donut shape applied to it (a). The base mesh in red is translated away for better visibility. Same shape with a different texture applied (b). 124
4.8 Displacement Map of Crater Lake applied to a flat base mesh (a). The head of Volker Blanz displaced from a cylindrical mesh, tetrahedral mesh show (b). 124
4.9 Sphere shaped base mesh with a earth displacement map and texture applied to it. Additionally the wire-frame of the tetrahedral mesh is shown (a). Different angle, this time showing Europe with slightly exaggerated displacements (b). 125

List of Tables

1.1 Comparison of memory footprints in kB for 128^3 voxel volume and different representations. 5

2.1 Comparison of memory footprints in kB for 128^3 voxel volume for rendering with simple shading. 26
2.2 Memory footprints in kB for 128^3 voxel volume for rendering with pre-classification. 28
2.3 Memory footprints in kB for 128^3 voxel volume for rendering with post-classification. 29
2.4 Memory footprints in kB for 128^3 voxel volume for rendering with pre-integration. 32
2.5 Memory footprints in kB for 128^3 voxel volume for rendering with material pre-integration. 38
2.6 Memory footprints in kB for 128^3 voxel volume for rendering with ray casting onto a 512^2 screen. 42
2.7 Performance for rendering a 64^3 data set (neghip64) onto a 512^2 screen with all possible rendering algorithms. [1] Some of the possibilities were not implemented because DirectX 9 hardware supports at least post-classified rendering. [2] Needed hardware feature not supported under DirectX 9. [3] Ray-casting was not implemented under OpenGL. 50
2.8 Performance for rendering a 128^3 data set (neghip128) onto a 512^2 screen with all possible rendering algorithms. [1] Some of the possibilities were not implemented because DirectX 9 hardware supports at least post-classified rendering. [2] Needed hardware feature not supported under DirectX 9. [3] Ray-casting was not implemented under OpenGL. 50
2.9 PSNR comparison between different wavelets using maximum quality, the popular MPEG like sequence and arithmetic compression. 55
2.10 Comparison regarding compression ratio and frames per second (using 2d textures on system 1 and 2) between different encoders using various qualities, the popular MPEG like sequence and the CFD wavelet with support 2/6. 57
2.11 Comparison between different sequences using CDF wavelet with support 2/6, arithmetic encoding and maximum (intermediate) quality. . . 57

3.1 Memory footprints in kB for a volume consisting of $2,097,152$ vertices (original and intermediate vertices) during rendering with simple shading. 83

3.2 Memory footprints in kB for a volume consisting of $2,097,152$ vertices (original and intermediate vertices) during rendering with simple shading. 84
3.3 Memory footprints in kB for a volume consisting of $2,097,152$ vertices (original and intermediate vertices) during rendering with simple shading. 88
3.4 Sorting times for the SPX data set with 12936 tetrahedra (Coolant simulation in a part of the Super Phoenix Reactor). The first line shows the results of using the plain MPVO algorithm which is fastest but does not render the SPX correctly. With an increasing number of tested cutting planes the total number of auxiliary cells decreases significantly and so does sorting time. 94
3.5 Basic quantities of the measured meshes. 109
3.6 Total number of encoded operations; relative frequencies of cut-border operations; relative frequency of non-manifold situations. 109
3.7 Cut-border machine: consumed storage for connectivity, border and quantized vertex coordinates. Running time for connectivity alone and together with vertex coordinates in tetrahedra per second on a Pentium II 350MHz. 109
3.8 Comparison of the different approaches. 110

Chapter 1

Introduction

Volume visualization in general is the art to visualize a function given in three-dimensional space in a single image. But a single image is not always sufficient. Most of the time interactive exploration of the volume or even play back time dependent data is desired. In order to clearly see the requirements for the algorithms to be implemented, the different settings and application areas to be encounter need closer examination.

Visualization of volumetric data is a very common task in many areas in medicine (see Figure 1.1), computational physics and various other disciplines. Besides being acquired, analyzed and stored for further processing, these data sets also require an adequate visualization. Since the acquiring and analysis of the data will not be investigated in this thesis, only storing and visualization will be discussed.

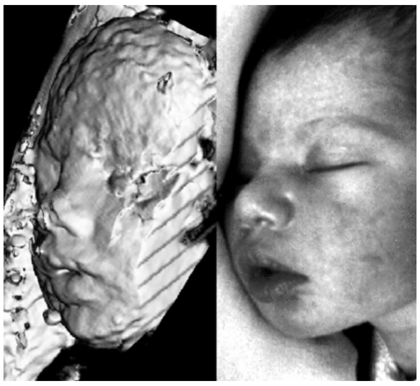


Figure 1.1: Visualization of ultrasonic data set in the 25.th week of pregnancy (left) and photograph taken 24 hours after birth (right) [SW95].

An important technique in interactively exploring these data sets is direct volume rendering. To achieve interactive frame rates, general purpose graphics hardware is used nowadays. This approach has two advantages. First, the central processing unit (CPU) is free for other processing, such as on the fly decompression. Second, the graphics processing unit (GPU) is a lot faster by now when it comes to volume rendering.

To store volume data on disc usually either the raw data or a simply compressed data (using the gzip [Deu96] library zlib) is used. In order to handle extremely large data sets a different compression approach has to be used [NH92, NH93, Wes94, CHF96, GLDH97, IP98, KS99, Rod99, GGS99, BIKP99, GS01, NS01, GWGS02, GS04]. However, a good compression scheme heavily depends on the structure of the data to be compressed.

1.1 Volume Representation

Depending on the source and application area of the volume data many different representations are possible. However, all these representations have one thing in common; they all represent a continuous function in space. Therefore not only the sample positions, i.e. the locations in space that have actual data attached to them, but also the space in between has to be represented correctly. Therefore the structure of the volume between these points, i.e. the grid, is very important.

1.1.1 Cartesian Grids

The simplest distribution of sample points is the cartesian grid. All sample points are evenly spaced along the three axes. The connectivity of the grid is regular, i.e. it only consists of cubes. Therefore only the number of sample points in each direction and the distance between the sample points has to be stored. This results in the minimal memory consumption, where only the data values at the sample point locations need to be stored. This representation can be seen as a three-dimensional image. These sample points are then called voxel (short for volume element), similar to pixel (picture element). The total amount of memory needed for this representation is n^3 for a volume with n voxel in each direction. Yet, this very simple volume representation is rarely used because the resolution is fixed for the whole volume and the same for all three axes, thus wasting a lot of memory in more or less constant regions.

During visualization only the voxel values are of interest and the volume data is treated as a three-dimensional array of sample points. Therefore the rendering can be done with a simple ray-casting approach [KH84, MJC02] or with a slicing approach [FGR85, Sab88] that can also be run on the GPU [Ake93, RSEB$^+$00]. The resulting frame rates are interactive to real-time[1] on todays graphics hardware [RSEB$^+$00, EKE01, MGS02] for volume data sets of a size of up to 256^3 voxel.

1.1.2 Regular Grids

In contrast to cartesian grids, regular grids don't have the same sample distance for all three axes. However, the sampling distance is still fixed for each axis. This type of grid is usually encountered in medical data sets (see Figure 1.2). If a medical data set is created using computed tomography (CT) or magnet resonance (MR), the distance along the w-axis is usually longer than along the u- and v-axis. This is due to the slice based construction of these apertures.

Since all voxel have the same extend, this representation is also as easy to handle as the cartesian grid, even with the different scaling for all the axis. Even if a rendering approach needs some modifications, the algorithm itself remains untouched. The amount

[1]Interactive frame rates are usually defined as 1 to 10 frames per second, whereas real-time is defined as more than 10 frames per second.

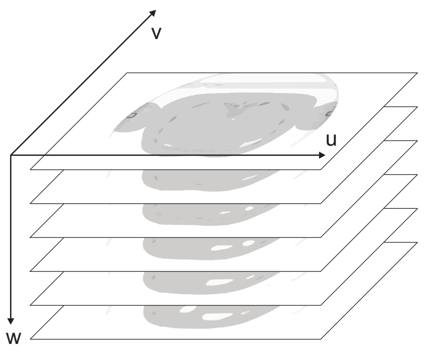


Figure 1.2: Slices of a regular sampled volume.

of memory needed is still n^3 for a volume with n voxel in each direction. Therefore, especially larger data sets, such as the Visible Human [Nat86] or the Christmas Tree [KTM+02], are using this kind of representation. Again the resulting frame rates are interactive to real-time on todays graphics hardware, even with the slight modifications to the rendering algorithm. Ray-casting on the other hand needs no modifications to handle regular grids.

1.1.3 Rectangular Grids

The difference between a regular and a rectangular grid is that the size of the voxel varies within the data set. Yet, each voxel still has a rectangular shape. Therefore the only thing that varies is the distance between the sample points along the u-, v- and w-axis. One can easily see that each slice has to be treated differently with this setting. Since the adaptation to local details in a simulation environment is very poor, this representation is rarely encountered and can always be treated as a structured grid instead. The amount of memory needed for this representation is now n^3 for the voxel data and $3 \times (n-1)$ for the spacing between the slices in each direction for a volume with n voxel in each direction.

As for the rendering, a software based ray-casting approach and a modified shear-warp algorithm [LL94] that may also run on the GPU, are the only two correct ways for rendering this representation directly.

1.1.4 Structured Grids

Structured grids are usually encountered in physical simulations where both, a regular connectivity and adaptation to local details, are necessary. While physical simulations mostly use curvilinear grids (see Figure 1.3), i.e. regular grids that have been deformed with a continuous function, structured grids can always be described as regular grids with arbitrary sample point positions.

In order to render these grids using graphics hardware, the grid cells have to be traversed in either a front-to-back or back-to-front order and will be processed individually. Since the structure is regular and the data set only consists of hexahedral cells, the sorting can be done very efficiently. The rendering of a single cell can either be done directly or by splitting each hexahedron into 5 or 6 tetrahedra.

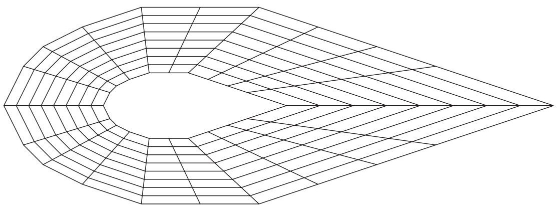


Figure 1.3: Data sampled on a structured grid.

These tetrahedra can then be rendered using any projection approaches running on the GPU [ST90, WMS98, Wit99, GRS$^+$02] or in software [FMS00].

For a structured grid the memory footprint contains n^3 voxel and n^3 three-dimensional coordinates. If 8 bits are used for the voxel data and floating point numbers (32 bits) for the coordinates, the voxel data is only $\frac{1}{13}$ of all the data.

1.1.5 Unstructured Grids

Unstructured grids (see Figure 1.4) do not only consist of arbitrary connectivity but also of arbitrary cell types. In order to render these grids they are usually decomposed into tetrahedral meshes. Unstructured grids are used in physical simulations whenever structured grids do not adapt enough to local details within a certain limit of sample points.

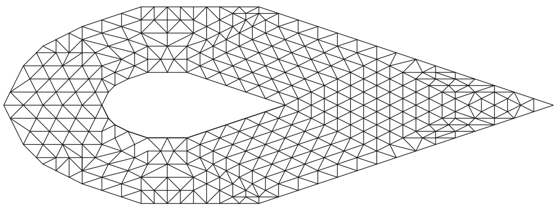


Figure 1.4: Data sampled on an unstructured grid.

In contrast to rendering structured grids, the sorting becomes much more of an issue [KE01, RSG$^+$04]. The rendering of individual cells on the other hand is the same as for structured grids, since the rendering algorithms were not created using the structure of the volume in the first place. Although order-independent rendering schemes, such as ray-casting, exist, they can hardly be mapped to graphics hardware if one wants to achieve a decent frame rate. For an unstructured grid the memory footprint contains n^3 voxel, n^3 three-dimensional coordinates and up to $n^3 - 2$ tetrahedra with 4 indices to the corresponding sample points. If 8 bits are used for the voxel data, floating point numbers (32 bits) for the coordinates and 32 bit integers for the indices, the voxel data is only about $\frac{1}{29}$ of all the data. A comparison between the memory footprints[2] for a volume containing 128^3 voxel is given in Table 1.1.

[2]The memory footprint is maximum amount of memory used by the rendering algorithm.

representation	voxel data	coordinate data	connectivity data
cartesian grid	2,048 kB	< 0.1 kB	0 kB
regular grid	2,048 kB	< 0.1 kB	0 kB
rectangular grid	2,048 kB	≈ 1.5 kB	0 kB
structured grid	2,048 kB	24,576 kB	0 kB
unstructured grid	2,048 kB	24,576 kB	8,192 kB - 32,768 kB

Table 1.1: Comparison of memory footprints in kB for 128^3 voxel volume and different representations.

1.2 Optical Model

After being able to represent a continuous function in space, the light interactions with the volume have to be defined. The interaction between light and volume is called optical model [Max95]. This model consists of a combination of emission, absorption and scattering or even more general light transportation equations. In order to set up these models, the volume is seen as a cloud of particles where the scalar voxel value represents the particle density ρ.

1.2.1 Absorption

The most simple particle medium is a black cloud, thus the optical model is a pure absorption model. For simplicity the particles are assumed to be identical spheres of radius r and projected area $A = \pi r^2$. With ρ being the number of particles per unit volume a small cylindrical slab with base B of area E and thickness Δs is considered. The slab has volume $E\Delta s$ and contains $N = \rho E\Delta s$ particles. With the light flowing perpendicular to Δs and Δs being small enough, the probability of particles to overlap is low, so the total area they occlude can be approximated by $NA = \rho AE\Delta s$. Thus the fraction of light being occluded is $\frac{\rho AE\Delta s}{E} = \rho A\Delta s$. In the limit case where Δs approaches zero and the probability of overlap also approaches zero, the differential equation is

$$\frac{dI}{ds} = -\rho(s)\, AI(s) = -\tau(s)\, I(s) \tag{1.1}$$

where s is the length parameter along the ray in the direction of the light flow and $I(s)$ is the light intensity at distance s. The quantity $\tau(s) = \rho(s)\, A$ is called extinction coefficient and defines the rate at which light is occluded. The solution to the differential equation is

$$I(s) = I_0 \exp\left(-\int_0^s \tau(t)\,\mathrm{d}t\right) \tag{1.2}$$

where I_0 is the intensity at $s = 0$ where the ray enters the volume. The transparency between 0 and s is defined as

$$T(s) = \exp\left(-\int_0^s \tau(t)\,\mathrm{d}t\right) \tag{1.3}$$

while the opacity of a voxel viewed parallel to one edge can be written as

$$\alpha = 1 - T(l) = 1 - \exp\left(-\int_0^l \tau(t)\,\mathrm{d}t\right). \tag{1.4}$$

Rendering with this model produces images similar to x-ray photos. However the x-ray itself is the negative of this image (see Figure 1.5). The main drawback of this approach is that it is order independent, i.e. there is no visual cue for the depth where a certain detail is to be found within the data set. Also the values along the viewing ray all approach zero very fast.

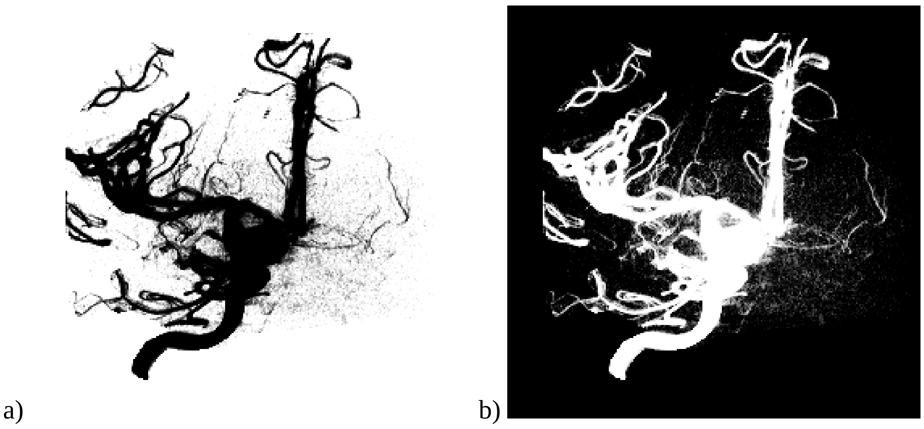


Figure 1.5: Positive (a) and negative (b) absorption-only rendering of aneurism data set.

1.2.2 Emission

In addition to extinction, the medium may also add light to the ray by emission or reflection. If the particles uniformly glow with the intensity κ per unit projected area, the projected area $\rho AE\Delta s$ will contribute a glow flux $\kappa\rho AE\Delta s$ to the base area E. Therefore the added flux per unit is $\kappa\rho A\Delta s$. Thus the equation for $I\left(s\right)$ is

$$\frac{dI}{ds} = \kappa\left(s\right)\rho\left(s\right)A = \kappa\left(s\right)\tau\left(s\right) = g\left(s\right). \tag{1.5}$$

The term $g\left(s\right)$ is called source term and includes all light that is emitted. The solution to this differential equation is

$$I\left(s\right) = I_0 + \int_0^s g\left(t\right)\mathrm{d}t. \tag{1.6}$$

As can clearly be seen in Equation 1.6, this model is also order independent. However, there are a couple of shortcomings in this model. Again the order independence turns out to be a major drawback instead of an advantage. The second problem is that if all values along a certain viewing ray are added up an overflow might occur, thus parts of the volume will saturate to white in the final image, even if the original values were different. This can be circumvented by re-normalizing the final pixel values to the valid range after summing up (see Figure 1.6). The images produced with this model will look like some kind of neon light since this kind of volume also only consists of emitting particles.

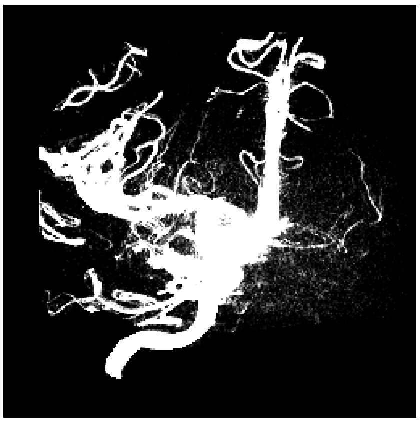
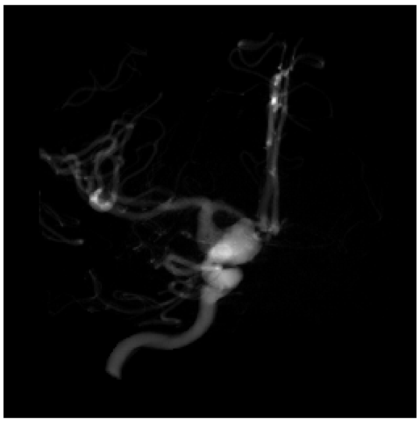
a)  b) 

Figure 1.6: Emission-only rendering of aneurism data set with saturation (a) and re-normalization (b).

Instead of adding up the values along the viewing ray, the maximum value can be used instead and the intensity will be

$$I\left(s\right) = I_0 + \sup_{t=0..s} \left(g\left(t\right)\right). \tag{1.7}$$

Rendering with this model is then called MIP (maximum intensity projection). This approach is mainly used for CT angiography (see Figure 1.7) where vessels have been enhanced using a contrast agent. The idea is that even small vessels will have high values but these values might be lost along the viewing ray if there are a lot of small values in front of them. While this behavior is expected by the viewer most of the time, these small vessels are very important for medical diagnosis and need to be visible for this kind of application.

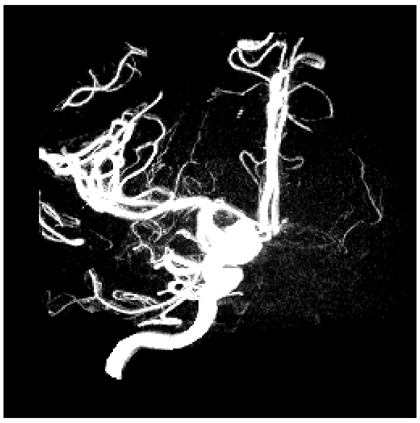


Figure 1.7: Maximum intensity projection of aneurism data set

1.2.3 Emission & Absorption

In a more realistic setting, both emission and absorption are present. So the differential equation should look like this:

$$\frac{dI}{ds} = g(s) - \tau(s) I(s). \tag{1.8}$$

According to Max [Max95] this equation can be solved as

$$I(u) = I_0 \exp\left(-\int_0^u \tau(t)\,\mathrm{d}t\right) + \int_0^u g(s) \exp\left(-\int_s^u \tau(t)\,\mathrm{d}t\right) \mathrm{d}s. \tag{1.9}$$

For a uniform emission, i.e. $g(s) = \kappa\tau(s)$, the differential intensity is

$$\frac{dI}{ds} = \kappa\tau(s) - \tau(s) I(s) = \tau(s)(\kappa - I(s)). \tag{1.10}$$

The solution to this is

$$I(s) = \kappa - (\kappa - I_0) \exp\left(-\int_0^s \tau(t)\,\mathrm{d}t\right). \tag{1.11}$$

Emission and absorption together not only solve the overflow problem of the emission-only setting, but also provide a bit more flexibility. However, a uniform emission results in exactly the same image as for the negative absorption-only setting (compare Equation 1.2 and Equation 1.11). So if a uniform emission was used, the same drawbacks as for the absorption-only approach will be present.

Instead of using the same value for emission and absorption, i.e. uniformly emitting particles, the emission can for example be scaled with the particle density. The resulting images in Figure 1.8 show that, with this scaling, the rendering is no longer order independent. This enables the algorithm to see which parts of the volume are facing to the viewer and which parts are backward facing. It also enables the user to better see the structure of the volume itself.

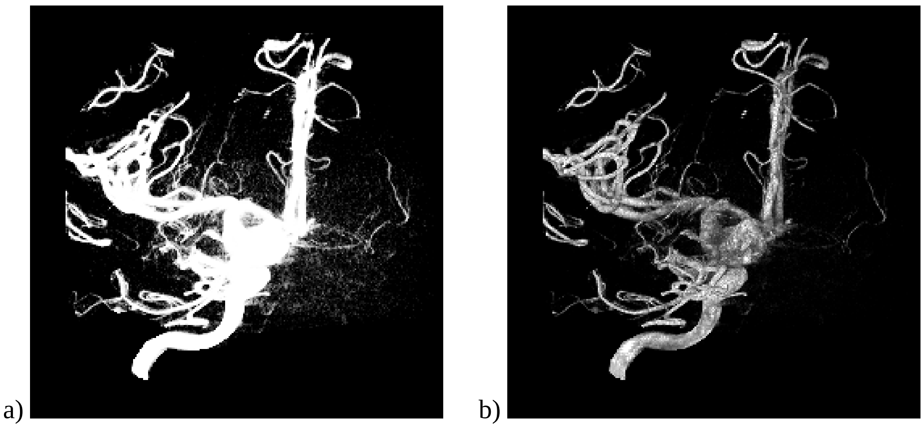


Figure 1.8: Uniform (a) and non-uniform (b) emission κ and absorption τ rendering of aneurism data set.

1.2.4 Lighting, Scattering & General Light Transportation

So far only self emitting particles were discussed. In order to see more small details within a data set, the way how light interacts with particles has to be defined. This includes shading and light transportation. Even if shading was originally supposed to work only for geometry, all that is needed to shade a voxel is a normal. Taking a look at surface extraction algorithms [LC87, Lev88, SW95], it can easily be seen that the normal of the extracted iso-surface is pointing in the opposite direction as the gradient of the volume data. For $\overrightarrow{n}(s)$ being the gradient at position s and $\overrightarrow{l}$ the direction to the light source the following equation for $g(s)$ can be found:

$$g(s) = \kappa(s)\,\tau(s)\,\overrightarrow{n}(s) \cdot \overrightarrow{l}. \tag{1.12}$$

Shading the volume with these gradients and a local lighting model already unveils most of the finer structures contained within the volume (see Figure 1.9a).

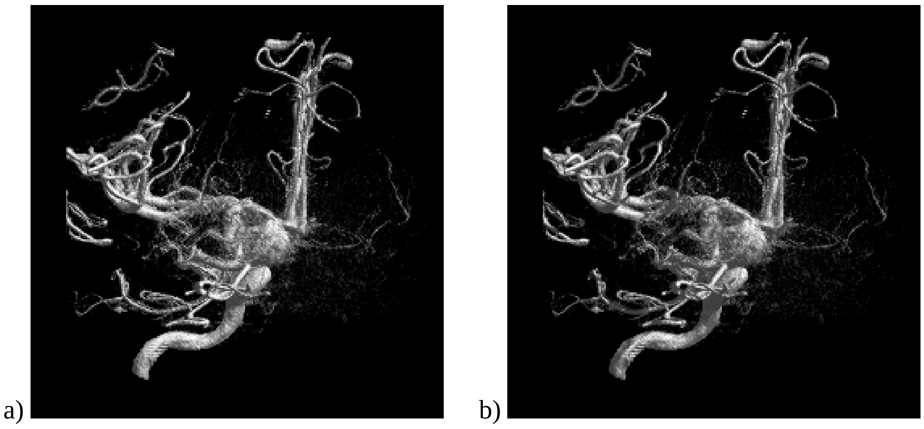


Figure 1.9: Local lighting (a) and global lighting (b) of aneurism data set.

For additional depth cues the lighting model can be changed to a global model, e.g. one that traces back to the light source in order to produce shadows (see Figure 1.9b). With l being the position of a directional light source, the resulting equation then looks like this:

$$g(s) = \kappa(s)\,\tau(s)\,\overrightarrow{n}(s) \cdot \overrightarrow{l} \exp\left(-\int_s^l \tau(t)\,\mathrm{d}t\right). \tag{1.13}$$

For a point light source, an attenuation that increases with the distance to the light source is also encountered. Thus g can be written as:

$$g(s) = \kappa(s)\,\tau(s)\,\overrightarrow{n}(s) \cdot \overrightarrow{l}\,\frac{\exp\left(-\int_s^l \tau(t)\,\mathrm{d}t\right)}{|s-l|}. \tag{1.14}$$

Introducing higher order lighting terms, i.e. scattering, increases the realism of the resulting images but also dramatically increases the rendering costs. Besides the increased realism there are very few additional details to be discovered so that most rendering schemes will only use a local light source for shading the voxel.

Max [Max95] also discusses a more general approach for scattering, but even the simple shadow calculation is too slow for interactive rendering. Therefore only a local lighting model will be used throughout this thesis.

1.2.5 Numeric Integration

Since an analytical evaluation of the volume rendering integrals is scarcely possible, a numeric integration needs to be carried out. The most common numeric integration used for volume rendering is Riemann-sums (see Figure 1.10) for n ray segments of equal length. This is the same numeric integration as used by Max [Max95]. However, the resulting equations can easily be adapted to segments of varying length as needed for adaptive sample of the volume data along each ray (see Section 2.1.6).

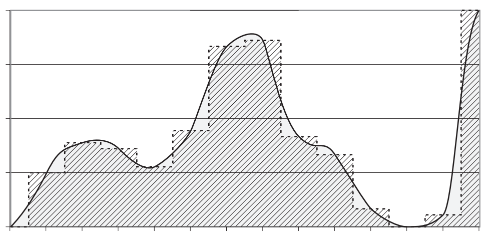


Figure 1.10: Approximation of an integral (grey) using Riemann-sums (dashed).

The absorption-only volume rendering integral in Equation 1.2 can be approximated by a Riemann-sum sampling at a distance $d = s/N$:

$$\begin{aligned} I(s) &= I_0 \exp\left(-\int_0^s \tau(t)\,\mathrm{d}t\right) \\ &\approx I_0 \exp\left(-\sum_{i=1}^{s/d} d\tau(id)\right) \\ &= I_0 \prod_{i=1}^{s/d} \exp(-d\tau(id)) \end{aligned} \tag{1.15}$$

In contrast to the integral, the approximation can be computed very efficiently. Even if the notation using a sum may be faster to compute, the approximation using a product is much more accurate and should therefore be used.

The volume rendering integral of emission-only rendering (Equation 1.6) is even easier to approximate since it contains only a single integral and no exponential function. Again a sampling distance of $d = s/N$ is used.

$$\begin{aligned} I(s) &= I_0 + \int_0^s g(t)\,\mathrm{d}t \\ &\approx I_0 + \sum_{i=1}^{s/d} dg(id) \\ &= I_0 + d\sum_{i=1}^{s/d} g(id) \end{aligned} \tag{1.16}$$

The rendering now only sums up all densities along the viewing ray, multiplies this sum with the sampling distance and adds it to the background intensity.

The general emission and absorption model using $\tau(s)$ and $g(s)$ as stated in Equation 1.9 is a bit more complicated to approximate. In order to achieve a good approximation, the self-occlusion within each sampling interval of size d has to be accounted for. The emission color g and opacity α are defined as $g(s) = C(s)\,\tau(s)$ and $\alpha(s) = 1 - \exp(-d\tau(s))$. With $C(s)$ being the emissive term. At a sampling distance $d = u/N$ the intensity is:

$$\begin{aligned}
I(u) &= I_0 \exp\left(-\int_0^u \tau(t)\,\mathrm{d}t\right) + \int_0^u g(s)\exp\left(-\int_s^u \tau(t)\,\mathrm{d}t\right)\mathrm{d}s \qquad (1.17)\\
&\approx I_0 \exp\left(-\sum_{i=1}^{u/d} d\tau(id)\right) + \int_0^u g(s)\exp\left(-\sum_{i=s/d+1}^{u/d} d\tau(id)\right)\mathrm{d}s\\
&= I_0 \prod_{i=1}^{u/d} \exp(-d\tau(id)) + \int_0^u g(s) \prod_{i=s/d+1}^{u/d} \exp(-d\tau(id))\,\mathrm{d}s\\
&\approx I_0 \prod_{i=1}^{u/d} \exp(-d\tau(id)) + \sum_{j=1}^{u/d} \alpha(jd)\,C(jd) \prod_{i=j+1}^{u/d} \exp(-d\tau(id)) .
\end{aligned}$$

As can easily be seen, Equation 1.17 is still very costly to evaluate since a sum needs to be calculated for every sample point. However, if this equation is written with ΔT_i as the transparency of the current sample, the following recursive definition can easily be found.

$$\begin{aligned}
\Delta T_n &= \exp(-d\tau(nd)) \qquad (1.18)\\
&= 1 - \alpha(dn)\\
I(0) &= I_0\\
I(u) &= I_N\\
I_n &= I_0 \prod_{i=1}^{n} \exp(-\tau(id)\,d) + \sum_{j=1}^{n} \alpha(jd)\,C(jd) \prod_{i=j+1}^{n} \exp(-d\tau(id))\\
&= I_0 \prod_{i=1}^{n} \Delta T_i + \sum_{j=1}^{n} \alpha(jd)\,C(jd) \prod_{i=j+1}^{n} \Delta T_i\\
&= I_0 \prod_{i=1}^{n} \Delta T_i + \sum_{j=1}^{n-1} \alpha(jd)\,C(jd) \prod_{i=j+1}^{n} \Delta T_i + d\alpha(jd)\,C(jd)\\
&= \left(I_0 \prod_{i=1}^{n-1} \Delta T_i + \sum_{j=1}^{n-1} \alpha(jd)\,C(jd) \prod_{i=j+1}^{n-1} \Delta T_i\right)\Delta T_n + d\alpha(nd)\,C(nd)\\
&= I_{n-1}\Delta T_n + \alpha(nd)\,C(nd)\\
&= (1 - \alpha(nd))\,I_{n-1} + \alpha(nd)\,C(nd)
\end{aligned}$$

In order to calculate $I(u) = I_N$ there are two possible approaches. Starting with I_0, I_n can be calculated using $I_n - 1$. This results in the so called back-to-front rendering order since the evaluation starts with the background color and ends the calculation at the closest sampling point. The opacity α of each sample point at distance d is then defined as $\alpha = 1-t = 1-\exp(-\tau d)$. According to Wilhelms and van Gelder [WG91] this can be approximated for small d by $\alpha \approx \min(1, td)$. Additionally Max [Max86] also proposes to use a quadratic approximation for α that meets $\alpha = 1$ smoothly for $d = 2/\tau$.

The simplest way to calculate I_N directly is using recursion. However, taking a closer look at the total transparency T_n in front, i.e. the visibility, of the sample point n and its emission ΔI_n, I_N can be calculated as follows:

$$\begin{aligned} T_N &= 1 \\ T_n &= T_{n+1} \Delta T_{n+1} \\ \Delta I_n &= \alpha(nd)\, C(nd) \\ \Delta I_0 &= I_0 \\ I(u) &= I_0 \prod_{i=1}^{N} \exp(-\tau(id)\, d) + \sum_{j=1}^{N} \alpha(jd)\, C(jd) \prod_{i=j+1}^{N} \exp(-d\tau(id)) \\ &= \Delta I_0 T_0 + \sum_{j=1}^{N} \Delta I_j T_j \\ &= \sum_{j=0}^{N} \Delta I_j T_j \end{aligned} \tag{1.19}$$

The resulting rendering order is called front-to-back since the evaluation starts with the closest sample and ends at the background color.

For emission and absorption rendering with uniform emission (see Equation 1.11), the approximation is a lot easier again.

$$\begin{aligned} I(s) &= \kappa - (\kappa - I_0) \exp\left(-\int_0^s \tau(t)\, \mathrm{d}t\right) \\ &\approx \kappa - (\kappa - I_0) \exp\left(-\sum_{i=1}^{s/d} d\tau(id)\right) \\ &= \kappa - (\kappa - I_0) \prod_{i=1}^{s/d} \exp(-d\tau(id)) \end{aligned} \tag{1.20}$$

Again, the resulting color $I(s)$ can be calculated directly. This special case produces a negative version of an absorption-only rendering if the inverted background color $I_0^n = 1 - I_0$ was used for the absorption-only rendering and $\kappa = 1$ for the uniform emission.

The shadow calculation for a direction light source in Equation 1.13 can also be approximated using Riemann-sums.

$$\begin{aligned} C(s) &= \kappa(s)\, \overrightarrow{n}(s) \cdot \overrightarrow{l} \exp\left(-\int_s^l \tau(t)\, \mathrm{d}t\right) \\ &\approx \kappa(s)\, \overrightarrow{n}(s) \cdot \overrightarrow{l} \exp\left(-\sum_{t=s/d+1}^{l/d} d\tau(td)\right) \\ &= \kappa(s)\, \overrightarrow{n}(s) \cdot \overrightarrow{l} \prod_{t=s/d+1}^{l/d} \exp(-d\tau(td)) \end{aligned} \tag{1.21}$$

As in Equation 1.15, the version containing only a single exponential evaluation is faster but less accurate than the version using a product of exponential functions.

Similar to directional lighting, the shadow calculation for point light source (see Equation 1.14) can be approximated the same way.

$$\begin{aligned} C(s) &= \kappa(s)\,\overrightarrow{n}(s)\cdot\overrightarrow{l}\,\frac{\exp\left(-\int_s^l \tau(t)\,\mathrm{d}t\right)}{|s-l|} \\ &\approx \kappa(s)\,\overrightarrow{n}(s)\cdot\overrightarrow{l}\,\frac{\exp\left(-\sum_{t=s/d+1}^{l/d} d\tau(td)\right)}{|s-l|} \\ &= \kappa(s)\,\overrightarrow{n}(s)\cdot\overrightarrow{l}\,\frac{\prod_{t=s/d+1}^{l/d}\exp(-d\tau(td))}{|s-l|} \end{aligned} \tag{1.22}$$

Note that the result only needs to be divided by the distance to the light source once since the distance between shadow sample point and light source does not influence the amount of light that is absorbed.

1.3 Transfer Functions

So far only a representation for a continuous function in space was defined. This function only has one optical property attached to it; the optical density $\tau(s)$. Also this function only returns the density of the volume at the sampling position. This is why the images produced so far were all black and white.

For a more general rendering of the given volume data a so called transfer function is defined. This function adds optical properties to the voxel values. The optical density $\tau(s)$ does not reflect the voxel value directly but applies any possible mapping to it (see Figure 1.11). In order to better distinguish between different voxel values (see Figure 1.12) the emission color $\kappa(s)$ can also be modified with the transfer function. Since a lighting equation has already been defined for volume rendering in Section 1.2.4, different ambient $\kappa_{ambient}(s)$ and diffuse colors $\kappa_{diffuse}(s)$ for the emission color can also be defined. Using the Phong lighting model [Pho75], an additional specular color $\kappa_{specular}(s)$ and a specular power $\kappa_{power}(s)$ has to be defined within the transfer function.

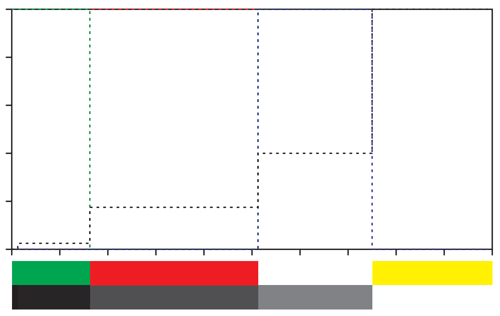


Figure 1.11: Simple transfer function used for rendering the lobster data set in Figure 1.12. The two color bars represent κ (top) and τ (bottom).

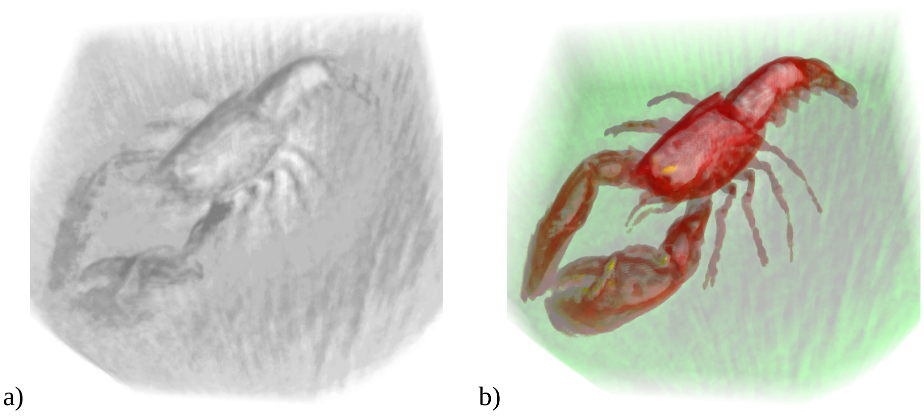


Figure 1.12: Rendering of lobster data set without (a) and with (b) transfer function applied.

With different shading models the number of output values of the transfer function may increase even more. In Section 1.3.1 and 1.3.2 the voxel value is not necessarily the only input to the transfer function but may contain additional information.

Besides the transfer function, there are two different shading paradigms. Pre-classification applies the transfer function only to the original voxel values and interpolates these lookups in order to get visual properties for each sample point. Post-classification first interpolates the sample point and then applies the transfer function to the interpolated value. While both of these rendering paradigms are being used, post-classification is usually considered to produce higher quality images but is also more computational complex as can be seen in the following sections. This is due to the fact, that the frequency spectrum of the transfer function and thus the frequency spectrum of the classified volume is unbound for post-classification rendering.

1.3.1 Gradient Magnitude Modulation

With the definition of a transfer function that only depends on the voxel value only materials that have different absorption coefficients can be distinguished. Also the problem that because of partial volume effects and interpolation there will always be a medium absorption coefficient between a low and a high absorption coefficient is encountered. Thus the rendering cannot distinguish between a material with a medium absorption coefficient and the boundary between a low and high absorption material. Therefore an additional input to the transfer function is needed that enables the rendering to separate those two cases.

For scalar volume data sets, the first derivative (the gradient) gives the direction of fastest change. This motivates its use as the ”surface normal” in shaded volume rendering as already mentioned in Section 1.2.4. The gradient magnitude itself is another fundamental local property of a scalar field, since it characterizes how fast values are changing. Using gradient magnitude as the second dimension in a transfer functions allows to vary opacity or color according to the magnitude of change within the voxel values. With this a large region of medium absorption coefficients and a partial volume effect resulting in the same coefficient can be distinguished.

In order to enhance the boundary between different sample values within the volume and to suppress homogeneous regions, the opacity can be scaled with the magnitude of the local gradient. This can simply be done by using $\alpha' = \alpha \left(\overrightarrow{n} \cdot \overrightarrow{n} \right)$ with $\overrightarrow{n}$ being the un-normalized gradient. Even if the resulting images look rather like surface rendering instead of volume rendering, the amount of detail they uncover (compare Figure 1.13a and b) is increased.

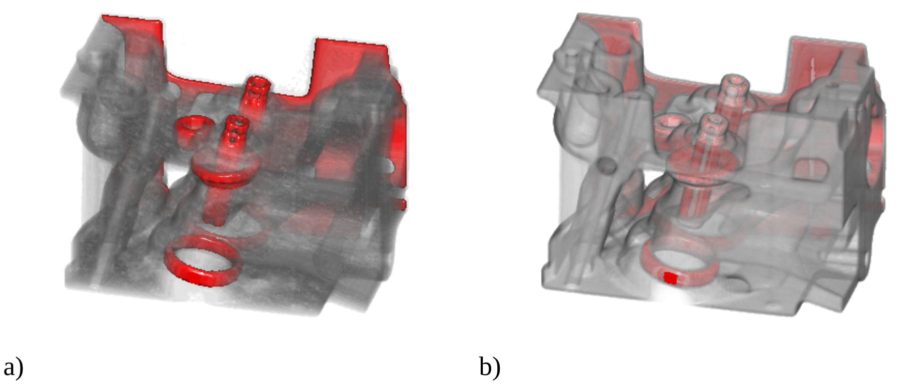


Figure 1.13: Engine data set rendered with normal transfer function (a) and gradient magnitude modulation (b).

The resulting transfer function now depends on two parameters but everything except the opacity only depends on one of these values. So this is the first simple multi-dimensional transfer function. However, this transfer function still has the problem, that it cannot differentiate between a boundary going from s_0 to s_3 and a boundary going from s_1 to s_2 with $s_0 < s_1 < s_2 < s_3$. In order to distinguish different settings within a volume even more, or to render data sets with more than one or two parameters, additional axes need to be introduced into the transfer function.

1.3.2 Multi-Dimensional Transfer Functions

A general approach for this kind of transfer functions is to define a multi-dimensional transfer function [KKH01] (see Figure 1.14). This allows for using not only the first derivative but also higher order derivatives or even multi-dimensional volumes. Transfer functions can better discriminate between various structures in the volume data when they have more variables - a larger vocabulary – with which to express the differences between them. These variables are the axes of the multi-dimensional transfer function.

Kniss et al. [KKH01] choose the third axis of the transfer function directly based on principles of edge detection. It is best suited for application areas concerned with the boundaries or interfaces between relatively homogeneous materials. Some edge detection algorithms (such as Marr-Hildreth [MH79]) locate the middle of an edge by detecting a zero-crossing in a second derivative measure, such as the Laplacian. In practice, a more accurate but computationally expensive measure, the second directional derivative along the gradient direction, which involves the Hessian, a matrix of second partial derivatives, is used. Details on these measurements can be found in

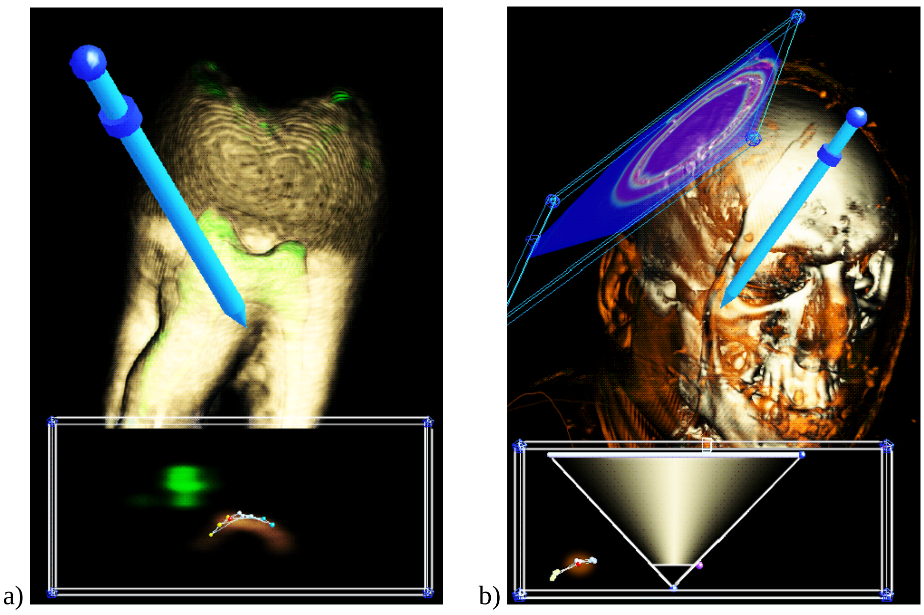


Figure 1.14: Tooth data set (a) and head of visible human data set (b) rendered with multi-dimensional transfer function [KKH01].

work on semi-automatic transfer function generation [KD98]. The usefulness of having a second derivative measure in the transfer function is that it enables more precise disambiguation of complex boundary configurations. Since 2D transfer functions are unable to accurately and selectively render the different material boundaries present, higher dimensional transfer functions are needed. A 3D transfer function can easily accomplish this.

Also multi-dimensional volumes are very interesting for medical application in order to combine CT and MR scans. Another medical application is the visualization of the visual human data set that consists of color images [Nat86]. Biological application such as confocal laser microscopy also produces multi-dimensional volume data.

1.3.3 Pre-Integration Technique

Pre-integrated classification is a technique used in volume rendering when classification is applied after interpolation [MHC90, EKE01]. Following the Nyquist theorem [Nyq28], one can generally ensure that the reconstruction of the volumetric function along the rays is accurate. However, a non continuous transfer function, e.g. binary classification with infinite frequencies, introduces well-known slicing artifacts, as shown in Figure 1.15a.

To circumvent this, pre-integration assumes a certain behavior of the volumetric function along the cast ray, i.e. linearity. Based on the conventional 1D classification table, each interval between two samples can be pre-integrated and stored in a 2D table. During rendering, two consecutive sample values are used as indices for the 2D table, instead of classifying each individual sample assuming the color to be constant for the distance to the next sample along the ray. The advantage of pre-integrated volume

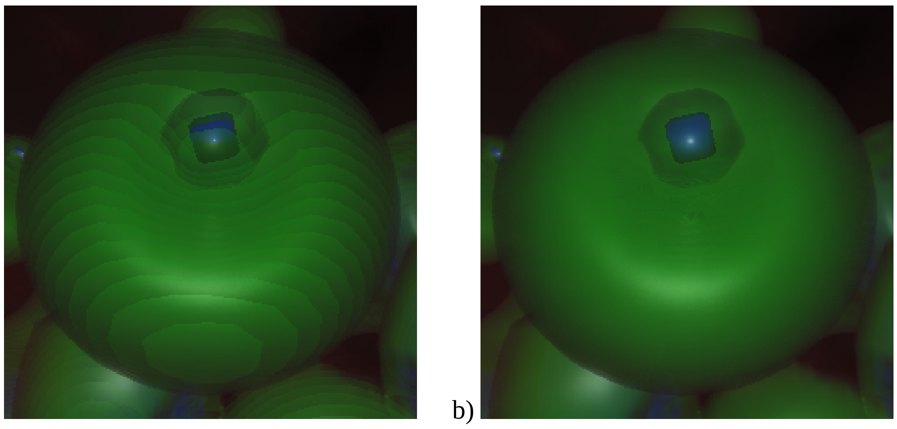
a) b)


Figure 1.15: High frequency transfer function applied to Neghip data set rendered without (a) and with pre-integration (b).

rendering is that even precise iso-surfaces can be rendered without any additional cost during ray casting, see Figure1.15b. While Max et al. [MHC90] pre-integrated opacity only, Engel et al. [EKE01] extended this to the color information $\tau(s)$ of the transfer function.

Instead of using a mapping $\tau(s)$ for each ray that takes the ray parameter and returns the optical density, $\tau(s(x(\lambda)))$ with the ray parameter λ is used. $x(\lambda)$ maps the ray parameter to the actual sampling position. $s(x)$ returns the scalar voxel value at position x. The one dimensional transfer function is now defined in $\tau(s)$ and is the same for every ray. This is a restriction to the more general notation of Section 1.2. Using a given sampling distance $d = \lambda/N$, Equation 1.2 can be written as:

$$\begin{aligned} I(\lambda) &= I_0 \exp\left(-\int_0^{\lambda} \tau(s(x(\varphi)))\,\mathrm{d}\varphi\right) \qquad (1.23) \\ &= I_0 \exp\left(-\sum_{i=0}^{\lambda/d-1} \int_{id}^{(i+1)d} \tau(s(x(\varphi)))\,\mathrm{d}\varphi\right) \end{aligned}$$

Assuming that s varies linearly between $x(id)$ and $x((i+1)d)$, $s(x(\varphi))$ can be substituted for $id \leq \varphi \leq (i+1)d$ with a linear interpolation between the first voxel value

$$s_f = s(x(id)) \qquad (1.24)$$

and the second voxel value

$$s_b = s(x((i+1)d)). \qquad (1.25)$$

With this approximation the following volume rendering equation is produced.

$$\begin{aligned} I(\lambda) &\approx I_0 \exp\left(-\sum_{i=0}^{\lambda/d-1} d \int_0^1 \tau((1-\omega)s_f + \omega s_b)\,\mathrm{d}\omega\right) \qquad (1.26) \\ &= I_0 \prod_{i=0}^{\lambda/d-1} \exp\left(-d \int_0^1 \tau((1-\omega)s_f + \omega s_b)\,\mathrm{d}\omega\right) \end{aligned}$$

This approximation is a lot closer to the actual solution of the volume rendering equation since it de-couples the frequencies of the transfer function and the volume itself. Thus the Nyquist theorem allows the rendering algorithm to sample the volume at a sampling distance half as large as the highest frequency within the volume again. Also one has to keep in mind that for a slicing the volume with axis aligned slices d is nearly constant. To get a constant d one has to use concentric shells around the viewer or a parallel projection instead of a perspective one. In order to evaluate this simplified equation, the transparency

$$\Delta T_i = \exp\left(-d\int_0^1 \tau\left(\left(1-\omega\right)s_f + \omega s_b\right)\mathrm{d}\omega\right) \tag{1.27}$$

is pre-computed for the constant d and each possible combination of s_f and s_b. For the emission and absorption model and back-to-front rendering the following approximation to the volume rendering equation is produced.

$$\begin{aligned} \Delta I_n &= d\int_0^1 g\left(\left(1-\omega\right)s_f + \omega s_b\right) \\ &\quad \exp\left(-d\int_0^\omega \tau\left(\left(1-\omega'\right)s_f + \omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega \\ I_n &= I_{n-1}\Delta T_n + \Delta I_n \\ I\left(u\right) &= I_N \end{aligned} \tag{1.28}$$

Changing the rendering order to front-to-back in order to implement ray-casting, the equation can be solved as follows.

$$\begin{aligned} T_N &= 1 \\ T_n &= T_{n+1}t_i \\ \Delta I_0 &= I_0 \\ I\left(u\right) &= \sum_{j=0}^{N}\Delta I_j T_j \end{aligned} \tag{1.29}$$

One of the main drawbacks of pre-integrated volume rendering is its incompatibility with gradient magnitude modulation. Since the color is pre-integrated based on the voxel value only, no gradient magnitude modulation is possible, unless using a 4D or even higher dimensional function which would prevent interactivity. Solving this is still topic of research.

1.3.4 Pre-Integrated Lighting

In order to implement accurate lighting, the different changing gradients in addition to the two sample values s_f and s_b for each interval between two sample points have to be handled. Meissner et al. [MGS02] extended the pre-integration approach to handle material properties and lighting since pre-integration suffers from shading artifacts when using more than one iso-surface or semi-transparent rendering in the original implementation [EKE01] (see Figure 1.16).

To handle the changing gradients, $g\left(s\right)$ is split into three parts, the ambient term $g_a\left(s\right)$ that does not depend on the gradient, the diffuse term $g_d\left(s\right)$ that roughly depends in a linear way on the dot product between gradient $\vec{n}$ and the light vector $\vec{l}$, and

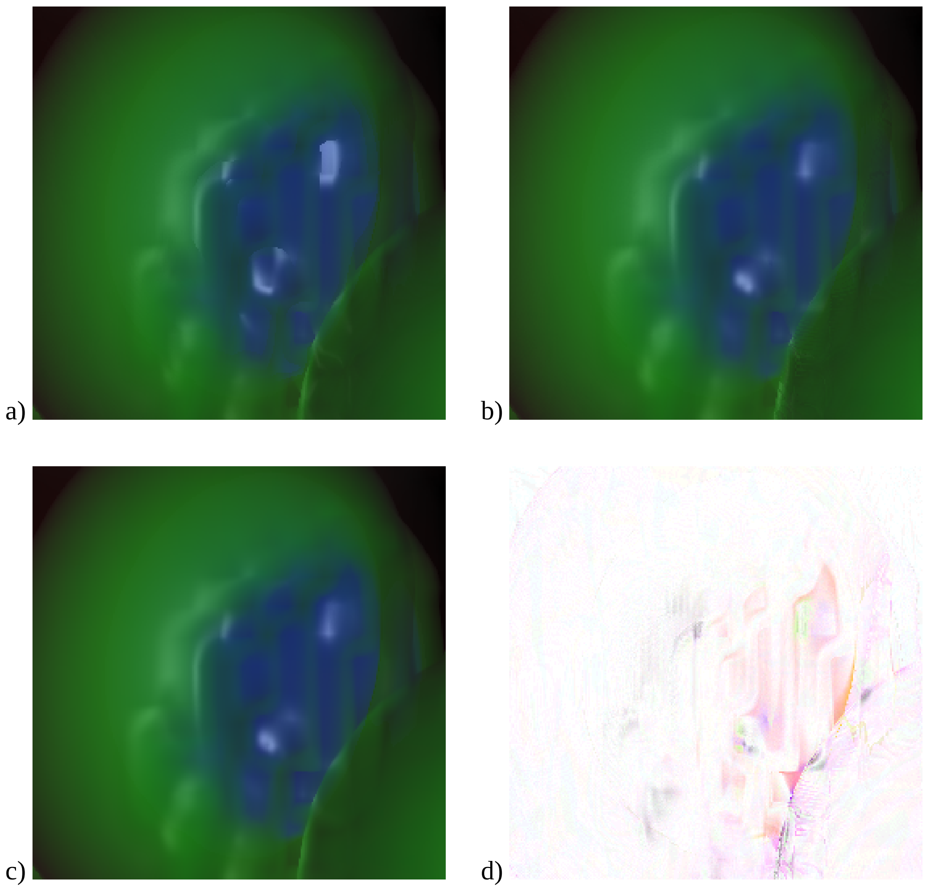


Figure 1.16: High frequency transfer function and Phong lighting applied to Neghip data set rendered without (a) and with pre-integrated lighting (b) compared against ray-casting using 5 times more samples (c) (negative difference image (d) 8 times amplified).

$g_s(s)$ that depends on the gradient in any non-linear way. It is also assumed that the gradient $\overrightarrow{n}$, just as the sample values s, only changes linearly between the two sample points. It can therefore be expressed as $\overrightarrow{n} = (1-\omega)\overrightarrow{n}_f + \omega\overrightarrow{n}_b$.

$$\begin{aligned}
g_a(s) &= \kappa_a(s)\,\tau(s) \\
g_d(s) &= \kappa_d(s)\,\tau(s) \\
g_s(s) &= \kappa_s(s)\,\tau(s) \\
l_d(\overrightarrow{n}) &= \max\left(\overrightarrow{n}\cdot\overrightarrow{l},0\right) \\
l_s(\overrightarrow{n}) &= \max\left(\overrightarrow{r}\cdot\overrightarrow{l},0\right)^{\kappa_p} \\
g(s,\overrightarrow{n}) &= g_a(s) + g_d(s)\,l_d(\overrightarrow{n}) + g_s(s)\,l_s(\overrightarrow{n})
\end{aligned} \tag{1.30}$$

The dependency between $g_d(s)$ and $\overrightarrow{n}$ can also depend on any other nearly linear mapping, i.e. fake-shading or even a very smooth cube map or light map. The only restriction is that g_d has to depend nearly linear on ω for each sample interval. The dependency between $g_s(s)$ and $\overrightarrow{n}$ can be defined in any way, including a reflection or environment map, or even a noise texture. Therefore any non-linear term should be handled in g_s while any linear term can be handled by g_d.

Then ΔI_n is split in a way that each resulting ΔI_n^x only depends on g_x, so that the total emission can be split into three parts.

$$\Delta I_n = \Delta I_n^a + \Delta I_n^d + \Delta I_n^s \tag{1.31}$$

Since the ambient emission ΔI_n^a does not depend on the gradient in any way, it can simply be pre-computed with the normal pre-integration table for this portion of the emission.

$$\begin{aligned}\Delta I_n^a &= d\int_0^1 g_a\left((1-\omega)s_f+\omega s_b\right) \\ &\quad \exp\left(-d\int_0^\omega \tau\left((1-\omega')s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega\end{aligned} \tag{1.32}$$

Assuming a linear change of the diffuse term g_d between s_f and s_b, the diffuse emission I_n^d is split into two separate terms. While I_n^{df} only depends on the first gradient $\overrightarrow{n}_f$, I_n^{nb} only depends on the second gradient $\overrightarrow{n}_b$.

$$\begin{aligned}\Delta I_n^d &= d\int_0^1 g_d\left((1-\omega)s_f+\omega s_b\right) l_d\left((1-\omega)\overrightarrow{n}_f+\omega\overrightarrow{n}_b\right) \\ &\quad \exp\left(-d\int_0^\omega \tau\left((1-\omega')s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega \\ &\approx d\int_0^1 g_d\left((1-\omega)s_f+\omega s_b\right)\left((1-\omega)l_d\left(\overrightarrow{n}_f\right)+\omega l_d\left(\overrightarrow{n}_b\right)\right) \\ &\quad \exp\left(-d\int_0^\omega \tau\left((1-\omega')s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega \\ &= dl_d\left(\overrightarrow{n}_f\right)\int_0^1 (1-\omega)g_d\left((1-\omega)s_f+\omega s_b\right) \\ &\quad \exp\left(-d\int_0^\omega \tau\left((1-\omega')s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega + \\ &\quad dl_d\left(\overrightarrow{n}_b\right)\int_0^1 \omega g_d\left((1-\omega)s_f+\omega s_b\right) \\ &\quad \exp\left(-d\int_0^\omega \tau\left((1-\omega')s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega \\ \Delta I_n^{df} &= d\int_0^1 (1-\omega)g_d\left((1-\omega)s_f+\omega s_b\right) \\ &\quad \exp\left(-d\int_0^\omega \tau\left((1-\omega')s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega \\ \Delta I_n^{db} &= d\int_0^1 \omega g_d\left((1-\omega)s_f+\omega s_b\right) \\ &\quad \exp\left(-d\int_0^\omega \tau\left((1-\omega')s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega\end{aligned} \tag{1.33}$$

In order to approximate l_s however, something else needs to be done. If no assumptions are made on the relation between l_s and $\overrightarrow{n}$, a linear combination between $l_s\left(n_f\right)$ and $l_s\left(n_b\right)$ can not be used. Instead of using an interpolation, all the normals between id and $\left(i+1\right)d$ are weighted with their maximum emission intensity to get a single gradient $\overrightarrow{n}\left(x\left(\omega^*\right)\right)$ that represents the whole interval.

$$\begin{aligned}
T\left(\omega\right) &= g_s\left(\left(1-\omega\right)s_f+\omega s_b\right) \\
&\quad \exp\left(-d\int_0^{\omega}\tau\left(\left(1-\omega'\right)s_f+\omega' s_b\right)\mathrm{d}\omega'\right) \\
\omega^* &= \frac{\int_0^1 \omega T\left(\omega\right)\mathrm{d}\omega}{\int_0^1 T\left(\omega\right)\mathrm{d}\omega} \\
\Delta I_n^s &= d\int_0^1 g_s\left(\left(1-\omega\right)s_f+\omega s_b\right) l_s\left(\left(1-\omega\right)\overrightarrow{n}_f+\omega\overrightarrow{n}_b\right) \\
&\quad \exp\left(-d\int_0^{\omega}\tau\left(\left(1-\omega'\right)s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega \\
&\approx d\int_0^1 g_s\left(\left(1-\omega\right)s_f+\omega s_b\right) l_s\left(\left(1-\omega^*\right)\overrightarrow{n}_f+\omega^*\overrightarrow{n}_b\right) \\
&\quad \exp\left(-d\int_0^{\omega}\tau\left(\left(1-\omega'\right)s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega \\
&= d l_s\left(\left(1-\omega^*\right)\overrightarrow{n}_f+\omega^*\overrightarrow{n}_b\right)\int_0^1 g_s\left(\left(1-\omega\right)s_f+\omega s_b\right) \\
&\quad \exp\left(-d\int_0^{\omega}\tau\left(\left(1-\omega'\right)s_f+\omega' s_b\right)\mathrm{d}\omega'\right)\mathrm{d}\omega
\end{aligned} \tag{1.34}$$

For the Phong lighting model used in Equation 1.30 the following resulting exponent can be calculated.

$$\kappa_p^* = \frac{\int_0^1 \kappa_p\left(\left(1-\omega\right)s_f+\omega s_b\right)T\left(\omega\right)\mathrm{d}\omega}{\int_0^1 T\left(\omega\right)\mathrm{d}\omega} \tag{1.35}$$

Instead of using the maximum emission intensity, the average intensity with $\kappa_p > 0$ over all reflected angles could also be used. This results in a slightly different formulation of the approximation stated in Equations 1.34 and 1.35.

$$\begin{aligned}
\hat{\tau}\left(s\right) &= \frac{\tau\left(s\right)}{\kappa_p\left(s\right)+1} \\
\hat{g}_s\left(s\right) &= \kappa\left(s\right)\hat{\tau}\left(s\right) \\
\hat{T}\left(\omega\right) &= \hat{g}_s\left(\left(1-\omega\right)s_f+\omega s_b\right) \\
&\quad \exp\left(-d\int_0^{\omega}\hat{\tau}\left(\left(1-\omega'\right)s_f+\omega' s_b\right)\mathrm{d}\omega'\right) \\
\hat{\omega}^* &= \frac{\int_0^1 \omega\hat{T}\left(\omega\right)\mathrm{d}\omega}{\int_0^1 \hat{T}\left(\omega\right)\mathrm{d}\omega} \\
\hat{\kappa}_p^* &= \frac{\int_0^1 \kappa_p\left(\left(1-\omega\right)s_f+\omega s_b\right)\hat{T}\left(\omega\right)\mathrm{d}\omega}{\int_0^1 \hat{T}\left(\omega\right)\mathrm{d}\omega}
\end{aligned} \tag{1.36}$$

Note that average and maximum intensity produce the same result if κ_p is constant or if a single cube-map has been used as reflection or environment map.

Chapter 2

Regular Grids

For a closer look at how to visualize volume data, to compress and render large data sets, the most simple volume representation, the regular grid, is investigated first. Most of the data sets to be encountered in this chapter originate in medical applications and are the results of CT or MR scans. These scanners were built in a way, that their output resolution ranges from $1mm$ down to $\frac{1}{3}mm$. The size is usually a power-of-two value ranging from 32 to 512 per slice with a variable, possibly non-power-of-two, number of slices. Also some data sets originated in physical simulations or even photographs of a cryosection of a human body [Nat86]. Regardless of the size and origin of each data set, they have to be rendered at interactive to real-time frame rates in order to allow for an interactive exploration of the data.

In order to guarantee these frame rates different rendering approaches will be explored in Section 2.1 that all run on graphics hardware. After that compression schemes for large and animated data sets will be investigated in Section 2.2. In order to render large data sets at interactive frame rates a multi-resolution rendering scheme will be presented in Section 2.3. Then some results for rendering and compressing volume data stored on a regular grid will be shown and some conclusions for this type of representation will be drawn.

2.1 Hardware Accelerated Rendering

In order to get the rendering load off the CPU, there are three possible solutions. One could use special volume rendering hardware [KS97, PHK$^+$99], special purpose programmable graphics hardware that has been designed for volume rendering [MKS98, MKW$^+$02, KWHM02], or general purpose graphics hardware, that has become programmable during the last couple of years. The three basic rendering approaches that allow for porting to general purpose graphics hardware are shear-warp [LL94] algorithms, simple algorithms designed for fast evaluation [Ake93] and full featured ray-casting [KH84, Kni00, RGW$^+$03].

The simplest rendering approach is the GPU based adaptation of the shear-warp algorithm. Since it doesn't make any sense to split the transformation into the shear-warp factorization on graphics hardware, the algorithm is reduced to rendering a stack of 2D textures along the main axis of the volume that is most perpendicular to the viewing plane (see Figure 2.1a). This approach will be used for the simple shading in Section 2.1.1. The main problem of this approach is that the distance between the slices changes when rotating the volume and for the cases where the rendering switches between different stacks.

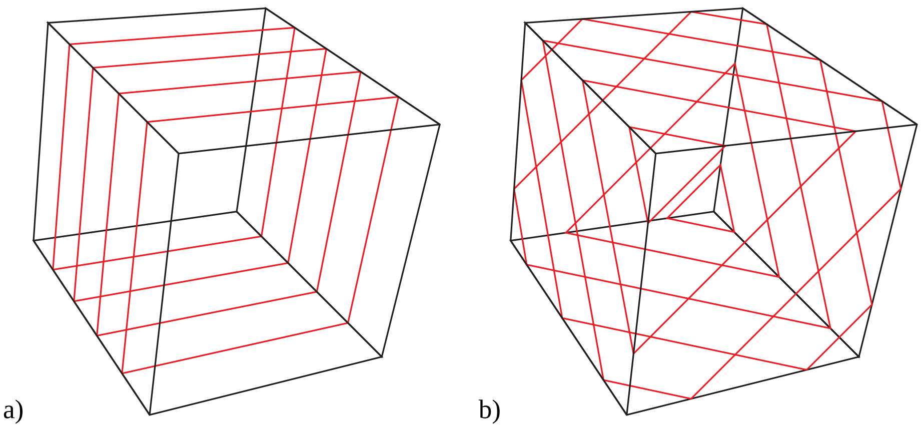


Figure 2.1: Slices for shear-warp (a) and 3D texture mapping based algorithm (b).

Since 3D textures are available on all of today's graphics hardware, the rendering approach proposed by Akeley [Ake93] (see Figure 2.1) can also be used. For this approach, slices that are aligned to the viewing plane are cut through the volume. The distance between these planes stays constant during rotation and no stack switches will happen. However, for perspective projection, the distance between each pair of slices depends on the angle between the view direction and the viewing ray. 3D textures will be used for the high quality shading algorithms in Section 2.1.2 to Section 2.1.5.

To fix the problem of non-constant sampling distances and to correctly render very high frequencies within the data sets, adaptive ray-casting will be used in Section 2.1.6 for highest quality results at the cost of lower frame rates.

2.1.1 Simple Shading

Rendering with a stack of 2D texture (see Figure 2.1) is both the fastest and most simple rendering approach for volume visualization. Applying only the classification to the volume data allows the rendering algorithm to use an index texture [WNDS99] and a color map so that the volume texture does not need to be changed whenever the classification changes. With this approach the 2D textures are just blended over each other and the first simple volume rendering algorithm (see Figure 2.2a) can be implemented. On current graphics hardware however index textures are no longer supported. What actually happens is that the graphics driver applies the classification to the texture slices before uploading. In order to increase the performance of this rendering approach, an RGBα texture to which the classification has already been applied is uploaded. Thus the texture will only be uploaded whenever the classification changes, whereas the index texture may force the graphics hardware to upload the texture every frame.

However, there are two drawbacks with this approach. First, if shading is to be incorporated a second texture or special extensions [MHS99] are used. The rendering pipeline then consists of two parts. The preprocessing and texture upload portion, that applies the classification to the original volume data and uploads the RGBα texture slices to the graphics hardware, and the rendering portion that blends the slices over

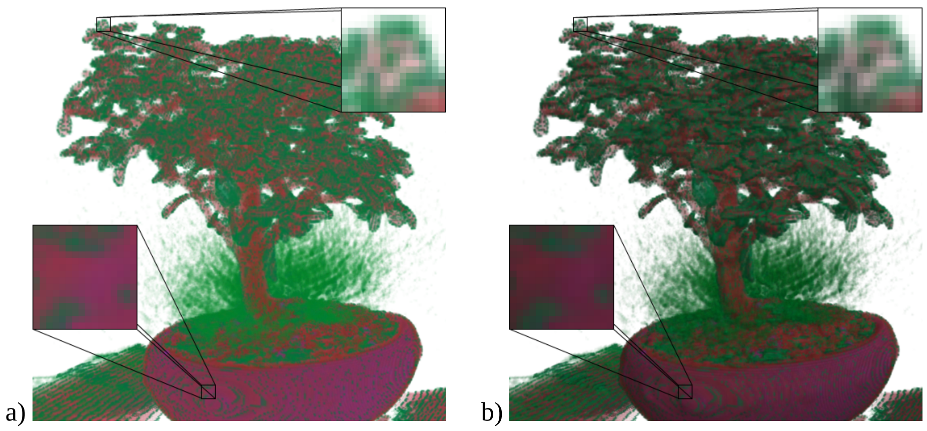


Figure 2.2: Simple shading using two-dimensional textures and object aligned slices without (a) or with lighting (b).

each other using alpha blending to produce the final image. Another approach also incorporates the lighting calculations into the pre-processing step. Additionally, the gradients have to be calculated after loading the volume. This computation however has to be carried out only once. Although this seems to increase the amount of graphics memory being used, index textures end up as RGBα textures on any current graphics hardware, as previously mentioned. A comparison between a rendering without and with lighting can be seen in Figure 2.2. The complete rendering pipeline for the simple shading approach can be seen in Figure 2.3.

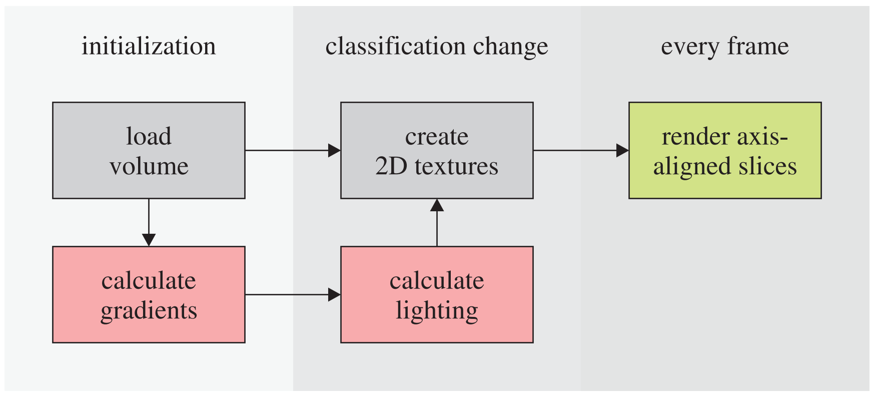


Figure 2.3: Rendering pipeline for simple shading with (red, green and grey) and without (green and grey only) lighting. The green portion is the rendering itself.

The memory consumption during rendering with lighting is four bytes per voxel in main memory and three times four bytes per voxel in texture memory. However, only one slice with four bytes per voxel needs to be present in graphics memory at the same time (minimal rendering footprint). In order to speed up the rendering however, the whole volume needs to be stored into a single 2D texture. Therefore the complete classified volume will be present in texture memory at a time (maximal rendering footprint). Table 2.1 shows a comparison of the two simple shading approaches. This approach uses only one texture unit in 2D texture mode at a time and the so called legacy OpenGL rendering mode [WNDS99]. It also runs on all graphics hardware that supports DirectX 9 [Mic03] in hardware rendering mode.

data type	memory locations	without lighting	with lighting
voxel data	system	2,048 kB	2,048 kB
gradients	system	0 kB	6,144 kB
classification	system	1 kB	1 kB
texture stacks	system/graphics	24,576 kB	24,576 kB
min. rendering footprint	graphics	256 kB	256 kB
max. rendering footprint	graphics	8,192 kB	8,192 kB

Table 2.1: Comparison of memory footprints in kB for 128^3 voxel volume for rendering with simple shading.

2.1.2 Pre-Classification

As already mentioned, 2D textures and axis aligned slices have three major drawbacks. First, the distance between the slices along the viewing ray is not constant during rotation. Second, there are three texture stacks containing the whole volume, one for each axis to be created. Third, whenever switching between different texture stacks occurs, popping artifact become visible. Even with the compensation approaches to these problems [EKE01, GS01], 3D textures and view plane aligned slices are the easiest way to circumvent all three of these problems. Axis aligned slices produce a near constant distance between the sample points along each viewing ray. The distance at the corner of the image may be larger due to the perspective used for generating the image. For a moderate perspective setting however, these artifacts are hardly visible and could be fixed easily with a short fragment program. Comparing the 2D texture rendering with the 3D texture rendering in Figure 2.4, it can clearly be seen that the transparency in the simple shading approach is too high. This is even more visible when the data set is explored interactively.

The rendering pipeline of pre-classified shading with 3D textures (see Figure 2.5) is very similar to the one for simple shading with 2D textures. The rendering itself blends each slice cut through the 3D texture into the frame buffer using alpha blending. The texture has still to be updated every time the classification or the lighting changes since the RGBα texture used for rendering contains the classified and lit voxel. However, there is only one 3D texture to be created instead of three 2D texture stacks packed into three separate textures. But therefore the geometry changes every frame, increasing the traffic between the CPU and the GPU. To circumvent this, one could use a fixed geometry that has to be transformed in world and texture space according to the current camera position. On the other hand, this would require a larger geometry in order to fit every possible rotation within this so called proxy-geometry and 6 user defined clipping planes to cut back the size of this geometry. Besides that, the approach with changing proxy-geometry is very similar to the one proposed by Akeley [Ake93], but without the lookup into the color table. This lookup has been removed since the standard OpenGL index texture will be changed into an RGBα texture whenever the texture is uploaded and no post-interpolative lookup is supported in standard OpenGL or in the legacy

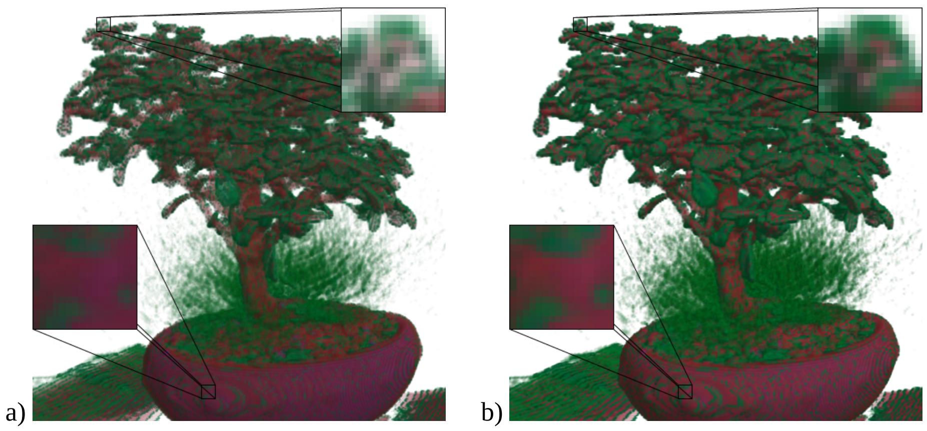


Figure 2.4: Comparison of simple shading (a) and pre-classified rendering with three-dimensional textures (b).

pipeline of DirectX 9. Rendering with this post-interpolative lookup is called post-classification. This approach uses OpenGL 1.3 extensions and only runs on hardware that supports these extensions or on hardware that supports DirectX 9 pixel shader. The rendering itself will be discussed in Section 2.1.3.

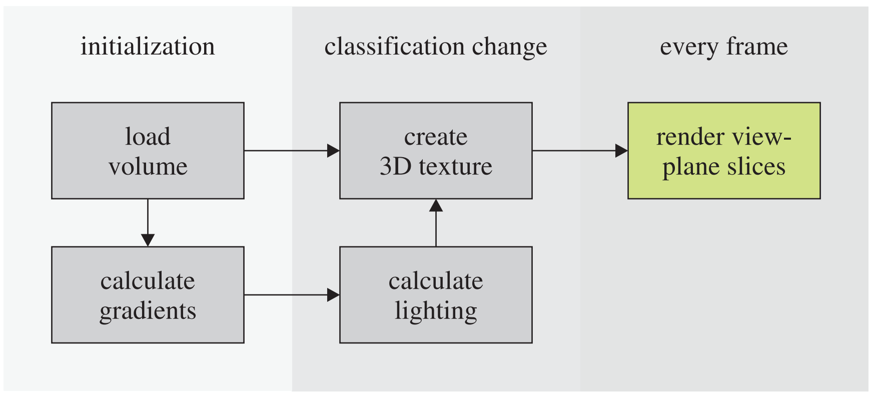


Figure 2.5: Rendering pipeline for pre-classification rendering with lighting. The green portion is the rendering itself.

The memory consumption during rendering is four bytes per voxel in main memory and four bytes per voxel in texture memory. In contrast to the simple shading approach, the complete classified volume has to be present in texture memory at a time since it is represented by a single texture. Table 2.2 shows the memory footprint during rendering and the location where the data is stored. This approach uses only one texture unit in 3D texture mode at a time and the so called legacy OpenGL rendering mode of OpenGL 1.3 or the legacy pipeline of DirectX 9. It also runs on all graphics hardware that supports DirectX 9 in hardware rendering mode.

data type	memory locations	pre-classification
voxel data	system	2,048 kB
gradients	system	6,144 kB
classification	system	1 kB
3D texture	graphics	8,192 kB
rendering footprint	graphics	8,192 kB

Table 2.2: Memory footprints in kB for 128^3 voxel volume for rendering with pre-classification.

2.1.3 Post-Classification

Using the post-classification paradigm, the rendering is somewhat more complicated. In addition to the post-interpolative lookup into the transfer function, the shading also has to happen based on interpolated gradients. Theoretically these interpolated gradients have to be normalized in order to evaluate the diffuse lighting correctly. In practice however, it turns out that the resulting image is only slightly darker than the correct solution so that the normalization is usually skipped. Also the visual quality of post-classified rendering turns out to be a lot higher than the quality of pre-classified rendering (see Figure 2.6).

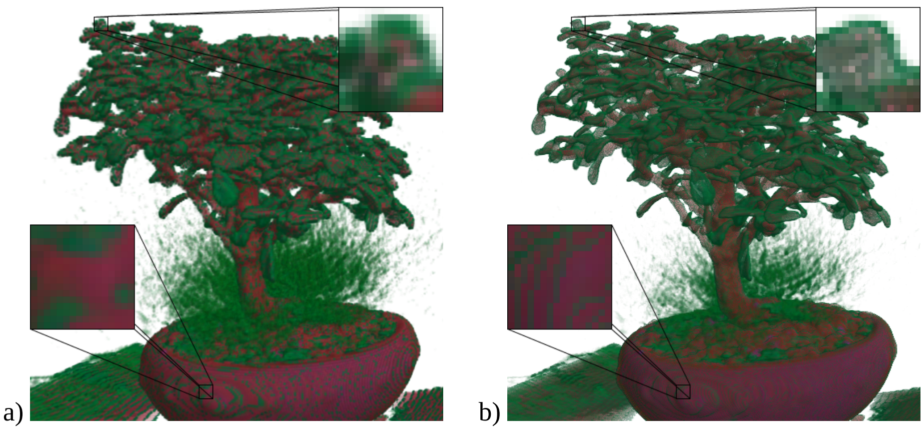


Figure 2.6: Comparison of pre-classified (a) and slice based post-classified (b) rendering on the graphics card.

The rendering pipeline of post-classified shading with 3D textures (see Figure 2.7) is still very similar to the previous ones. The main change is that a 1D texture now contains a lookup table that stores the transfer function. Therefore the 3D texture needs not to be changed if the transfer function changes. Also none of the textures needs to be changed if the light position is updated since the shading itself now takes place during rendering. Even if this rendering paradigm seems to result in a higher performance because of the reduced traffic between the CPU and the GPU for changing the transfer function, the actual performance is a lot lower than for pre-classified rendering. This is because of the dependent texture lookup and the per-pixel shading.

The memory consumption during rendering is four bytes per voxel in main memory and four bytes per voxel in texture memory. In addition to this the transfer function is

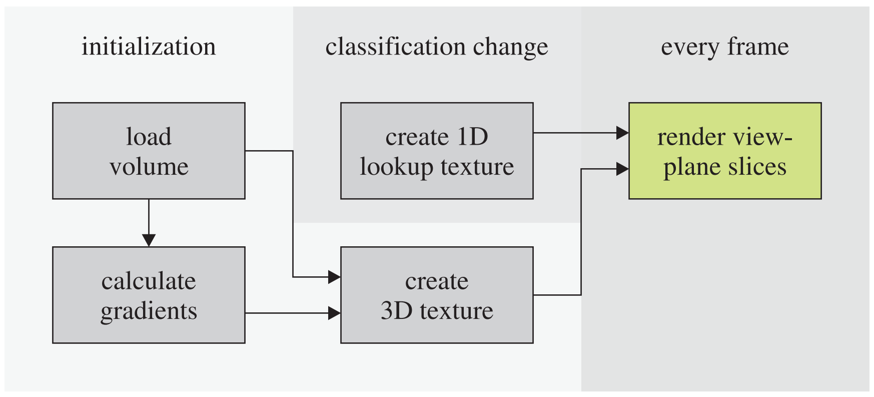


Figure 2.7: Rendering pipeline for post-classification rendering with lighting. The green portion is the rendering itself.

stored in main and texture memory with four bytes per entry. Table 2.3 shows the memory footprint during rendering and the location where the data is stored. This approach uses two texture units, one in 3D texture mode and one in 1D dependent mode.

data type	memory locations	post-classification
voxel data	system	2,048 kB
gradients	system	6,144 kB
classification	system	1 kB
3D texture	graphics	8,192 kB
1D lookup texture	graphics	1 kB
rendering footprint	graphics	8,193 kB

Table 2.3: Memory footprints in kB for 128^3 voxel volume for rendering with post-classification.

OpenGL fragment program

While the geometry transformation uses the legacy pipeline of both, OpenGL 1.3 and DirectX 9, a fragment program or pixel shader is used for the per-pixel computations. The OpenGL fragment program uses shader code version 1.0 while the DirectX pixel shader requires version 1.1. The per pixel computations can be expressed in these five steps.

1. Sample volume data $s(x(\omega))$ and gradient $\overrightarrow{n}(x(\omega))$.

2. Get opacity $\alpha(s(x(\omega))) = 1 - exp(-d\tau(s(x(\omega))))$ and emission color $\kappa(s(x(\omega)))$.

3. Calculate diffuse emission intensity $l_d(x(\omega)) = max(\overrightarrow{n}(x(\omega)) \cdot \overrightarrow{l}, 0)$.

4. Calculate emission $C(x(\omega)) = \left(\frac{1}{2}l_d(x(\omega)) + \frac{1}{2}\right)\kappa(x(\omega))$.

5. Write emissive color $\Delta I_\omega = C\left(x\left(\omega\right)\right)\alpha\left(s\left(x\left(\omega\right)\right)\right)$.

For back-to-front rendering the blending is set to multiply $1-\alpha$ with the previously stored color values. For front-to-back rendering, the output color is multiplied by $1-\alpha^*$ with α^* being the transparency of all previously rendered pixels.

```
!!ARBfp1.0

ATTRIB    texPos   =   fragment.texcoord[0];
ATTRIB    lightDir =   fragment.texcoord[1];

TEMP      voxel;
TEMP      color;
PARAM     const    = { 0.5, 1.0, 2.0, 0.0 };

# sample volume data at given position                ①
TEX       voxel, texPos, texture[0], 3D;

# use sample value to lookup into classification      ②
TEX       color, voxel.a, texture[1], 2D;

# calculate diffuse emission intensity l_d            ③
MAD       voxel, voxel, const.b, -const.g;
DP3_SAT   voxel, voxel, lightDir;

# 50% diffuse and 50% ambient light                   ④
MAD       voxel, voxel, const.r, const.r;
MUL       color.rgb, color, voxel;

# write pre-multiplied color                          ⑤
MUL       color.rgb, color, color.a;
MOV       result.color, color;

END
```

DirectX 9 pixel shader

Even if the order of the computations is different for the DirectX 9 pixel shader, the functionality remains the same. Furthermore, the hardware requirements for the DirectX 9 implementation are lower.

```
ps_1_1

// sample volume data at given position                ①
TEX       t0;

// pass light direction
TEXCOORD  t1;
```

```
// use sample value to lookup into classification          ②
TEXREG2AR t2, t0;

// calculate diffuse emission intensity l_d                 ③
  DP3_SAT r0.rgb, t1_bx2, t0_bx2;

// 50% diffuse and 50% ambient light                        ④
  MAD_D2  r0.rgb, r0, t2, t2;

// write pre-multiplied color                               ⑤
+ MOV     r0.a, t2.a;
  MUL     r0.rgb, r0, r0.a;
```

2.1.4 Pre-Integration

As already mentioned, the post-classification introduces very high frequencies into the volume data. To reduce the sampling rate for post-classified rendering Engel et al. [EKE01] presented a pre-integrated rendering approach using graphics hardware. The resulting image quality is greatly increased with this rendering scheme, especially for iso-surface-like transfer functions (see Figure 2.8).

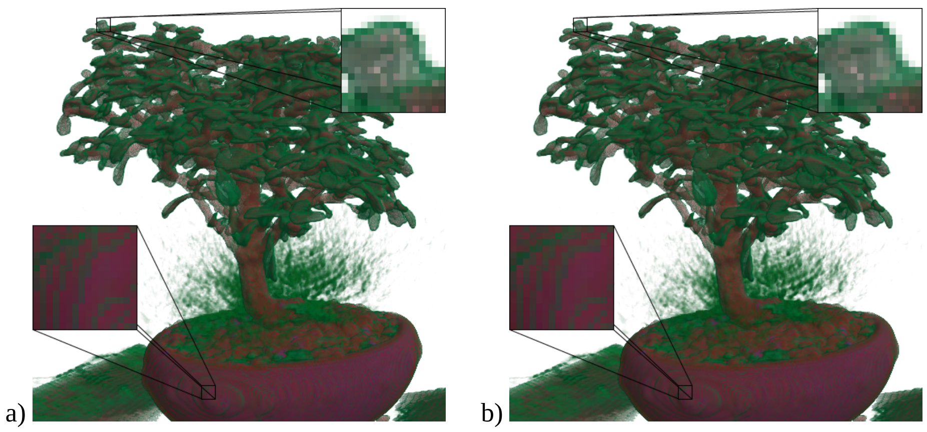


Figure 2.8: Comparison of simple post-classified (a) and slab based pre-integrated rendering (b).

The rendering pipeline of pre-integrated rendering with 3D textures (see Figure 2.9) is the same as the previous one with only one exception. The transfer function is now residing within a 2D texture to implement the pre-integrated lookup. The rest of the pipeline has the same advantages and limitations as post-classified shading.

The memory consumption during rendering is four bytes per voxel in main memory and four bytes per voxel in texture memory. In addition to this the transfer function is stored in main and texture memory with four bytes per entry. Table 2.4 shows the memory footprint during rendering and the location where the data is stored. This approach uses two texture units, one in 3D texture mode and one in 2D dependent

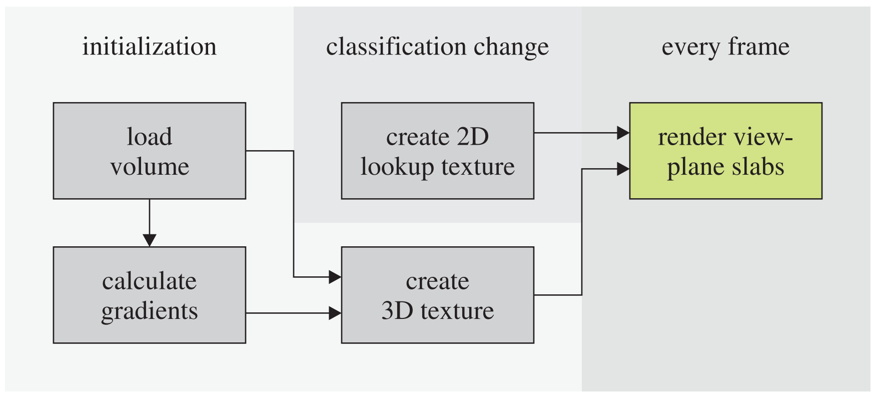


Figure 2.9: Rendering pipeline for pre-integrated rendering with lighting. The green portion is the rendering itself.

mode. With certain hardware limitations two texture units may be used for sampling the 3D texture, but both units will sample the same texture memory.

data type	memory locations	pre-integration
voxel data	system	2,048 kB
gradients	system	6,144 kB
classification	system	1 kB
3D texture	graphics	8,192 kB
2D lookup texture	graphics	256 kB
rendering footprint	graphics	8,448 kB

Table 2.4: Memory footprints in kB for 128^3 voxel volume for rendering with pre-integration.

OpenGL vertex program

The OpenGL or DirectX 9 vertex program is very close to part of the legacy pipeline. The only difference is the calculation of the texture coordinates for sampling the 3D texture containing the volume data. The functionality can be described with the following six steps.

1. Transform the vertex coordinates with the model-view-projection matrix (just like the legacy pipeline).
2. Calculate the sampling distance for the slices $x\,(d)$ for each vertex and interpolate in between.
3. Rotate the reciprocal of the scaling factors between world and texture space, and the eye direction into texture space.
4. Calculate front and back offset for texture coordinates.
5. Write texture coordinates to the output.

6. Write rotated eye direction and light direction to output.

After the vertex shader, the per-fragment computations sample the volume textures and calculate the per-fragment lighting.

```
!!ARBvp1.0

ATTRIB    iPos     =   vertex.position;
ATTRIB    iTex0    =   vertex.texcoord[0];
ATTRIB    iTex1    =   vertex.texcoord[1];

PARAM     mvp[4]   = { state.matrix.mvp };
PARAM     lightDir =   program.env[0];
PARAM     location =   program.env[1];
PARAM     viewDir  =   program.env[2];
PARAM     iScale   =   program.env[3];
PARAM     rotation =   program.env[4];

TEMP      eyeDir;
TEMP      offsetFront;
TEMP      offsetBack;
TEMP      iScaleTex;
TEMP      eyeDirTex;

OUTPUT    oPos     =   result.position;
OUTPUT    oTex0    =   result.texcoord[0];
OUTPUT    oTex1    =   result.texcoord[1];
OUTPUT    oTex2    =   result.texcoord[2];
OUTPUT    oTex3    =   result.texcoord[3];

# transform vertex                                      ①
DP4       oPos.x, mvp[0], iPos;
DP4       oPos.y, mvp[1], iPos;
DP4       oPos.z, mvp[2], iPos;
DP4       oPos.w, mvp[3], iPos;

# calculate texture coordinate offsets                  ②
SUB       eyeDir, iPos, location;
DP3       eyeDir.w, eyeDir, viewDir;
RCP       eyeDir.w, eyeDir.w;
MUL       eyeDir, eyeDir, eyeDir.w;

# rotate inverse scaling factors and eye direction      ③
# into texture space
MUL       iScaleTex, rotation.x, iScale.xyzw;
MAD       iScaleTex, rotation.y, iScale.yzxw, iScaleTex;
MAD       iScaleTex, rotation.z, iScale.zxyw, iScaleTex;
MUL       eyeDirTex, rotation.x, eyeDir.xyzw;
MAD       eyeDirTex, rotation.y, eyeDir.yzxw, eyeDirTex;
MAD       eyeDirTex, rotation.z, eyeDir.zxyw, eyeDirTex;
```

```
# calculate front and back offset with correct scaling  ④
MUL         offsetFront, iScaleTex, iTex1.x;
MUL         offsetBack, iScaleTex, iTex1.y;

# write texture coordinates                              ⑤
MAD         oTex0, offsetFront, eyeDirTex, iTex0;
MAD         oTex1, offsetBack, eyeDirTex, iTex0;

# write eye direction and light direction for lighting  ⑥
MOV         oTex2, eyeDir;
MOV         oTex3, lightDir;

END
```

OpenGL fragment program

While the geometry transformation mainly implements the legacy pipeline for both, OpenGL 1.3 and DirectX 9, a more complex fragment program or pixel shader is used for the per-pixel computations. The OpenGL fragment program uses shader code version 1.0 while the DirectX pixel shader requires version 1.4. The per pixel computations can be expressed in these seven steps.

1. Sample volume data $s_f = s(x(\omega))$, $s_b = s(x(\omega + d))$ and gradient $\vec{n}_f = \vec{n}(x(\omega))$, $\vec{n}_b = \vec{n}(x(\omega + d))$.
2. Combine s_f and s_b into a single texture coordinate (s_f, s_b)
3. Get opacity $\alpha(s_f, s_b)$ and emission color $\kappa(s_f, s_b)$.
4. Calculate diffuse emission intensities $l_{df}(x(\omega)) = max(\vec{n}_f \cdot \vec{l}, 0)$ and $l_{db}(x(\omega)) = max(\vec{n}_b \cdot \vec{l}, 0)$.
5. Weight l_{df} and l_{db} based on α and combine them into l_d.
6. Calculate emission $C(x(\omega)) = \left(\frac{1}{2} l_d + \frac{1}{2}\right) \kappa(s_f, s_b)$.
7. Write emissive color $\Delta I_\omega = C(x(\omega))\, \alpha(s_f, s_b)$.

For back-to-front rendering the blending is set to multiply $1 - \alpha$ with the previously stored color values. For front-to-back rendering, the output color is multiplied by $1 - \alpha^*$ with α^* being the transparency of all previously rendered pixels.

```
!!ARBfp1.0

ATTRIB      frontPos =   fragment.texcoord[0];
ATTRIB      backPos  =   fragment.texcoord[1];
ATTRIB      eyeDir   =   fragment.texcoord[2];
ATTRIB      lightDir =   fragment.texcoord[3];

TEMP        frontVoxel;
TEMP        backVoxel;
TEMP        color;
PARAM       const    = { 0.5, 1.0, 2.0, 0.0 };
```

```
# sample volume data at given position                                ①
TEX       frontVoxel, frontPos, texture[0], 3D;
TEX       backVoxel, backPos, texture[0], 3D;

# calculate texture coordinate                                        ②
LRP       color, const.aggg, frontVoxel.a, backVoxel.a;

# use sample values to lookup into classification                     ③
TEX       color, color, texture[1], 2D;

# calculate diffuse emission intensities l_df and l_db                 ④
MAD       frontVoxel, frontVoxel, const.b, -const.g;
DP3_SAT   frontVoxel, frontVoxel, lightDir;
MAD       backVoxel, backVoxel, const.b, -const.g;
DP3_SAT   backVoxel, backVoxel, lightDir;

# weight l_df and l_db to get l_d                                      ⑤
LRP       frontVoxel, color.a, frontVoxel, backVoxel;

# 50% diffuse and 50% ambient light                                    ⑥
MAD       frontVoxel, frontVoxel, const.r, const.r;
MUL       color.rgb, color, frontVoxel;

# write pre-multiplied color                                           ⑦
MUL       color.rgb, color, color.a;
MOV       result.color, color;

END
```

DirectX 9 vertex shader

The DirectX 9 vertex program basically looks like the OpenGL vertex program except for the usage of the macro commands that are not available in OpenGL.

```
vs_1_1

// vertex position (xyz1)
DCL_POSITION0 v0

// texcoord at slice location (xyz)
DCL_TEXCOORD0 v1

// thickness of slice -d/2,d/2(xy)
DCL_TEXCOORD1 v2

DEF       c8, 0.5, 0.0, 0.0, 0.0;

// transform vertex                                                    ①
M4X4      oPos, v0, c0;
```

```
// calculate texture coordinate offsets                           ②
SUB       r3.xyz, v0.xyz, c7.xyz;
DP3       r3.w, r3.xyz, c10.xyz;
RCP       r3.w, r3.w;
MUL       r3.xyz, r3.xyz, r3.www;

// rotate inverse scaling factors and eye direction              ③
// into texture space
MOV       r4.xyz, c8.xyz;
MUL       r5.xyz, c11.x, r4.xyz;
MAD       r5.xyz, c11.y, r4.yzx, r5.xyz;
MAD       r5.xyz, c11.z, r4.zxy, r5.xyz;
MUL       r4.xyz, c11.x, r3.xyz;
MAD       r4.xyz, c11.y, r3.yzx, r4.xyz;
MAD       r4.xyz, c11.z, r3.zxy, r4.xyz;

// calculate front and back offset with correct                  ④
// scaling
MUL       r1.xyz, r5.xyz, v2.xxx;
MUL       r2.xyz, r5.xyz, v2.yyy;

// write texture coordinates                                      ⑤
MAD       oT0.xyz, r1.xyz, r4.xyz, v1.xyz;
MAD       oT1.xyz, r2.xyz, r4.xyz, v1.xyz;

// write eye direction and light direction for                   ⑥
// lighting
MOV       oT2.xyz, r3.xyz;
MOV       oT3.xyz, c4.xyz;
```

DirectX 9 pixel shader

The DirectX 9 pixel shader looks a lot different from the OpenGL implementation. The eye and light vectors have to be passed to the second stage (a) and the first and second stage need to be separated with the *phase* command (b). The functionality however is the same, even if the hardware requirements of the DirectX 9 implementation are lower.

```
ps_1_4

DEF       c0, 0.0, 1.0, 1.0, 0.5;

// sample volume data at given position                          ①
TEXLD     r0, t0;
TEXLD     r1, t1;

// pass eye and light vector                                      ⓐ
TEXCRD    r2.rgb, t2.xyz;
TEXCRD    r3.rgb, t3.xyz;
```

```
// calculate diffuse emission intensities l_df and          ④
// l_db
  DP3_SAT r0.rgb, r0_bx2, r3;
  DP3_SAT r1.rgb, r1_bx2, r3;

// calculate texture coordinate                              ②
  LRP     r3.rgb, c0, r1.a, r0.a;

PHASE                                                        ⓑ

// use sample values to lookup into classification           ③
TEXLD     r2, r3;

// weight l_df and l_db to get l_d                           ⑤
  LRP     r0.rgb, r2.a, r0, r1;

// 50% diffuse and 50% ambient light                         ⑥
  MAD_D2  r0.rgb, r0, r2, r2;

// write pre-multiplied color                                ⑦
+ MOV     r0.a, r2.a;
  MUL     r0.rgb, r0, r0.a;
```

2.1.5 Pre-Integrated Lighting

To further increase the image quality, the lighting equations are also pre-integrated as proposed by Meissner et al. [MGS02]. A comparison between pre-integration and pre-integrated lighting can be seen in Figure 2.10.

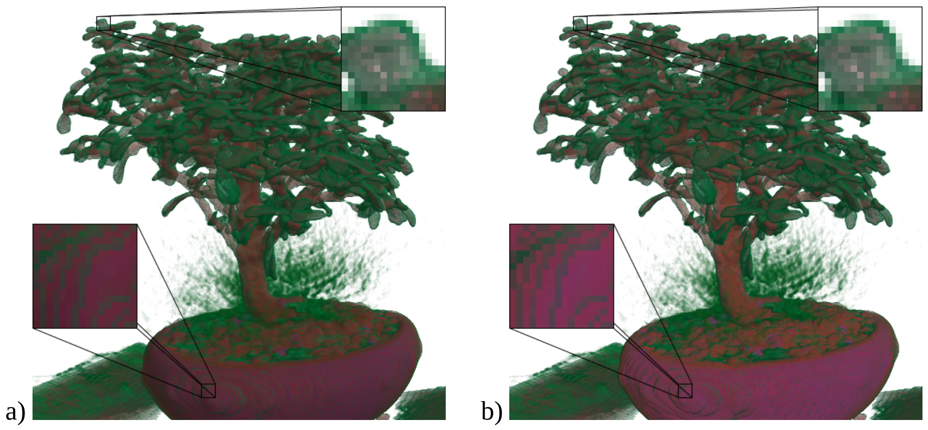


Figure 2.10: Comparison of pre-integrated (a) with material pre-integrated rendering (b) for higher quality lighting of the volume.

The rendering pipeline of pre-integrated lighting Figure 2.11 is again the same as the previous one, except that more than one 2D texture is used for the transfer function.

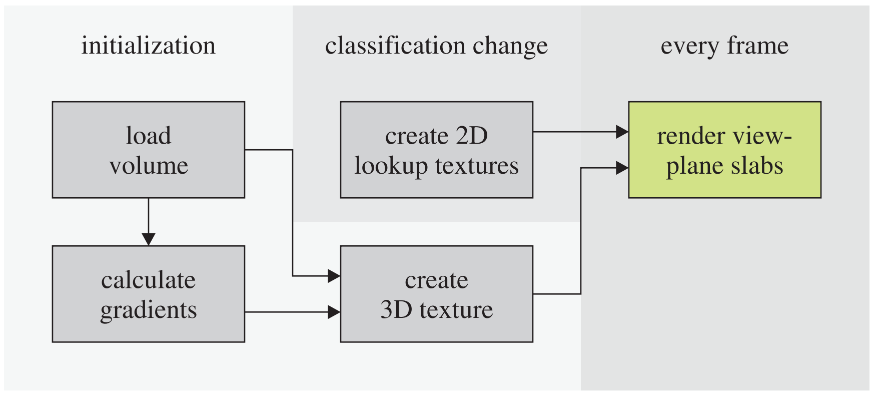


Figure 2.11: Rendering pipeline for material pre-integrated rendering with lighting. The green portion is the rendering itself.

The memory consumption during rendering is four bytes per voxel in main memory and four bytes per voxel in texture memory. In addition to this the transfer function is stored in main and texture memory with four bytes per entry. Table 2.5 shows the memory footprint during rendering and the location where the data is stored. This approach uses four texture units, one in 3D texture mode and three in 2D dependent mode. With certain hardware limitations two texture units may be used for sampling the 3D texture, but both units will sample the same texture memory.

data type	memory locations	mat. pre-integration
voxel data	system	2,048 kB
gradients	system	6,144 kB
classification	system	2 kB
3D texture	graphics	8,192 kB
2D lookup textures	graphics	768 kB
rendering footprint	graphics	8,960 kB

Table 2.5: Memory footprints in kB for 128^3 voxel volume for rendering with material pre-integration.

The vertex program remains, for both OpenGL and DirectX 9, exactly the same since only the dependent texture lookups and the shading changed.

OpenGL fragment program

The OpenGL fragment program for pre-integrated lighting uses shader code version 1.0 while the DirectX pixel shader requires version 1.4. The per pixel computations can be expressed in these six steps.

1. Sample volume data $s_f = s\,(x\,(\omega))$, $s_b = s\,(x\,(\omega + d))$ and gradient $\overrightarrow{n}_f = \overrightarrow{n}\,(x\,(\omega))$, $\overrightarrow{n}_b = \overrightarrow{n}\,(x\,(\omega + d))$.

2. Combine s_f and s_b into a single texture coordinate (s_f, s_b)

3. Get opacity $\alpha\left(s_f, s_b\right)$ and emission colors $\kappa_a\left(s_f, s_b\right)$, $\kappa_{df}\left(s_f, s_b\right)$ and $\kappa_{db}\left(s_f, s_b\right)$.
4. Calculate diffuse emission intensities $l_{df}\left(x\left(\omega\right)\right) = max(\overrightarrow{n}_f \cdot \overrightarrow{l}, 0)$ and $l_{db}\left(x\left(\omega\right)\right) = max(\overrightarrow{n}_b \cdot \overrightarrow{l}, 0)$.
5. Calculate emission $C\left(x\left(\omega\right)\right) = \kappa_a + l_{df}\left(x\left(\omega\right)\right)\kappa_{df}\left(s_f, s_b\right) + l_{db}\left(x\left(\omega\right)\right)\kappa_{db}\left(s_f, s_b\right)$.
6. Write emissive color $\Delta I_\omega = C\left(x\left(\omega\right)\right)\alpha\left(s_f, s_b\right)$.

For back-to-front rendering the blending is set to multiply $1 - \alpha$ with the previously stored color values. For front-to-back rendering, the output color is multiplied by $1-\alpha^*$ with α^* being the transparency of all previously rendered pixels.

```
!!ARBfp1.0

ATTRIB    frontPos =   fragment.texcoord[0];
ATTRIB    backPos  =   fragment.texcoord[1];
ATTRIB    eyeDir   =   fragment.texcoord[2];
ATTRIB    lightDir =   fragment.texcoord[3];

TEMP      frontVoxel;
TEMP      backVoxel;
TEMP      color;
TEMP      front;
TEMP      back;
TEMP      coord;
PARAM     const    = { 0.5, 1.0, 2.0, 0.0 };

# sample volume data at given position                                ①
TEX       frontVoxel, frontPos, texture[0], 3D;
TEX       backVoxel, backPos, texture[0], 3D;

# calculate texture coordinate                                        ②
LRP       coord, const.aggg, frontVoxel.a, backVoxel.a;

# use sample values to lookup into classification                     ③
TEX       color, coord, texture[1], 2D;
TEX       front, coord, texture[2], 2D;
TEX       back, coord, texture[3], 2D;

# calculate diffuse emission intensities l_df and l_db                ④
MAD       frontVoxel, frontVoxel, const.b, -const.g;
DP3_SAT   frontVoxel, frontVoxel, lightDir;
MAD       backVoxel, backVoxel, const.b, -const.g;
DP3_SAT   backVoxel, backVoxel, lightDir;

# calculate final color with l_df, l_db and l_a                       ⑤
MAD       color.rgb, frontVoxel, front, color;
MAD       color.rgb, backVoxel, back, color;
```

```
# write pre-multiplied color                                        ⑥
MUL        color.rgb, color, color.a;
MOV        result.color, color;

END
```

DirectX 9 pixel shader

The DirectX 9 pixel shader looks a lot different from the OpenGL implementation. Again the eye and light vectors are passed to the second stage (a) and the first and second stage need to be separated with the *phase* command (b). The functionality however is the same, even if the hardware requirements of the DirectX 9 implementation are lower.

```
ps_1_4

DEF        c0, 0.0, 1.0, 1.0, 0.5;

// sample volume data at given position                             ①
TEXLD      r0, t0;
TEXLD      r1, t1;

// pass eye and light vector                                        ⓐ
TEXCRD     r2.rgb, t2.xyz;
TEXCRD     r3.rgb, t3.xyz;

// calculate diffuse emission intensities l_df and                  ④
// l_db
  DP3_SAT r0.rgb, r0_bx2, r3;
  DP3_SAT r1.rgb, r1_bx2, r3;

// calculate texture coordinate                                     ②
  LRP      r4.rgb, c0, r1.a, r0.a;

PHASE                                                               ⓑ

// use sample values to lookup into classification                  ③
TEXLD      r2, r4;
TEXLD      r3, r4;
TEXLD      r4, r4;

// calculate final color with l_df, l_db and l_a                    ⑤
  MAD      r0.rgb, r0, r3, r2;
  MAD      r0.rgb, r1, r4, r0;

// write pre-multiplied color                                       ⑥
+ MOV      r0.a, r2.a;
  MUL      r0.rgb, r0, r0.a;
```

2.1.6 Ray-casting

Ray-casting can be used as a different rendering approach that is able to better adapt to the data set and the transfer function to both increase image quality and performance. The main advantage of ray-casting is that the sampling distance can be changed dynamically for each ray. For this another volume is used that holds the maximum sampling distance from a certain point in any possible direction. Combined with pre-integrated rendering, a dramatic increase in image quality can again be seen (see Figure 2.12).

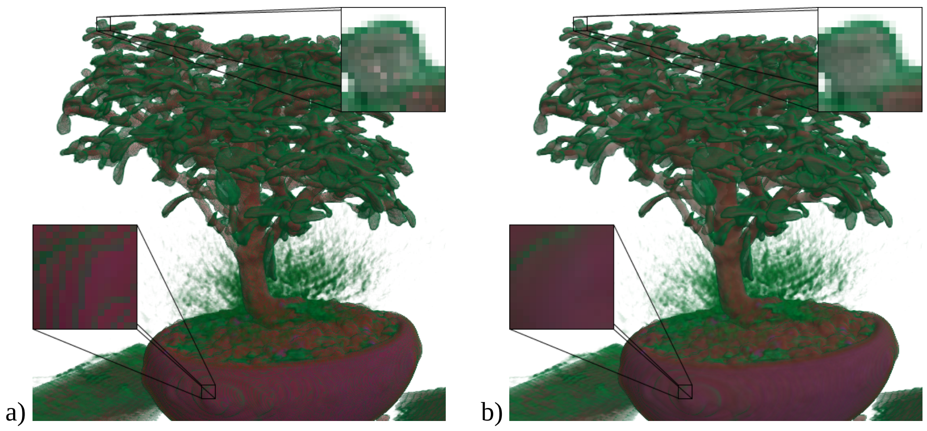


Figure 2.12: Comparison of pre-integrated rendering (a) with pre-integrated ray-casting (b) for highest quality images.

In contrast to the previous rendering pipelines, the ray-casting pipeline (see Figure 2.13) has more than one stage running on the GPU and also implements a loop-back on the GPU. To efficiently implement the loop-back and thus the termination of completed rays, the z-buffer is used in combination with early z-test to skip any terminated ray. Also the rendering consists of a combined termination & copy pass to keep the buffers used for rendering consistent.

The memory consumption during rendering is five bytes per voxel in main memory and five bytes per voxel in texture memory. In addition to this the transfer function is stored in main and texture memory with four bytes per entry. Additionally ten bytes per output pixel are needed for the loop-back textures. Table 2.6 shows the memory footprint during rendering and the location where the data is stored. This approach uses four texture units, one in 3D texture mode and three in 2D with one in dependent mode.

Since both the OpenGL and DirectX 9 vertex and fragment calculations are exactly the same, only the DirectX 9 vertex and pixel shader are presented because of their higher portability between different graphics cards.

DirectX 9 vertex shader

The vertex shader basically performs the ray setup and starts the per-fragment computations. The shader can be described as four steps.

1. Transform the vertex coordinates with the model-view-projection matrix (just like the legacy pipeline).

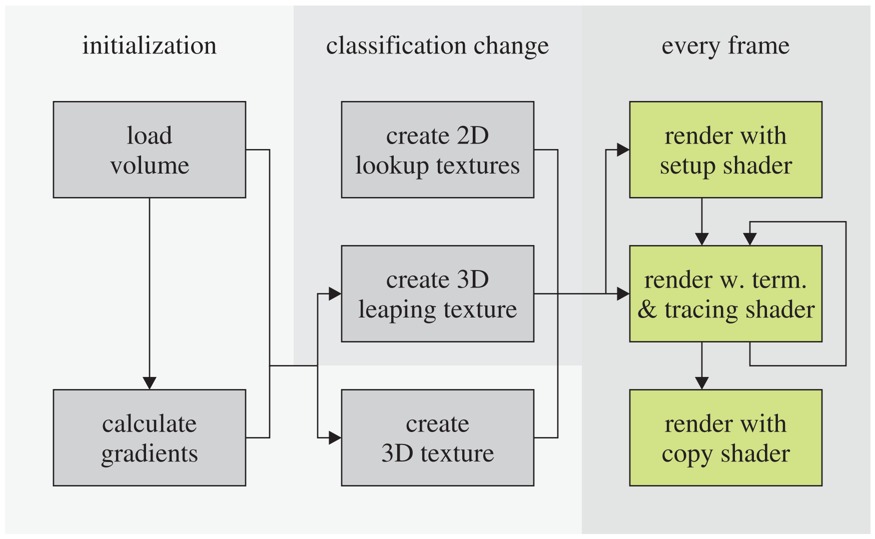


Figure 2.13: Rendering pipeline for ray-casting with pre-integration and lighting. The green portion is the rendering itself.

data type	memory locations	pre-integration
voxel data	system	2,048 kB
gradients	system	6,144 kB
classification	system	1 kB
3D texture	graphics	8,192 kB
3D acceleration texture	graphics	2,048 kB
2D lookup texture	graphics	256 kB
2D loop-back textures	graphics	2,560 kB
rendering footprint	graphics	13,056 kB

Table 2.6: Memory footprints in kB for 128^3 voxel volume for rendering with ray casting onto a 512^2 screen.

2. Calculate the ray increment $x(d)$ for each vertex and interpolate in between.
3. Calculate the scaling factors between world and texture space.
4. Write everything to the output.

Since the vertex shader is used to start the per-fragment computations, the same vertex shader is used for all rendering passes.

```
vs_1_1

// vertex position (xyz1)
DCL_POSITION0 v0
// texture coordinate at start position
DCL_TEXCOORD0 v1
```

```
DEF         c12, 0.5, 1.0, 0.002, 63.75;
DEF         c13, 0.25, 0.25, 0.25, 0.25;

// transform vertex                                    ①
M4X4        r0, v0, c0;

// calculate increment for ray                         ②
MOV         r1, c8;
MUL         r1.xyz, r1, c12.w;
ADD         r2.xyz, v0.xyz, -c7.xyz;
MOV         r3, r0;
MOV         r3.y, -r3.y;
ADD         r3.xyz, r3, r3.w;
MAD         r3.xyz, r3.w, c12.z, r3;
MUL         r3.xyz, r3, c12.x;

// calculate scaling factor                            ③
RCP         r4.x, c8.x;
RCP         r4.y, c8.y;
RCP         r4.z, c8.z;

// write position, ...                                 ④
MOV         oPos.xyzw, r0.xyzw;
// ...  texture coordinate, ...
MOV         oT0.xyz, v1.xyz;
// ...  tracing direction, ...
MOV         oT1.xyz, r2.xyz;
// ...  stepping distance, ...
MUL         oT2.xyz, r1.xyz, r1.w;
// ...  light direction, ...
MOV         oT3.xyz, c4.xyz;
// ...  pixel position, ...
MOV         oT4.xyzw, r3.xyzw;
// ...  scaling factor and ...
MOV         oT5.xyz, r4;
// ...  inverse scaling factor
MUL         oT6.xyz, c8, c8.w;
MOV         r5, c8;
MUL         oT6.w, r5.w, c13.x;
```

DirectX 9 pixel shader

The rendering is started with the *setup pass*. Here the following computations are carried out.

1. Sample the volume data at the starting location, get the first sampling distance multiplier $\Delta\omega$ and calculate the ray parameters for the lower and upper bound of the volume data.

2. Store the current and maximum sampling distance, i.e. ray parameter ω.

3. Save old sampling value s_f and normal $\overrightarrow{n}_f$ and update ray parameter ω.

4. Get new sample value s_b, normalize $\overrightarrow{n}_f$ and combine s_f, s_b, and $\Delta\omega$ into a single texture coordinate.

5. Lookup $\kappa\left(s_f, s_b, \Delta\omega\right)$ and $\tau\left(s_f, s_b, \Delta\omega\right)$.

6. Calculate $\alpha\left(s_f, s_b, \Delta\omega\right) = \tau\left(s_f, s_b, \Delta\omega\right)^{\Delta\omega}$.

7. Calculate light intensity $l_d = max(\overrightarrow{n}_f \cdot \overrightarrow{l}, 0)$ and pre-multiplied emission color $C = \alpha\left(s_f, s_b, \Delta\omega\right) \kappa\left(s_f, s_b, \Delta\omega\right) \left(\frac{1}{2} l_d + \frac{1}{2}\right)$. Also split color for output.

8. Output RGBα, current and maximum ray parameter as half-float values.

This rendering pass is only used once to initialize the maximum ray parameters. Any consecutive rendering pass will use the value stored in the third output texture.

Setup pass

```
ps_2_0

DEF        c0, 0.0, 2.0, 63.75, 1.0;
DEF        c1, 1.0, 0.0, 0.0, 0.5;
DEF        c4, 0.002, 0.3333333333, 0.0, 0.0;

DCL        t0.xyz;
DCL        t1.xyz;
DCL        t2.xyz;
DCL        t3.xyz;

// volume
DCL_VOLUME s0;
// leap map
DCL_VOLUME s1;
// pre-int table with correct self attenuation
DCL_VOLUME s2;
```

```
// ---- setup ----                                              ①
// scalar value at starting point
TEXLD      r2, t0, s0;
// multiplier for sampling distance
TEXLD      r5, t0, s1;
NRM        r0.xyz, t1;
// scaled distance
MUL        r0.xyz, r0, t2;
// signed distance to lower bound of volume
ADD        r4.xyz, c2, -t0;
// signed distance to upper bound of volume
ADD        r3.xyz, c3, -t0;
```

```
// calculate maximum ray length
RCP        r6.x, r0.x;
RCP        r6.y, r0.y;
RCP        r6.z, r0.z;
MUL        r3.xyz, r3, r6;
MUL        r4.xyz, r4, r6;
MAX        r6.xyz, r3, r4;
MIN        r4.x, r6.x, r6.y;
MIN        r1.y, r4.x, r6.z;
```

```
// store distance, maximum distance and previous       ②
// distance
MOV        r1.xzw, c0.x;
```

```
// ---- stepping ----                                   ③
// save old position and add sampling distance to new
// one
MAD        r1.xz, r5.x, c1, r1.x;
// clamp to maximum sampling position
MIN        r1.x, r1.x, r1.y;
// new sampling position
MAD        r3.xyz, r1.x, r0, t0;
```

```
// ---- pre-integration ----                            ④
// scalar value at end point
TEXLD      r4, r3, s0;
MAD        r2.xyz, r2, c0.y, -c0.w;
// normalize gradient
NRM        r8.xyz, r2;
// include sampling distance for pre-integration
ADD        r6.w, r1.x, -r1.z;
LRP        r6.xy, c1, r2.w, r4.w;
MAD        r6.z, r6.w, c4.y, -c4.y;
```

```
// lookup into pre-integration table                    ⑤
TEXLD      r7, r6, s2;
```

```
// correct opacity for sampling distance                ⑥
MUL        r6.x, c0.z, r6.w;
ADD        r7.w, c0.w, -r7.w;
POW        r10.w, r7.w, r6.x;
ADD        r7.w, c0.w, -r10.w;
```

```
// do lighting                                          ⑦
DP3_SAT    r8.w, r8, t3;
MUL        r7.xyz, c1.w, r7;
MAD        r7.xyz, r8.w, r7, r7;
MUL        r7.xyz, r7, r7.w;
```

```
// write pre-multiplied color
MOV       r0, r7.z;
MOV       r0.y, r7.w;
```

```
// ---- final ----                                         ⑧
MOV       oC0, r7;
MOV       oC1, r0;
MOV       oC2, r1;
```

The *termination pass* itself is rather simple. The idea behind this pass is to keep the two ping-pong buffers containing the color information consistent if a ray terminates and to write a z-value in the case of a termination. The shader does the following. Terminate if the α value exceeds a certain threshold (early ray termination) or if the volume is left. The termination itself is skipped with *texkill* if neither of these conditions is fulfilled.

Termination pass

```
ps_2_0

DEF       c0, -1.0, -1.0, 0.0, 0.0;
DEF       c1, 0.995, 0.0, 0.0, 0.0;

// pixel position
DCL       t4.xyzw;

// frame buffer (red and green)
DCL_2D    s3;
// frame buffer (blue and alpha)
DCL_2D    s4;
// distance buffer (current and maximum)
DCL_2D    s5;
```

```
// ---- copy terminated ----                                ①
// load old distances
TEXLDP    r1, t4, s5;
MOV       r1.zw, c0.z;
// load alpha of old color
TEXLDP    r8, t4, s4;
// calculate distance to end
ADD       r0, r1.x, -r1.y;
// distance -1 means done
MOV       r2, c0;
// remaining transparency
ADD       r3, c1.x, -r8.y;
// ray left volume
CMP       r1, r0, r2, r1;
// do early ray termination
CMP       r1, r3, r1, r2;
```

```
// invert texkill instruction
MOV        r0, -r1;
// only copy terminated pixels!
TEXKILL    r0;
// load red and green of old color
TEXLDP     r7, t4, s3;
// don't care for distances, termination uses z-buffer
MOV        oC0, r7;
MOV        oC1, r8;
```

The *tracing pass* does the same as the *setup pass*, except that it reads the maximum and current ray parameter.

1. Sample the volume data at the current location, get the current sampling distance multiplier $\Delta\omega$ and load the maximum ray parameter.
2. Save old sampling value s_f and normal $\vec{n}_f$, combine the colors of the previous pass and update ray parameter ω.
3. Get new sample value s_b, normalize $\vec{n}_f$ and combine s_f, s_b, and $\Delta\omega$ into a single texture coordinate.
4. Lookup $\kappa\left(s_f, s_b, \Delta\omega\right)$ and $\tau\left(s_f, s_b, \Delta\omega\right)$.
5. Calculate $\alpha\left(s_f, s_b, \Delta\omega\right) = \tau\left(s_f, s_b, \Delta\omega\right)^{\Delta\omega}$.
6. Calculate light intensity $l_d = max(\vec{n}_f \cdot \vec{l}, 0)$ and pre-multiplied emission color $C = \alpha\left(s_f, s_b, \Delta\omega\right)\kappa\left(s_f, s_b, \Delta\omega\right)\left(\frac{1}{2}l_d + \frac{1}{2}\right)$.
7. Since α blending for floating point buffer is not available, the blending is calculated in the shader.
8. Split the color again for output.
9. Output RGBα, current and maximum ray parameter as half-float values.

This rendering pass is carried out with the asynchronous occlusion culling test activated in order to check if there are any fragments still being processed.

Tracing pass

```
ps_2_0

DEF        c0, 0.0, 2.0, 63.75, 1.0;
DEF        c1, 1.0, 0.0, 0.0, 0.5;
DEF        c4, 0.002, 0.3333333333, 0.0, 0.0;

DCL        t0.xyz;
DCL        t1.xyz;
DCL        t2.xyz;
DCL        t3.xyz;
DCL        t4.xyzw;
```

```
// volume
DCL_VOLUME s0;
// leap map
DCL_VOLUME s1;
// pre-int table with correct self attenuation
DCL_VOLUME s2;
// frame buffer (red and green)
DCL_2D     s3;
// frame buffer (blue and alpha)
DCL_2D     s4;
// distance buffer
DCL_2D     s5;
```

```
// ---- setup ----                                                    ①
// load old distances
TEXLDP    r1, t4, s5;
// load old color
TEXLDP    r7, t4, s3;
TEXLDP    r8, t4, s4;
NRM       r0.xyz, t1;
// scaled distance
MUL       r0.xyz, r0, t2;
// reconstruct previous sampling position
MAD       r3.xyz, r1.x, r0, t0;
// scalar value at starting point
TEXLD     r2, r3, s0;
// multiplier for sampling distance
TEXLD     r5, r3, s1;
```

```
// ---- stepping ----                                                 ②
// save old position and add sampling distance to new
// one
MAD       r1.xz, r5.x, c1, r1.x;
// clamp to maximum sampling position
MIN       r1.x, r1.x, r1.y;
// new sampling position
MAD       r3.xyz, r1.x, r0, t0;
// combine old color
MOV       r7.z, r8.x;
MOV       r7.w, r8.y;
```

```
// ---- pre-integration ----                                          ③
// scalar value at end point
TEXLD     r4, r3, s0;
MAD       r2.xyz, r2, c0.y, -c0.w;
// normalize gradient
NRM       r8.xyz, r2;
```

```
// include sampling distance for pre-integration
ADD        r6.w, r1.x, -r1.z;
LRP        r6.xy, c1, r2.w, r4.w;
MAD        r6.z, r6.w, c4.y, -c4.y;

// lookup into pre-integration table                    ④
TEXLD      r9, r6, s2;

// correct opacity for sampling distance                ⑤
MUL        r6.x, c0.z, r6.w;
ADD        r9.w, c0.w, -r9.w;
POW        r10.w, r9.w, r6.x;
ADD        r9.w, c0.w, -r10.w;

// do lighting                                          ⑥
DP3_SAT    r8.w, r8, t3;
MUL        r9.xyz, c1.w, r9;
MAD        r9.xyz, r8.w, r9, r9;
MUL        r9.xyz, r9, r9.w;

// do alpha blending                                    ⑦
ADD        r8.w, c0.w, -r7.w;
MAD        r7, r9, r8.w, r7;

// write pre-multiplied color                           ⑧
MOV        r0, r7.z;
MOV        r0.y, r7.w;

// ---- final ----                                      ⑨
MOV        oC0, r7;
MOV        oC1, r0;
MOV        oC2, r1;
```

The *final pass* is used to copy the floating point values from the two color buffers into the frame buffer after all rays, i.e. fragments, have been terminated. Like the *setup pass* this pass is only used once.

Final pass

```
ps_2_0

DCL        t4.xyzw;

DCL_2D     s0;
DCL_2D     s1;

// load old color (red and green)                       ①
TEXLDP     r0, t4, s0;
```

```
// load old color (blue and alpha)
TEXLDP    r1, t4, s1;
// combine color and write to frame buffer
MOV       r0.z, r1.x;
MOV       r0.w, r1.y;
MOV       oC0, r0;
```

2.1.7 Results

While all of these approaches run on the GPU they greatly differ in both image quality and rendering performance. The Table 2.7 and 2.8 give an overview of all rendering algorithms running under different configurations.

	Radeon 9700 pro		GeForceFX 5600	
	DirectX 9	OpenGL	DirectX 9	OpenGL
simple shading	–[1]	242.42 fps	–[1]	84.10 fps
emulated 3D textures	–[1]	82.16 fps	–[1]	48.34 fps
pre-classified	–[1]	49.96 fps	–[1]	42.53 fps
post-classified	36.20 fps	41.13 fps	15.71 fps	13.43 fps
pre-integrated	30.68 fps	22.86 fps	7.90 fps	7.98 fps
pre-int. lighting	14.51 fps	16.00 fps	5.58 fps	5.99 fps
ray-casting	0.98 fps	–[3]	–[2]	–[3]

Table 2.7: Performance for rendering a 64^3 data set (neghip64) onto a 512^2 screen with all possible rendering algorithms. [1] Some of the possibilities were not implemented because DirectX 9 hardware supports at least post-classified rendering. [2] Needed hardware feature not supported under DirectX 9. [3] Ray-casting was not implemented under OpenGL.

	Radeon 9700 pro		GeForceFX 5600	
	DirectX 9	OpenGL	DirectX 9	OpenGL
simple shading	–[1]	97.13 fps	–[1]	55.75 fps
emulated 3D textures	–[1]	36.22 fps	–[1]	32.36 fps
pre-classified	–[1]	16.22 fps	–[1]	26.52 fps
post-classified	15.68 fps	14.46 fps	7.86 fps	7.84 fps
pre-integrated	10.80 fps	8.74 fps	3.67 fps	5.40 fps
pre-int. lighting	5.15 fps	6.74 fps	2.69 fps	4.19 fps
ray-casting	0.67 fps	–[3]	–[2]	–[3]

Table 2.8: Performance for rendering a 128^3 data set (neghip128) onto a 512^2 screen with all possible rendering algorithms. [1] Some of the possibilities were not implemented because DirectX 9 hardware supports at least post-classified rendering. [2] Needed hardware feature not supported under DirectX 9. [3] Ray-casting was not implemented under OpenGL.

Even if the performance seems worse for the GeForceFX 5600, the image quality is a lot higher than for the Radeon 9700. The main difference seems to be the α-blending of the GeForceFX 5600 that produces images of a quality comparable to the ray-casting images of the Radeon 9700.

2.2 Compression

As already mentioned in the introduction, a representation of the volume data that allows for fast and random access to every voxel has to be used. However, it has also been mentioned that this representation should consume as few memory as possible. In order to fulfill both requirements, a suitable compression scheme has to be used. In image and video compression wavelets are known to produce not only the best image quality/bits per pixel ratio, but also to allow for fast decompression. Therefore this compression will be used in this thesis.

2.2.1 Wavelet Transformation

As basis functions, symmetric biorthogonal spline wavelets [Dau92] are a good choice, as they lead to good compression results (they are also used in the JPEG 2000 standard). The tensor product construction (non standard decomposition [SDS96]) is used to obtain a three-dimensional basis of these functions. This means that the three-dimensional filtering is performed by applying the one dimensional filter in all three dimensions successively. The filtering was implemented to use the integer wavelet transformation algorithm by Calderbank et al. [CDSY96] based on lifting steps. It provides some performance benefits: Firstly, all calculations can be performed using 16 bit integer arithmetic [SS96], saving memory and bandwidth in comparison with the floating point algorithm. The operations can be implemented efficiently using SIMD instructions like MMX. The Intel C++ compiler that applies some of these optimizations automatically has been used. Secondly, the algorithm needs only about half the number of operations of the normal wavelet transformation algorithm.

Throughout this thesis a linearly interpolating spline wavelet will mainly be used. This wavelet basis already allows for a very good compression ratio but it has still a small filter support (5/3 for the lowpass/highpass filter). A small support is desirable as the running time of the (de-)compression algorithm is linear in the number of non-zero entries in the (reconstruction) filter matrix. However, the strongest argument for choosing this wavelet is the property that an increase of the resolution with zero wavelet coefficients leads to a linear interpolation of the low resolution function, which is consistent with the interpolation performed by the texturing hardware used for rendering. For multi-resolution rendering, this results in fewer popping artifacts when the resolution changes.

Symmetric extension [Dau92] is used in this thesis. This means the original data is just mirrored at the border. This allows for a reconstruction without storing additional wavelet coefficients for values outside the block because the basis functions are symmetric.

2.2.2 Entropy Coding

The compression consists of two steps: Firstly, wavelet coefficients of low importance are discarded and secondly, the wavelet coefficients must be encoded in a compact bit stream.

The number of wavelet coefficients to be stored is reduced by defining a threshold below which all coefficients are mapped to zero. Setting the threshold to zero leads to lossless compression: Due to the integer wavelet transform, there is no quantization error [CDSY96]. The fully lossless setting already permits compression ratios of up to 4:1 for typical data sets.

After choosing the relevant wavelet coefficients, they must be encoded efficiently. Code book based approaches such as LZW (Lemple Ziv Welsh) or LZH (Lemple Ziv Huffman) are only useful if the code book is also compressed. Progressive and embeded-ded encoding schemes [EWG99, FF01, MZFM98] on the other hand are not suitable for the difference images of the animated volumes of the multi-resolution hierarchy. To circumvent this, entropy coding with a suitable encoding model is used:

The coefficients are first mapped to positive values: even values represent positive coefficients ($c \rightarrow c \times 2$) while odd values represent negative coefficients ($c \rightarrow c \times (-2) - 1$). For compression of these values, two different algorithms have been implemented. Storing non-zero values with a significance-map combined with arithmetic coding, using the same model as Guthe and Straßer [GS04], is the best choice for maximum compression at a lossless or nearly lossless setting.

A significance-map combined with a fixed Huffman encoder on the other hand results in a very fast decompression, about ten times faster than arithmetic coding. The fixed model for the Huffman coder is defined as follows. Any coefficient is converted into a positive value and stored by using n 1 bits, with n being the minimum number of bits needed to represent the coefficient. After a 0 bit the coefficient is stored using n-1 bits without the first bit.

The compression ratio for significance-map Huffman coding at a lossless setting is lower (in practice about 5-10%) than for arithmetic coding. For a very lossy setting, the run-length Huffman coder is sometimes even able to outperform the arithmetic coder in terms of compression ratio since the adaptive model of the arithmetic coder is optimized for a large number of non-zero coefficients. To obtain higher compression ratios, sub-trees of the hierarchy containing only zero coefficients are completely stripped away. This stripping has more influence on the compression ratio than the coding of the blocks itself. For the compression setting used in the example walkthroughs, the increase of compression ratio is about 3%. This gain increases dramatically if the compression becomes lossier.

2.2.3 Animated Volume Data

To store animated volumes rather than single volumes effectively requires exploiting the temporal coherency between consecutive volumes. The simplest way to do this is storing the difference between the current and the previous volume rather than the current volume. Although this already reduces the compressed data significantly, it is not sufficient for storing large volumetric animations; therefore the motion compensation used for video encoding has to be adapted.

Motion Compensation

The easiest way is to store differential volumes only, as seen in Figure 2.14 (shown for 2D images for clarification). As this is not very effective for the wavelet compression scheme, a simple motion prediction and compensation has been implemented. The motion prediction is done by simple block matching of 8^3 blocks between the two volumes. The motion compensation is done before the differential encoding to reduce the differential content. A block of high similarity, i.e. minimum mean square error, in the previous image is computed by searching this minimum starting from a motion vector of length 0 using 15 steps in all three directions. This is similar to finding the correct motion vector using optical flow methods. This results in 31^3 possible motion vectors that have to be stored using some kind of encoding. The search for the local minimum mean-square error guarantees that most of the resulting motion vectors will be of zero length or at least close to zero length making an arithmetic encoding with a simple adaptive model the best choice as encoder.

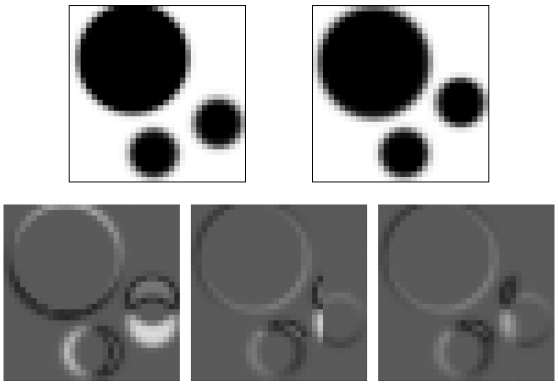


Figure 2.14: Previous image and current image (upper row). Differential image, standard motion compensation and windowed motion compensation (lower row).

The usual, i.e. MPEG, method for computing a motion compensated image is to map each block of the to be constructed volume onto a block of the previous volume as seen in Figure 2.14 for two dimensional images. However in combination with wavelet transformation, this results in severe problems if the motion vectors of two neighboring blocks are different, i.e. regions of high contrast are present. Using wavelet transformation, these high contrasts result in large wavelet coefficients in regions that correspond to high frequencies and therefore low compression ratios. There is another drawback using this simple approach. Due to the nature of the quantization scheme applied to the differential images these wavelet coefficients will be quantized very strongly and therefore result in a large error.

The solution to this problem is the windowed motion compensation introduced by Watanabe and Singhal [WS91]. The blocks are extended to 12^3 overlapping blocks and filtered by a function with a cosine falloff at both ends (see Figure 2.15) in all three directions as the sum of all neighboring window scaling functions and therefore the sum of all voxel weights with the window is already one. Although the computational overhead introduced by these overlapping blocks is with a theoretical value of 2.375 very high, it shows up that this has no severe impact on the performance in practice, due to the higher number of cache hits if the volume is reconstructed voxel by voxel, rather than block by block. The volume that has to be encoded using wavelet transforms does no longer have regions of high contrast, as seen in Figure 2.14 and therefore less wavelet coefficients differ from zero.

Similar to the naming conventions of the MPEG compression [ISO93, ISO96], a wavelet compressed volume is called I-volume (see Figure 2.16). Two different ways to reconstruct an individual volume using motion compensation were implemented. The

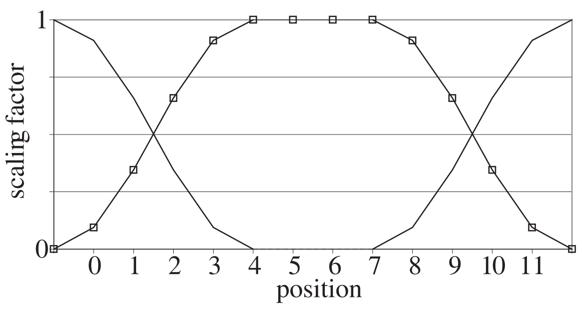


Figure 2.15: Scaling function used for the cosine windowed motion compensation with neighboring windows.

P-volume is reconstructed by using a motion compensated version of the previous I- or P-volume and wavelet compression of the difference between this predicted and the actual volume. The B-volume is reconstructed by using a weighted average between a motion compensated version of the last and the next I- or P-volume similar to the compression scheme used in MPEG compression as seen in Figure 2.16. There is no fixed sequence of I-, P- and B-volumes but any sequence of P- or B-volumes between two I-volumes can be defined. Note that the volumes are not stored in their original order, but in the order of their first usage, i.e. the B-volumes are stored after the next P- or I-volume. In the experiments, various sequences for compression ratio and visual impression were analyzed. It turned out that using the popular MPEG sequence (Figure 2.16c) delivers the best results.

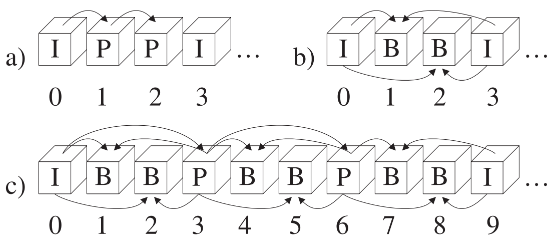


Figure 2.16: Reconstruction and storage order for different types of volumes and sequences, similar to MPEG. a) motion compensated differences b) motion compensation only, c) popular MPEG order

Playback

The single volumes of the animation are decompressed in their storage order rather than in the order of playback. To compensate for the different times needed for the decompression of an individual volume, the decompression of the volumes for each new time step has to be investigated. For this the third sequence in Figure 2.16 has been chosen. In a setup step, the first I-volume and the first P-volume are decoded to make sure that all the volumes for interpolation of the next volume (a B-volume) are present. The second and the third volume are decoded without any special treatment. Reaching the fourth volume, the already decoded P-volume, the next P-volume is decoded as this one is needed for the next B-volume. After decoding the next two B-volumes, the next sequence is being decoded starting with its I-volume. Before displaying the first volume of this new sequence, the next P-volume is decoded and the same state as at the beginning of the decoding is reached. Thus only one volume at a time needs to be decoded, resulting in smoother playback of the volumetric animation.

To achieve an on the fly decompression in real time, further optimizations need to be applied to the decompression. Using LZH coding the bottleneck of the algorithm is writing the decoded wavelet coefficients into their correct position, therefore the compression is restricted to storing the coefficients line by line rather than in the octree depth-first order. Most of the time this reduced the compression ratio slightly but makes better usage of the write cache and thus speeds up the decompression significantly. Using the arithmetic coding, there is an integer multiplication and an integer division as part of the main interval coder and therefore no need to optimize the writing of the wavelet coefficients in terms of cache hits, as this does not result in any noticeable speedup.

2.2.4 Large Volume Data

So far the decompression order and the data to be decompressed were defined by the frame numbering. For large volume data however, there is a different access scheme.

Assuming that all parts of the volume that are needed for producing the final image need to be decompressed, only a subset of the complete volume is needed. Furthermore, the voxel will be processed in a certain order, either front-to-back or back-to-front on a per ray or per slice basis.

Combined with the properties of the wavelet compression, the resulting data structure is more like a tree of wavelet coefficients than a monolithic compressed block. Therefore the compression scheme used for animated volume data has to be modified in order to allow for better support of large volume data.

2.2.5 Results

All tests have been carried out on an AMD K7 running at 800 MHz using a GeForce 2 graphics adapter (first configuration) or on an AMD K7 running at 1000 MHz using a Radeon graphics adapter (second configuration).

As expected the usage of higher order wavelets does not only reduce the size of the compressed volumes while enlarging the Peak-Signal to Noise Ratio (PSNR) as seen in table 2.9, but does also significantly improve the visual impression (see Figure 2.17 and 2.18) and better preserves features. The original volume can be seen in the color plates.

wavelet	I-vol.	P-vol.	B-vol.	ratio
Haar	47.363	47.022	40.613	1:27.30
Daubechies 4	48.673	47.784	40.756	1:30.96
Coiflets 6	48.615	47.751	40.750	1:33.27
CDF 2/6	49.015	47.990	40.787	1:34.28

Table 2.9: PSNR comparison between different wavelets using maximum quality, the popular MPEG like sequence and arithmetic compression.

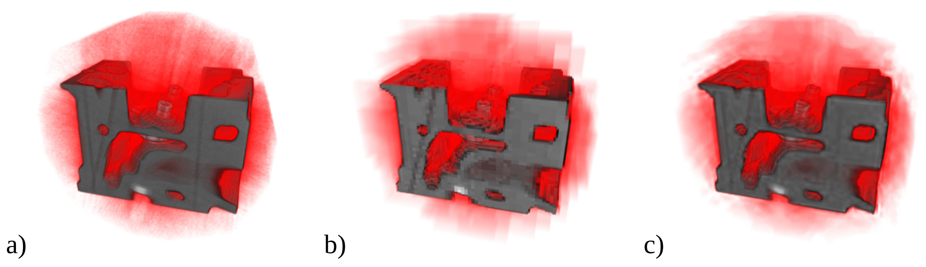


Figure 2.17: Original engine data set (a) and compressed data set at ratio 200:1 using the Haar wavelet (b) and a CDF wavelet (c).

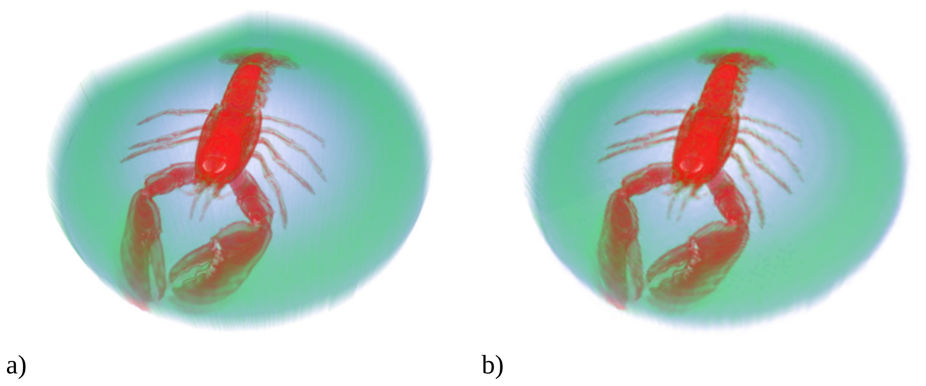


Figure 2.18: Original lobster data set (a) and compressed data set at ratio 100:1 using the CDF wavelet (b).

The PSNR of I- and P-volumes also depends on the compression ratio as seen in Figure 2.19. The quality of the B-volumes on the other hand does only roughly depend on this quality setting (see Figure 2.19), as these volumes are not reconstructed using wavelet transforms but using motion compensation only.

As already mentioned, the decompression times achieved using arithmetic encoding depend on the quantization used, as both the number of zeros increases and the depth of the tree that has to be updated dynamically decreases. Using LZH encoding, neither of these two properties does have any effect on the decompression times as seen in Table 2.10. The LZH encoding performs a lot better than the arithmetic encoding in terms of speed (up to two times faster using high quality settings) but worse in terms of compression ratio (about 30% more compressed data at moderate to high quality). This allows for about 4 frames per second regardless of the chosen quality. Note that the speed on the second configuration is not limited by the decompression time, but rather by the memory bandwidth during the 3d wavelet transform.

As seen in Figure 2.20 the PSNR of the B-volumes (the dotted line) is significantly lower for high quality volume animations. Removing the motion compensation at these high quality settings results in an improvement of the PSNR but also increases the size of the compressed data, as seen in Table 2.11. At a lower quality setting, the

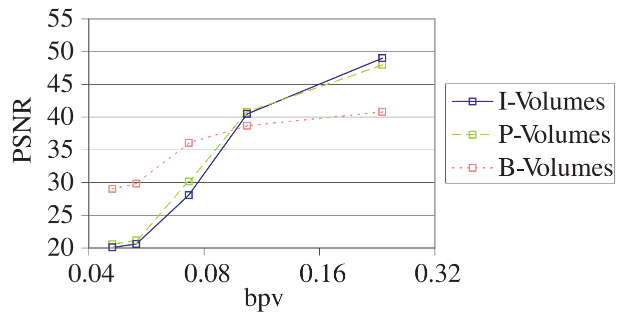


Figure 2.19: Average PSNR of each different kind of volume against bits per voxel (CDF wavelet with support 2/6 and popular MPEG like sequence).

quality	arithmetic	fps1	fps2	LZH	fps1	fps2
255	34.28:1	1.58	1.77	25.89:1	3.23	3.62
127	77.28:1	2.06	2.34	56.10:1	3.25	3.62
63	109.69:1	2.35	2.74	82.32:1	3.25	3.62
31	150.34:1	2.64	3.01	134.59:1	3.27	3.62
15	173.64:1	2.80	3.17	171.15:1	3.28	3.62

Table 2.10: Comparison regarding compression ratio and frames per second (using 2d textures on system 1 and 2) between different encoders using various qualities, the popular MPEG like sequence and the CFD wavelet with support 2/6.

PSNR increases if motion compensation is used again while also decreasing the size of the compressed data. Testing different qualities and sequences demonstrated that the popular MPEG sequence (the first sequence in Table 2.11) should always be used with only one exception. If a nearly lossless compression of every volume is desired, rather than every third volume, only I-volumes or I-volumes and P-volumes should be used as this will result in the highest possible PSNR.

sequence	ratio (255)	PSNR	ratio (15)	PSNR
IBBPBBPPB	34.28:1	43.308	173.64:1	26.159
IBB	28.84:1	43.702	160.70:1	26.040
IPP	13.77:1	48.408	168.85:1	20.396
I	10.09:1	49.004	153.43:1	20.089

Table 2.11: Comparison between different sequences using CDF wavelet with support 2/6, arithmetic encoding and maximum (intermediate) quality.

If the compression of a volume animation without generating too many noticeable visual artifacts is wanted that will playback at interactive frame rates of about 4 frames per second on the testing configurations, compression ratios of about 50:1 are achieved using CDF wavelets, a quality setting of about 127, the popular MPEG sequence and LZH encoding. If high compression ratios are needed rather than fast visualization, a ratio of about 75:1 can be reached by replacing the LZH encoding by arithmetic encoding without any additional loss of data. Using a transfer function of high contrast will emphasize visual artifacts. The example animation (Figure 2.21) has an absolute

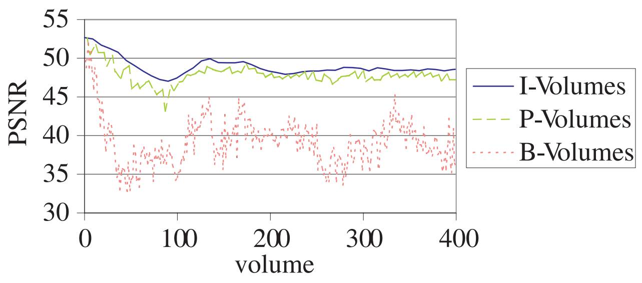


Figure 2.20: PSNR of each kind of volume of the first 500 volumes of a 2000 volume animation (CDF wavelet with support 2/6, maximum quality and popular MPEG like sequence).

derivative of 5 in the alpha component of the transfer function. A transfer functions of lower contrast will allow for a compression ratio of 100:1 and beyond without visual artifacts.

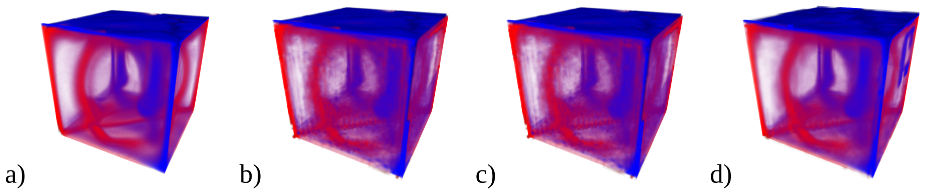


Figure 2.21: Original volume 1000 of an animation of 2000 volumes (a). I-only sequence, quality 63, compression ratio 31:1, I-Volume (b); P-only sequence, quality 63, compression ratio 50:1, P-Volume (c); MPEG sequence, quality 63, compression ratio 110:1, B-Volume(d) of the same animation

2.3 Multi-Resolution Rendering

Firstly, the data set is divided into cubic blocks of $(2k)^3$ voxel (in practice, $k = 16$ is a good choice). Then, the wavelet filters are applied to each block. This results in a lowpass filtered block of k^3 voxel and $(2k)^3 - k^3$ wavelet coefficients representing different high frequency components that are no longer present in the lowpass filtered block (see Figure 2.22). This scheme is carried on hierarchically: A cube of 8 adjacent lowpass filtered blocks is grouped to again obtain a block of $(2k)^3$ voxel. Then the filtering algorithm is applied to this block recursively until only a single block is left. The result of this procedure is an octree (see Figure 2.22): Each node of the octree describes a volume of k^3 voxel and contains a set of high frequency coefficients that allow for the reconstruction of the child nodes from the current node. The resolution of a child node is twice as high (in each dimension) as that of a parent node. The lowpass filter of the specific wavelets that is used assures that the downsampled data in the inner nodes does not show relevant aliasing artifacts.

From the perspective of the rendering algorithm, a representation of the volume

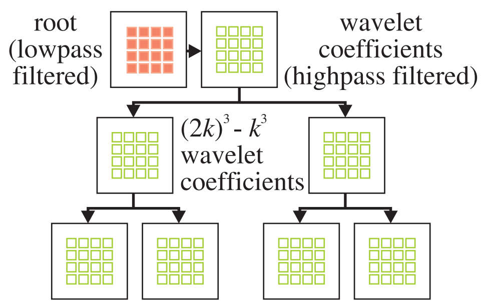


Figure 2.22: Representation of a large volume data set as a wavelet tree.

data in form of a multi-resolution octree has been constructed: The root node in the tree contains a very rough approximation of the data set and the resolution can be increased by a factor of 2 (in each dimension simultaneously) by going downwards the hierarchy to a child node. The task is to extract the information relevant for a certain point of view. This is done in two steps: Firstly, a projective classification step is performed to adjust the resolution of the data set to the screen resolution. Secondly, a consideration of the approximation error is incorporated into the classification algorithm to further reduce the amount of data to be processed in each frame. After extracting a suitable level of detail from the wavelet tree, the volume data is rendered using hardware texture mapping. Rendering of walkthrough animations can be accelerated substantially by applying a suitable caching scheme.

2.3.1 Multi-Resolution Hierarchy

The first step of the algorithm is to convert the volume data, which is given as a three-dimensional array of integers with fixed precision (usually 8-16 bits), into a compressed wavelet representation during preprocessing. This representation is much more compact and allows for an efficient extraction of different levels of detail of the data set, since the wavelet transformation is equivalent to applying a series of lowpass and highpass filters to the original data. To be able to decompress parts of the data set efficiently, a blockwise wavelet compression strategy is applied.

How efficient is octree-based projective classification? To answer this question it is assumed that the volume is discretized into voxel of arbitrary size (see Figure 2.23). Parameters to the algorithm are a camera position and a constant vertical viewing angle of α. It is also assumed w.l.o.g.[1] that the original resolution of the voxel grid exactly matches the display resolution of $w \times h$ pixel at the near clipping plane z_{near}. To cover the whole volume, m layers of resampled cube-shaped voxel are added with side length $voxelsize\,(i)$, $i = 1..m$, so that the projected size of the larger voxel still matches the display resolution. Let z_i be the depth of voxel layer i. Then obviously $z_{i+1} = z_i + voxelsize\,(i)$ and

$$voxelsize\,(i) = \frac{\tan\frac{\alpha}{2}}{\frac{h}{2}} z_i = q \cdot z_i \text{ with } q = \frac{\tan\frac{\alpha}{2}}{\frac{h}{2}}. \qquad (2.1)$$

[1]There is no problem if the near clipping plane is closer to the viewer: As the discretization in voxel is never finer than the original resolution of the data set, there are always less than $w \cdot h \cdot \cot\alpha \in \mathrm{O}(1)$ voxel in front of z_{near}.

This recurrence leads to $z_i = z_{near}(1+q)^i$. Let $z + far$ be the largest depth of a voxel in the volume. Then the number of layers of resampled voxel is bound to:

$$m = \frac{\log \frac{z_{far}}{z_{near}}}{\log(q+1)} \tag{2.2}$$

Thus, the number of resampled voxel is $m \cdot w \cdot h$. Note that the ratio $\frac{z_{near}}{z_{far}}$ is always bounded by the maximum diameter of the data set (measured in voxel). For a volume of n^3 voxel the diameter is at most $3n \in \mathrm{O}(n)$. Therefore, a total amount of $\mathrm{O}(\log n)$ resampled voxel is obtained.

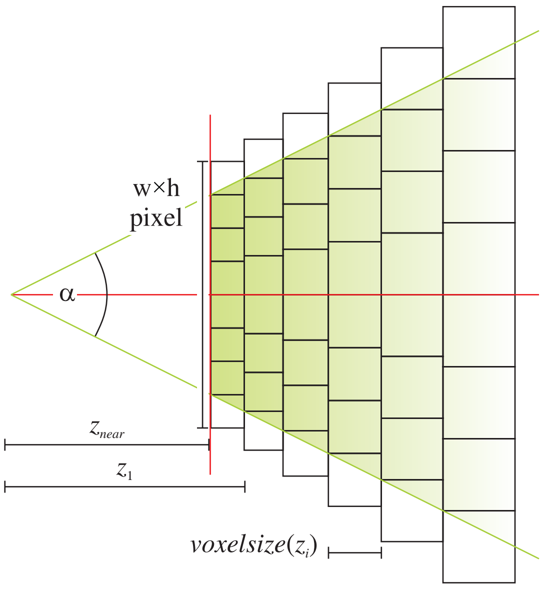


Figure 2.23: Analysis of the projective classification strategy.

Up to now, the analysis still neglects the fact that access to resampled voxel of arbitrary size is not possible but only octree nodes. This leads to two different kinds of overhead: Firstly, this enforces the use of blocks with k^3 voxel (typically $k = 16$) of the same, fixed resolution. Secondly, the resolution can only be chosen in powers of 2 (in each dimension). Considering the overhead due to the blocking first produces: Using some elementary trigonometry, it can be seen that the number of voxel per unit length does not increase by more than a factor of

$$\rho_{max} = 1 + \sqrt{3}\frac{2k\tan\frac{\alpha}{2}}{h} \tag{2.3}$$

between the foremost and the most distant voxel in each block. The bound can be derived by considering blocks diagonal to the viewing direction and comparing the number of voxel per unit length. The voxel density per unit area is given by the density per unit length squared. Thus, the average factor of increase of voxel due to the blocking in blocks of k^3 voxel is given by:

$$overhead_{block} = \frac{\int_1^{\rho_{max}} x^2 \mathrm{d}x}{\rho_{max} - 1} = \frac{\rho_{max}^3 - 1}{3(\rho_{max} - 1)} \tag{2.4}$$

For typical block sizes k, this leads only to a small overhead ($h = 256$, $\alpha = 45°$):

k	8	16	32	64	128
overhead	4.6%	9.2%	19.0%	40.2%	88.9%

However, the overhead is increased due to the fact that the resolution can be changed only in powers of two. This is easy to quantify: Assuming that all scales of resolution between 1^3 and 2^3 voxel are needed with equal probability, the average oversampling factor is $\int_1^2 x^3 dx = 3.75$. This factor usually dominates due to the blocking. Example: For a resolution of 256^2 pixel, $90°$ vertical viewing angle, and a depth of 2048 voxel 858 layers containing 56 million resampled voxel are obtained. The approximation with an octree with blocks of 16^3 voxel increases the amount of voxel to at most 230 million voxel. A 2048^3 data set contains 8.6 billion voxel. In conclusion, it can be seen that projective classification using an octree leads to a running time logarithmic in the size of the input data. However, the constants hidden in the O-notation are fairly high. Thus, the algorithm scales very good but additional techniques are necessary to obtain interactive performance.

2.3.2 Projective Classification

Firstly, nodes from the octree need to be extracted so that the resolution of these nodes matches the display resolution. Nodes outside the view frustum should be excluded from rendering. The task can be done using a straightforward algorithm originally proposed by Chamberlain et al. [CDL$^+$96]: The hierarchy is traversed recursively, starting from the root node. Each node is tested whether it is located completely outside the view frustum. In this case, the traversal is stopped, ignoring the current node. Otherwise, the spacing between the voxel grid and its projection to the screen is determined. If it is equal to or below the screen resolution, the node is passed to the renderer. Otherwise, if the voxel resolution is still too coarse, the node is subdivided and the algorithm is applied recursively to all 8 children.

This technique was already applied to volume data by LaMar et al. [LHJ99]. It has already been proven that the technique reduces the rendering time for an n^3 voxel grid from $\mathrm{O}\left(n^3\right)$ to $\mathrm{O}\left(\log n\right)$. However, the analysis also shows that the constants hidden in the O-notation are very high. For a close-up of a volume with a depth of 2048 voxel, still more than 230 million voxel are obtained after projective classification. This is about 4 times more than the texture memory of a typical contemporary graphics board (230MB versus 64MB). Therefore, a refined classification criterion for a further reduction is needed.

2.3.3 View-dependent Priority Schedule

In most data sets, only a few regions contain high frequency details (e.g. due to sharp borders). Most regions can be sampled at a low sampling rate without sacrificing detail resolution. This observation is utilized to reduce the amount of voxel that has to be processed by the renderer: For each node in the wavelet tree, the L2 error compared to the original data is measured during compression. During rendering this error is used as weight for the selection of nodes: Let $E(i)$ be the L^2 error of the normalized basis functions for the wavelets in the subtree below the node i. For leaf nodes, $E(i) = 0$. To each node i a priority $P(i) = \frac{E(i)}{z(i)}$, with $z(i)$ being the minimum depth of a voxel in the node is assigned. Dividing by $z(i)$ accounts for the projection on the screen:

The priority of nodes near the viewer should be higher than that of nodes far away. If $z(i) = 0$, the priority is set to infinity.

Using this priority function, a generalized projective classification is performed: A maximum amount of voxel that the rendering is able to process is chosen. This is usually determined by the texture memory of the graphics board. A priority queue is created and the root node r of the hierarchy is inserted into the queue with priority $P(r)$. Then the node with the highest priority is successively fetched from the queue, its high frequency wavelet coefficients are decompressed and the child with the highest priority is inserted into the queue. A flag is set for the node to indicate that the child node has been added to the queue (all other children would still be drawn using the low resolution representation from the parent node). If all children are in the priority queue, the parent node is removed from the queue. Nodes with a projected voxel distance that is already equal to or below the screen resolution are not subdivided. The algorithm stops if the maximum amount of voxel for the nodes in the queue is reached.

2.3.4 Error Estimation

To increase the rendering quality without any further impact on the performance of the algorithm, it will first be discussed how to choose the optimal working set for rendering. To find this set the simple greedy strategy is extended by finding the block that produces the most error and replace it with higher resolution blocks. Therefore the rendered image using the current set against the image rendered using the blocks that map one voxel to a single pixel need to be compared. As can easily be seen this is even slower than rendering the image in full resolution in the first place, so the screen-space error needs to be estimated in some way.

For estimating the screen-space error, an appropriate error metric has to be defined. The screen-space error used in this thesis is defined as the difference between the final color of a pixel using a low resolution block compared against the final color using the highest applicable resolution. This difference is the sum of the three color channels and the remaining transparency of each block. This error therefore depends on the current transfer function (color C and density D) in a natural way. In order to reduce the error, only the maximum error produced by each block multiplied by its importance, i.e. size in screen-space, is of interest. Since the maximum error per block err is the difference between the color err_C plus the difference between the opacity err_O, it can be calculated by

$$\begin{aligned}
err_O &= \left| e^{-\int_0^d D\left(V_l(x)\right)\mathrm{d}x} - e^{-\int_0^d D\left(V_h(x)\right)\mathrm{d}x} \right|, \qquad (2.5)\\
err_C &= \left| \int_0^d D\big(V_l(x)\big)C\big(V_l(x)\big)e^{-\int_0^x D\left(V_l(x)\right)\mathrm{d}t} \right. \\
&\quad \left. -D\big(V_h(x)\big)C\big(V_h(x)\big)e^{-\int_0^x D\left(V_h(x)\right)\mathrm{d}t}\mathrm{d}x \right|, \\
err &= err_O + err_C.
\end{aligned}$$

To calculate this error, both integrations have to be carried out for each pixel in the final image. Since this is equivalent to render the complete image twice using ray-casting, a conservative estimation has to be used. Assume that each voxel in the high resolution blocks is known and the corresponding voxel value in the low resolution block. Since, the integral is a linear operation, the error can be estimated as the maximum over all possible color and opacity combinations given by

$$\begin{aligned} err'_O &= \left|e^{-d\,D_l} - e^{-d\,D_h}\right|, \\ err'_C &= \left|C_l\left(1 - e^{-d\,D_l}\right) - C_h\left(1 - e^{-d\,D_h}\right)\right|, \\ err &\leq err'_O + err'_C. \end{aligned} \tag{2.6}$$

However, the corresponding child block would still have to be decompressed. Storing the maximum deviation of each voxel value in the low resolution block, the high resolution data is no longer needed and the estimation can be simplified even more. With C_{min}, C_{max}, O_{min} and O_{max} as the component wise minimum and maximum of the transfer function within the deviation results in:

$$\begin{aligned} err''_O &= e^{-d\,D_{min}} - e^{-d\,D_{max}} \\ err''_C &= \left(C_{max} - C_{min}\right)\left(1 - e^{-d\,D_{max}}\right) + C_{min}\,err''_O \\ err &\leq err''_O + err''_C \end{aligned} \tag{2.7}$$

With the maximum of err''_O at $d = \frac{\ln \frac{D_{max}}{D_{min}}}{D_{max} - D_{min}}$ and the maximum of err''_C at d being the diagonal of the block. If the value of d for the calculation of err''_O is larger than the diagonal of the block, eg. $D_{min} = 0$, the diagonal is also used for calculating this error.

Storing the values C_{min}, C_{max}, O_{min} and O_{max} into a two-dimensional table, the maximum error for all voxel values present in the low resolution block can now be calculated, but the amount of memory needed for this is still twice the size of the decompressed voxel. To store the deviation of all voxel values efficiently, a small histogram that contains the maximum positive and negative deviation for a small range of values is build. In practice a histogram with a fixed number of 8 entries showed up to be very efficient. The calculation of the estimated screen-space error is very costly since all possible combinations for each voxel value in the given range have to be considered. To speed up the calculation, another two-dimensional table is used that stores the minimum and maximum colors and opacities for a given range. Therefore only a single calculation has to be done for each interval in the given range of voxel values. On the other hand this again leads to a certain overestimation. Since the errors for a given resolution only change whenever the transfer function changes, the calculation can be sped up by storing the error produced by each octant prior to the multiplication with its size in screen-space with the decompressed block. The resulting space usage of all additional data is only 68 bytes per compressed block and an additional 40 bytes for each uncompressed block.

2.3.5 Visibility Testing

After selecting the appropriate resolution for each block, the visible blocks have to be rendered in a front-to-back or back-to-front order. During this traversal each block can be tested whether it is completely transparent under the current transfer function. If so, it is skipped completely without any further processing. The reduced depth complexity can be seen in Figure 2.24. Using this technique, the rendering performance increases by about 35%. A per sample alpha test after applying the transfer function can be used to skip the alpha blending, however this only delivers a speedup of another 1%.

Up to now, the rendering is still waiting more than 80% of the time for the graphics hardware, so more invisible blocks, in this case the occluded ones, need to be removed.

Taking occlusion into account on a per block basis is a bit more complicated than detecting transparency, especially if a back-to-front rendering order is used. Therefore the occlusion calculation is done prior to the rendering. Calculating the exact occlusion of each block is similar to render a complete image using ray-casting. Although no shading of any kind has to be taken into account, this step is too slow to gain a speedup. However exact occlusion information is not needed, but rather a number of blocks that are not visible. In order to approximate the occlusion, a uniform opacity for each block is assumed, i.e. to be on the safe side, its minimum opacity is used. Then a software ray-caster based on cell projection, similar to the one presented by Mora et al. [MJC02], is used to calculate an occlusion map at a fixed resolution. During the cell projection there are some cases to consider. Blocks that have a minimum opacity below a certain threshold only have to be tested for their visibility since they don't change the occlusion map. This test can also be skipped until the first block with opacity above this threshold has been processed. If a block is found to be occluded, the occlusion map also does not need to be updated. Therefore only very few updates of the occlusion map are typically necessary. With the current implementation based on optimized SSE2 code, an occlusion map of 256^2 pixel showed up to be a good value for all possible cases, ranging from no occlusion to very high occlusion.

Although this is a very rough approximation, the depth complexity can be reduced, as seen in Figure 2.24. To reduce the traversal costs the block based empty space skipping is done during the selection of the level-of-detail. Each of these optimizations reduces the number of shaded samples by about 30%. The transfer function used for the example in Figure 2.24 is not an iso-surface transfer function, but rather a linear ramp.

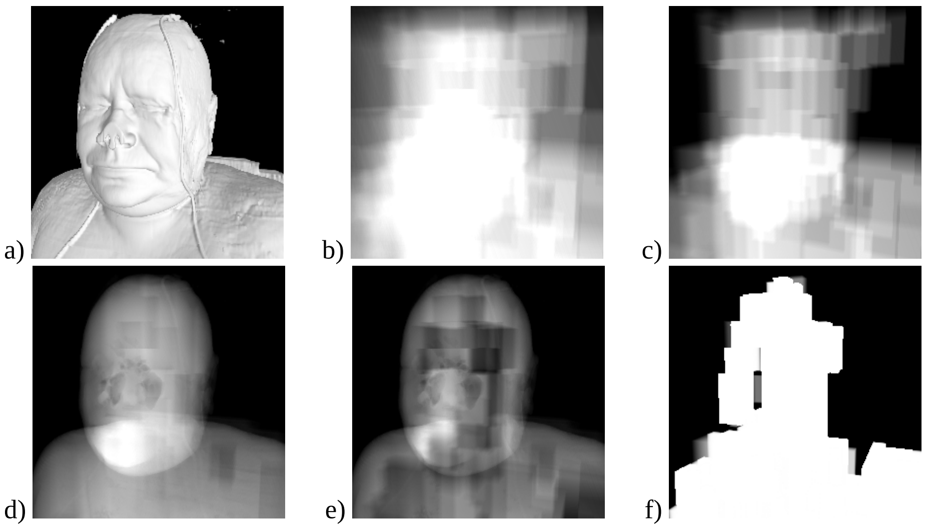


Figure 2.24: Depth complexity for a given image (a) without optimizations (b), with block based empty space leaping (c), alpha test (d) and occlusion culling (e) with corresponding occlusion map (f). About 30% of all sample points have been removed with each optimization.

2.3.6 Rendering of Blocks

Up to now, a set of tree nodes was chosen, each containing k^3 voxel (on a regular grid) that provide a suitable approximation to the original volume for the current view point. To render these voxel, hardware texture mapping is used: All blocks are drawn in back-to-front order. The order can be established easily by enforcing a back-to-front traversal order of the octree. For each block, a 3D-texture is created and loaded onto the graphics hardware. Each block is rendered using one of the previously discussed shading approaches.

As hierarchy blocks of different resolution also have different slice spacing, a different transfer function, i.e. pre-integration table, is stored for each possible slice spacing. The tri-linear interpolation performed by the texturing hardware needs special attention: The hardware is not able to interpolate across the borders of the octree blocks. This can lead to objectionable artifacts that reveal the underlying block structure. The solution to this problem is straightforward: For each block to be rendered, the algorithm also fetches its 7 neighbors with the next higher x-, y- and z-coordinates from the octree (Figure 2.25). If these nodes are not present in the rendering set, the corresponding node is also decompressed and cached, but the neighbor's neighbors are of course not reconstructed.

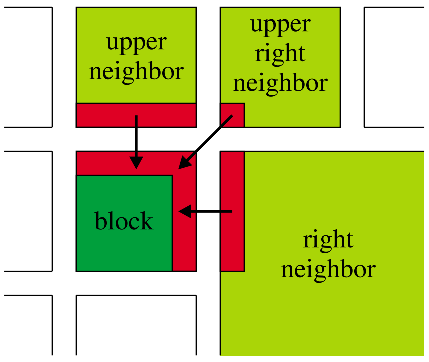


Figure 2.25: Copying data from neighbors for the 3D-texture blocks.

This lookup is not very expensive as a neighbor search in an octree can be done in expected time of $\mathrm{O}(1)$. The block to be rendered is enlarged by one voxel in x-, y- and z-direction and we store the neighboring values there[2]. Using the additional voxel, the rendering can perform a continuous linear interpolation (Figure 2.26). The texture memory necessary for rendering is increased by this technique because adjacent blocks overlap each other by one voxel. The overhead is $k^3 - (k-1)^3$ for k^3 voxel. For 16^3 voxel blocks an overhead of 21% is obtained. For 32^3 voxel blocks, the overhead is only 10%. In addition to this, the gradient information necessary is computed on-the-fly after the decompression of the volume data using a three-dimensional Sobel operator.

[2]As textures must have extents of a power of two, blocks of size $2n - 1$ in the wavelet tree (e.g. 15^3 voxel) have to be used.

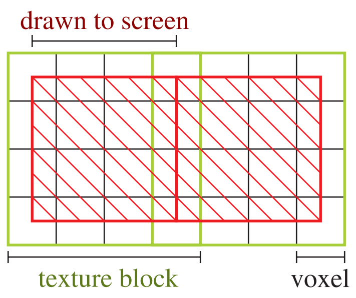


Figure 2.26: Texture interpolation, the blocks overlap each other by half a voxel.

2.3.7 Caching Strategy

Although the wavelet decompression algorithm already achieves a very high performance, it would not be able to perform an interactive walkthrough if it decompresses the wavelet representation for each frame from scratch. It is not possible to perform a decompression and texture upload at a similar speed as the 3d texturing is done by the graphics card on current hardware architectures. Fortunately, this is not necessary either, as the algorithm may anticipate reusing most of the decompressed data for subsequent frames. Therefore, three cache areas are used to store blocks for reuse:

Firstly, decompressed volume blocks from the octree are cached. To obtain a node in the octree, its parent node needs to be accessed, the high frequency coefficients stored in the node have to be decompressed and the reconstruction filter to obtain all 8 child nodes has to be applied. The nodes consist of blocks of k^3 16 bit integers. The decompressed wavelet coefficients are not cached as these are only needed once to obtain the child nodes which are already cached. Caching is done according to an LRU-scheme. To maximize the performance of the algorithm, the user defines a fixed amount of cache memory. If the algorithm runs short of memory, it always deletes the decompressed leaf node in tree that was not accessed by the renderer for the longest time.

Secondly, the algorithm has to create 3D-textures from the cache. The texture contains the scalar values and optionally the corresponding gradient field for advanced shading effects[3]. Using again an LRU scheme, the most recently used subset of decompressed blocks is fetched and converted into OpenGL texture objects. Gradient maps are computed at this point, if necessary.

Thirdly, the texture objects must be uploaded to the texture memory of the graphics adapter before rendering. This is done automatically by the OpenGL driver, again using an LRU caching scheme. By setting corresponding memory restrictions, the renderer assures that it does not use more texture objects per frame as fit into a given amount of video memory, thus avoiding cache thrashing.

2.3.8 Results

In this section, the results obtained with a prototype implementation of the proposed algorithm are discussed. The algorithm was implemented in C++ using OpenGL with

[3]The gradients are stored as 8 bit RGB values and the scalars are stored in the alpha channel of the RGBα texture. The shading is done using pixel shaders similar to the approach of Meissner et al. [MGS02].

extensions or DirectX 9 for rendering. All benchmarks were performed on a 3Ghz Pentium 4 PC with 1GB of Ram and an nVidia GeForce FX 5800 Ultra or an ATI Radeon 9700 graphics board with 128MB of local video memory. This section starts with a description of three example data sets that were used to evaluate the algorithm. Then, the influence of the compression efficiency on the running time and image quality is discussed. After that, the results for interactive examination of the three example data sets are presented.

Example Data Sets

Three different data sets were used for the evaluation of the algorithm. All three are too large to be visualized at interactive frame rates using conventional brute-force rendering approaches.

The first data set is a computer tomography scan of a Christmas tree [KTM$^+$02] at a resolution of $512 \times 512 \times 999$ voxel with 12 bits per voxel. The data set was acquired at the Technical University of Vienna to provide a large benchmark scene for volume rendering algorithms. The other two data sets are the visible human male and female data sets [Nat86]. Both are computer tomography scans of a male and a female human body. The variants of the data sets that are registered against the cryosection RGB images were used in this thesis. The visible human male data set has a resolution of $2048 \times 1216 \times 1877$ voxel and the visible human female data set has a resolution of $2048 \times 1216 \times 1734$ voxel. The example renderings were made using gradient based lighting and a classification function with several semi-transparent iso-surfaces. The iso-surfaces correspond to high derivatives in the classification function. These settings are very sensitive to noise and other reconstruction errors in the volume data and thus allow a good evaluation of the errors introduced by the rendering technique.

Compression Efficiency

In the compression algorithm, different encoding algorithms can be used for the wavelet coefficients. Two alternatives haven been implemented here: arithmetic coding and run-length Huffman coding. The decompression speed heavily depends on the compression algorithm. Using arithmetic coding, a decompression speed of 4.5 MB/s can be achieved, including the wavelet reconstruction. The run-length Huffman codec is able to decompress 50 MB/s (including the wavelet reconstruction). The compression ratio of the arithmetic coding is typically only about 10% to 15% higher than that of the run-length Huffman coding. Therefore, the run-length Huffman coding has been used for all examples in this thesis.

A second parameter of the compression algorithm is the threshold for removing small wavelet coefficients prior to encoding. If all coefficients are kept, a lossless compression scheme is obtained. Using lossless compression, a compression ratio of 3.9:1 (arithmetic coding) and 3.4:1 (RLE-Huffman coding) can be achieved for the Christmas tree data set. The visible human data sets could not be compressed using the lossless settings because the compressed data and the caches would exceed the 2GB address space. For higher compression ratios, a lossy compression needs to be applied: Figure 2.27 and 2.28 show the dependency between compression ratio and reconstructed signal quality for the three different test data sets: A peak signal-to-noise ratio (PSNR) of 60 dB for a compression ratio[4] of about 12:1 (1 bit per voxel) were obtained, while a PSNR of 50 allows a compression ratio of roughly 50:1 (0.25 bits

[4]All compression ratio measurements are based on 12 or 3×8 bit data sets.

per voxel). Figure 2.29 shows a visual comparison of the rendering results for the Christmas tree data set. The compression ratios obtained by the algorithm at a given PSNR are close to the results of Nguyen and Saupe [NS01]. These results show that it is possible to achieve good compression results although only linear interpolating wavelets and blockwise compression were used.

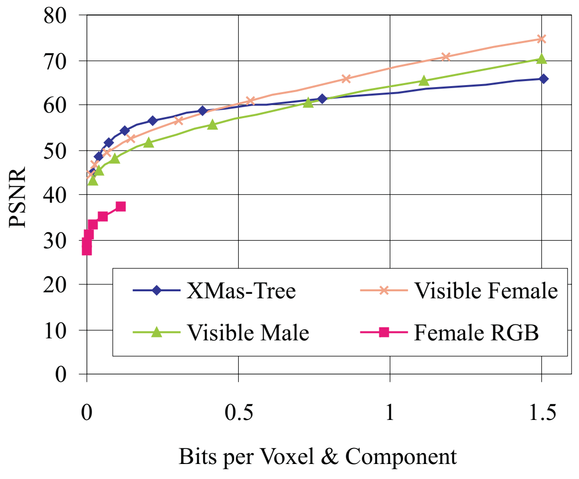


Figure 2.27: PSNR for Christmas tree data set, and the visible human data set.

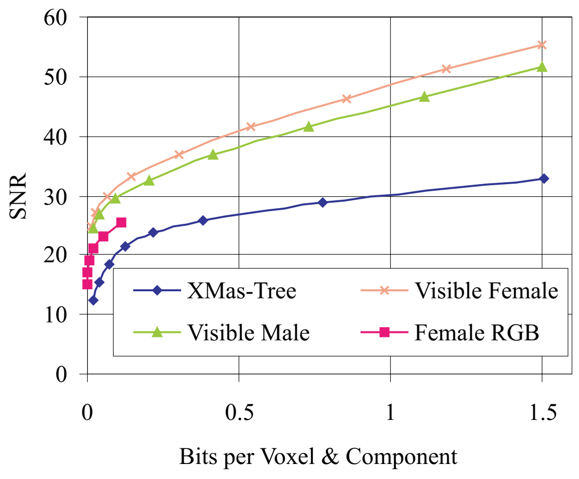


Figure 2.28: SNR for Christmas tree data set, and the visible human data set.

Another important parameter is the block size used for the construction of the wavelet hierarchy. If small blocks are used, the algorithm is able to classify the data according to local frequency spectra and projected size very accurately. However, the algorithm has high hierarchy traversal costs. If larger blocks are used, the traversal costs decrease but the algorithm must process more voxel for the same image quality because the classification is less accurate. Additionally, the block size must be a power

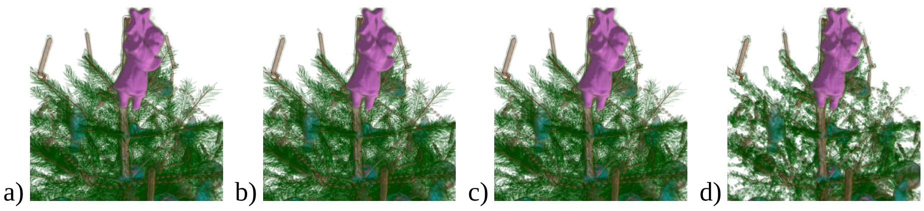


Figure 2.29: Quality comparison for different compression ratios for the Christmas Tree data set.

of two (minus one, for neighboring voxel) due to OpenGL restrictions. In practice, 31^3 blocks are not adaptive enough and 7^3 blocks introduce too much overhead. 15^3 blocks are a good compromise. This block size is used in all examples in this thesis.

Interactive Walkthroughs

The algorithm was used to render an interactive walkthrough of the three test data sets. The results are shown in Figure 2.30 and 2.31. The resolution of the output image is 512^2 pixel for all tests. The Christmas tree data set was compressed using lossless compression (3.4:1), the visible human data sets were compressed using lossy compression. (40:1 for the female and 30:1 for the male data set). The preprocessing time was 1 hour for the Christmas tree and about 5 hours for each of the two visible human data sets. The preprocessing times are dominated by hard disk access (seek times). The CPU-utilization was only 6-7% during compression.

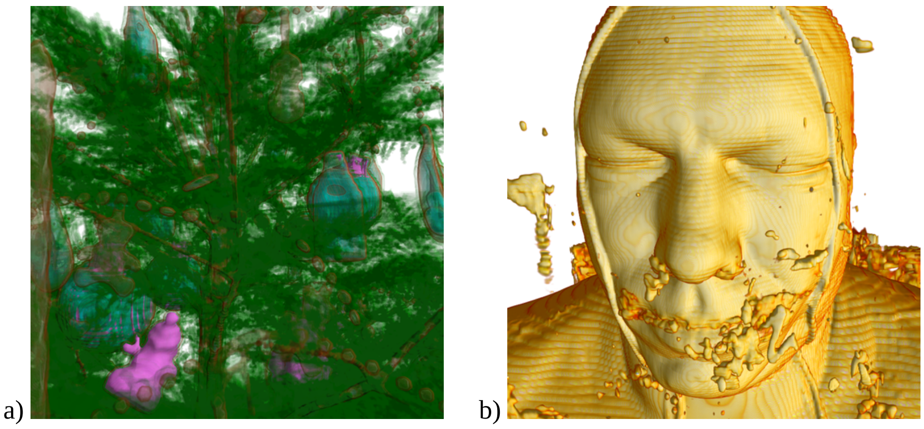


Figure 2.30: Christmas Tree data set (a) and visible human male data set (b) rendered at interactive frame rates.

During the walkthrough, the quality parameter can be adjusted to trade off image quality for rendering speed. The quality parameter is given as the maximum projected error value for the rendered hierarchy nodes. Three different settings with high, medium and lower image quality have been used. The high quality settings uses up to 2048 blocks and a maximum projective error of $\frac{1}{128}$ (an average error of $\frac{1}{128}$ of the

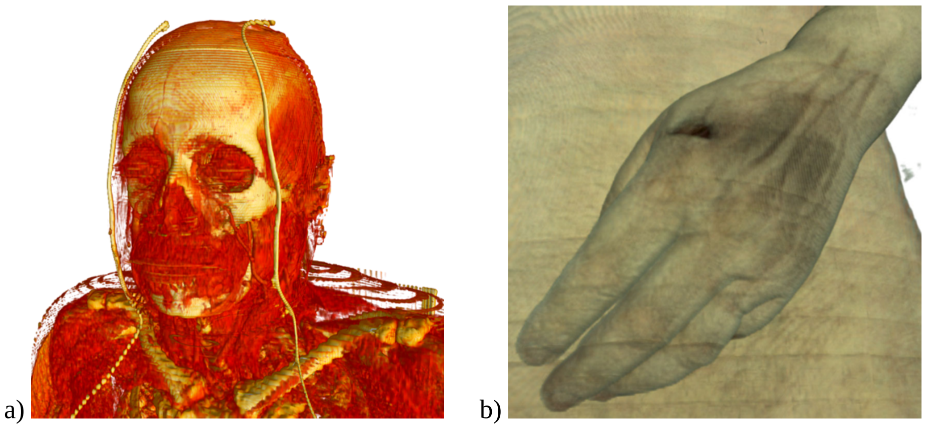


Figure 2.31: Visible human female data set (a) and visible human female RGB data set (b) rendered at interactive frame rates.

peak signal per pixel for each block) and therefore shows only very little artifacts due to a reduced resolution. Nevertheless, an average frame rate of 3-4 frames per second is obtained during the walkthrough. The low quality settings uses only 512 blocks and a maximum projective error of $\frac{1}{32}$ thus permitting frame rates of about 10 frames per second at an acceptable image quality. The medium quality setting is a good compromise with 1024 blocks and a maximum projective error of $\frac{1}{64}$: The image quality is still high at a rendering speed of about 7 frames per second. The rendering speed for the visible human male data set is lower than that of the other test data sets, because the data set contains more noise. Thus, a higher voxel resolution is necessary to obtain the same projected error as in the other example scenes.

The cache efficiency for the walkthrough settings is very high. During the high quality rendering of the test data set, only 40-60 blocks have to be decompressed per frame and 20-30 textures have to be constructed on the average. If the caching is deactivated, i.e. the algorithm performs wavelet decompression, gradient calculation, and transfer to graphics memory from scratch for each frame, an average frame rate of 0.3 fps can be obtained for all of the test scenes. This is also the limit frame rate if there was no temporal coherence, i.e. a turn of 180 degrees or moving to a random position within the data set. For this test, the renderer was configured for highest quality, i.e. to use exactly 2048 volume blocks. Thus, the frame rate corresponds to a processing speed of 614 blocks per second or 10 MB of texture data per second.

To measure the exact timing of each part of the visualization is not an easy task in itself. This is due to the concurrent execution, i.e. decompression and gradient calculation are already executed while waiting for the last frame to complete rendering. For the example walkthrough animations about 6% of the time are spend for decompressing blocks and an additional 5% are spend for the gradient calculations. Transferring the textures onto the graphics board consumes another 1% of the time (part of this already runs in parallel), while the vast majority of the time with 88% is spend for the actual rendering, i.e. the processor is waiting for the graphics hardware.

The animation still shows some popping and discontinuity artifacts due to different resolutions in the rendered blocks. This is only a minor problem for high quality settings, but clearly visible for the low resolution settings. It should be quite straightforward to reduce these artifacts by employing mip mapping and techniques similar to the one proposed by Weiler at al. [WWH$^+$00].

2.4 Conclusion & Future Work

A novel approach for accomplishing artifact free shading of volumetric data using shading and pre-integration was presented. Additionally, the presented approach allows to specify material properties on a per sample base. Furthermore, it was also shown that pre-integrated classification can be combined with shading by additionally pre-integrating an interpolation weight used to interpolate the two respective gradients at sample location. It was also shown how to implement ray-casting to further increase the image quality.

A very efficient approach to decompress and visualize animated volume data sets in real time on standard PC hardware was presented. The favored compression scheme uses a quality setting of 63 and the popular MPEG sequence with a significance map and then either arithmetic or Huffman coding. The presented algorithm does not exploit the possibility for parallelization of the wavelet transform or the motion compensation and therefore leaves a lot of room for further optimization using a single processor (3DNow or SSE2 instructions) and multiple processors. Although the sole visualization of each volume is quite fast, this part can also be split up into several sub-volumes that are to be rendered using a cluster of standard PCs.

A rendering algorithm for the visualization of very large data sets was implemented. The algorithm uses a hierarchical wavelet representation to store very large data sets in main memory. The algorithm extracts the levels of detail necessary for the current view point on-the-fly. An error metric that minimized the loss of high frequency information in the projected image is used to determine a suitable level of detail. This technique allows interactive walkthroughs of large volume data sets like the visible human data set on a single commodity PC. This algorithm is the first that achieves an interactive visualization of data set of this size on a single PC. The rendering algorithm scales provably good. Therefore even much larger data sets than the visible human data set can be processed. To overcome the storage problems if even the compressed data set does not fit into main memory any longer, the caching technique should be generalized to swapping to hard disk. The compressed representation will also be useful in an out-of-core scenario, as it can significantly reduce the necessary bandwidth. A special problem of out-of-core rendering is latency due to hard disk seek times. To circumvent this problem, the data must be transferred in large blocks and stored in caches in main memory. Other future directions should include improved rendering techniques to minimize discontinuity artifacts between different resolutions [WWH$^+$00] and a generalization to full RGBα volume data without classification, for example for rendering the cryosection visible human data, too. It would also be interesting to examine whether the wavelet coefficients in each block can be used more effectively to obtain a better adaptation of the rendering resolution to the local frequency spectrum.

Chapter 3

Unstructured Grids

Unstructured tetrahedral grids are a common data representation for three-dimensional scalar fields. The most efficient method to visualize these fields is the so called projected tetrahedra algorithm which takes a sorted list of tetrahedra and composes the pre-integrated foot-print of each tetrahedron in a back to front fashion. With the upcoming of programmable graphics hardware the computationally expensive determination of the footprints can be performed by the graphics hardware, so that the construction of the visibility ordering has become the main bottle-neck of unstructured volume rendering. Currently known visibility sorting algorithms can only handle well-shaped meshes efficiently and have a significant performance penalty for ill shaped meshes or may even produce sorting errors for more complex mesh topologies. A new sorting method is developed which is guaranteed to work for arbitrary mesh topologies including cyclic meshes. Moreover the method has no performance penalty and therefore outruns the previously known general sorting strategies. Additionally it is demonstrated that tetrahedra can be lit according to their ambient, diffuse, and specular material properties. This significantly improves visual appearance of the volumetric data and can be thought of as the three-dimensional counter part of the shading step in the OpenGL rendering pipeline.

In 1990 Shirley and Tuchman invented a popular method for the direct volume visualization of unstructured tetrahedral grids [ST90]. These grids are encountered in finite element simulations, where they are the natural simulation domain. In general, unstructured tetrahedral grids are utilized whenever complex three-dimensional data must be represented with a minimum of volumetric elements.

The algorithm of Shirley and Tuchman is commonly called the projected tetrahedra (PT) algorithm. It takes a scalar volume constructed from tetrahedra as input, and composes the projected tetrahedral cells in a back to front fashion. The footprint of a projected tetrahedron either consists of three or four triangles centered around the thick vertex of the tetrahedron as illustrated in Figure 3.1 (not counting degenerate cases). In this way the volumetric primitive of a tetrahedron is transferred into a triangular representation that can be rendered efficiently by the graphics hardware. This explains the popularity of the algorithm, since its performance directly relates to the number of cells in the data set and is almost independent of the size of the viewing window. Another advantage is that for the purpose of visualization the volume does not have to be converted into intermediate data representations.

In recent years the original approach has been extended in numerous ways and despite its simplicity is still under active research. The first improvement was presented

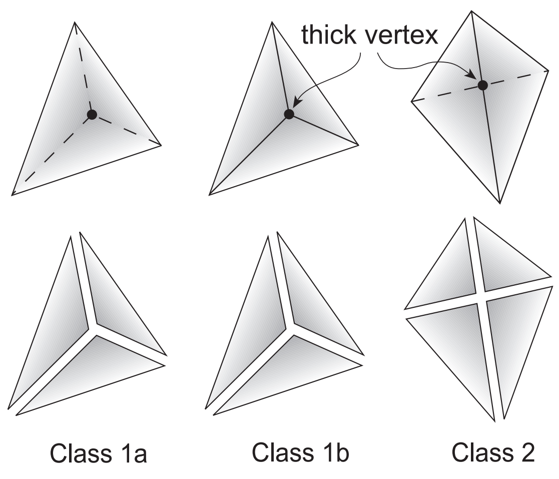


Figure 3.1: Classification of non-degenerated projected tetrahedra (top row) and the corresponding decomposition (bottom row) according to [ST90].

by Stein et al. [SBM94]. They used a more accurate exponential interpolation of opacities inside the tetrahedra instead of the linear approximation of the original approach. In principle, the colors and opacities assigned to each triangle of the tetrahedral decomposition correspond to the line integral along the intersection of each viewing ray with the tetrahedron. Using the volume density optical model of Williams [WM92, Max95], the complexity of the line integral depends on the transfer functions of the optical model. The integral can be solved analytically for the special case of a linear transfer function as proposed by Stein et al. Later this was extended for piecewise linear transfer functions in the HIAC system [WMS98]. For arbitrary transfer functions, however, a numerical integration of the transfer function is necessary. While the numerical integration cannot be performed in real-time, the line integral can be pre-computed and stored in a three-dimensional table. This approach is called pre-integrated cell-projection [MHC90, RKE00].

Most recently the upcoming of programmable graphics hardware has lead to further improvements: The large size of the three-dimensional pre-integration table prevented the use of high-resolution transfer functions. This drawback was resolved by a polynomial reconstruction of the pre-integration table in the pixel shader of modern graphics accelerators [GRS+02]. Using this approach only the polynomial coefficients for the approximation of the line integral need to be stored instead of the memory consuming numerical integral itself.

Using the increasing capabilities of graphics accelerators the decomposition of the tetrahedra into triangles can also be performed in the vertex shader. This is called hardware-accelerated cell-projection [WKE02, WMFC02]. Although this approach not yet leads to a significant performance bump it is expected that graphics accelerators of the next generation will be much more efficient. Then the rendering speed primarily does not depend on the performance of the tetrahedral decomposition, but rather on the speed by which the CPU feeds the GPU over the AGP bus (also compare Wittenbrink et al. [Wit99]). Since the tetrahedra must be processed in a sorted order that is usually in a back to front fashion the overall system performance will be determined by the efficiency of the visibility sorting algorithm. As a consequence, the goal of this section is to devise a visibility sorting algorithm that keeps pace with the growing speed of the graphics accelerators. For this purpose first a brief survey of existing visibility sorting methods is given and their advantages and their limitations are discussed.

3.1 Visibility Sorting

By definition an unstructured tetrahedral grid is a collection of tetrahedra, where it is assumed that the intersection of two tetrahedra is either empty or a common face. An unstructured grid is said to be convex, if the faces which are not shared between two tetrahedra form the convex hull of the data set. This definition includes the connectivity of all parts and excludes interior holes. The task of visibility sorting is closely related to graph theory: For a convex grid the set of tetrahedral pairs with a common face define the edges of a graph. For a specific point of view the edges of the graph point to the occluded neighbors of a tetrahedron. The direction can be determined quickly by computing the dot product of the normal of the common face with the viewing direction. This directed graph imposes an ordering on the set of tetrahedra which is said to be the visibility ordering. If the graph contains no cycles the ordering is total but does not need to be unique. The well known MPVO algorithm of Williams et al. [Wil92] constructs this directed graph for each point of view and traverses it in depth-first fashion. Whenever the algorithm encounters a cell which does not occlude unvisited neighbors it outputs the cell. In this fashion a depth sorted list of tetrahedra is constructed for each specific point of view.

Note that in the majority of cases the graph does not contain a cycle. However, a cycle can occur much easier than one may think in the first place. For example think of a synchronized gear with slanted teeth. When looking along the axis of the gear the teeth may occlude each other in a cyclic way. In such a case any sorting algorithm will produce rendering artifacts because of improper sorting (for other examples see [Wil92]). This case can be resolved in two ways: Either the cycle is cut apart by the selection of an appropriate cutting plane or the cycle is treated differently by using a rendering algorithm that can handle cycles. The selection of an appropriate cutting plane is a very difficult task even for simple cycles, and is not well understood in the general case. A better solution to this problem is to use the so called MPVOC algorithm of Kraus et al. [KE01]. This algorithm can handle arbitrary cyclic meshes without the need of sorting, but has quadratic runtime in contrast to the linear run time of the MPVO algorithm. Although the run time is quadratic it is affordable to use MPVOC, since cycles occur seldom and usually only make up for a tiny fraction of the whole data set. As a summary, one has to keep in mind that cycles may naturally occur in unstructured tetrahedral grids and have to be accounted for to guarantee artifact-free rendering.

For unstructured grids with a concave hull (or data sets with partial connectivity) the MPVO algorithm still produces a sorted list of tetrahedra, but this list cannot be guaranteed to be totally sorted. This fact is illustrated in Figure 3.2 which shows a gear rendered with correct ordering and a difference image showing the artifacts produced by the MPVO algorithm. The incorrect sorting is due to missing relations between the boundary faces of the tetrahedral grid. The straightforward solution to this problem is thus to add complementary edges to the directed graph.

Several algorithms are known which compute the missing relations: First, the MPVONC algorithm is an extension presented by Williams in the original MPVO paper. It uses an easy to compute heuristic for the determination of the additional face relations, but this heuristic is only a first guess of the correct set of relations. The XMPVO algorithm presented by Silva et al. [SMW98], utilizes a sweep plane parallel to the viewing plane to process the faces in correct order and thus is able to find the correct relations. Because of the expensive sweep plane calculations this method was later improved by Comba et al. [CKM$^+$99] who introduced a BSP tree for the efficient computation of the boundary face relations. Hence, it is called BSP-XMPVO. Note that these sorting algorithms for concave meshes assume that the mesh is acyclic, otherwise rendering artifacts occur.

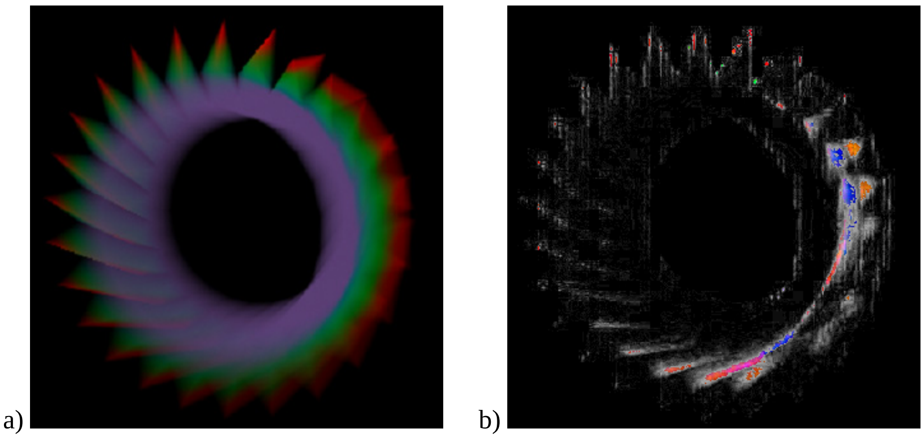


Figure 3.2: Artifacts produced by incorrect visibility sorting using the MPVO algorithm. Correct image (a) and difference image (b).

3.1.1 Tetrahedral Convexification

An analysis of the presented visibility sorting algorithms shows that BSP-XMPVO on the one hand produces correct results for all types of meshes excluding cyclic meshes. On the other hand it is significantly slower than MPVONC, which is guaranteed to produce correct results for convex meshes only. One naive idea to combine the advantages of both methods would be to fill out the concavities of a data set with additional tetrahedra so that it could be handled by the MPVO algorithm. The two-dimensional analogue to this approach is the convexification of a concave polygon. While in two-dimensions this is a well understood problem, in three dimensions the situation is completely different. Here, the addition of complementary tetrahedra to a given concave or even unconnected data set requires the specification of auxiliary vertices. The placement of these vertices is a very difficult task even for simple configurations and up to now is an unsolved problem in the general case.

In the following a solution for the convexification problem will be presented. The key idea is not to add tetrahedra in the first place but rather to break up the concavities into convex polyhedra. Then these convex polyhedra can be filled with auxiliary tetrahedra easily. Both the auxiliary and the original tetrahedra are fed into the MPVO algorithm which is now able to construct the correct visibility ordering. If the mesh contains cycles the MPVOC algorithm is used to take care of the cycles.

Since the runtime of the MPVO algorithm is linear in terms of the processed cells, the performance decreases with the number of auxiliary tetrahedra. However, it turns out that for the special purpose of sorting it is not necessary to break up the convex polyhedra into tetrahedra at all. This is due to the fact that the MPVO algorithm works with all types of convex cells. So just the set of auxiliary convex polyhedra is added to the original set of tetrahedra and a generalized MPVO algorithm is used to sort both. In practice the number of auxiliary polyhedra is small in comparison to the total number of tetrahedra so that sorting performance is decreased only slightly. This is analyzed in more detail in the results section. In the following a detailed description of the proposed tetrahedral convexification algorithm will be given.

Basic Algorithm

Let S be a set of triangles that form the closed boundary surface of a volume. The only assumption being made is that the normals of such a triangle set uniquely determine the exterior of the volume that is that the normals point outwards. Then the volume is said to be concave if the opening angle at the common edge of two triangles is less than 180°. The volume is said to be connected if all triangles can be reached by traveling along the edges of the boundary. With these definitions the tetrahedral convexification algorithm can be described as follows:

P1) S_0 is set to the boundary surface of the tetrahedral mesh

P2) flip the normals of all triangles in S_0

P3) add the convex hull of the tetrahedral mesh to S_0 with normals pointing outwards

P4) remove triangles from S_0 which are part of both the boundary surface and the convex hull

Now S_0 contains the boundary description of the smallest exterior volume which needs to be added to form a convex mesh (compare left side of Figure 3.3). If S_0 is empty, the tetrahedral mesh is already convex and can be fed directly into the MPVO or MPVOC sorting algorithm. Alternatively, one could replace Steps P3) and P4) by just adding the faces of the bounding box (see right side of Figure 3.3).

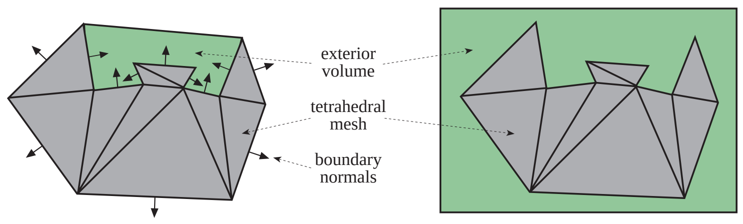


Figure 3.3: Determining the exterior volume (depicted in light green). **Left**: minimal volume using convex hull. **Right**: easy setup with bounding box.

As mentioned above, the exterior volume will not be filled with tetrahedra but it will be broken up into a set of n convex polyhedra $S_i, i = 1..n$. This is achieved by cutting away one concavity after another. For each detected concavity the exterior volume is split into two sub-volumes similar to binary space partition:

C1) $n = 1, S_1 = S_0$

```
C2) while $S_i$ is concave or disconnected for any $i = 1..n$ do
C3) choose triangle $T \in S_i$ so that the plane through $T$ cuts $S_i$ into two non-empty
    sub-volumes $S_i'$ and $S_i''$
C4) triangulate the intersection of the cutting plane with $S_i$ and add the resulting
    triangles to both $S_i'$ and $S_i''$
C5) $S_i = S_i'$, $S_n = S_i''$, $n = n + 1$
C6) optional: discard sub-volumes that are entirely separated from the tetrahedral
    mesh by the cutting plane
C7) endwhile
```

The triangles of a sub-volume that intersect with the cutting plane have to be split and the resulting sub-triangles have to be moved into the corresponding triangle subset. Note that the tetrahedral mesh is not split at all. Only a triangle of the boundary surface may be split so that the directed graph has multiple dependencies for the corresponding face of the attached tetrahedron (compare bottom right of Figure 3.5). The intersection of the cutting plane with a sub-volume is a polygon which may be concave or even disconnected. This polygon has to be triangulated and added to both subsets, since otherwise the sub-volumes are not valid closed surfaces. Triangles which lie in the cutting plane are a special case and must be added to only one sub-volume. Step C6) is an optimization for the bounding box setup.

The described convexification algorithm does not introduce complex volumetric operations but rather is a combination of well known algorithms working on surfaces alone. In this way the seemingly intractable goal of filling an arbitrarily complex volume with tetrahedra is broken down to a number of well analyzed operations on triangular meshes. In the worst case one iteration of the cutting algorithm is needed for each face of the boundary. For each cut the triangulation in Step 4) is the most expensive operation with $O(b \log b)$ runtime and b being the number of boundary triangles. Therefore, the total runtime for the preprocessing of the tetrahedral mesh is $O(b^2 \log b)$. In practice, however, the number of necessary cuts is much less, so that total preprocessing time is nearly linear in terms of the number of tetrahedra.

Cutting Plane Selection

The runtime of the MPVO sorting algorithm for the convexified mesh is directly related to the number of auxiliary cells. Since each cut produces at least one additional cell, the goal is to keep the number of cuts as low as possible to reduce both preprocessing and sorting times. The correct selection of the cutting plane is critical to minimize the number of necessary cuts.

A naive approach would be to select the cutting plane which halves each sub-volume best. This is the analogue strategy to the construction of BSP trees in computer games. Here the BSP performance relies on equal sized nodes, for which the halving strategy works well. In this case, however, the size of the nodes is not relevant, since they won't be rendered. Instead the total number of cuts needs to be minimized.

In principle, the cutting plane should be chosen so that the corresponding triangle T has at least one neighbor with an opening angle less than 180°. This ensures that at least one concavity is cut away from the sub-volume. Since there are usually many triangles that fulfill this condition (see center and right case in Figure 3.4), a different selection

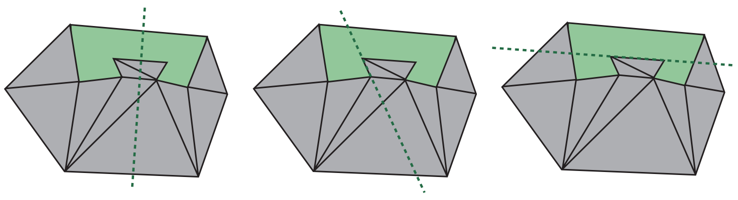


Figure 3.4: Selection of cutting plane. **Left**: bad BSP strategy (no concavity cut away). **Center**: elimination of one concavity. **Right**: elimination of two concavities (but total number of auxiliary cells is the same, since the bottom sub-volume is disconnected and needs to be split by another cut).

criterion is introduced: The criterion is based on the fundamental observation that the number of generated auxiliary cells depends on the number of cut surface elements. For each cut at least one additional auxiliary cell is generated. In the best case each sub-volume is divided into two cells. If the cutting plane intersects the sub-volume boundary multiple times the number of generated cells is usually higher.

This behavior is illustrated in Figure 3.5 which shows the process of convexification for a simple two-dimensional object. Since cuts with as few as possible intersections with the boundary are preferred, the triangles on the convex hull of the data set are processed first (top left of Figure 3.5). The generated sub-volumes outside the convex hull are redundant, hence they are discarded by Step C6) of the cutting algorithm. After processing all possible cuts on the convex hull the algorithm could vote for the horizontal cut on the top right of Figure 3.5. But then the small cell depicted in bright red would be generated. This cell is redundant, since it does not eliminate any concavity. So the algorithm votes for the vertical cut. The left sub-volume is convexified easily by one additional vertical cut, which is preferred over the horizontal cut because it does not intersect with the boundary. The right sub-volume requires two cuts because it is more complex. The algorithm can choose any of the three possible cutting planes, since all have the same intersection count. Finally the algorithm has found a convexification consisting of a total of five auxiliary cells (bottom right of Figure 3.5).

In three dimensions the mesh topology can be more complex than in two dimensions, for example cycles can only occur in three dimensions. But the tendency that cuts with few intersections produce less auxiliary cells is basically the same. As a consequence, the triangle for which the cutting plane has the smallest number of intersections with the boundary of each sub-volume is selected. Since it is infeasible to calculate the number of cuts for every triangle, the algorithm randomly selects a small set of triangles and chooses the best of this group. This strategy is also used by the BSP-XMPVO sorting algorithm. It usually will not find the global minimum, but experimental tests have shown that the results are pretty close to the optimum.

Using the described bounding box setup for tetrahedral convexification Figure 3.6 shows the convexified Blunt Fin data set. For this data set a single auxiliary cell is generated. Only this cell, which is depicted below the wire-frame representation, needs to be added, to yield a convex mesh. Usually the data sets encountered in practice cannot be handled as easy as this, but the Blunt Fin example illustrates that automatic convexification can be achieved very easily for a wide variety of grid types such as curvilinear grids. The sorting performance for the convexified Blunt Fin unsurprisingly is the same as for using MPVO on the original Blunt Fin. A more complex convexification example

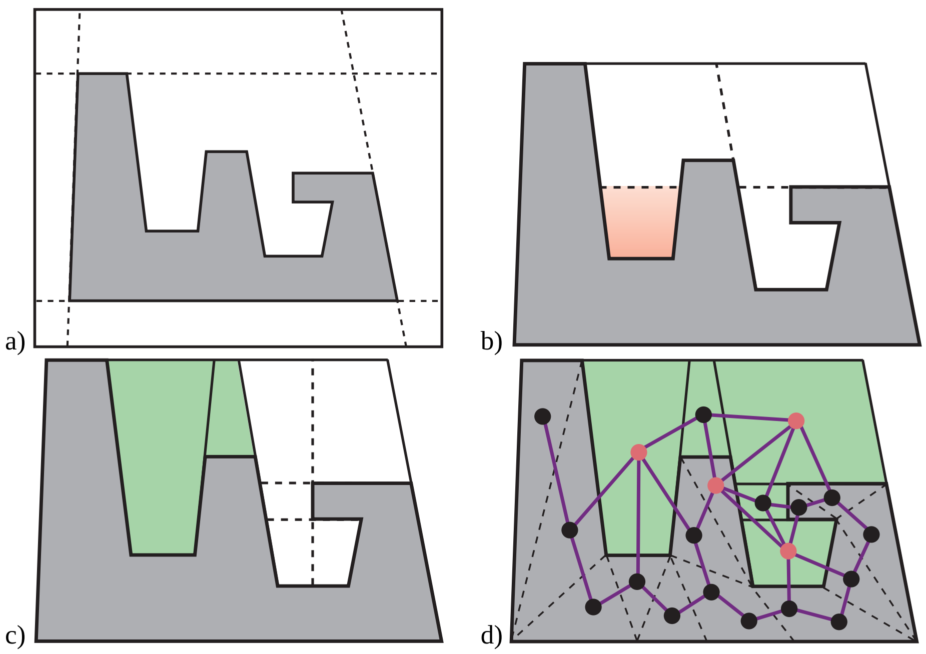


Figure 3.5: 2D convexification example with resulting sorting graph. The light red balls depict cells with multiple dependencies for a face.

with timings is given in Section 3.2.4.

3.1.2 Handling Cycles

Although the tetrahedral convexification as outlined above seems to be a straightforward extension to the MPVO algorithm, there is one issue which needs further explanation. Besides the observation that a tetrahedral mesh can naturally contain cycles, the convexification process can produce cycles as well. This fact is demonstrated in Figure 3.7 a) showing a simple setup which after three cuts generates a cycle. The order in which the cuts are performed has a main influence whether cycles occur or not. For example, a reordering the three cuts as shown in Figure 3.7 b) easily eliminates the cycle. Conceptually, the generation of cycles should be an additional criterion for cutting plane selection. However, it is not yet well understood how to detect cycles during convexification and it is also not well understood whether or not cycles can be prevented in general by proper cutting plane selection. As can be seen in the above example the strategy to select the cutting plane with the least number of boundary intersections also prevents the cycle. Therefore the cutting strategy is also suited for cycle avoidance, but one cannot guarantee acyclicity. So for now the algorithm has to deal with the appearance of cycles. The solution for data sets that already contain cycles is to use the MPVOC algorithm. The solution for cycles generated by convexification is fortunately as simple as this: By definition the cycles are generated by the addition of auxiliary cells. Since these cells need not be rendered the detected cycle can be broken up by deleting one ore more auxiliary cells from the cycle.

In the easiest case the cycle is a simple loop containing one strand of auxiliary cells.

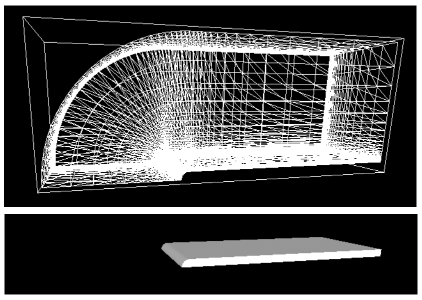
a)
b)


Figure 3.6: To convexify the BluntFin data set (a) only one auxiliary cell needs to be added (b).

Then the algorithm breaks up the cycle by deleting one of the auxiliary cells. After that all elements of the cycle are still connected. This means that the cycle does not break up into several parts, which cannot be sorted using the plain MPVO algorithm. For more complex cycle graphs the deletion of a single auxiliary cell usually also works well, but in rare cases two or more cells have to be deleted. From this experience the most frequent cases are cycles which contain up to two strands of auxiliary cells. In the tests all detected cycles could be eliminated by deleting either a single or at most two auxiliary cells. So the strategy is as follows: If a cycle is detected, the algorithm first subsequently checks if the deletion of a single auxiliary cell resolves the problem. In practice this works for the vast majority of cases. Otherwise the algorithm checks all combinations of deleting two cells. If this still doesn't work the algorithm triggers the MPVOC algorithm for this cycle. In practice it shows up that the latter is triggered only if the tetrahedral mesh itself contains a cycle.

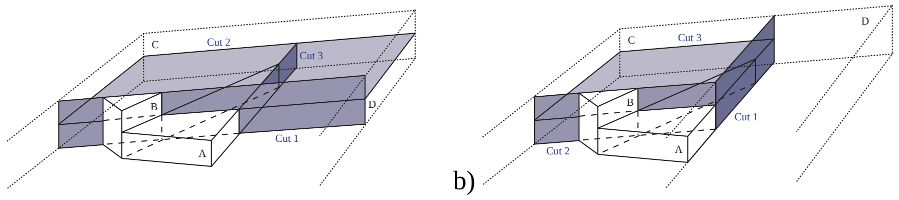
a)
b)


Figure 3.7: a) A cycle generated by three cuts ($A > B > C > D > A$ with $>$ meaning "in front of"). b) A different cut selection resolves the cycle ($A > B > C < D > A$).

3.2 Hardware Accelerated Rendering

3.2.1 Simple Shading

Rendering a tetrahedral data set without any textures or shading enabled already produces a quick preview of the final rendering. However, assigning only a single color and opacity for each vertex of the tetrahedral mesh and each vertex constructed for the PT algorithm produces some severe artifacts (see Figure 3.8).

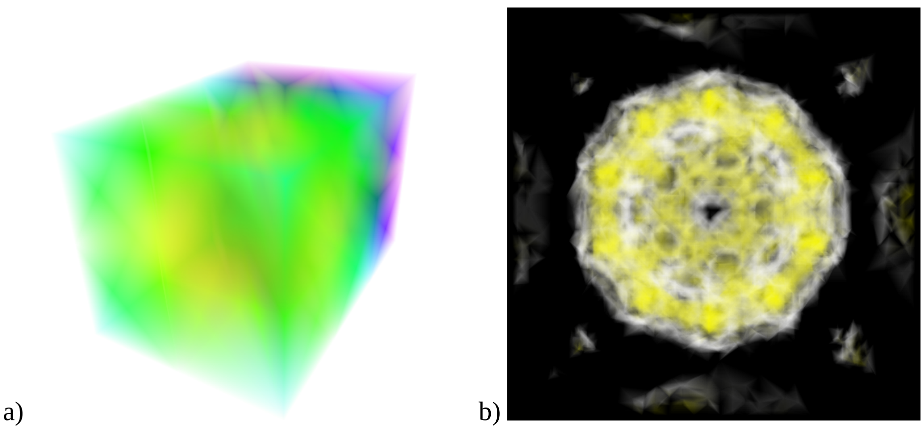


Figure 3.8: Simple shading using no textures and per vertex coloring.

The complete rendering pipeline for the simple shading approach can be seen in Figure 3.9. The only thing that has to change whenever the transfer function is modified are the attributes assigned for each vertex. During rendering, the additional vertices are constructed and their attributes are interpolated from the attributes of the original vertices.

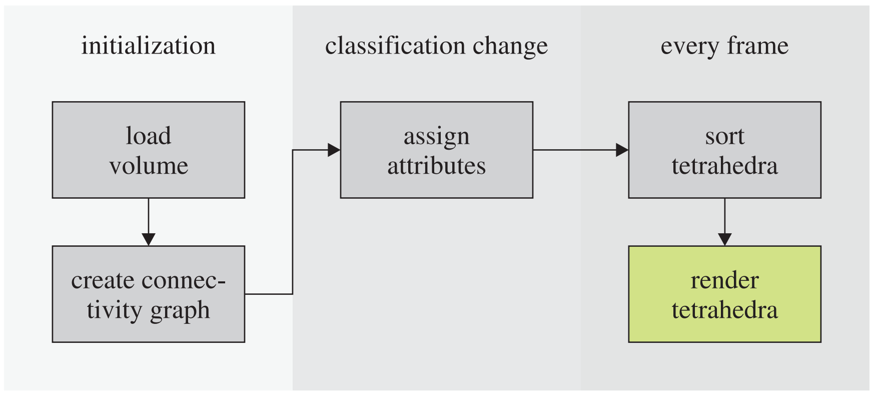


Figure 3.9: Rendering pipeline for simple shading. The green portion is the rendering itself.

The memory consumption during rendering is 3 floats and 1 byte per vertex in system memory and 3 floats and 4 bytes of graphics memory during rendering. The number of vertices also includes the vertices introduced by the projected tetrahedra algorithm. Table 3.1 shows the rendering footprint of this approach.

3.2.2 Pre-Integration

There are a lot of different algorithms for rendering tetrahedral meshes with pre-integrations. While most approaches need large 3D or 2D textures in order to store the pre-

data type	memory locations	simple shading
vertex location	system/graphics	24,576 kB
scalar vertex values	system	2,048 kB
attribute data	graphics	8,192 kB
rendering footprint	graphics	32,768 kB

Table 3.1: Memory footprints in kB for a volume consisting of $2,097,152$ vertices (original and intermediate vertices) during rendering with simple shading.

integration table, these tables can also be approximated while shading [GRS$^+$02]. All of these approaches have one thing in common, to work correctly the rendering needs to interpolate texture coordinates and other attributes on the front and back face of each spat. Thus the rendering has to use homogeneous coordinates and need to implement a per-fragment perspective divide or a dependent projective texture lookup[1] in the shader. The resulting image quality can be seen in Figure 3.10. For this approach, a 3D texture is used and no l_{max} as maximum ray segment length is fixed. Instead $l' = 1 - 2^{-l}$ is used rather than l as third texture coordinate.

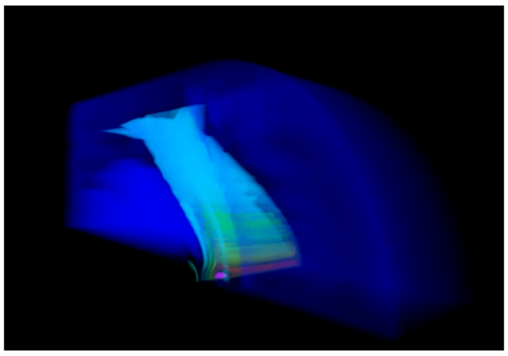
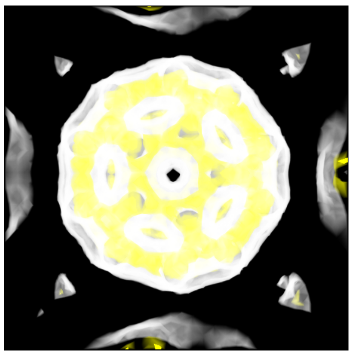
a)  b) 

Figure 3.10: Pre-integrated rendering of the Blunt Fin (a) and Bucky Ball (b) data sets.

The complete rendering pipeline for the pre-integration approach can be seen in Figure 3.11. The 3D textures now have to change whenever the transfer function is modified. During rendering, only texture coordinates for the additional vertices need to be interpolated.

The memory consumption during rendering is 3 floats and 1 byte per vertex in system memory and the same amount of graphics memory during rendering. The number of vertices also includes the vertices introduced by the projected tetrahedra algorithm. Additionally a 3D texture is needed for storing the pre-integration table. A size of $256^2 \times 128$ with 128 entries for the ray segment length l' was chosen. Table 3.2 shows the rendering footprint of this approach.

OpenGL vertex program

The vertex program converts all texture coordinates to homogenous texture coordinates with a w-value and then divides them by the w-value of the transformed front vertex.

[1]Since the rendering also needs to calculate the correct length of the ray segment within the spat, it has to divide anyway. It therefore will not use the projective texture lookup.

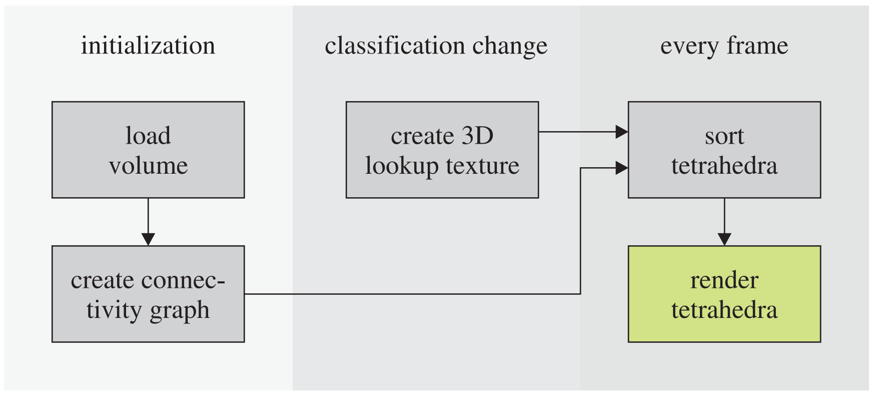


Figure 3.11: Rendering pipeline for pre-integrated rendering. The green portion is the rendering itself.

data type	memory locations	simple shading
vertex location	system/graphics	24,576 kB
scalar vertex values	system/graphics	2,048 kB
pre-integration table	graphics	32,768 kB
rendering footprint	graphics	59,392 kB

Table 3.2: Memory footprints in kB for a volume consisting of $2,097,152$ vertices (original and intermediate vertices) during rendering with simple shading.

This is because the hardware will interpolate on the front face but homogenous coordinates that are interpolated in screen-space are needed. The vertex program can be summarized to implement the following steps.

1. Transform the front vertex coordinates with the model-view-projection matrix (just like the legacy pipeline).
2. Transform the homogenous part w of the back vertex coordinates with the model-view-projection matrix and calculate the scaling between the w parts of both transformed vertices.
3. Calculate the scalar front and homogenous back values.
4. Calculate the distance vector between eye point and front vertex.
5. Calculate the distance vector between eye point and back vertex in homogeneous coordinates.

The shader now has the correctly interpolated values on both the front and back face of each spat.

```
!!ARBvp1.0

ATTRIB    iPosF      =   vertex.position;
ATTRIB    iPosB      =   vertex.texcoord[0];
ATTRIB    iTex       =   vertex.texcoord[1];
```

```
TEMP      fPos;
TEMP      tmp;
TEMP      rPos;

PARAM     mvp[4]     = { state.matrix.mvp };
PARAM     iEye       =   state.matrix.modelview[0].
                         invtrans.row[3];

OUTPUT    oPos       =   result.position;
OUTPUT    oTex       =   result.texcoord[0];
OUTPUT    oPosF      =   result.texcoord[1];
OUTPUT    oPosB      =   result.texcoord[2];

# transform front vertex                                   ①
DP4       fPos.x, mvp[0], iPosF;
DP4       fPos.y, mvp[1], iPosF;
DP4       fPos.z, mvp[2], iPosF;
DP4       fPos.w, mvp[3], iPosF;
MOV       oPos, fPos;

# transform homogeneous part of back vertex                ②
DP4       rPos, mvp[3], iPosB;
RCP       rPos, rPos;
MUL       rPos, fPos.w, rPos;

# texture 0 contains correct scalar front value and        ③
# homogeneous back value
MOV       oTex.x, iTex.x;
MUL       oTex.y, iTex.y, rPos;
MOV       oTex.zw, rPos;

# texture 1 contains correct front (homogeneous)           ④
ADD       tmp, iPosF, -iEye;
MUL       oPosF.xyz, tmp, iTex.w;
MOV       oPosF.w, 1.0;

# texture 2 contains correct back (homogeneous)            ⑤
ADD       tmp, iPosB, -iEye;
MUL       tmp, tmp, rPos;
MUL       oPosB.xyz, tmp, iTex.w;
MOV       oPosB.w, rPos;

END
```

OpenGL fragment program

The fragment program now treats all input coordinates as homogeneous coordinates and does the required texture coordinate calculations and lookups.

1. Calculate the texture coordinates for accessing the pre-integration table. The

   texture coordinates consist of: front value s_f, back value s_b (needs to be divided) and distance l between front and back vertex (also needs to be divided by the same number). The third coordinate is actually 2^{-l} instead of the distance itself.

2. Lookup into pre-integration table.

3. Convert the opacity from distance 1 to distance l.

4. Write final pre-multiplied color.

The resulting fragment program then looks like this.

```
!!ARBfp1.0

ATTRIB    iTex      =   fragment.texcoord[0];
ATTRIB    iPosF     =   fragment.texcoord[1];
ATTRIB    iPosB     =   fragment.texcoord[2];

TEMP      texcoord;
TEMP      color;
TEMP      length;

# calculate texture coordinate for lookup                  ①
MOV       texcoord.r, iTex;
RCP       texcoord.a, iPosB.a;
MAD       length, texcoord.a, iPosB, -iPosF;
DP3       length, length, length;
RSQ       length, length.r;
RCP       length, length.r;
EX2       texcoord.b, -length.r;
MAD       texcoord.b, texcoord.b, -1.0, 1.0;

# look up pre-integrated value                             ②
TEX       color, texcoord, texture[1], 3D;

# convert optical density into transparency                ③
ADD       color.a, 1.0, -color.a;
POW       color.a, color.a, length.r;

# write final color                                        ④
MOV       result.color.rgb, color;
ADD       result.color.a, 1.0, -color.a;

END
```

3.2.3 Pre-Integrated Lighting

Since the rendering performance only depends on the sorting time and the vertex performance, quite long fragment programs can be used to improve the visual quality compared to simple pre-integrated rendering or the rendering used by Guthe et al. [GRS$^+$02]. Meissner et al. [MGS02] already proposed the pre-integration of not only one emissive color but also lighting properties like ambient, diffuse and specular coefficient. This has already been explained in the pre-integrated lighting section for regular grids.

Since the scalar values and the gradients are interpolated linearly within each tetrahedra, they are also linear along each viewing ray. Therefore most of the approximations used for regular grids are actually much more accurate for irregular grids. Again, to allow for pre-integration of the diffuse color, the assumption that the intensity of the light varies linearly is used. Therefore the light intensity ΔI_n for each segment n can be split into the ambient intensity ΔI_n^a, the diffuse intensities ΔI_n^{df}, ΔI_n^{db} and the specular intensity ΔI_n^s.

Figure 3.12 shows the improved visual perception of the surfaces while increasing the diffuse coefficient within the transfer function. Up to now, no specular highlight has been used.

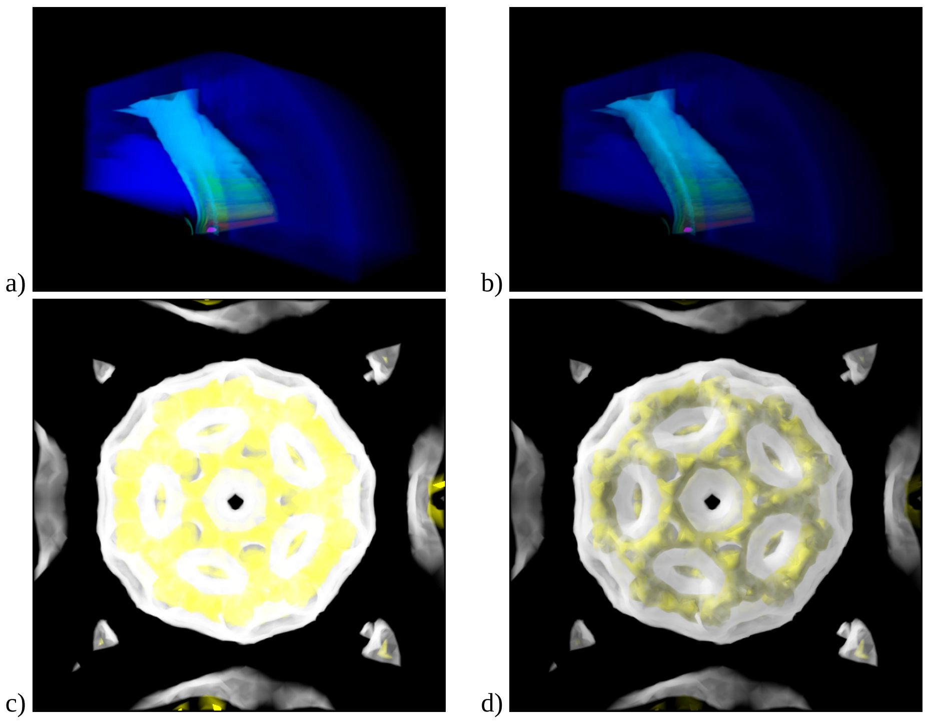


Figure 3.12: Blunt Fin (a, b) and Bucky Ball (c, d) data set using per-ray lighting. With ambient only (a, c) and diffuse lighting (b, d).

With this approach a highlight is rendered correctly over multiple opaque isosurfaces within a single tetrahedra and is approximated efficiently for semi-transparent transfer functions as seen in Figure 3.13 and Figure 3.14.

In order to further increase the accuracy of both diffuse lighting and the specular highlight, additional samples along each ray can be used. Although the resulting image quickly converges to the correct solution, the increasing number of samples into the pre-integration tables may introduce the fill rate as the limiting performance factor again.

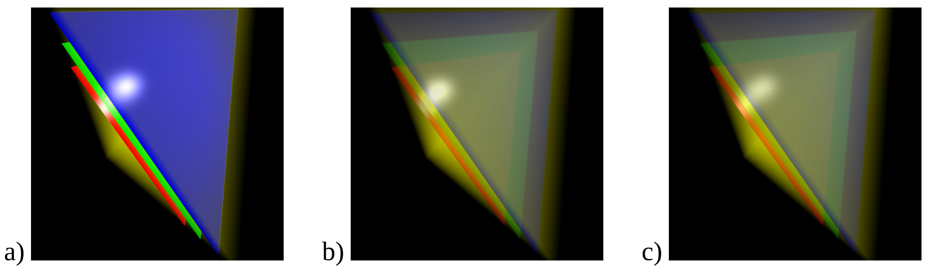


Figure 3.13: Specular highlight on different iso-surfaces within a single tetrahedron (a), approximation for non-iso-surface setting (b) and correct solution (c).

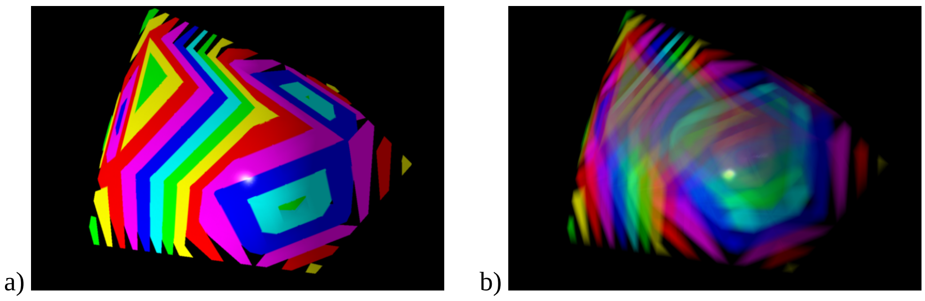


Figure 3.14: Specular highlight on different iso-surfaces for a data set containing 60 tetrahedra (a) and highlights for non-iso-surface setting (b).

The rendering pipeline seen in Figure 3.15 is again very similar to the one used for pre-integrated rendering. The only difference is the calculation of normals per vertex. This operation has to be carried out only once, after loading the volume data and constructing the connectivity graph.

The memory consumption during rendering is 3 floats and 1 byte per vertex in system memory and the same amount of graphics memory during rendering. The number of vertices also includes the vertices introduced by the projected tetrahedra algorithm. Additionally three 3D textures are needed for storing the pre-integration table. A size of $256^2 \times 128$ with 128 entries for the ray segment length l' was chosen. Table 3.3 shows the rendering footprint of this approach.

data type	memory locations	simple shading
vertex location	system/graphics	24,576 kB
scalar vertex values	system/graphics	2,048 kB
pre-integration tables	graphics	98,304 kB
rendering footprint	graphics	124,928 kB

Table 3.3: Memory footprints in kB for a volume consisting of $2,097,152$ vertices (original and intermediate vertices) during rendering with simple shading.

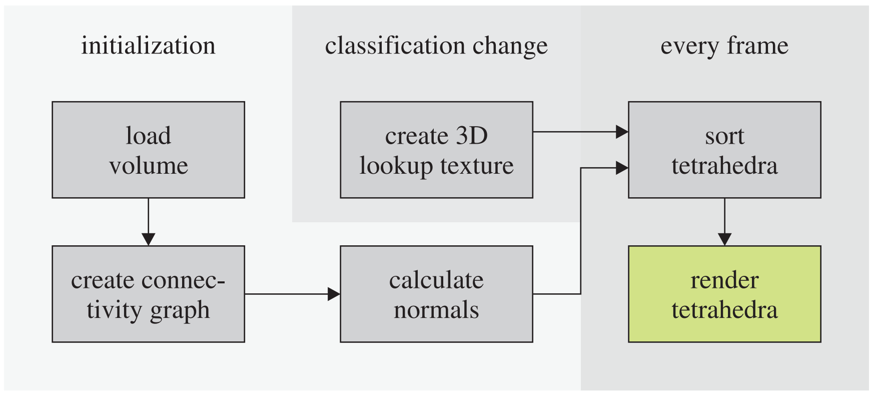


Figure 3.15: Rendering pipeline for pre-integrated rendering. The green portion is the rendering itself.

OpenGL vertex program

The vertex program converts all texture coordinates to homogenous texture coordinates with a w-value and then divides them by the w-value of the transformed front vertex. This is because the hardware will interpolate on the front face but homogenous coordinates that are interpolated in screen-space are needed. The vertex program can be summarized to implement the following steps.

1. Transform the front vertex coordinates with the model-view-projection matrix (just like the legacy pipeline).
2. Transform the homogenous part w of the back vertex coordinates with the model-view-projection matrix and calculate the scaling between the w parts of both transformed vertices.
3. Calculate the scalar front and homogenous back values.
4. Calculate the distance vector between eye point and front vertex.
5. Calculate the distance vector between eye point and back vertex in homogeneous coordinates.
6. Store front gradient.
7. Calculate and store back gradient in homogeneous coordinates.
8. Calculate the distance vector between light source and front vertex.
9. Calculate the distance vector between light source and back vertex in homogeneous coordinates.

The shader now has the correctly interpolated values on both the front and back face of each spat.

```
!!ARBvp1.0

ATTRIB    iPosF      =   vertex.position;
ATTRIB    iPosB      =   vertex.texcoord[0];
ATTRIB    iTex       =   vertex.texcoord[1];
ATTRIB    iGradientF =   vertex.texcoord[2];
ATTRIB    iGradientB =   vertex.texcoord[3];

TEMP      fPos;
TEMP      tmp;
TEMP      rPos;
TEMP      light;
TEMP      eye;

PARAM     mvp[4]     = { state.matrix.mvp };
PARAM     iEye       =   state.matrix.modelview[0].
                         invtrans.row[3];
PARAM     iLight     =   state.light[0].position;

OUTPUT    oPos       =   result.position;
OUTPUT    oTex       =   result.texcoord[0];
OUTPUT    oPosF      =   result.texcoord[1];
OUTPUT    oPosB      =   result.texcoord[2];
OUTPUT    oGradientF =   result.texcoord[3];
OUTPUT    oGradientB =   result.texcoord[4];
OUTPUT    oLightF    =   result.texcoord[5];
OUTPUT    oLightB    =   result.texcoord[6];

# transform front vertex                                    ①
DP4       fPos.x, mvp[0], iPosF;
DP4       fPos.y, mvp[1], iPosF;
DP4       fPos.z, mvp[2], iPosF;
DP4       fPos.w, mvp[3], iPosF;
MOV       oPos, fPos;

# transform homogeneous part of back vertex                 ②
DP4       rPos, mvp[3], iPosB;
RCP       rPos, rPos;
MUL       rPos, fPos.w, rPos;

# texture 0 contains correct scalar front value and         ③
# homogeneous back value
MOV       oTex.x, iTex.x;
MUL       oTex.y, iTex.y, rPos;
MOV       oTex.zw, rPos;

# texture 1 contains correct front (homogeneous)            ④
ADD       tmp, iPosF, -iEye;
MUL       oPosF.xyz, tmp, iTex.w;
```

```
MOV        oPosF.w, 1.0;

# texture 2 contains correct back (homogeneous)                  ⑤
ADD        tmp, iPosB, -iEye;
MUL        tmp, tmp, rPos;
MUL        oPosB.xyz, tmp, iTex.w;
MOV        oPosB.w, rPos;

# texture 3 contains correct front gradient                      ⑥
# (homogeneous)
MOV        oGradientF.xyz, iGradientF;
MOV        oGradientF.w, 1.0;

# texture 4 contains correct back gradient                       ⑦
# (homogeneous)
MUL        oGradientB.xyz, iGradientB, rPos;
MOV        oGradientB.w, rPos;

# texture 5 contains correct front light direction               ⑧
# (homogeneous)
ADD        light, -iPosF, iLight;
DP3        light.w, light, light;
RSQ        light.w, light.w;
MUL        oLightF.xyz, light, light.w;
MOV        oLightF.w, 1.0;

# texture 6 contains correct back light direction                ⑨
# (homogeneous)
ADD        light, -iPosB, iLight;
DP3        light.w, light, light;
RSQ        light.w, light.w;
MUL        light, light, light.w;
MUL        oLightB.xyz, light, rPos;
MOV        oLightB.w, rPos;

END
```

OpenGL fragment program

The fragment program now treats all input coordinates as homogeneous coordinates and does the required texture coordinate calculations and lookups.

1. Calculate the texture coordinates for accessing the pre-integration table. The texture coordinates consist of: front value s_f, back value s_b (needs to be divided) and distance l between front and back vertex (also needs to be divided by the same number). The third coordinate is actually 2^{-l} instead of the distance itself.
2. Lookup into pre-integration tables.
3. Convert the opacity from distance 1 to distance l and weight ambient color with ambient light intensity.

4. Calculate diffuse term of front sample s_f.
5. Calculate diffuse term of back sample s_b.
6. Calculate and normalize the representing gradient.
7. Calculate and normalize the eye vector at the same location.
8. Calculate reflected vector.
9. Calculate and normalize the light vector at the same location.
10. Calculate specular highlight.
11. Write final pre-multiplied color.

The resulting fragment program then looks like this.

```
!!ARBfp1.0

ATTRIB    iTex       =   fragment.texcoord[0];
ATTRIB    iPosF      =   fragment.texcoord[1];
ATTRIB    iPosB      =   fragment.texcoord[2];
ATTRIB    iGradientF =   fragment.texcoord[3];
ATTRIB    iGradientB =   fragment.texcoord[4];
ATTRIB    iLightF    =   fragment.texcoord[5];
ATTRIB    iLightB    =   fragment.texcoord[6];

PARAM     lAmbient   =   state.light[0].ambient;
PARAM     lDiffuse   =   state.light[0].diffuse;
PARAM     lSpecular  =   state.light[0].specular;

TEMP      texcoord;
TEMP      color;
TEMP      front;
TEMP      back;
TEMP      eye;
TEMP      tmp;
TEMP      length;

# calculate texture coordinate for lookup                  ①
MOV       texcoord.r, iTex;
RCP       texcoord.a, iPosB.a;
MAD       length, texcoord.a, iPosB, -iPosF;
DP3       length, length, length;
RSQ       length, length.r;
RCP       length, length.r;
EX2       texcoord.b, -length.r;
MAD       texcoord.b, texcoord.b, -1.0, 1.0;

# look up pre-integrated value                             ②
TEX       color, texcoord, texture[1], 3D;
TEX       front, texcoord, texture[2], 3D;
TEX       back, texcoord, texture[3], 3D;
```

```
# convert optical density into transparency                  ③
ADD        color.a, 1.0, -color.a;
POW        color.a, color.a, length.r;
MUL        color.rgb, color, lAmbient;

# diffuse lighting front                                      ④
DP3        tmp.r, iGradientF, iGradientF;
DP3        tmp.g, iLightF, iLightF;
DP3        tmp.b, iGradientF, iLightF;
MUL        tmp.r, tmp.r, tmp.g;
RSQ        tmp.r, tmp.r;
MUL_SAT    tmp.r, tmp.r, tmp.b;
MUL        tmp, tmp.r, lDiffuse;
MAD        color.rgb, tmp, front, color;

# diffuse lighting back                                       ⑤
DP3        tmp.r, iGradientB, iGradientB;
DP3        tmp.g, iLightF, iLightF;
DP3        tmp.b, iGradientB, iLightF;
MUL        tmp.r, tmp.r, tmp.g;
RSQ        tmp.r, tmp.r;
MUL_SAT    tmp.r, tmp.r, tmp.b;
MUL        tmp, tmp.r, lDiffuse;
MAD        color.rgb, tmp, back, color;

# Calculate and normalize representing gradient               ⑥
MUL        tmp, back.a, iGradientB;
MUL        tmp, tmp, texcoord.a;
MAD        tmp, front.a, iGradientF, tmp;
DP3        tmp.a, tmp, tmp;
RSQ        tmp.a, tmp.a;
MUL        tmp.rgb, tmp, tmp.a;

# Calculate and normalize eye vector                          ⑦
MUL        eye, back.a, iGradientB;
MUL        eye, eye, texcoord.a;
MAD        eye, front.a, iGradientF, eye;
DP3        eye.a, eye, eye;
RSQ        eye.a, eye.a;
MUL        eye.rgb, eye, eye.a;

# Calculate reflected vector                                  ⑧
MAD        eye, tmp.a, tmp, -eye;
MAD        eye, tmp.a, tmp, eye;

# Calculate and normalize representing light vector           ⑨
MUL        tmp, back.a, iLightB;
MUL        tmp, tmp, texcoord.a;
MAD        tmp, front.a, iLightF, tmp;
```

```
DP3       tmp.a, tmp, tmp;
RSQ       tmp.a, tmp.a;
MUL       tmp.rgb, tmp, tmp.a;

# Calculate specular highlight                                 ⑩
DP3_SAT   tmp.rgb, tmp, eye;
LG2       tmp.rgb, tmp.r;
MUL       tmp.rgb, tmp, 128.0;
EX2       tmp.rgb, tmp.r;
ADD       tmp.a, front.a, back.a;
MUL       tmp.rgb, tmp, tmp.a;
MUL       tmp, tmp, lSpecular;

# write final color                                            ⑪
ADD_SAT   result.color.rgb, color, tmp;
ADD       result.color.a, 1.0, -color.a;

END
```

3.2.4 Results

The current implementation of the described convexification algorithm is not yet designed for speed but rather is a proof of concept. Right now the algorithm processes roughly 1,000 tetrahedra per second, but there is plenty of room for improvement. Besides the convexification, which is only a preprocessing step, an implementation of the MPVO algorithm with a performance of about 1,2 million tetrahedra per second on an AMD Athlon 1.33 GHz processor was used. At such clock-speeds the sorting algorithm is already memory bound. To overcome the limitations of the memory bus a simple trick was applied. By not only using an indexed vertex but also an indexed normal data structure the traffic on the memory bus was reduced by about 20-50% depending on the regularity of the data set. Together with a hand-tweaked cell-projection algorithm written in SSE2 an overall rendering performance of about 880,000 tetrahedra per second was achieved on an NVIDIA GeForce4 Ti4200 graphics accelerator. For example the Blunt Fin Data set in Figure 3.6 with 224,874 tetrahedra renders at approximately 3.9 frames per second. The performance for the complex SPX data set is given in Table 3.4.

tested cuts	auxiliary cells	sorting time	total time
0	0	7.2 ms	15.4 ms
1	1562	17.9 ms	
5	1115	13.7 ms	
10	1000	12.9 ms	
20	836	12.2 ms	20.4 ms

Table 3.4: Sorting times for the SPX data set with 12936 tetrahedra (Coolant simulation in a part of the Super Phoenix Reactor). The first line shows the results of using the plain MPVO algorithm which is fastest but does not render the SPX correctly. With an increasing number of tested cutting planes the total number of auxiliary cells decreases significantly and so does sorting time.

Taking the number of generated auxiliary cells into account the total number of cells is increased by only 13% but sorting time increases by 69% and total rendering time by 32%. The reason for this overproportinal progression are the memory latencies for the irregular auxiliary cells. In comparison with Guthe et al. [GRS$^+$02] this algorithm achieves about the same performance although it uses a slower test platform. This is mainly due to the optimizations of the original algorithm, that is the indexed normals and the hand-crafted cell projection algorithm. This algorithm is also faster in comparison to the hardware-accelerated cell projection algorithms [WKE02, WMFC02] which achieve only approximately 500,000 tetrahedra per second excluding times for sorting.

In a direct comparison with BSP-XMPVO the algorithm has the same average runtime behavior but the sorting stage is much faster since the space partitioning calculations have been swapped into the preprocessing step. During runtime only very simple graph operations as in the original MPVO are required. An additional advantage is that the explicit representation of the exterior volume in contrast to the implicit BSP-tree eases postprocessing and storage of the convexified mesh. However the main advantage of the presented algorithm is that all types of meshes including cyclic meshes can be handled correctly and at the same time almost the same performance as the MPVO algorithm is achieved.

In Figure 3.16 an additional convexification example of a more complex mesh is given: It shows two interleaving gears with many concavities and a complex structure in between the gears.

3.3 Compression

Tetrahedral meshes have been around in finite element simulations on volumetric domains for a long time. With the growing need of visualizing simulation data, tetrahedral meshes established themselves also in volume visualization. There are several beautiful properties of tetrahedral meshes which make them the natural choice for volume data representation. The flexibility of a tetrahedral mesh is ideally suited for irregular samplings and multiresolution analysis. The convex nature of a single tetrahedron allows for a simple visibility sorting algorithm, which is essential in volume visualization.

In most application areas of tetrahedral meshes some data is attached to the mesh elements. The data can be attached to the vertices, edges, the faces, the border faces or the tetrahedra. A density might be attached to the vertices, the intensity of a flow to the edges or material identifiers to the tetrahedra. The tetrahedral mesh serves several different purposes. It can be used to store nearest neighbors, to subdivide a volume into convex primitives or to sample and, by using the barycentric coordinates, to parameterize the domain of a function. The function can be scalar, a vector field or even a tensor field as for example the stress tensor of an inhomogeneous material. The compression algorithm can be extended in a natural way to support compression of all three types of data functions defined on all different types of mesh elements.

Basic Definitions and Notations

This section deals with tetrahedral meshes in the three-dimensional Euclidean space given by a set of tetrahedra such that any two tetrahedra either are disjunctive or share a face, an edge or a vertex. The number of vertices, edges, faces, border faces and tetrahedra is denoted with v, e, f, b and t respectively.

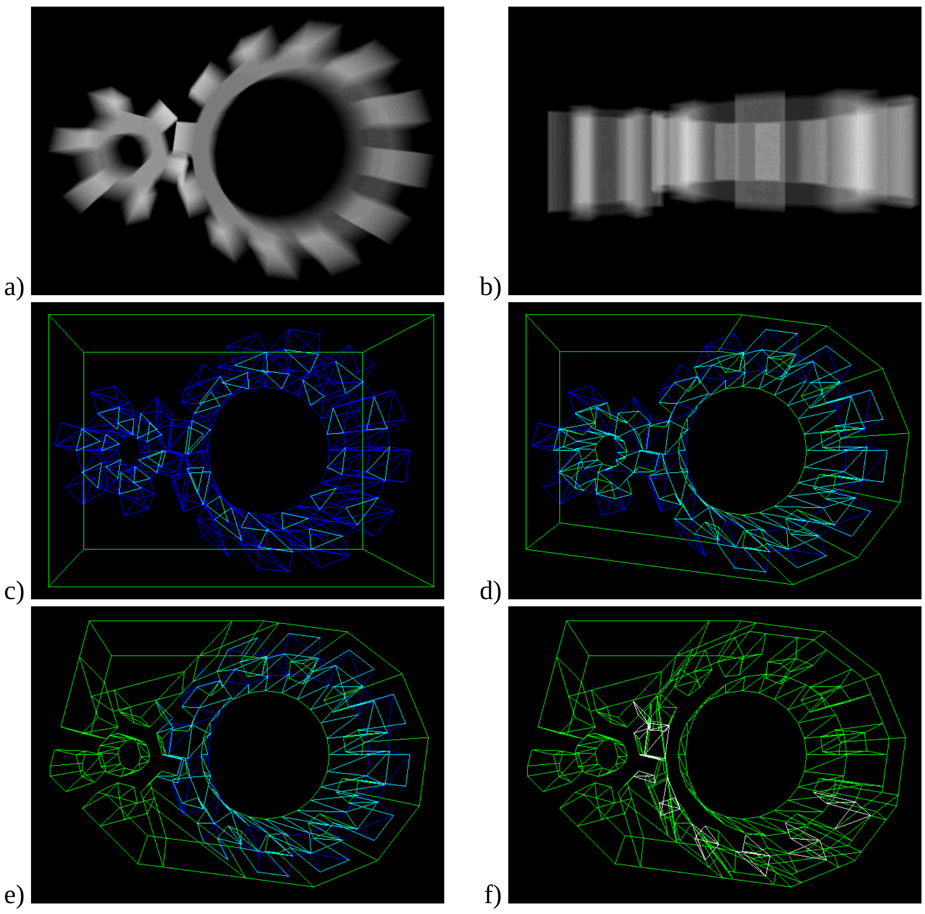


Figure 3.16: Convexification of two interleaving gears. The top row (a, b) shows the data set from the side and from above. For the final convexified mesh (f) a total of 42 cuts are necessary for a data set consisting of 202 tetrahedra.

The total amount of bits consumed by a tetrahedral mesh is denoted with $\mathcal{S}$, where a right subscript is used to express a special representation type. $\mathcal{S}_{std}$ denotes for example the standard representation with a list of vertex coordinate triples, a list of vertex index quadruples representing the tetrahedra and additional lists for the attached data. The storage space $\mathcal{S}$ is split into the bits $\mathcal{L}$ consumed by the locations of the vertices, $\mathcal{C}$ consumed by the connectivity and $\mathcal{D}$ consumed by the data attached to the mesh elements. If no data is present only the geometry consisting of connectivity and vertex locations has to be encoded in $\mathcal{G}$ bits. For reasonable representations the following approximation can be used:

$$\mathcal{S} \leq \mathcal{G} + \mathcal{D} \quad \mathcal{G} \leq \mathcal{C} + \mathcal{L} \tag{3.1}$$

The combined representation of two and more components of the tetrahedral mesh can be more efficient since better predictions might improve delta coding or just because the coding mechanism can combine some fractional bits.

Basic Equations and Approximations

The basic equation for a tetrahedral mesh as defined in the previous section is the Euler equation:

$$v - e + f - t = \chi \tag{3.2}$$

where χ is the Euler characteristic of the mesh and in most cases negligibly small. If the tetrahedron-face instances are counted once for each tetrahedron and once for each face a second equation including the number of border faces b can be set up:

$$f = 2t + \frac{b}{2} \tag{3.3}$$

In the case of triangle meshes the corresponding equations are sufficient to determine the average face-order of a vertex and the number of triangles per vertex in a mesh with small Euler characteristic and few boundary edges, but not in the tetrahedral case as Figure 3.17 illustrates. The tetrahedron-order of a vertex might vary between one as in Figure 3.17 a) and v for the mesh in b)[2]. Thus for the number of tetrahedra in an arbitrary tetrahedral mesh only the following is known:

$$\frac{v}{4} \le t \in \Omega\left(v^2\right) \tag{3.4}$$

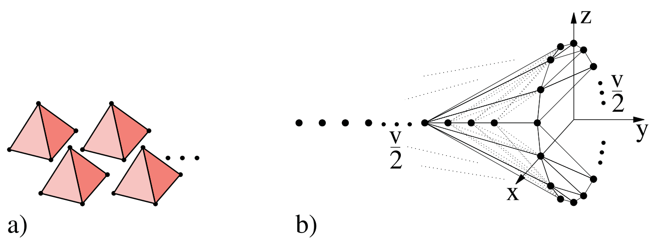


Figure 3.17: tetrahedral meshes with a) minimum and b) maximum vertex tetrahedron-order $\frac{4t}{v}$.

None of the tetrahedral meshes of Figure 3.17 are used to sample volumetric functions for volume visualization or finite element analysis. The tetrahedral meshes of interest normally have a limited edge-order of the vertices, a small border portion and low Euler characteristics of the mesh and of the border mesh, respectively. Therefore, the fraction between t and v is expressed in terms of the average number of edges around a vertex $\overline{o}_{v\to e} = \frac{2e}{v}$, the number of border vertices v_b, χ and χ_b the Euler characteristic of the border. As the border is closed the relation $3b = 2e_b$, where e_b is the number of border edges, can be expressed. Using the Euler equation for triangular meshes $v_b + b = e_b + \chi_b$, produces $v_b = \chi_b - \frac{b}{2}$. This equation together with equations 3.2 and 3.3 yields

$$\frac{t}{v} = \frac{\overline{o}_{v\to e}}{2} - 1 - \frac{v_b}{v} + \frac{\chi + \chi_b}{v}. \tag{3.5}$$

To find a basic approximation for the relation between t and v in a typical tetrahedral mesh with small border portion and low Euler characteristics the estimation of

[2]The mesh is one of the delaunay tetrahedrizations of the shown set of points.

$\overline{o}_{v\rightarrow e}$ for a regular tetrahedral mesh is left. Unfortunately, the Euclidean space can not be tetrahedralized with equilateral tetrahedra. But the fraction of 4π over the steradian occupied by an equilateral tetrahedron yields[3] with 11.64 a good approximation of the average vertex edge-order. The tetrahedralization of a cubic grid yields $\overline{o}_{v\rightarrow e} \overset{v\rightarrow\infty}{\longrightarrow} 12$ for a 1 : 5 zoning[4] and $\overline{o}_{v\rightarrow e} \overset{v\rightarrow\infty}{\longrightarrow} 14$ for a 1 : 6 zoning. Considering this and the measured average edge-orders in Table 3.5, we assume in the following an average vertex order of thirteen. For tetrahedral meshes with small Euler characteristic and border portion the following equation in agreement with Table 3.5 is produced.

$$v : e : f : t \approx 1 : 6.5 : 11 : 5.5. \tag{3.6}$$

The algorithm uses this approximation to estimate the storage consumption of a tetrahedral mesh in the standard representation, where each vertex is given by three 32bit floating point coordinates and each tetrahedron by four vertex indices:

$$\mathcal{L}_{std} = 96v, \quad \mathcal{C}_{std} = 4t \cdot \lceil \log_2 v \rceil \overset{v\approx 10^5}{=} 374v. \tag{3.7}$$

For a typically sized tetrahedral mesh with a hundred thousand vertices the connectivity consumes about four times more storage space than the vertex coordinates. The algorithm can reduce the connectivity to about eleven bits per vertex (see Table 3.7). This reduces the storage requirements to a quarter without loosing information.

Entropy and Arithmetic Coding

Let $\mathcal{A} = \{a_1, ..., a_m\}$ be an alphabet with m different symbols and $\mathcal{S} = s_1 s_2 ... s_n$ a sequence of n symbols $s_i \in \mathcal{A}$. Then the relative frequency ν_{a_i} of the symbol a_i is defined by

$$\nu_{a_i}(\mathcal{S}) = \frac{|\{j | s_j = a_i\}|}{n}. \tag{3.8}$$

If besides the relative frequencies no further information about the sequence $\mathcal{S}$ is known, it can be shown that at least

$$\mathcal{E}(\mathcal{S}) = \mathcal{E}(n, \nu_{a_1}, ..., \nu_{a_m}) = -n \cdot \sum_{a\in\mathcal{A}} \nu_a(\mathcal{S}) \log_2 \nu_a(\mathcal{S}) \tag{3.9}$$

bits are needed to encode the sequence $\mathcal{S}$. The quantity $\mathcal{E}$ is called binary entropy. Arithmetic coding (see [WNC87] for an introduction) allows to encode a sequence with only slightly more bits than the binary entropy.

Related Work

As shown in the previous section tetrahedral meshes consume disproportional storage space in comparison to the data functions they sample. There are two approaches to reduce the size of tetrahedral meshes. The first one is mesh simplification as described for tetrahedral meshes in [CPS97, PH97, SG98, VV98, ZCK97]. Most methods are based on the edge collapse operation introduced by Hoppe [Hop96] which is easily generalized from the triangular case. All these methods are lossy compression schemes. In the area of lossless compression previously only the Grow&Fold method proposed by

[3]The Euler equation for spherical triangle meshes has been applied.
[4]Each cube is split into five tetrahedra.

Szymczak [SR99] was available. The compressed representation consists of a tetrahedra spanning tree and a folding string. The spanning tree is rooted at an arbitrary tetrahedron and grown by attaching all other tetrahedra to external triangles of the current spanning tree. For each added tetrahedron three bits encode whether further tetrahedra are attached to the three external triangles of this tetrahedron or not. The folding string contains for each external triangle of the spanning tree a 2-bit code defining one of the three edges or no edge. If an edge is specified, this external triangle is folded together with the triangle adjacent through the specified edge. As the spanning tree contains t tetrahedra and $2t + 1$ external faces the storage requirements so far are $7t + O(1)$ bits. External triangles of the spanning tree without folding edge are either border triangles of the tetrahedral mesh or must be glued to a triangle which is not edge-adjacent. The glue operations consume over two bits but appear in reasonable meshes seldom enough such that the total storage space increases only slightly to more than seven bits per tetrahedron. This thesis will also compare the storage space consumed by the compressed representation of the cut-border machine to the lower bound of seven bits per tetrahedron for the Grow&Fold representation:

$$\mathcal{C}_{G\&F} = 7t. \tag{3.10}$$

The limitation of the Grow&Fold approach is that it cannot handle non manifold borders.

Grow&Fold combines ideas from the triangle mesh compression techniques ”geometry compression through topological surgery” introduced by Taubin [TR98] and ”edgebreaker” proposed by Rossignac [Ros99]. The compression scheme generalizes the cut-border machine proposed by Gumhold and Straßer [GS98] which is similar to the edgebreaker approach. But the cut-border machine is much easier to generalize to the tetrahedral case than the edgebreaker. The ideas of the triangular cut-border machine are briefly repeated in section 3.3.1. Basically, the cut-border machine traverses the mesh in a region filling way, which is determined by the connectivity of the mesh, and encodes for each newly added triangle one operation, which describes how the triangle is formed upon the current border edge of the growing region.

Toumas [TG98] triangle mesh connectivity compression scheme allows the encoding of regular triangle meshes with better space efficiency. This method is in a way similar to the cut-border machine. By encoding the vertex triangle-orders with a runlength encoding scheme half of the operations encoded by the cut-border machine can be saved. For regular triangle meshes the vertex triangle-orders can be encoded very space efficiently such that the compressed representation may only consume half the storage space of the cut-border representation. Touma also proposes a simple prediction for the vertex coordinates. The coordinates are estimated as the fourth vertex of a parallelogram which is formed from the triangle inside the growing region and adjacent to the current border edge of the growing region. The crease angle at this edge is estimated from the other crease angles at the interior triangle. Encoded is only the difference between the estimation and the actual location of the vertex.

3.3.1 Cut-Border Machine

The connectivity compression is based on the generalization of the cut-border machine described in [GS98]. In the triangular case the cut-border machine is very simple to implement and also extremely fast. The generalization to the tetrahedral case requires a more sophisticated data structure for non manifold triangle meshes.

Triangle Connectivity Compression with the Cut-Border Machine

The cut-border machine compresses triangle meshes which consist of a list of vertices and a list of triangles, each triangle containing the three vertex indices and the indices of the three edge-adjacent triangles. If the latter adjacency information is not known it can be easily computed through hashing.

The cut-border machine is based on a region growing traversal of the triangle mesh starting with an arbitrary triangle. The border of the growing region is called the cut-border. It divides the mesh into the inner and the outer part, which contain the already compressed and the remaining triangles respectively. Triangles are added to the inner part at a distinguished current cut-border edge which will be called the gate as proposed in [Ros99]. After each addition of a triangle the gate moves on to another cut-border edge, until all triangles of an edge-connected component of the triangle mesh have been compressed. This is done for each edge-connected component.

The compressed representation contains for each triangle a bit code of an operation identifier which tells how the triangle was formed upon the gate. There are three cases: the gate is an edge of the mesh border, the gate forms a triangle with a vertex on the cut-border or the triangle is formed with a new vertex. The different operations are called ”border”, ”new vertex” and ”connect”. The ”connect”-operations take one parameter which specifies the offset of the third vertex relative to the gate. The ”connect”-operations with offset one and minus one are also called ”connect forward” and ”connect backward”. All other ”connect”-operations split the cut-border into two loops. As the triangle meshes describe two dimensional surfaces in three-dimensional space, two cut-border loops can grow together again, actually once for each handle and each hole of the triangle mesh. The operation which unifies two loops is called ”union” and takes two parameters, the index of the second loop and the offset of the third triangle vertex within this loop.

The cut-border data structure basically consists of a stack of doubly linked lists containing the vertices or their indices , which are adjacent to triangles of the inner and the outer part at the same time. Data at the faces, edges or vertices such as their coordinates are included in the compressed representation each time a new mesh element is added to the cut-border for example the vertex coordinates of vertex v_i are encoded after the ”new vertex”-operation which introduces v_i into the cut-border.

The decompression algorithm builds the mesh in the same order as the compression algorithm traverses the original mesh. With the help of the indices attached to the ”connect”-operations the original connectivity can be reconstructed with permuted vertices and triangles. During decompression the edge-adjacency information can be reconstructed with no additional cost.

The success of the method results from the high frequency of the ”new vertex”- and the ”connect forward”-operations. Together they constitute in most meshes over 95% of all operations. The high frequency of the ”connect forward”-operation and the low frequency of ”connect”-operations with large offsets depends in a high degree on the traversal order, which is determined by the choice of the gate after each operation.

From Triangle Connectivity to Tetrahedral Connectivity Compression

As in the triangular case the uncompressed tetrahedral mesh is stored as a list of vertices and a list of tetrahedra, each tetrahedron containing the indices of the incident vertices and the face adjacent tetrahedra. The inner and the outer part consist of a set of tetrahedra. The cut-border is the triangular surface between the inner and the outer part and the gate is a triangle of the cut-border. For each face-connected component of the mesh the traversal begins with an arbitrary tetrahedron and successively adds face-adjacent tetrahedra at the gate to the inner part. The different cut-border operations are described in the next section. The cut-border may become the surface of an arbitrary face-connected tetrahedral mesh and therefore contain non manifold vertices and edges. We assume that the tetrahedral mesh is embedded in three-dimensional space and that the tetrahedra do not penetrate each other. As in the triangular case the traversal order highly influences the distribution of the ”connect”-operations with different offsets.

Cut-Border Operations and Situations

There are three possibilities for the fourth vertex of a newly added tetrahedron at the gate: the gate is a border triangle of the tetrahedral mesh, the gate forms a tetrahedron with a new vertex or the gate is connected through a tetrahedron to another cut-border vertex. The corresponding cut-border operations will again be called ”border”, ”new vertex” and ”connect” and are abbreviated with the symbols Δ $*$ and ∞_i.

Although only three different types of cut-border operations exist, ten different situations are distinguished which describe the surrounding of the cut-border around the gate for the different cut-border operations. All the situations are illustrated in Figures 3.18 and 3.19. Figure 3.18 shows the situations which do not introduce non manifold vertices or edges. For the ”border”- and the ”new vertex”- operation only one situation exists which is depicted in Figure 3.18 c) and b), respectively. The ”connect” operation comes along with a whole variety of situations. The most frequent of these is the ”flip” operation shown in Figure 3.18 a). Here the newly added tetrahedron connects the gate to an adjacent triangle of the cut-border. The common edge of these two cut-border triangles is kind of flipped if the two former cut-border triangles are replaced by the two new cut-border triangles introduced by the new tetrahedron. The ”top” and the ”close” operations are very similar to the ”flip” operation. The only difference is that not only two faces of the newly added tetrahedron are part of the cut-border but three of them in the case of the ”top” operation or even all in the case of the ”close” operation. The ”close”-operation eliminates or closes an edge-connected component of the cut-border triangle mesh. The ”close”-situation cannot be seen from the outside of the cut-border. Therefore, in Figure 3.18 e) the cut-border is rendered with transparent triangles. In the front long edges of outer triangles are visible.

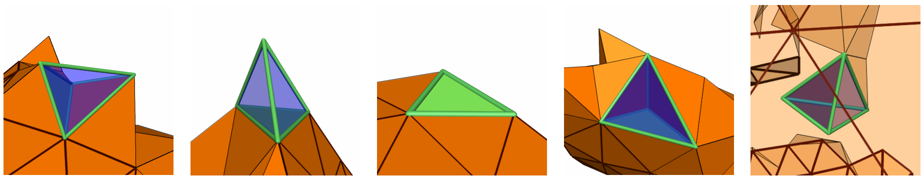

a) flip b) new vertex c) border d) top e) close

Figure 3.18: The different manifold cut-border situations. The gate is shown as green triangle and the newly added tetrahedron with green edges and blue transparent faces.

As mentioned earlier, the cut-border can be a non manifold triangle mesh. Figure 3.19 portrays all types of situations which introduce a non manifold vertex or edge.

In Figure 3.19 a) the non manifold counterpart of the ”flip” situation is shown. Here the free edge of the ”flip” situation is touched by the cut-border and therefore already belongs to the cut-border. The touched edge becomes non manifold after application of the ”connect” operation. The ”join” situation in Figure 3.19 b) is the non manifold counterpart of the ”new vertex” operation. The fourth vertex of the newly added tetrahedron is part of a region of the cut-border triangle mesh which is further apart from the gate. This vertex becomes non manifold. Finally, in the ”join” situations depicted in Figures 3.19 c), d) and e) not only the fourth vertex of the newly added tetrahedron belongs to the cut-border but also one two or all three free edges of the ”join” situation. Thus one, two or three non manifold edges are introduced.

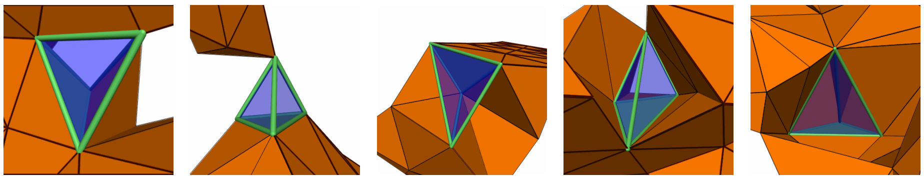

a) nm flip b) join c) j. 1 nm edge d) j. 2 nm edges e) j. 3 nm edges

Figure 3.19: The different types of non manifold cut-border situations.

The situations depicted in Figures 3.18 and 3.19 constitute all possible situations, which can be easily verified by considering a newly added tetrahedron: the three triangles of the tetrahedron which are unequal to the gate may all be part of the cut-border or not be part. The same holds true for the fourth vertex and the three edges not incident to the gate. All of these seven mesh elements might be present in the cut-border or not. The presence of one of the three triangles implies the presence of the fourth vertex and the two incident edges. If such implications are taken into account each possible assignment of presence to the three triangles, three edges and the fourth vertex yields exactly one of the discussed situations. Thus each face-connected component of the tetrahedral mesh can be compressed without any vertex repetitions. Only if two components of the tetrahedral mesh are exclusively connected through edge-adjacency and vertex-adjacency the involved non manifold vertices are repeated. In a simple way the ”border”-operation allows for the encoding of all possible border surfaces of tetrahedral meshes including non manifold borders.

The ”connect” operation takes one index as parameter, which specifies the fourth vertex in the cut-border. The fourth vertex is with high probability near to the gate. This fact can be exploited for a more efficient encoding by mapping near fourth vertices to small connect indices. This is achieved by a breadth-first traversal through the triangles of the cut-border starting at the gate as shown in the illustration of Figure 3.20. The enumeration is not uniquely defined before one edge of the gate is specified at which the enumeration with the zero connect index will begin. This edge will be called the zero edge and is specified by the traversal strategy. Figure 3.20 gives pseudo code for the vertex enumeration. The algorithm is similar to the cut-border traversal in the case of a triangle mesh. In a fifo these edges of the cut-border are stored which are adjacent both to a visited triangle and to a not visited triangle at the same time. The zero edge is firstly placed into the fifo. Triangles are visited by extracting the next edge from the fifo and addressing the adjacent triangle which has not been visited yet. If the third vertex of the newly visited triangle is reached the first time, the next available connect index is assigned to it. In this way the vertices obtain the indices illustrated in Figure 3.20.

```
fifo.pushback(gate.zeroEdge())
fifo.pushback(gate.oneEdge())
fifo.pushback(gate.twoEdge())
while not fifo.empty do
  edge = fifo.popfront()
  tgl = edge.rightTriangle()
  if not marked(tgl) then
    mark(tgl)
    vtx = tgl.oppositeVtx(edge)
    if not marked(vtx) then
      mark(vtx)
      enumerate(vtx)
    fifo.pushback(tgl.nextEdge(edge))
    fifo.pushback(tgl.prevEdge(edge))
```

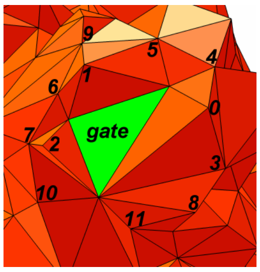


Figure 3.20: Vertex enumeration.

The ”flip” situation can arise for the operations ∞_0, ∞_1 and ∞_2, the ”top” situation for ∞_0 and ∞_1 and ”close” only for ∞_0. The different ”join” situations correspond to ”connect” operations with larger index and are less frequent. The traversal strategy has to optimize the choice of the zero edge in a way that most ”flip” and ”top” situations are encoded with ∞_0.

Compressed Representation

In the triangular case the ”new vertex”-operation $*$ is performed in about half the cases and is most frequent. In the tetrahedral case the relative frequency of $*$ is only about $\frac{1}{5.5}$, whereas the connect operations with small index are most frequent. For optimal encoding of the operation symbols arithmetic coding was used since the relative frequencies are unequal to 2^{-k} and therefore Huffman-coding is not appropriate.

The connectivity of the tetrahedral mesh is given by the sequence of cut-border operations. As each operation adds one tetrahedron or specifies one border face, $t + b$ operations are encoded. The binary entropy defined in Equation 3.9 gives a good lower bound

$$\mathcal{C}_{CB} = \mathcal{E}\left(n, \nu_\Delta, \nu_*, \nu_{\infty_0}, \nu_{\infty_1}, \ldots\right) < \mathcal{C}_{CB}^{adapt} \tag{3.11}$$

for the storage space $\mathcal{C}_{CB}^{adapt}$ consumed by the arithmetic coder with adaptive relative frequencies, which are initialized to the average values given in the last row of Table 3.6. Table 3.7 shows that the arithmetic coder almost achieves the optimum.

The vertex coordinates and the data at the vertices, edges, faces and tetrahedra are incorporated in the arithmetic coding stream with separate coding models. Each time a cut-border operation produces a new mesh element, the corresponding data is added to the stream. The representation of a 1:6 zoning of a cube with vertex data $v_0, v_1, \ldots, v_7$ and tetrahedral data $t_0, t_1, \ldots, t_5$ might look as follows:

$t_0 x_0 y_0 z_0 v_0 x_1 y_1 z_1 v_1 x_2 y_2 z_2 v_2 x_3 y_3 z_3 v_3 \Delta\Delta$
$* t_1 x_4 y_4 z_4 v_4 \Delta * t_2 x_5 y_5 z_5 v_5 \Delta$
$* t_3 x_6 y_6 z_6 v_6 \infty_0 t_4 \Delta\Delta * t_5 x_7 y_7 z_7 v_7 \Delta\Delta\Delta\Delta\Delta\Delta.$

Traversal Order

The traversal strategy chooses after each cut-border operation the next gate and zero edge. The aim is to favor a small number of different kinds of operations. To avoid most connect operations with large indices it turned out that a good strategy is to stay at one cut-border vertex until all adjacent tetrahedra have been visited. The cut-border vertices are processed in a fifo order. For the choice of the zero edge and the order in which the triangles around a cut-border vertex are added, two heuristics were tried that favor the ∞_0-operation. The first one cycles around edges and tries to close up with a ∞_0-operation by setting the zero edge of the gate to the edge around which the cut-border machine cycles. The second strategy defines the zero edge of each cut-border triangle at the time when the triangle is created. The zero edge is set to the edge which is shared by the gate and the new triangle. In case of a new vertex operation it is obvious that with this choice the zero edge is the edge with the smallest angle in the outer part. This still holds true to some extent for the other operations. The first heuristic increased the frequency of the ∞_0-operation to 45% and the second heuristic even to 60%. Thus the second strategy was chosen, which is documented in Table 3.6.

Mesh-Border Encoding

In order to allow for a non manifold mesh border, the border operations are encoded explicitly. The border symbol can be avoided when an edge-adjacent triangle of the gate has already been encoded as border triangle. In this case the corresponding connect symbol can be used. This optimization helped to decrease the additional amount of storage for the mesh border to one bit per border triangle as tabulated in Table 3.7. The same optimization improves the border encoding in the triangular case of the cut-border machine.

Cut-Border Data Structure

Data Structure 1 Cut-Border

```
CutBorder
  CutBorderTriangle       triangles[]
  Fifo<CutBorderVertex>   vertices
  TriangleIndex           gate
CutBorderTriangle
  VertexIndex             vertexIndices[3]
  TriangleIndex           adjacentTriangles[3]
  TetraIndex              innerTetra
  Boolean                 meshBorder
  Integer                 zeroEdge
CutBorderVertex
  VertexIndex             meshVertexIndex
  Set<TriangleIndex>      adjacentTriangles
```

Data structure 1 shows the cut-border data structure. Three relations between the cut-border vertices and the cut-border triangles are stored: for each triangle the three incident vertices and three edge-adjacent triangles; for each vertex all incident triangles. The latter relation is stored in a set data structure which allows insertion and elimination of elements and the intersection of two sets. This relation allows for the handling of non manifold vertices and edges. For each cut-border triangle the incident tetrahedron of the inner part is stored in order to find the new tetrahedron if the triangle becomes the gate. The meshBorder-flag tells the algorithm if the cut-border triangle has already been encoded as border triangle of the mesh and therefore does not have to be visited again. With the help of this flag the optimized border encoding is realized. As the traversal order defines the zero edge for each triangle at creation time, an index between zero and two is stored for each cut-border triangle defining the zero edge. The cut-border vertices are organized in a fifo as demanded by the chosen traversal strategy.

For each vertex of the tetrahedral mesh a field which stores the index of the cut-border vertex is generated and initialized before compression to minus one. In this way the algorithm can not only map a tetrahedral mesh vertex index to a cut-border vertex index but does also know which of the tetrahedral mesh vertices are part of the cut-border.

Why is it sufficient to keep for each triangle only three edge-adjacent neighbors even at non manifold edges? At any time the cut-border describes the surface of a tetrahedral mesh. Thus the faces around a non manifold edge divide the space into regions alternately belonging to the inner and the outer part. These regions around a non manifold edge are called inner/outer regions. The faces bounding the same outer region can be set to be edge-adjacent as illustrated in Figure 3.21. This definition correctly reflects the proximity needed in enumerating the vertices relative to the gate. Faces of different outer regions can not be connected through a tetrahedron without intersecting an inner region.

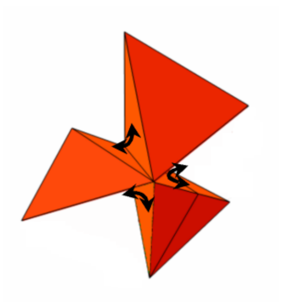


Figure 3.21: Edge-adjacency of cut-border triangles around non manifold edge.

Finally, the updates of the cut-border data structure are described for the different situations depicted in Figures 3.18 and 3.19. During the ”connect” operation of a manifold ”flip” situation (see Figure 3.18 a)) the two present triangles in the cut-border are replaced with two new ones where the common edge is flipped. The vertices and face-adjacent triangles of the two new triangles can be easily determined from the old triangles. For each new triangle the zero edge is set to the edge, which is incident to the gate. The innerTetra index of the newly added triangles is set to the newly added tetrahedron, as in all other situations of all operations. Finally, the old triangles are removed from the triangle sets of the vertices and the new triangles are added.

The first step during the update of the ”new vertex” operation is to create a new cut-border vertex for the fourth vertex of the newly added tetrahedron and store its vertex index of the tetrahedral mesh in the corresponding field. Conversely, the index of the new cut-border vertex is stored within the corresponding field of the tetrahedral mesh vertex. Next the gate triangle is removed and three new triangles are inserted. Again their zero edges are set to the edges incident to the gate. The ”border” operation just sets the border flag of the gate triangle. For the border optimization the border flags of the three edge-adjacent cut-border triangles are checked and if one of them is set, the operation is encoded with the corresponding ”connect” operation. The ”top” situation is similar to the ”flip” situation except that three triangles are removed and only one is added. As last manifold situation the ”close” operation eliminates all involved triangles and these vertices for which the set of adjacent triangles becomes empty. If a cut-border vertex is removed the index stored with the corresponding tetrahedral mesh vertex is set to minus one again.

In order to distinguish between manifold and non manifold situations the algorithm has to clear up how to decide whether an edge of the newly added tetrahedron belongs to the cut-border or not. The question is trivially answered positively if an incident triangle of the newly added tetrahedron already belongs to the cut-border. Otherwise the answer can be determined by intersecting the set of adjacent triangles of the incident vertices of the edge in question. If the intersection is empty no cut-border triangle contains the edge and therefore the edge cannot belong to the cut-border. The intersection test must be performed for all edges of the non manifold situations in Figure 3.19 which are not incident to a cut-border triangle. In case of the ”nm flip” situation this is one edge and in case of the four ”join” situations these are three edges. Only if the non manifold edges are detected, the face-adjacency can be updated according to Figure 3.21. And this is the only difference in the update process between the ”nm flip” and ”flip” situations and between the four different ”join” situations and the ”new vertex” operation.

The ”nm flip” operation is distinguished from the ”flip” situation by checking if the edge connecting the two newly added triangles belongs to the cut-border or not. This check can be done after the update performed for the ”flip” situation, such that the face-adjacency of the two new triangles can be corrected if necessary. This is only possible if the vertex coordinates are known and given in three-dimensional space. For more general tetrahedral meshes the neighbors of the newly added triangles must be explicitly encoded. This can be done with few bits and as the non manifold situations are much less frequent as the manifold situations, the total storage space will not increase significantly for typical meshes.

The family of ”join” situations is detected whenever the three triangles of the newly added tetrahedron, which are not equal to the gate, are not part of the cut-border but the fourth vertex is part of the cut-border. The latter condition is checked with the help of the cut-border index field attached to the tetrahedral mesh vertices. The update of the ”join” situations is the same as in the case of the ”new vertex” operation except that the three newly added triangles must also be inserted to the triangle set of the fourth vertex. Finally, the three potential non manifold edges are checked for their presence in the cut-border and the face-adjacency of the corresponding triangles are corrected if necessary as in the case of the ”nm flip” situation.

Coordinate Compression

In a first step each vertex coordinate is quantized to 16 bits according to the diagonal of the bounding box of all vertices. Thus the compression is lossy and for some applications not appropriate. All the meshes were in ASCII format with six to eight valid digits which is equivalent to 19-26 bits. The quantization step will loose some information and the shape of small tetrahedra changed slightly, but no tetrahedron changed its orientation.

To encode the 16 bit coordinates arithmetically it turned out to be economical to split each coordinate into four packages of four bits. For each package a different set of adaptive frequencies was used for the arithmetic coder. This strategy dramatically reduced the storage space consumed by the arithmetic coder and increased the compression speed.

The next step in coordinate compression is delta coding. The vertex coordinates are encoded during the compression of the connectivity. After each new vertex operation the difference vector from the center of the gate triangle to the new vertex is encoded. Thus the proximity information given by the tetrahedralization of the vertices was used. The number of bits saved through delta coding can be estimated with the following simple argument. Suppose the vertices are uniformly distributed. Then there are approximately $\sqrt[3]{v}$ vertices per coordinate axis and it should be possible to save $\log_2 \sqrt[3]{v}$ bits per coordinate. Thus the storage space consumed per vertex can be estimated with $48 - \log_2 v$ bits, which is about three bits above the actually achieved storage space.

A final improvement of two bits less storage space per vertex could be achieved by rotating the coordinate system such that the z-axis is the normal of the gate and the x-axis parallel to the zero edge. Quantization is done after changing to the new coordinate system. To avoid accumulation of rounding errors it is very important that during compression the center of the gate is computed with the same quantized coordinates which are available to the decompression algorithm. The change of the coordinate system saved two bits in the x- and y-axis. The final storage space consumed per vertex by the coordinates is tabulated in Table 3.7 in the column labeled $\frac{\mathcal{L}_{CB}^{16Bit}}{v}$.

3.3.2 Results

The Tetrahedral Meshes

Figure 3.22 shows the six tetrahedral meshes which were used for the measurements. They differ in their sizes and their origin. The ”Random” mesh was generated by delaunay tetrahedralization of a cloud of randomly distributed points. In order to show that the interior of this mesh is more complex than the surface, a cut through the mesh was blended with its surface. The ”Proto” mesh is a quite regular tetrahedralization of an object with non trivial boundary. The ”Bubble” is the output of a simplification algorithm applied to a spherical symmetric scalar function. Again the blending technique shows part of the interior. The ”Torso” meshes are regularly tetrahedralized real world meshes and the ”Blunt Fin” is a curvilinear grid.

Measurements

Table 3.5 shows the basic quantities of the different meshes and average values which confirm Equation 3.6.

In Table 3.6 the distribution of the cut-border symbols is analyzed. The first column shows for each mesh the total number $t + b$ of encoded operations. In the following columns the relative frequencies of the different cut-border symbols are shown. ∞_0 is with 60% the most frequent operation, followed by $*$, ∞_1 and ∞_2. With the border optimization the frequency of the border symbol became negligibly small. The last column shows the fraction of the non-manifold situations in Figure 3.19 which arose

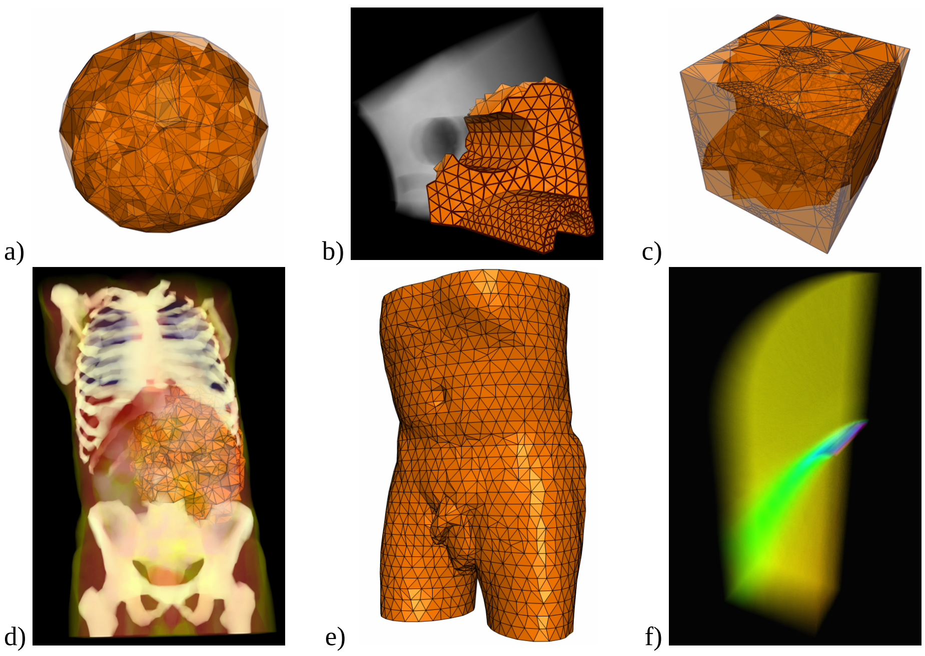


Figure 3.22: The measured tetrahedral meshes Random (a), Proto (b), Bubble (c), Torso I (d), Torso II (e) and Blunt Fin (f). The transparent meshes were rendered with projected tetrahedra. To the tetrahedra of the ”Torso I”-mesh a material identifier is attached. The ”Blunt Fin”-mesh was rendered with false colors.

during compression. This number is important for the optimal running time of the compression and decompression algorithms as the non-manifold operations consume more computing power.

Table 3.7 illustrates different aspects of the consumed storage space and running time for the cut-border machine. The first column shows the storage space consumed by the arithmetic coder for the connectivity. The second and third columns tabulate the binary entropy of the cut-border operations in bits per vertex and bits per tetrahedra. Comparison of the first two columns shows that the arithmetic coder is near the optimum. The cut-border machine consumes on average about two bits per tetrahedron, even for the randomly generated mesh which forces more connect operations with a high index. $\mathcal{C}_{CB,\Delta}$ is the binary entropy of the sequence of cut-border operations which were used to encode the border faces. The fourth column of Table 3.7 shows that the border could be encoded with about one bit per triangle. As the best triangle mesh compression methods consume also about one bit per triangle, the initializing of the cut-border machine with the border of the tetrahedral mesh would not improve the border encoding. The fifth column of Table 3.7 documents the compression speed in tetrahedra per second for connectivity alone. The decompression speed is approximately the same. The speed does not depend on the size but more on the frequency of non-manifold operations (compare the last column of Table 3.6). The last but one column contains the storage space consumed by the vertex coordinates. Finally, the last column shows that the vertex compression doesn't decrease the compression speed significantly.

mesh	v	v: e: f: t	$\frac{v_b}{v}$	b	$\overline{o}_{v\rightarrow e}$	$\overline{o}_{e\rightarrow t}$
Random	2000	1:7.39:12.67:6.29	0.101	400	14.77	5.11
Proto	2896	1:5.94: 9.41:4.47	0.477	2760	11.89	4.51
Bubble	5715	1:6.89:11.64:5.74	0.150	1710	13.78	5.00
Torso I	11140	1:6.55:10.91:5.35	0.197	4380	13.10	4.90
Torso II	15164	1:6.61:11.04:5.43	0.180	5454	13.22	4.93
Blunt Fin	40960	1:5.74: 9.32:4.58	0.165	13516	11.48	4.78
average		1:6.52:10.83:5.31	0.212		13.04	4.87

Table 3.5: Basic quantities of the measured meshes.

mesh	t+b	ν_Δ	ν_*	ν_{∞_0}	ν_{∞_1}	ν_{∞_2}	$\nu_{\infty_{i>2}}$	νnm
Random	12971	0.001	0.154	0.519	0.118	0.108	0.101	0.116
Proto	15695	0.001	0.184	0.631	0.073	0.067	0.044	0.046
Bubble	34526	0.001	0.165	0.549	0.106	0.091	0.088	0.109
Torso I	64028	0.002	0.174	0.607	0.080	0.072	0.064	0.069
Torso II	87788	0.001	0.173	0.603	0.083	0.075	0.065	0.069
Blunt Fin	200910	0.000	0.204	0.707	0.045	0.044	0.000	0.000
average		0.001	0.176	0.602	0.084	0.076	0.060	0.068

Table 3.6: Total number of encoded operations; relative frequencies of cut-border operations; relative frequency of non-manifold situations.

Table 3.8 compares the cut-border machine to the standard representation and the Grow&Fold compression of Szymczak [SR99]. The results of the cut-border machine are convincing and improve the standard representation by a factor of 20 to 50 depending primarily on the size of the tetrahedral mesh but also on the regularity.

3.4 Conclusion & Future Work

The sorting algorithm presented here is capable of sorting any convex or concave tetrahedral mesh with or without cycles, or even a number of disconnected meshes. The number of auxiliary cells and thus additional sorting time is so low, that these meshes can now be sorted correctly and rendered interactively.

The rendering quality is comparable to the quality of structured volume rendering.

mesh	$\frac{\mathcal{C}_{CB}^{adapt}}{v}$	$\frac{\mathcal{C}_{CB}}{v}$	$\frac{\mathcal{C}_{CB}^{adapt}}{t}$	$\frac{\mathcal{C}_{CB,\Delta}}{b}$	$\left(\frac{t}{sec}\right)_\mathcal{C}$	$\frac{\mathcal{L}_{CB}^{16Bit}}{v}$	$\left(\frac{t}{sec}\right)_\mathcal{G}$
Random	15.12	15.02	2.39	1.37	84831	34.40	73866
Proto	9.55	9.48	2.12	0.90	93603	30.86	74259
Bubble	13.52	13.43	2.34	1.11	85774	30.09	74146
Torso I	11.02	10.99	2.05	1.29	92508	30.41	76749
Torso II	11.15	11.14	2.05	1.20	92574	29.64	76992
Blunt Fin	6.00	5.99	1.31	0.54	98587	26.36	78493
average	11.06	11.01	2.04	1.07	91313	30.29	75751

Table 3.7: Cut-border machine: consumed storage for connectivity, border and quantized vertex coordinates. Running time for connectivity alone and together with vertex coordinates in tetrahedra per second on a Pentium II 350MHz.

mesh	$\frac{\mathcal{C}_{std}}{\mathcal{C}_{CB}^{adapt}}$	$\frac{\mathcal{C}_{CB}^{adapt}}{v}$	$\frac{\mathcal{C}_{G\&F}}{v}$
Random	18.39	15.12	44.03
Proto	22.61	9.55	31.29
Bubble	22.22	13.52	40.18
Torso I	27.27	11.02	37.45
Torso II	27.29	11.15	38.01
Blunt Fin	48.90	6.00	32.06

Table 3.8: Comparison of the different approaches.

Additionally the performance only depends on the number of tetrahedra so, if the original data is unstructured, there is no further need to resample the data for rendering. Thus unstructured volumes can be rendered directly in real-time at very high quality. Making this adaptive representation ideal for rendering extended volumes.

The lossless connectivity compression scheme for tetrahedral meshes can handle non manifold borders. The implementation of the cut-border machine showed that it achieves very high compression rates and is able to compress tetrahedral connectivity to about two bits per tetrahedron, which is between three and four times better than previously reported results. Lossy compression of vertex coordinates turned out to be not as efficient as in the triangular case but still valuable for most applications. Future work must concentrate on more sophisticated compression techniques for the vertex coordinates and further data attached to the tetrahedral mesh.

Chapter 4

Displacement Mapping

Displacement mapping adds real surface detail to objects in three-dimensional scenes by using two dimensional maps containing height data. Displacement mapping can be used for generating large scale objects such as terrain and for adding smaller scale detail such as bumps. Displacement mapping is used in offline cinematic content creation packages to add this surface detail and as the capabilities of graphics hardware increases it can also be used in real time applications.

Displacement mapping is performed by displacing the position of a surface along the normal to the surface by a distance sampled from a map of scalar values associated with the surface. The displacement can be applied to vertices that make up the mesh of the base surface with displacement values associated with each vertex. Alternatively many algorithms insert new vertices into the surface to increase the base mesh detail either using a fixed tessellation factor or by inserting vertices adaptively based on the detail in the displacement map. The displacement map is typically a two dimensional area used in a similar manner to a texture map with the base mesh containing coordinates that indicate where the displacement map is to be sampled.

Displacement mapping was first mentioned by Cook [Coo84] in the context of software based rendering. Techniques for ray tracing displacement maps have been presented previously by Heidrich et al.[HS98] and Pharr et al.[PH96]. But they are still more complex than can be implemented on the current generation of programmable graphics hardware.

In recent years several proposals for dedicated hardware have been proposed for displacement mapping. Doggett et al. [DK99, DKS01] presented a level of detail driven rasterization approach that inserts new vertices into the base mesh. A similar technique is presented by Gumhold et al. [GH99]. Doggett and Hirche [DH00] presented an adaptive tessellation scheme that inserts new vertices dependent on the average displacement within the displacement map and the variance of the surface. Moule and McCool [MM02] improved upon the area coverage for detecting change in the displacement map to drive a similar adaptive scheme. Hirche and Ehlert [HE02] use a pre-computed decision to drive tessellation eliminating the need for computing tessellation decisions. All of these approaches require the creation of new vertices in the vertex shader stage of existing graphics hardware, a feature which has only recently been exposed in a limited fashion through the concept of output from the vertex shader being sent to vertex arrays. This requires that the target vertex arrays are sized correctly for a CPU calculated number of vertices.

Recently introduced hardware by Matrox [Mat02] allows fetching from a displacement map within the vertex shader, a feature not available on other hardware. But like DirectX 9 displacement mapping tessellation is done at a pre set tessellation level which is not controllable from within the vertex shader.

Using a similar approach to that presented in this chapter, Kautz et al. [KS01] extrude the base mesh to enclose the entire displaced surface and then composite together slices through the extruded volume using a technique similar to volume rendering. Since all slices through the volume are rendered whether visible or not this technique requires high fill rates and a high texture bandwidth.

Hirche et al. [HEG$^+$04] presents an approach to displacement mapping using currently available programmable graphics hardware that creates the appearance of a displaced surface on a per pixel basis. Unlike previous techniques it doesn't require any insertion of vertices to re-tessellate the mesh. Displacement map sampling occurs in the pixel shader so all texture filtering modes can be applied. Many of the schemes above include level of detail control tied to triangle size computed in screen space, this step is not required using this technique since the sampling of the displacement map is relative to the number of pixels contained within the bounding prisms of each displaced triangle.

4.1 General Purpose Algorithm

Most approaches to displacement mapping require that the geometry of a given base mesh can be modified, especially in the sense of adding more detail in the form of re-tessellated triangles. Currently available hardware, at which this algorithm is targeted, does not allow vertices to be added once the geometry has been transferred to the graphics card. To work with this restriction this algorithm does not generate geometry on the card, but instead creates triangles that cover the area on the screen that could be affected by the displaced base mesh triangle. When the covering triangles are rasterized a per pixel calculation is performed to detect an intersection with the displaced surface. The number of covered triangles should be kept to a minimum to reduce the geometry transfer overhead. The bounding volume of the surface with a displacement map applied to it is given by a prism obtained by displacing the base triangle along the vertex normals to the maximum displacement height.

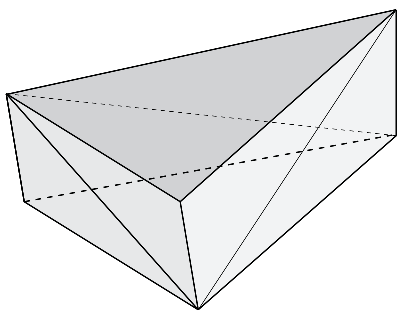


Figure 4.1: The prism with its resulting triangles used for rendering.

At each pixel of the prism's triangles a non-trivial intersection of the ray with the prism has to be performed, placing a very high burden on the pixel shader pipeline.

Since back-facing triangles can be culled, the amount of used pixels to be drawn is relatively limited. The resulting triangles are shown in Figure 4.1. The sides of the prism are quads and have to be split into two triangles, the bottom and top of the prism remain unchanged resulting in eight triangles to be rendered per base triangle.

The fundamental problem with the prism rendering is, that the faces of the prism cannot be interpolated linearly but are bi-linear. Thus the shader may intersect a ray with the displacement map at two different positions. Thus the prism have to be split into tetrahedra to avoid any disambiguates.

4.2 Hardware Acceleration

4.2.1 Single pass prism Renderer

The first approach renders the displacement prism by splitting it up into eight triangles and casting rays into the prism from every rendered pixel. The rays are cast in the viewing direction from each pixel position. To find out whether the ray intersects the displacement map, the height of the sampling position is compared to the height of the displacement map at the interpolated texture coordinate of the sampling position. The height ranges from zero at the base mesh level to one on the top of the prism. The 3D texture coordinates need to be interpolated inside the prism, along the viewing direction. The texture coordinates are local to the prism and a base transform has to be made at all vertex positions to obtain the viewing direction in local texture space. Given a triangle with vertices $V_i = (x_i, y_i, z_i)$ with normals N_i and texture coordinates $U_i = (u_i, v_i)$ for i = 1,2,3, the first step is to add a third coordinate defined by the height of the vertex in the prism:

$U'_i := (u_i, v_i, 0)$ for vertices of the base triangle and $U'_i := (u_i, v_i, 1)$ for vertices of the displaced triangle. To calculate the transformation for vertex V_1 for example, on the base triangle, we define a local base B_{Texture} with the texture directions e_1, e_2 along the triangle edges:

$$\begin{aligned} e_1 &:= U'_2 - U'_1 \\ e_2 &:= U'_3 - U'_1 \\ B_{\text{Texture}} &:= (e_1, e_2, 1) \end{aligned} \tag{4.1}$$

In the same manner we define a local base B_{World} with the world coordinates of the vertices:

$$\begin{aligned} f_1 &:= V_2 - V_1 \\ f_2 &:= V_3 - V_1 \\ B_{\text{World}} &:= (f_1, f_2, N_1) \end{aligned} \tag{4.2}$$

The basis transformation from B_{World} to B_{Texture} can be used to move the viewing direction at the vertex position V_1 to local texture space.

To avoid sampling outside of the prism, the exit point of the viewing ray has to be determined. In texture space the edges of the prism are not straightforward to detect and a 2D intersection calculation has to be performed. This can be overcome by defining a second local coordinates system which has its axes aligned with the prism edges. For this we assign 3D coordinates to the vertices as shown in Figure 4.2. The respective name for the new coordinate for a vertex V_i is O_i.

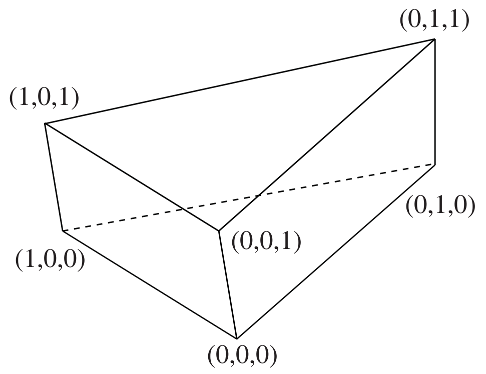


Figure 4.2: The vectors used to define the second local coordinate system for simpler calculation of the ray exit point.

Then the the viewing direction can be transformed in exactly the same manner to the local coordinate system defined by the edges between the O_i vectors:

$$\begin{aligned} g_1 &:= O_2 - O_1 \\ g_2 &:= O_3 - O_1 \\ B_{\text{Local}} &:= (g_1, g_2, 1). \end{aligned} \tag{4.3}$$

Again this is the example for the vertex V_1. In the following the local viewing direction in texture space is called *View*$_T$, and in the B_{Local} base representation *View*$_L$. We assume that the viewing direction changes linearly over the face of a prism triangle and put the local viewing direction in both coordinate systems in 3D texture coordinates and use them as input to the fragment shader pipeline in order to get linearly interpolated local viewing directions. The interpolated *View*$_L$ allows us to very easily calculate the distance to the backside of the prism from the given pixel position as it is either the difference of the vector coordinates to 0 or 1 depending which side of the prism we are rendering. With this Euclidean distance we can define the sampling distance in a sensible way which is important as the number of samples that can be read in one pass is limited, and samples should be evenly distributed over the distance. An example of this algorithm is shown in Figure 4.3. In this case four samples are taken inside the prism. The height of the displacement map is also drawn for the vertical slice hit by the viewing ray. The height of the third sample which is equal to the third coordinate of its texture coordinate as explained earlier, is less than the displacement map value and thus a hit with the displaced surface is detected.

Although the pixel position on the displaced surface is now calculated, the normal at this position is still the interpolated normal of the base mesh triangle, it has to be perturbed for correct shading, in this case standard bump mapping using a pre-calculated bump map derived from the used displacement map is used. The bump map can be stored together with the displacement map in one texture, with the displacement stored in the alpha channel.

There is a fundamental problem in this approach. The surface of the prism is not planar, but bi-linear for the sides or even a bezier patch on the top. Therefore the calculation of the sampling positions is very complex and we have to ensure that we always render the outer surface of the prism. In order to solve this correctly, we would

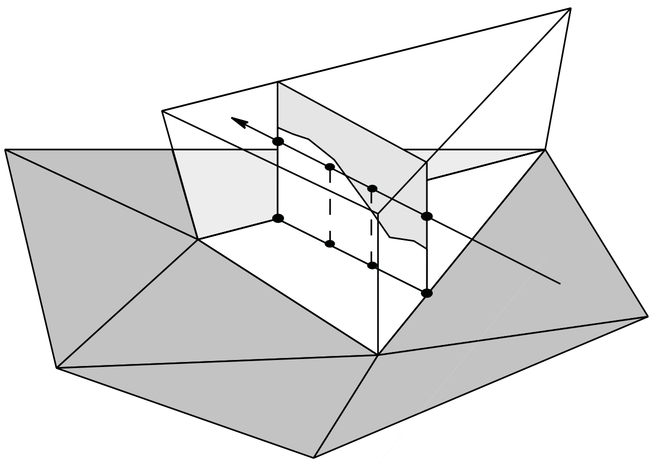


Figure 4.3: Sampling within the extruded prism with a slice of the displacement map shown.

have to solve an equation with 3 unknowns that is only partly linear and therefore not solvable in general. If we restrict the prisms to have flat sides, we still run into numerical problems with this approach.

4.2.2 Tetrahedral Renderer

Numerical problems can be reduced by simplifying the shape used that the rays are cast through. The prism is geometrically complex for performing intersection calculations. Obviously the prism can be split into three tetrahedra as shown in Figure 4.4. This also reduces the bi-linear surface into two triangles. The main difference in using tetrahedra instead of the prism is that the texture space coordinates of the entry and exit point can be interpolated at the same time by the rasterization units. The sampling points between the entry and exit point can then be obtained by just linearly interpolating in between them. The tetrahedra can be rendered using an adaptation of the projected tetrahedra (PT) Algorithm by Shirley and Tuchman[ST90].

4.2.3 Mesh Construction

Using tetrahedra requires the construction of a tetrahedral mesh from the given triangle base mesh. When this is done we have to ensure that neighboring tetrahedral edges are aligned in a consistent way to avoid aliasing between adjacent triangles. This can be achieved without using any knowledge of connectivity in the tetrahedral mesh. All that is needed is a consistent numbering of the vertices in the mesh which is usually just given by the vertex indices in a given array. The algorithm iterates over all faces in the triangle mesh folding up a prism by displacing every vertex of the base triangle along the vertex normal direction. To adjust the amount of displacement you can multiply the normal with a user defined scalar. Every prism is then split into three tetrahedra following the ordering scheme as schematically shown in Figure 4.4. We assign the indices v0, v1, v2 to the lower vertices and v3, v4, v5 to the upper base vertices. Now every prism is tiled into the three tetrahedra v0-v1-v2-v5, v0-v1-v4-v5 and v0-v3-v4-v5. An additional requirement is that v0 $<$ v1 $<$ v2 with respect to the consistent numbering scheme of the mesh as noted before. Hence the algorithm simply works this way:

```
FOR_EVERY_TRIANGLE_FACE(f)

  IF(v0 > v1)
    SWAP(v0, v1)
    SWAP(v3, v4)

  IF(v0 > v2)
    SWAP(v0, v2)
    SWAP(v3, v5)

  IF(v1 > v2)
    SWAP(v1, v2)
    SWAP(v4, v5)

  CREATE_TETRA(v0, v1, v2, v5)
  CREATE_TETRA(v0, v1, v4, v5)
  CREATE_TETRA(v0, v3, v4, v5)
```

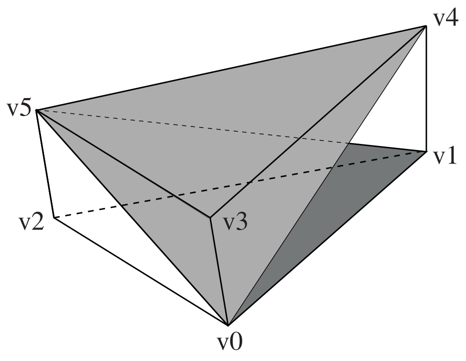


Figure 4.4: Subdivision of prism into three tetrahedra (v0-v1-v2-v5, v0-v1-v4-v5, v0-v3-v4-v5).

4.2.4 Rendering

To adapt the PT-algorithm to displacement mapping only a few modifications have to be applied. In contrast to the standard algorithm where each vertex needs color and opacity, each vertex is attributed with its respective tangent space consisting of normal, tangent and bi-normal, each a 3d-vector. Additionally two texture coordinates, one for the bump-/displacement map, the other for a freely usable texture, are assigned to each vertex. Before the geometry is sent down the rendering pipeline a view-dependent preprocessing step has to be performed where the tetrahedra are decomposed into triangles according to the PT-algorithm. This way one tetrahedron is decomposed into four to six triangles, a possible decomposition of one tetrahedron is depicted in Figure 4.5. The point S marks the intersection between a front side and a backside edge in the view-plane. The intersection point on the backside edge in world coordinates is referred to as secondary vertex later in the text. So far all the processing has to be done on the driver side by the host computer's CPU.

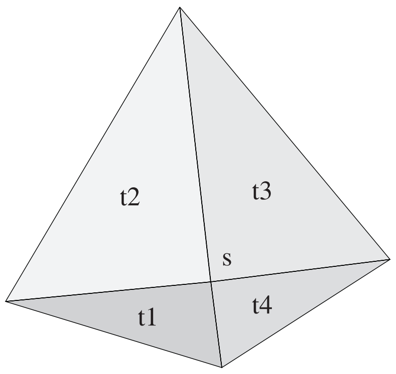


Figure 4.5: One possible decomposition of tetrahedron into triangles. Intersection of front and back edge at point S in the viewplane.

Every triangle vertex (primary vertex) sent into the first stage is attributed with texture coordinates and tangent space vectors. Likewise the vertex on the backside (secondary vertex) of the decomposed tetrahedron is transferred as attribute including its texture coordinates and tangent space vectors. With these parameters the vertex shader computes homogeneous texture coordinates for the primary and secondary vertex. It also computes the model-view-projection transformation of the vertices and finally transforms per vertex viewing and light direction into tangent space.

In the second stage of this pipeline the pixel shader performs the intersection calculation between eye vector and the displacement map. To achieve this the pixel shader performs four lookups in the displacement map given by the interpolated texture coordinates of the primary and secondary vertex and two interpolated positions in between. The intersection between eyevector and displaced surface is then calculated by subtraction of the sampled displacement value from the interpolated texture coordinates. A sign change indicates the interval where the eye-vector hits the displaced surface. In case no surface was hit the pixel is killed. Otherwise the pixel can now undergo a final shading step. In our case bump mapping was used to perturb the interpolated normal and Phong shading performed using the fragment shader stage.

OpenGL vertex program

The vertex program of the displacement mapping is very similar to the tetrahedral renderer. However, the number of attributes to be passed to the fragment program has increased. The functionality can be explained as follows.

1. Transform the front vertex coordinates with the model-view-projection matrix (just like the legacy pipeline).

2. Project the transformed vertex into camera space.

3. Project back vertex coordinates with the model-view-projection matrix.
4. Write homogeneous texture coordinates (front and back).
5. Write eye vectors (front and back) in homogeneous coordinates.
6. Write light vectors (front and back) in homogeneous coordinates.

Again, the shader now has the correctly interpolated values on both the front and back face of each spat.

```
!!ARBvp1.0

ATTRIB    iPos0    =   vertex.attrib[ 0];
ATTRIB    iNormal0 =   vertex.attrib[ 1];
ATTRIB    iDisp0   =   vertex.attrib[ 2];
ATTRIB    iTex0    =   vertex.attrib[ 3];
ATTRIB    iTang0   =   vertex.attrib[ 4];
ATTRIB    iBinorm0 =   vertex.attrib[ 5];
ATTRIB    iPos1    =   vertex.attrib[ 6];
ATTRIB    iNormal1 =   vertex.attrib[ 7];
ATTRIB    iDisp1   =   vertex.attrib[ 8];
ATTRIB    iTex1    =   vertex.attrib[ 9];
ATTRIB    iTang1   =   vertex.attrib[10];
ATTRIB    iBinorm1 =   vertex.attrib[11];

PARAM     mvp[4]   = { state.matrix.mvp };
PARAM     iEye     =   state.matrix.modelview[0].
                       invtrans.row[3];
PARAM     iLight   =   state.light[0].position;

TEMP      bPos;
TEMP      fPos;
TEMP      tmp;
TEMP      rPos;
TEMP      light;
TEMP      eye;

OUTPUT    oPos     =   result.position;
OUTPUT    oDisp0   =   result.texcoord[0];
OUTPUT    oDisp1   =   result.texcoord[1];
OUTPUT    oLight0  =   result.texcoord[2];
OUTPUT    oLight1  =   result.texcoord[3];
OUTPUT    oEye0    =   result.texcoord[4];
OUTPUT    oEye1    =   result.texcoord[5];
OUTPUT    oTex0    =   result.texcoord[6];
OUTPUT    oTex1    =   result.texcoord[7];
```

```
# transform front vertex                                   ①
DP4       fPos.x, mvp[0], iPos0;
DP4       fPos.y, mvp[1], iPos0;
DP4       fPos.z, mvp[2], iPos0;
```

```
DP4       fPos.w, mvp[3], iPos0;
MOV       oPos, fPos;

# make un-homogeneous                                          ②
MOV       rPos, fPos.w;
RCP       tmp, fPos.w;
MUL       fPos, fPos, tmp;

# transform back vertex                                        ③
DP4       bPos.x, mvp[0], iPos1;
DP4       bPos.y, mvp[1], iPos1;
DP4       bPos.z, mvp[2], iPos1;
DP4       bPos.w, mvp[3], iPos1;
RCP       tmp, bPos.w;
MUL       rPos, rPos, tmp;
MUL       bPos, bPos, tmp;

# write homogeneous texture coordinates and z-values           ④
MOV       oDisp0.xyz, iDisp0;
MOV       oDisp0.w, fPos.z;
MOV       oTex0.xyz, iTex0;
MOV       oTex0.w, 1.0;
MUL       oDisp1.xyz, iDisp1, rPos;
MUL       oDisp1.w, bPos.z, rPos;
MUL       oTex1.xyz, iTex1, rPos;
MOV       oTex1.w, rPos;

# write homogeneous eye vectors                                ⑤
ADD       eye, -iPos0, iEye;
DP3       eye.w, eye, eye;
RSQ       eye.w, eye.w;
MUL       eye, eye, eye.w;

DP3       oEye0.x, eye, iTang0;
DP3       oEye0.y, eye, iBinorm0;
DP3       oEye0.z, eye, iNormal0;
MOV       oEye0.w, 1.0;

DP3       tmp.x, eye, iTang1;
DP3       tmp.y, eye, iBinorm1;
DP3       tmp.z, eye, iNormal1;
MOV       tmp.w, 1.0;
MUL       oEye1, tmp, rPos;

# write homogeneous light vectors                              ⑥
ADD       light, -iPos0, iLight;
DP3       light.w, light, light;
RSQ       light.w, light.w;
MUL       light, light, light.w;
```

```
DP3       oLight0.x, light, iTang0;
DP3       oLight0.y, light, iBinorm0;
DP3       oLight0.z, light, iNormal0;
MOV       oLight0.w, 1.0;

ADD       light, -iPos1, iLight;
DP3       light.w, light, light;
RSQ       light.w, light.w;
MUL       light, light, light.w;

DP3       tmp.x, light, iTang1;
DP3       tmp.y, light, iBinorm1;
DP3       tmp.z, light, iNormal1;
MOV       tmp.w, 1.0;
MUL       oLight1, tmp, rPos;

END
```

OpenGL Fragment Program

The main tasks of the fragment program are to determine if and where the viewing ray intersects the displaced surface and to shade the fragment according to the location of the intersection. These tasks are implemented in the following way.

1. Project the back vertex position into 3D space.
2. Compute the sample positions for the displacement map.
3. Sample the displacement map.
4. Calculate the position of the intersection.
5. Exit if the ray did not hit the surface or continue calculating the exact position.
6. Project the remaining homogeneous coordinates into 3D space. Then interpolate front and back values for the sample position.
7. Calculate the final color using bump mapping.
8. Lit the fragment with Phong shading.

The final fragments are then drawn into the frame buffer with the z-buffer enabled in order to find the closest intersections for all tetrahedra.

```
!!ARBfp1.0

ATTRIB    iDisp0    =   fragment.texcoord[0];
ATTRIB    iDisp1    =   fragment.texcoord[1];
ATTRIB    iLight0   =   fragment.texcoord[2];
ATTRIB    iLight1   =   fragment.texcoord[3];
ATTRIB    iEye0     =   fragment.texcoord[4];
ATTRIB    iEye1     =   fragment.texcoord[5];
ATTRIB    iTex0     =   fragment.texcoord[6];
ATTRIB    iTex1     =   fragment.texcoord[7];
```

```
PARAM     lAmbient  =   state.light[0].ambient;
PARAM     lDiffuse  =   state.light[0].diffuse;
PARAM     lSpecular =   state.light[0].specular;

TEMP      disp1;
TEMP      smp0;
TEMP      smp1;
TEMP      signs;
TEMP      persp;
TEMP      tmp;
TEMP      loc;
TEMP      tmp1;
TEMP      tmp2;
TEMP      tmp3;

ALIAS     light1    =   smp0;
ALIAS     eye1      =   smp1;
ALIAS     tex1      =   signs;
ALIAS     color     =   loc;

# project disp1                                         ①
RCP       persp, iTex1.w;
MUL       disp1, iDisp1, persp;

# compute sample positions                              ②
ADD       tmp, -iDisp0, disp1;
MAD       smp0, 0.333, tmp, iDisp0;
MAD       smp1, 0.667, tmp, iDisp0;

# sample displacement map                               ③
TEX       tmp, iDisp0, texture[0], 2D;
TEX       tmp1, smp0, texture[0], 2D;
TEX       tmp2, smp1, texture[0], 2D;
TEX       tmp3, disp1, texture[0], 2D;
ADD       loc.r, -tmp.a, iDisp0.b;
ADD       loc.g, -tmp1.a, smp0.b;
ADD       loc.b, -tmp2.a, smp1.b;
ADD       loc.a, -tmp3.a, disp1.b;
ADD       loc, loc, {0.001, 0.0, 0.0, -0.001};

# calculate hit location                                ④
SWZ       signs, loc, g, b, a, r;
MUL       signs, loc, signs;
CMP       tmp, -signs, 2.0, {0.0, 0.333, 0.667, 2.0};
MIN       tmp1, tmp, tmp.abgr;
MIN       tmp, tmp1.r, tmp1.g;
ADD       tmp1, 1.0, -tmp.r;
```

```
# kill if surface was not hit                                  ⑤
KIL       tmp1;
ADD       tmp2, tmp, {0.167, -0.167, -0.5, -0.833};
CMP       loc.r, tmp2.g, loc.r, loc.g;
CMP       loc.g, tmp2.g, loc.g, loc.b;
CMP       loc.r, tmp2.b, loc.r, loc.b;
CMP       loc.g, tmp2.b, loc.g, loc.a;
ADD       tmp2, loc.r, -loc.g;
MUL       tmp2, tmp2, 3.0;
RCP       tmp2, tmp2.r;
MAD       tmp2, tmp2, loc.r, tmp;

# project light1, eye1 and tex1 the interpolate                ⑥
ADD       disp1, -iDisp0, disp1;
MAD       disp1, tmp2, disp1, iDisp0;
MAD       tex1, persp, iTex1, -iTex0;
MAD       tex1, tmp2, tex1, iTex0;
MAD       light1, persp, iLight1, -iLight0;
MAD       light1, tmp2, light1, iLight0;
MAD       eye1, persp, iEye1, -iEye0;
MAD       eye1, tmp2, eye1, iEye0;

# calculate final color using bump mapping                     ⑦
TEX       color, tex1, texture[1], 2D;
TEX       tmp, disp1, texture[0], 2D;
MAD       tmp, tmp, 2.0, -1.0;
DP3       tmp.a, tmp, tmp;
RSQ       tmp.a, tmp.a;
MUL       tmp.rgb, tmp, tmp.a;
DP3       tmp1.a, light1, light1;
RSQ       tmp1.a, tmp1.a;
MUL       tmp1.rgb, light1, tmp1.a;
DP3       tmp2.a, eye1, eye1;
RSQ       tmp2.a, tmp2.a;
MUL       tmp2.rgb, eye1, tmp2.a;

# lit fragment with Phong shading                              ⑧
DP3       tmp2.a, tmp2, tmp;
MAD       tmp2.rgb, tmp, tmp2.a, -tmp2;
MAD       tmp2.rgb, tmp, tmp2.a, tmp2;
DP3_SAT   tmp, tmp, tmp1;
MUL       tmp, tmp, lDiffuse;
ADD       tmp, tmp, lAmbient;
DP3_SAT   tmp1, tmp2, tmp1;
LG2       tmp1, tmp1.r;
MUL       tmp1, tmp1, 128.0;
EX2       tmp1, tmp1.r;
MUL       tmp1, tmp1, lSpecular;
MAD       result.color.rgb, color, tmp, tmp1;
```

```
MOV       result.color.a, 1.0;

END
```

4.2.5 Performance Optimizations

Up to now, the performance is still limited by the fill rate of the graphics card, i.e. the fragments that are processed during rendering. There are two possibilities to reduce the number of fragments that pass through the pixel shader.

The most commonly used fill rate optimization is to use the early-z-test in order to terminate fragments before they reach the fragment shader. This can easily be implemented by roughly sorting the tetrahedra into a front-to-back order. Since we are not interested in the exact visibility order of the tetrahedra, we can use the center of each tetrahedra and a simple bucket sort algorithm.

But even with this optimization, the rendering is still very slow since most of the fragments are terminated within the fragment program. This is especially true for the silhouette of the displaced surface. So the second optimization is to adapt the height of the tetrahedra above the base surface.

Since each prism above a triangle is the bounding box of the displaced triangle, we can simply shrink the prism using the values stored in the displacement map. For each triangle we now sample the displacement map on the CPU, storing only minimum and maximum displacement value. After that we update the minimum and maximum values for each vertex. The minimum value of the vertex is the lowest minimum value of all adjacent triangles, while the maximum value corresponds to the highest maximum value of all adjacent triangles. The optimized mesh can be seen in Figure 4.6.

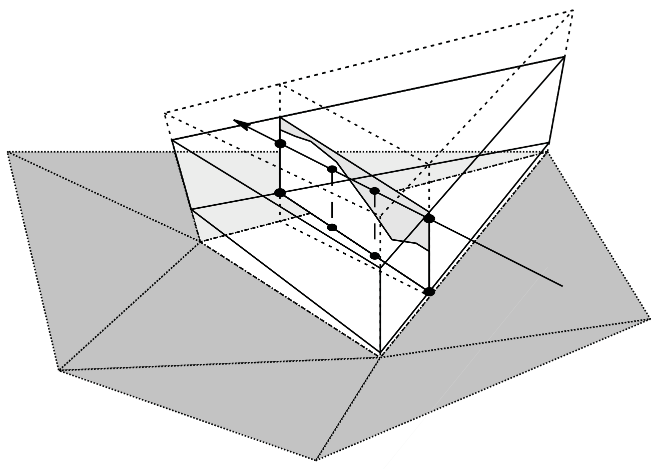


Figure 4.6: Optimized base mesh.

These optimizations together reduce the fill rate in a way, that real-time rendering with displacement mapping becomes feasible.

4.2.6 Results

The tetrahedral renderer was implemented using OpenGL vertex and fragment programs. The problems of using tetrahedra as primitives are of course the amount of additional geometry that has to be transformed and rendered. The pixel shader needs a single rendering pass on an nVidia GeForceFX 5800 or ATI RADEON 9700.

In our implementation four samples along a ray inside a tetrahedra are taken and compared with the displacement map. To avoid sampling artifacts by missing a surface completely the size of the tetrahedra and thus the size of the used base triangle mesh has to be chosen appropriately. Longer pixel shader programs will allow more samples to be taken improving the sampling quality. Figure 4.7 a) and b) shows a flat base triangle meshes with half donut as a displacement map. Figure 4.8 a) shows the crater lake displacement map applied to the same base mesh.

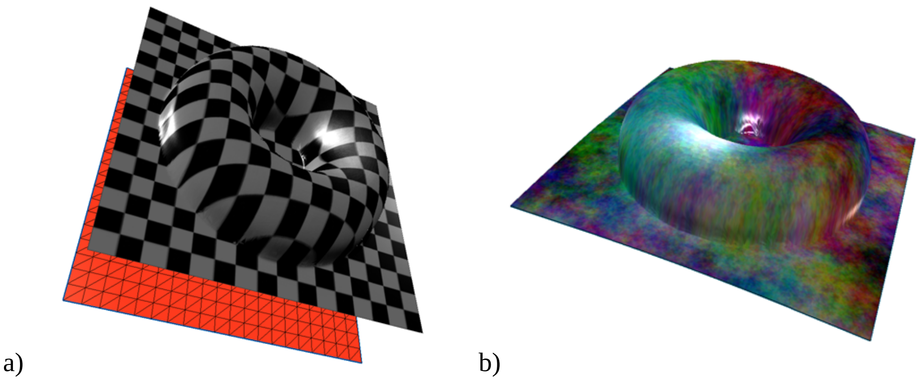


Figure 4.7: Flat base mesh with a half donut shape applied to it (a). The base mesh in red is translated away for better visibility. Same shape with a different texture applied (b).

In Figure 4.8 b) we used a cylinder shaped mesh and applied the displacement map of a laser range scan of a human head.

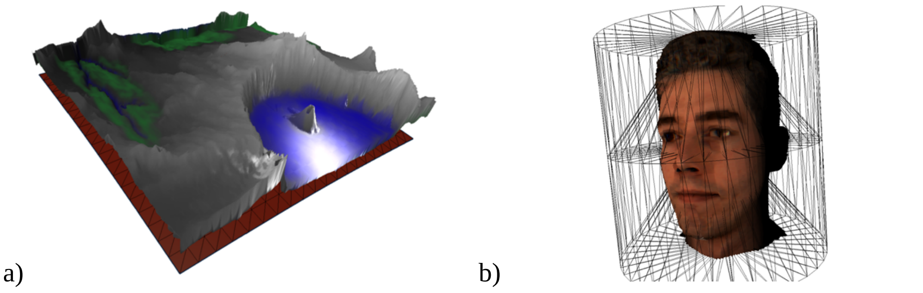


Figure 4.8: Displacement Map of Crater Lake applied to a flat base mesh (a). The head of Volker Blanz displaced from a cylindrical mesh, tetrahedral mesh show (b).

We then used a sphere shaped mesh and applied the displacement map of an earth height field (see Figure 4.9). It is possible to add a texture additionally to displacement mapping, even with different texture coordinates than the displacement maps, allowing for light maps, etc.

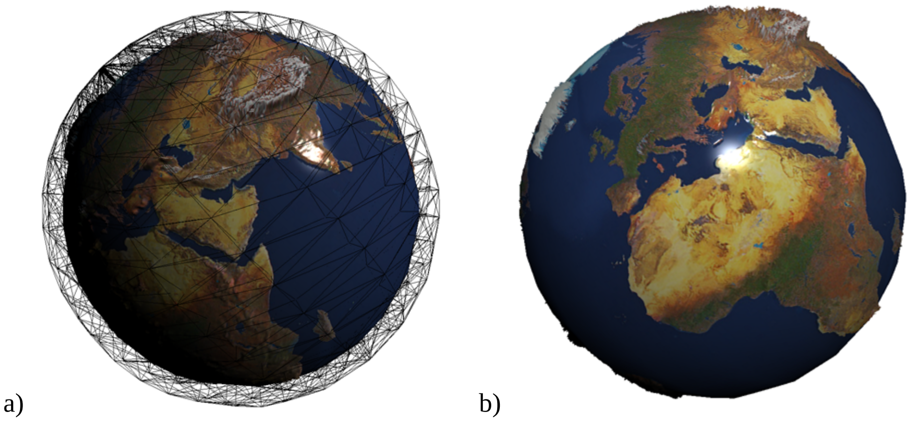


Figure 4.9: Sphere shaped base mesh with a earth displacement map and texture applied to it. Additionally the wire-frame of the tetrahedral mesh is shown (a). Different angle, this time showing Europe with slightly exaggerated displacements (b).

Frame rates for the shown examples were clearly pixel shader limited. Our implementation is capable of rendering at 20fps at 500x500 pixels resolution without the optimizations and about 40fps with the silhouette optimizations in place. As all rendering was done in immediate mode, there is certainly an opportunity for optimizations as soon as we are no longer fill rate limited.

4.3 Conclusion & Future Work

Displacement mapping can be used to reduce the bandwidth from CPU to GPU by only requiring displacement maps to be sent to the GPU's memory to generate more complex geometry without sending large vertex arrays. This also reduces the constraints of limited GPU memory.

The pixel based algorithm presented in this paper performs at interactive rates on currently available hardware. Sampling of the displacement map is driven by visible pixels unlike most previous displacement mapping approaches that are driven by retessellation of the base mesh using various schemes.

The approach does not require the use of render to vertex and does not require any modifications to existing programmable graphics hardware. But could be improved by increased pixel shader lengths which would allow more samples of the displacement map to be taken avoiding undersampling. Also control flow in the pixel shader could allow loops to increment along the ray until the intersection with the surface is found. This approach could be improved using many existing ray tracing techniques that improve intersection calculations with surfaces, for example octrees and space leaping.

The availability of displacement mapping for real time rendering can be used to raise the level of realism in applications that generate three-dimensional worlds.

Bibliography

[Ake93] K. Akeley. RealityEngine Graphics. In *Computer Graphics*, Proceedings of ACM SIGGRAPH, pages 109–116, August 1993.

[BIKP99] C. Bajaj, I. Ihm, G. Koo, and S. Park. Parallel Ray Casting of Visible Human on Distributed Memory Architectures. In *Data Visualization*, Eurographics, pages 269–276, May 1999.

[CDL$^+$96] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, and J. Snyder. Fast Rendering of Complex Environments Using a Spatial Hierarchy. In *Graphics Interface*, pages 132–141, May 1996.

[CDSY96] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. Wavelet Transforms that Map Integers to Integers. Technical report, Department of Mathematics, Princeton University, 1996.

[CHF96] W. O. Cochran, J. C. Hart, and P. J. Flynn. Fractal Volume Compression. *IEEE Transactions on Visualization and Computer Graphics*, 2(4):313–322, December 1996.

[CKM$^+$99] J. Comba, J. T. Klosowski, N. L. Max, J. S. B. Mitchell, C. T. Silva, and P. L. Williams. Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids. *Computer Graphics Forum (Proceedings of Eurographics '99)*, 18(3):369–376, 1999.

[Coo84] R. L. Cook. Shade Trees. In *Computer Graphics*, Proceedings of ACM SIGGRAPH, pages 223–231, July 1984.

[CPS97] P. Cignoni, E. Puppo, and R. Scopigno. Multiresolution Representation and Visualization of Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):352–369, October 1997.

[Dau92] I. Daubechies. *Ten Lectures on Wavelets*, volume 61 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[Deu96] L. P. Deutsch. RFC 1952: GZIP file format specification version 4.3, May 1996.

[DH00] M. Doggett and J. Hirche. Adaptive View Dependent Tessellation of Displacement Maps. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000*, pages 59–66, 2000.

[DK99] M. Doggett and A. Kugler. A Hardware Architecture for Displacement Mapping using Scan Conversion. Technical Report WSI–99–12, Wilhelm-Schickard-Institut für Informatik, University of Tübingen, Germany, 1999.

[DKS01] M. Doggett, A. Kugler, and W. Straßer. Displacement Mapping using Scan Conversion Hardware Architectures. *Computer Graphics Forum*, 20(1):13–26, March 2001.

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 9–16, August 2001.

[EWG99] T. Ertl, R. Westermann, and R. Grosso. Multiresolution and hierarchical methods for the visualization of volume data. *Future Generation Computer Systems*, 15(1):31–42, February 1999.

[FF01] J. E. Fowler and D. N. Fox. Embedded Wavelet-Based Coding of Three-Dimensional Oceanographic Images with Land Masses. *IEEE Transactions on Geoscience and Remote Sensing*, pages 284–290, February 2001.

[FGR85] G. Frieder, D. Gordon, and R. A. Reynolds. Back-to-Front Display of Voxel-Based Objects. *IEEE Computer Graphics & Applications*, 5(1):52–60, January 1985.

[FMS00] R. Farias, J. S. B. Mitchell, and C. T. Silva. ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering. In *IEEE Symposium on Volume Visualization*, pages 91–99, 2000.

[GGS99] S. Gumhold, S. Guthe, and W. Straßer. Tetrahedral Mesh Compression with the Cut-Border Machine. In *IEEE Visualization*, pages 51–58, October 1999.

[GGS01] S. Guthe, S. Gumhold, and W. Straßer. Texture Particles: Interactive Visualization of Volumetric Vector Fields. In *Workshop über Trends und Höhepunkte der Graphischen Datenverarbeitung*, pages 13–23, 2001.

[GGS02] S. Guthe, S. Gumhold, and W. Straßer. Interactive Visualization of Volumetric Vector Fields Using Texture Based Particles. In *Proceedings of WSCG*, 2002.

[GH99] S. Gumhold and T. Hüttner. Multiresolution Rendering with Displacement Mapping. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 55–66, August 1999.

[GLDH97] M. H. Gross, L. Lippert, R. Dittrich, and S. Häring. Two Methods for Wavelet-Based Volume Rendering. *Computers & Graphics*, 21(2):237–252, 1997.

[GRS$^+$02] S. Guthe, S. Roettger, A. Schieber, W. Straßer, and T. Ertl. High-Quality Unstructured Volume Rendering on the PC Platform. In *ACM Siggraph/Eurographics Hardware Workshop*, 2002.

[GS98] S. Gumhold and W. Straßer. Real Time Compression of Triangle Mesh Connectivity. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 133–140, July 1998.

[GS01] S. Guthe and W. Straßer. Real-time Dekompression and Visualization of Animated Volume Data. In *IEEE Visualization*, pages 349–356, October 2001.

[GS04] S. Guthe and W. Straßer. Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. *Computers & Graphics*, 28(1):51–58, February 2004.

[GWGS02] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive Rendering of Large Volume Data Sets. In *IEEE Visualization*, pages 53–60, October 2002.

[HE02] J. Hirche and A. Ehlert. Curvature-Driven Sampling of Displacement Maps. presented at ACM SIGGRAPH 2002 as a Technical Sketch, July 2002.

[HEG$^+$04] J. Hirche, A. Ehlert, S. Guthe, M. Dogget, and W. Straßer. Hardware Accelerated Per-Pixel Displacement Mapping. In *Graphics Ingterface*, May 2004. to be published.

[Hop96] H. Hoppe. Progressive meshes. In *ACM SIGGRAPH 96 Conference Proceedings*, pages 99–108, August 1996.

[HS98] W. Heidrich and H.-P. Seidel. Ray-tracing Procedural Displacement Shaders. In *Graphics Interface*, pages 8–16, 1998.

[IP98] I. Ihm and S. Park. Wavelet-Based 3D Compression Scheme for Very Large Volume Data. In *Graphics Interface*, pages 107–116, June 1998.

[ISO93] ISO/IEC. MPEG-1 Coding of Moving Pictures and Associated Audio for Digital Storage Media at Up to About 1,5 Mbit/s. *ISO/IEC 11172*, 1993.

[ISO96] ISO/IEC. MPEG-2 Generic coding of moving pictures and associated audio information. *ISO/IEC 13818*, 1996.

[KD98] G. Kindlmann and J. W. Durkin. Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering. In *Symposium on Volume Visualization (VOLVIS-98)*, pages 79–86, October 1998.

[KE01] M. Kraus and T. Ertl. Cell-Projection of Cyclic Meshes. In *IEEE Visualization*, pages 215–222, October 2001.

[KH84] J. T. Kajiya and T. Von Herzen. Ray Tracing Volume Densities. *Computer Graphics*, 18(3):165–173, July 1984.

[KKH01] J. Kniss, G. Kindelmann, and C. Hansen. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. In *IEEE Visualization*, pages 255–262, October 2001.

[Kni00] G. Knittel. The UltraVis System. Technical Report HPL-2000-100, Hewlett Packard Laboratories, August 2000.

[KS97] G. Knittel and W. Straßer. VIZARD: Visualization Accelerator for Realtime Display. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 139–147. ACM SIGGRAPH / Eurographics, August 1997.

[KS99] T. Kim and Y. Shin. An Efficient Wavelet-based Compression Method for Volume Rendering. In *Pacific Graphics*, pages 147–157, October 1999.

[KS01] J. Kautz and H.-P. Seidel. Hardware Accelerated Displacement Mapping for Image Based Rendering. In *Graphics Interface*, pages 61–70, June 2001.

[KTM$^+$02] A. Kanitsar, T. Theußl, L. Mroz, M. Šrámek, A. V. Bartrolí, B. Csébfalvi, J. Hladůvka, D. Fleischmann, M. Knapp, R. Wegenkittl, P. Felkel, S. Röttger, S. Guthe, W. Purgathofer, and E. Gröller. Christmas Tree Case Study: Computed Tomography as a Tool for Mastering Real World Objects with Applications in Computer Graphics. In *IEEE Visualization*, pages 489–492, October 2002.

[KWHM02] U. Kanus, G. Wetekam, J. Hirche, and M. Meißner. VIZARDII: An FPGA-based Interactive Volume Rendering System. In *Field-Programmable Logic and Applications*, Proceedings of the 12th International Conference on Field-Programmable Logic, pages 1114–1117, September 2002.

[LC87] W. E. Lorensen and H. E. Cline. Marching–Cubes: A High Resolution 3D Surface Construction Algorithm. In *Computer Graphics*, Proceedings of ACM SIGGRAPH, pages 163–169, July 1987.

[Lev88] M. Levoy. Display of Surfaces From Volume Data. *IEEE Computer Graphics & Applications*, 8(5):29–37, May 1988.

[LHJ99] E. C. LaMar, B. Hamann, and K. I. Joy. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *IEEE Visualization*, pages 355–362, October 1999.

[LL94] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp factorization of the Viewing Transform. In *Computer Graphics*, Proceedings of ACM SIGGRAPH, pages 451–457, July 1994.

[Mat02] Matrox. Parhelia. Features presented at http://www.matrox.com/mga/products/parhelia512/technology/disp_map.cfm, 2002.

[Max86] N. Max. Light Diffusion Through Clouds and Haze. *Computer Vision, Graphics, and Image Processing*, 33(3):280–292, March 1986.

[Max95] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[MGS01] M. Meißner, S. Guthe, and W. Straßer. Higher Quality Volume Rendering on PC Graphics Hardware. Technical Report WSI-2001-12, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/GRIS), University of Tübingen, April 2001.

[MGS02] M. Meißner, S. Guthe, and W. Straßer. Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. In *Graphics Interface*, pages 209–218, May 2002.

[MH79] D. Marr and E. Hildreth. Theory of Edge Detection. Technical Report AIM-518, MIT Artificial Intelligence Laboratory, April 1979.

[MHC90] N. Max, P. Hanrahan, and R. Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *San Diego Workshop on Volume Visualization, Computer Graphics*, 24(5):27–33, 1990.

[MHS99] M. Meißner, U. Hoffmann, and W. Straßer. Enabling Classification and Shading for 3D Texture Mapping Based Volume Rendering using OpenGL and Extensions. In *IEEE Visualization*, pages 207–214, October 1999.

[Mic03] Microsoft. DirectX 9 Documentation. Technical documentation, available at http://msdn.microsoft.com, 2003.

[MJC02] B. Mora, J.-P. Jessel, and R. Caubet. A New Object-Order Ray-Casting Algorithm. In *IEEE Visualization*, pages 203–210, October 2002.

[MKS98] M. Meißner, U. Kanus, and W. Straßer. VIZARD II: A PCI-card for Real-Time Volume Rendering. In *Proceedings of the Eurographics / Siggraph Workshop on Graphics Hardware (EUROGRAPHICS-98)*, pages 61–68, August 1998.

[MKW$^+$02] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. VIZARDII: A Reconfigurable Interactive Volume Rendering System. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 137–146, September 2002.

[MM02] K. Moule and M. D. McCool. Efficient Bounded Adaptive Tessellation of Displacement Maps. In *Graphics Interface*, May 2002.

[MZFM98] R. Machiraju, Z. Zhu, B. Fry, and R. Moorhead. Structure Significant Representation of Computational Field Simulation Datasets. *IEEE Transactions on Visualization and Computer Graphics*, April-June 1998.

[Nat86] National Library of Medicine. The Visible Human Project. Data set download and documentation, available at http://www.nlm.nih.gov/research/visible/visible_human.html, 1986.

[NH92] P. Ning and L. Hesselink. Vector Quantization for Volume Rendering. In *Workshop on Volume Visualization*, pages 69–74, October 1992.

[NH93] P. Ning and L. Hesselink. Fast Volume Rendering of Compressed Data. In *IEEE Visualization*, pages 11–18, October 1993.

[NS01] K. G. Nguyen and D. Saupe. Rapid High Quality Compression of Volume Data for Visualization. *Computer Graphics Forum*, 20(3), 2001.

[Nyq28] H. Nyquist. Certain Topics in Telegraph Transmission Theory. *Transactions on A.I.E.E.*, pages 617–644, February 1928.

[PH96] M. Pharr and P. Hanrahan. Geometry Caching for Ray-Tracing Displacement Maps. In *Eurographics Workshop on Rendering*, June 1996.

[PH97] J. Popović and H. Hoppe. Progressive Simplicial Complexes. *Computer Graphics*, 31(Annual Conference Series):217–224, August 1997.

[PHK$^+$99] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro Real-Time Ray-Casting System. In *Computer Graphics*, Proceedings of ACM SIGGRAPH, pages 251–260, August 1999.

[Pho75] B. T. Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18(6):311—317, June 1975.

[RGW$^+$03] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Straßer. Smart Hardware-Accelerated Volume Rendering. In *Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, pages 231–238, May 2003.

[RKE00] S. Roettger, M. Kraus, and T. Ertl. Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection. In *IEEE Visualization*, pages 109–116, 2000.

[Rod99] F. Rodler. Wavelet based 3D Compression with Fast Random Access for Very Large Volume Data. In *Pacific Graphics*, pages 108–117, October 1999.

[Ros99] J. Rossignac. Edgebreaker: Connectivity Compression for Triangle Meshes. In *IEEE Transactions on Visualization and Computer Graphics*, volume 5 (1), pages 47–61. IEEE Computer Society, 1999.

[RSEB$^+$00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Texturing and Multi-Stage Rasterization. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 109–118, August 2000.

[RSG$^+$04] S. Roettger, A. Schieber, S. Guthe, W. Straßer, T. Ertl, and M. Stamminger. Tetrahedral Convexification and Lighting. In *IEEE Visualization*, 2004. submitted for publication.

[Sab88] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. In *Computer Graphics*, Proceedings of ACM SIGGRAPH, pages 59–64, August 1988.

[SBM94] C. M. Stein, B. G. Becker, and N. L. Max. Sorting and Hardware Assisted Rendering for Volume Visualization. In *IEEE Symposium on Volume Visualization '94*, pages 83–89, 1994.

[SDS96] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgann Kaufmann, 1996.

[SG98] O. G. Staadt and M. H. Gross. Progressive Tetrahedralizations. In *IEEE Visualization*, pages 397–402, 1998.

[SMW98] Claudio T. S., Joseph S. B. M., and Peter L. W. An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes. In *IEEE Symposium on Volume Visualization* , pages 87–94, 1998.

[SMW$^+$04] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2004. submitted for publication.

[SR99] A. Szymczak and J. Rossignac. Grow & Fold: Compression of Tetrahedral Meshes. In *Proceedings of the Fifth Symposium on Solid Modeling and Applications*, pages 54–64, June 9–11 1999.

[SS96] W. Sweldens and P. Schröder. Building Your Own Wavelets at Home. In “Wavelets in Computer Graphics”, ACM SIGGRAPH Course Notes, 1996.

[ST90] P. Shirley and A. Tuchman. A Polygonal Approximation for Direct Scalar Volume Rendering. In *Proceedings of San Diego Workshop on Volume Visualization SIGGRAPH*, November 1990.

[SW95] G. Sakas and S. Walter. Extracting Surfaces from Fuzzy 3D Ultrasonic Data. In *Computer Graphics*, Proceedings of ACM SIGGRAPH, pages 465–474, August 1995.

[TG98] C. Touma and C. Gotsman. Triangle Mesh Compression. In *Graphics Interface*, pages 26–34, June 1998.

[TR98] G. Taubin and J. Rossignac. Geometric Compression Through Topological Surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.

[VV98] V. Verma and A. VanGelder. Decimation of Tetrahedral Grids with Error Control. Technical Report UCSC-CRL-97-25, University of California, Santa Cruz, Jack Baskin School of Engineering, June 23, 1998.

[Wes94] R. Westermann. A Multiresolution Framework for Volume Rendering. In *IEEE/SIGGRAPH Symposium on Volume Visualization*, pages 51–58, October 1994.

[WG91] J. Wilhelms and A. Van Gelder. A Coherent Projection Approach for Direct Volume Rendering. In *Computer Graphics*, Proceedings of ACM SIGGRAPH, pages 275–284, August 1991.

[Wil92] P. L. Williams. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

[Wit99] C. M. Wittenbrink. CellFast: Interactive Unstructured Volume Rendering. Technical Report HPL-1999-81R1, Hewlett Packard Laboratories, September 1999.

[WKE02] M. Weiler, M. Kraus, and T. Ertl. Hardware-Based View-Independent Cell Projection. In *IEEE Symposium on Volume Visualization*, pages 13–22, 2002.

[WKG$^+$03] M. Weiler, M. Kraus, S. Guthe, T. Ertl, and W Straßer. Ray Casting with Programmable Graphics Hardware. In *Scientific Visualization: Extracting Information and Knowledge from Scientific Data Sets (DAGSTUHL 2003)*, October 2003.

[WM92] P. L. Williams and N. L. Max. A Volume Density Optical Model. In *ACM Computer Graphics (Workshop on Volume Visualization '92)*, pages 61–68, 1992.

[WMFC02] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral Projection using Vertex Shaders. In *IEEE Symposium on Volume Visualization*, pages 7–12, 2002.

[WMS98] P. L. Williams, N. L. Max, and C. M. Stein. A High Accuracy Volume Renderer for Unstructured Data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, January/March 1998.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6), June 1987.

[WNDS99] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley, Third edition, 1999.

[WS91] H. Watanabe and S. Singhal. Windowed Motion Compensation. In *Proc. SPIE's Visual Comm. and Image Proc.*, volume 1605, pages 582–589, 1991.

[WWH$^+$00] M. Weiler, R. Westermann, C. Hansen, K. Zimmermann, and T. Ertl. Level-of-Detail Volume Rendering via 3D Textures. In *IEEE/SIGGRAPH Symposium on Volume Visualization*, pages 7–13, October 2000.

[ZCK97] Y. Zhou, B. Chen, and A. Kaufman. Multiresolution Tetrahedral Framework for Visualizing Regular Volume Data. In *IEEE Visualization*, pages 135–142, October 1997.

Lebens- und Bildungsgang

7. Juni 1976	geboren in Castrop-Rauxel
1982 - 1986	Grundschule, Henrichenburg (Castrop-Rauxel)
1986 - 1995	Jugenddorf Christophorus Gymnasium, Altensteig Abschluss Abitur
1995	Teilnahme am 13. Bundeswettbewerb Informatik und erfüllen der Teilnahmebedingungen für die zweite Runde.
1995	Preis des Jugenddorf Christophorus Gymnasiums für herausragende Leistungen in den Fächern Mathematik, Physik und Informatik
1995 - 2000	Studium der Informatik an der Eberhard-Karls-Universität Tübingen
seit 2000	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Graphisch Interaktive Systeme am Wilhelm-Schickard-Institut für Informatik der Eberhard-Karls-Universität Tübingen (Prof. Straßer)
2003	NVIDIA Internship, September bis November
2003 - 2004	NVIDIA Graduate Research Fellowship